IBM® Information Management Software

IBM solidDB Delivering Data with Extreme Speed

Provides low latency, high throughput, and extreme availability

Offers fully featured relational in-memory database software

Has universal cache with shared memory access

Chuck Ballard
Dan Behman
Asko Huumonen
Kyosti Laiho
Jan Lindstrom
Marko Milek
Michael Roche
John Seery
Katriina Vakkila
Jamie Watters
Antoni Wolski

Redbooks

ibm.com/redbooks

International Technical Support Organization

IBM solidDB: Delivering Data with Extreme Speed

May 2011

SG24-7887-00

Note: Before using this information and the product it supports, read the information in "Notices" on page ix.

First Edition (May 2011)

This edition applies to Version 6.5 of IBM solidDB (product number 5724V17) and IBM solidDB Universal Cache (product number 5724W91).

Contents

Notices

Trademarks

Preface

The world seems to move more quickly these days; businesses want what they want right now. As such, we are moving quickly towards a real-time environment, which means instant access to information, immediate responses to queries, and constant availability worldwide. You must keep pace or be surpassed.

The world is also rushing to an open environment. Having separate systems, languages, and transactions to access data in multiple environments is becoming less of an accepted option. At IBM®, the move to higher levels of systems integration has been continuous, and is part of IBM solidDB®. For example, the Universal Cache in solidDB can speed access to data in IBM DB2® and IBM Informix®, and also to other relational databases such as Oracle, Sybase, and Microsoft® SQL Server.

Worrying about reliability is unnecessary because with solidDB all the data is always accessible from the in-memory cache. The reason is because solidDB also writes updated data to disk to ensure recoverability of the data. To do that, checkpoint and transaction logging functions exist. Therefore, even if the server fails between checkpoints, a transaction log contains all the committed transactions for recovery. In addition, to maintain a balance between performance requirements and logging capabilities, there is *strict logging* and *relaxed logging* functionality. Also, solidDB provides a *hot-standby* capability to give you extreme availability.

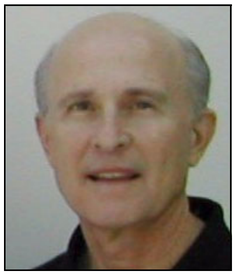
To get all these capabilities you might expect an expensive solution, with high maintenance costs. But solidDB helps avoid the costs associated with both planned and unplanned outages. For example, the costs for maintenance can be better controlled because most of it is performed by applications, which can run virtually unattended, resulting in reduced costs for administration.

When a database system can produce more throughput per hardware cost unit, with a constant software cost, the result is a higher return on investment (ROI). Further, it can deliver shorter response times, which increases the value of the service and which also increases your ROI. In-memory database systems can fill these expectations.

There is an evolution towards a real-time environment, bringing the potential for faster and more efficient access to data, analysis of that data, and the delivery of information for making more informed business decisions. We think that as you read further in this book and better understand the capabilities of solidDB, you will agree that this product can enable you to more easily realize those benefits.

The team who wrote this book

This IBM Redbooks® publication was produced by a team of specialists from around the world working with the International Technical Support Organization. The team began their work on the project at the IBM solidDB Development Lab in Helsinki, Finland and completed it working remotely at their home locations.



Chuck Ballard is a Project Manager at the International Technical Support organization, in San Jose, California. He has over 35 years of experience, holding positions in the areas of product engineering, sales, marketing, technical support, and management. His expertise is in the areas of database, data management, data warehousing, business intelligence, and process re-engineering. He has written extensively on these subjects, taught classes, and presented at conferences and seminars worldwide. Chuck has both a Bachelors degree and a Masters degree in Industrial Engineering from Purdue University.

Dan Behman is currently the solidDB Universal Cache Performance and Benchmarking Team Lead at the IBM Toronto Lab. He has worked for IBM for over 12 years in various positions in the DB2 for Linux®, UNIX®, Windows® area such as development, support, performance, and management. Besides database technology, Dan has extensive experience with using, troubleshooting, and developing applications for the Linux operating system, including co-authoring a book and writing several articles. Dan has an HBSc Computer Science Degree from Lakehead University.

Asko Huumonen is a Technical Sales Manager and worldwide Tiger team member for solidDB products. He joined IBM when Solid® Information Technology was acquired in 2008. Asko had worked for Solid since 1995 in various technical and management positions running professional services and technical support in Europe, in the U.S. and worldwide. He developed a deep understanding of the capabilities of the product and of customer systems by using the product. Prior to joining Solid, he worked 6 years in consulting, focusing on various software technologies. Asko has an MSc degree in Technical Physics from Helsinki University of Technology.

Kyosti Laiho is Technical Sales Manager for IBM solidDB, with 15 years of experience in working with customer solidDB environments on worldwide basis. He joined IBM with the acquisition of Solid Information Technology in 2008; he joined Solid in 1996. Kyosti (or "Koppa") had worked in multiple technical roles at Solid, including technical sales, managing support, consulting, training and worked both in North America (Silicon Valley) and Europe. Before joining Solid Information Technology Kyosti worked at Andersen Consulting (Accenture) for six years. His expertise is mostly in high performance database systems including high availability requirements, and the IBM solidDB related technologies. Kyosti has MSc in Computer Science from the University of Technology of Helsinki, Finland.

Jan Lindstrom is the development manager for solidDB core development. He joined IBM with the acquisition of Solid Information Technology in 2008. Before joining Solid in 2006, Jan spent almost 10 years working in the database field as a researcher, developer, author, and educator. He has developed experimental database systems, and has authored, or co-authored, a number of research papers. His research interests include real-time databases, in-memory databases, transaction processing and concurrency control. Jan has an MSc. and Ph.D. in Computer Science from the University of Helsinki, Finland.

Marko Milek is a Senior Manager responsible for worldwide development of the IBM solidDB product family. Prior to joining the solidDB development team at the IBM Helsinki Lab, he was a member of the DB2 LUW development team at the IBM Toronto Lab. Marko has 10 years of experience in software development, and release, project, and product management. His areas of expertise include relational database technologies, performance optimization and benchmarking, and in-memory databases. Marko holds a Bachelors degree in Physics from California Institute of Technology, a Masters degree in Computer Science and a Doctorate in Physics from McGill University.

Michael Roche is an Architect and Team Lead for the solidDB Universal Cache product. He joined IBM in 2009. Prior to that, Michael spent over 10 years designing and implementing enterprise middleware solutions, such as the IONA Orbix and Artix Mainframe products. A previous winner of the IBM-DCU U-18 All Ireland Programming competition, Michael was selected to coach the Irish team for the International Olympiad in Informatics (IOI). Michael holds a B.Sc. degree in Computer Applications and is currently pursuing an M.Sc. degree in Management of Information Systems at Trinity College, in Dublin Ireland.

John Seery is Development Manager of solidDB Universal Cache product development in IBM Ireland. He has 16 years experience in software engineering, including 10 in telecommunications. Prior to joining the solidDB team he worked in financial services solutions in IBM. He holds a B.Sc. and M.Sc. in Computer Science.

Katriina Vakkila is an Information Developer for the IBM solidDB product family. She has been writing user documentation for solidDB since 2008, prior to which she worked as a Documentation Specialist in the telecommunications field for more than five years. Katriina has a Masters degree in Applied Linguistics from University of Jyväskylä, Finland.

Jamie Watters is the Senior Product Manager of the IBM solidDB product family, which includes the solidDB database server and the solidDB Universal Cache. Prior to this role, Jamie held product and solution management positions at IBM for the InfoSphere™ Industry Data and Service Models and InfoSphere Master Data Management Server. Jamie has 13 years of experience working in the IT and software industry at both large corporations and start-ups.

Antoni Wolski is Chief Researcher for solidDB. He joined IBM with the acquisition of Solid Information Technology in 2008. Before joining Solid in 2001, Antoni spent almost 20 years working in the database field as a researcher, consultant, developer, author and educator. He has developed experimental database systems, and has authored and co-authored a number of research papers, books and software patents. His expertise is within special purpose database systems, including in-memory databases, active databases, heterogeneous databases and high-availability databases. Antoni has an MSc. and Ph.D. in Computer Science from the Technical University of Warsaw, Poland.

Other Contributors

We thank others who contributed to this book, in the form of advice, written content, and project support.

Darragh Coy: solidDB Universal Cache Engineer, IBM Software Group,
solidDB Universal Cache Development, Mulhuddart, Ireland

Mary Comianos: Publications Management
Emma Jacobs: Graphics Support
Ann Lund: Residency Administration
Diane Sherman: Editor
International Technical Support Organization, San Jose, CA

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

Introduction

In this chapter, we offer an insight into why a product such as IBM solidDB was conceived and delivered to the market. We discuss differences between the new in-memory database technology and traditional disk-based database technology. We introduce the idea of database caching and show how solidDB Universal Cache fills the need. We discuss the issues of throughput and response times in solidDB. Also, we discuss the competing solutions and guide the reader through the contents of this book.

1.1 The opportunity of the in-memory database

Consumers of IT products consistently work to increase application productivity and return on investment (ROI) in their enterprises. That applies, to a great degree, to database systems that are core to many solutions. If a database system can produce more throughput per hardware cost unit, with a constant software cost, that serves a higher ROI. If it can deliver shorter response times, that increase the value of the offered service, and ROI is again increased. In-memory database systems can fill both expectations.

To achieve that result, developers of in-memory databases strive to use the power of new computer hardware to its best potential.

What could not escape attention is a tremendous progress in the computer technology, bringing processors of increasingly growing processing power. The trend has increased the gap between the raw computing power captured in the processors and the capability of the I/O system to bring the data in and out fast enough. In the same time, the cost of main memory (RAM) has dropped significantly, enabling companies to have as much main memory capacity as is typically found in a modest size disk drive. Systems are already on the market that can accommodate up to a few terabytes of main memory.

The progress in processor technology and memory volumes has changed the landscape of computer platforms forever. It created a new opportunity for database management systems (DBMS) to minimize their dependency on disk-based systems and instead realize huge performance gains by using fast processors with large main memories.

In the following sections we describe why traditional database systems cannot stand up to the challenge, and why solidDB can. We also confront the most common misconceptions about in-memory databases.

1.1.1 Disk databases cannot expand to memory

From the beginning of the database era, disk drives were the only place to store and access large amounts of data in a reasonable time. DBMS designers concentrated on optimizing disk I/O and tried to align the data access patterns with the block structure imposed by the disk drives. Design strategy frequently centered on a shared buffer pool where data blocks were kept for reuse; advances in access methods produced solutions such as block-optimized indexes used to find the path to the data faster. Moving disk arms were a special challenge because each arm movement increased the access time. The ways to reduce arm movement were adjacent placing of data (data clustering), and writing transaction logs sequentially.

In the fierce battle for performance, disk I/O was often the deadliest enemy, and processing efficiency was sacrificed to avoid disk access. For example, with typical block page sizes of 8 KB or 16 KB, in-page processing is still inherently sequential and less CPU-efficient than random data access. Nevertheless, the page structures remained a popular way to reduce disk access. Meanwhile, query optimization tactics focused on minimizing page fetches where possible.

When the era of abundant memory arrived, many database administrators increased their buffer pools until the pools were large enough to contain an entire database, creating the concept of a *fully cached database*. However, within the large RAM buffer pools, the DBMSs were still hostage to all the structural inefficiencies of the block-oriented I/O strategy that had been created to deal with hard disk drives.

This way is all to be changed when developing an in-memory DBMS is to be done without compromising any of the *good* characteristics of a database, such as data persistency and transactional capabilities. The result is the IBM solidDB in-memory database (IMDB).

1.1.2 IBM solidDB IMDB is memory-friendly

By “memory-friendly” software, we mean one that executes memory-efficient code. Let us start with the block structures.

One of the most noticeable differences of an in-memory database system is the absence of large data block structures. IBM solidDB eliminates them. Table rows and index nodes are stored independently in memory, so that data can be added without reorganizing big block structures. In-memory databases also forgo the use of large-block indexes, sometimes called *bushy trees*, in favor of slim structures (*tall trees*) where the number of index levels is increased and the index node size is kept to a minimum to avoid costly in-node processing. IBM solidDB uses an index called *trie* (or prefix tree), which was originally created for text searching, but can be perfect for in-memory indexing.

A trie (the name comes from the word *retrieval*) is made up of a series of nodes where the descendants of a given node have the same prefix of the string associated with that node. For example, if the word *cat* is stored in a trie as a leaf node, it would descend from the node containing *ca*, which would descend from the node containing *c*. Trie indexes increase performance by reducing the need for key value comparisons, and practically eliminate in-node processing.

Another area of being memory-friendly, is checkpoint execution. A *checkpoint* is a persistent image of the whole database, allowing the database to be restarted after a system crash or other case of down-time. IBM solidDB executes a snapshot-consistent checkpoint that is alone sufficient to recover the database to a consistent state that existed at some point of time in the past. Other database products do not normally allow that; the transaction log files must be used to recalculate the consistent state. However, solidDB allows transaction logging to be turned off, if desired. The solidDB solution is memory-friendly by the ability to allocate row images and row shadow images (different versions of the same row) without using inefficient block structures. Only those images that correspond to a consistent snapshot are written to the checkpoint file, and the row shadows allow the currently executing transactions to run unrestricted during checkpointing.

There are more examples of “rethinking” of the principles of operation of a database system, resulting from the main-memory orientation. For example, differences can be seen in the way the query optimizer works, in the way a transaction log is implemented. and in the way applications connect to a database server, to name a few. Those solutions are described in a more detail in Chapter 2, “IBM solidDB details” on page 13.

To summarize, what makes a DBMS an in-memory DBMS? Look for the following characteristics:

- Has memory-optimized access methods
- Has memory-optimized checkpointing and logging
- Has both transient and persistent data structures. The latter ones are fully recoverable after a shutdown or a crash
- Does not have to wait for I/O on any read query execution.

Looking at it another way, taking a disk-based database system that was developed on day one, with a focus on reducing disk I/O, and converting it easily into an in-memory database system, focused on reducing CPU and memory, is not possible. Design choices and code implemented in the disk-based database remain, and continue to affect the performance and resources of the system. Another misconception is that increasing the size of the buffer pool, so that recently used data can be accessed quickly and without incurring the cost of I/O, is the same as getting an in-memory database. The truth is that managing the buffer pool requires substantial memory and CPU cycles, and the solution under performs as compared to an in-memory database. Essentially all you get is putting data blocks in RAM instead of disk, which, by itself, is not enough to proclaim the product to be a true in-memory database.

Alternatively, the in-memory database is not necessarily the best cure for all problems. The benefit gained from the memory-centered processing is sensitive to work loads and usage scenarios. See more about this in Chapter 7, “Putting solidDB and the Universal Cache to good use” on page 219.

Fortunately, with solidDB you can have a disk-based database also, if it fits the purpose. IBM solidDB is a *dual-engine DBMS*. Both engines, the main-memory engine (MME) and disk-based engine (DBE) can be used to implement a seamless *hybrid database* having both disk-based and in-memory tables.

To know more on how to deploy a database based on solidDB, see Chapter 4, "Deploying solidDB and Universal Cache" on page 67. Ways to monitor and optimize the database operation are described in Chapter 6, "Performance and troubleshooting" on page 147.

1.1.3 Misconceptions

In this section, we respond to a number of misconceptions about in-memory database. The term itself *in-memory database* can have various connotations, and some of them can be misconceptions, as follows:

- Data is volatile and non-persistent

  Not true. The fact that the main memory is volatile does not make the database volatile. The methods of checkpointing and transaction logging make sure the solidDB database is fully recoverable.

- Not a complete DBMS

  Not true. In terms of external interfaces, power of expression and utilities, solidDB is as complete as any other DBMS.

- SSD is just as fast

  Not true. Solid-state disks (SSDs) are slightly faster disks, with shorter access times. By replacing rotating disks with SSDs, you can improve the performance of a DBMS but the change does not turn the system into a in-memory database system. The load on the I/O system remains as huge as before, and in-memory operation remains as inefficient as before.

- No structured query language (SQL)

  Not true. IBM solidDB is a full-function SQL-based database system, including standard complying ODBC and JDBC interfaces

- Not reliable

  Not true. With solidDB, the database stands up to the standard definition of "a persistent data repository shared among concurrent applications." That also includes a concurrency control mechanism preventing data errors that can result from concurrent data usage. IBM solidDB maintains the transactional quality standard depicted as ACID (atomicity, consistency, isolation and durability), making it as reliable as any traditional DBMS.

- Poor availability

  Not true. Traditional high-availability solutions such as hot standby are available to in-memory databases too. IBM solidDB high availability offers the corresponding capabilities. For more, see Chapter 5, "IBM solidDB high availability" on page 109.

Misconceptions also exist about traditional databases. One example is as follows:

- High throughput means low response times

  Not with a concurrent load. On the contrary, the measures taken to increase the throughput degenerate response times. A good example is a common technique called *group commit* whereby several consecutive transactions are bundled for a single durable write in the disk storage. Consequently, the number of synchronous I/O operations is reduced and thus the throughput is improved. However, some of the transactions may wait for return from commit longer than they would do without the group commit.

1.1.4 Throughput and response times

In solidDB, the main focus is on short response times that naturally result from the fact that the data is already there (that is, it is in memory). Additional techniques are available to avoid any unnecessary latencies in the local database usage, such as linking applications with the database server code by way of special drivers. By using those approaches, one can shorten the query response time to a fraction (one tenth or even less) of that available in a traditional database system. The improved response times also fuels high throughput. Nevertheless, techniques of improving the throughput of concurrent operations are applied too. The outcome is a unique combination of short response times and high throughput.

The advantages of solidDB in-memory database over a disk-based database are illustrated in Figure 1-1 and Figure 1-2 on page 8. They unveil the results of an experiment involving a database benchmark called Telecom Application Transaction Processing (TATP)[1] that was run on a middle-range system platform[2]. For other configurations, both throughput (in transactions per second) and response time (shown as the longest response time of the best 90% of the transactions) are shown.

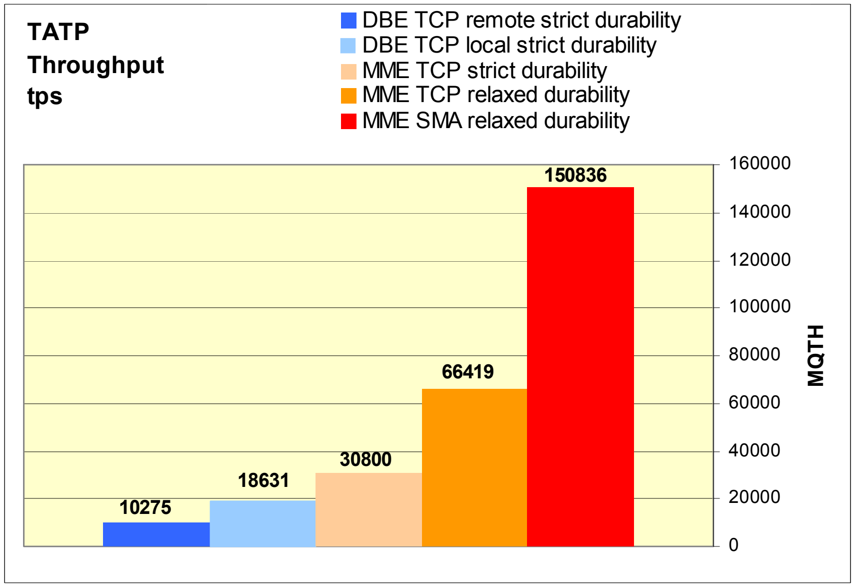


Figure 1-1 Throughput of the disk-base an in-memory database.

[1] http://tatpbenchmark.sourceforge.net/

[2] Two Intel® Xeon E5410 processors, total of 8 cores, 16 GB of memory, 2 SATA disks, under Linux RHEL 5.2. IBM solidDB v. 6.5.0.2. was used. TATP was configured for a 1 million subscriber database, and read/write mix of 80/20. The number of concurrent application clients was eight.

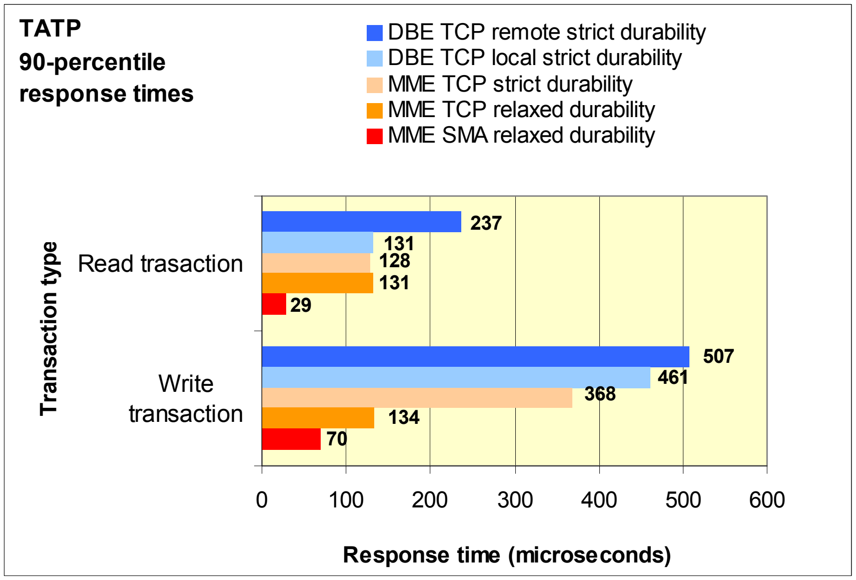


Figure 1-2 Response times in various configurations

The solidDB product was used for both the on-disk and in-memory databases. The charts illustrate how single configuration change steps can affect the throughput and response times. Between each two adjacent bars, only one configuration variable is changed. We describe the changes in more detail here. For each bar in Figure 1-1 on page 7, starting from the leftmost one, the corresponding configuration is as follows:

- DBE TCP remote strict durability

  In the chart, *DBE* means disk-based engine, *Remote* means accessing the database over the network, and *TCP* means a driver using TCP/IP. Strict durability means that the write operations are safely stored in a persistent storage before a commit request can be acknowledged. The configuration is typical for distributed client/server database applications.

- DBE TCP local strict durability

  Here, the term *remote* is replaced with *local*. That means running the application in the same node with the database server. The performance gain results from avoiding inter-computer communications. This is one example of bringing the data closer to the application. In all the remaining configurations the application is run locally.

- MME TCP strict durability

  DBE is replaced with *MME* (main-memory engine). The performance increase is attributed to faster operation of the database engine that is optimized for in-memory data. The faster are the interactions between the application and the server the bigger is the gain brought by MME. Running the applications locally is key to uncover that gain.

- MME TCP relaxed durability

  Here the term *strict* is replaced with *relaxed*. With relaxed durability, the commit acknowledgment is returned before the commit record is persistently written to the disk storage. The commit record is written to disk asynchronously, with a delay of few seconds. With relaxed durability, MME does not have to wait for I/O operations to complete, and that can be seen in the response time of write transactions. Overall, more CPU time can be utilized to process the in-memory data.

- MME SMA relaxed durability

  By replacing the TCP/IP based driver with an SMA driver, you ultimately bring the data as close to the application as possible, which is into the application's process space. Shared memory access (SMA) allows the user to link the application with a library containing the whole query processing code. At this point, all obstacles in the way of bringing the data to the application with a full CPU speed are removed.

In the experiment, the disk-based database was reasonably buffered: the size of the page buffer pool (called *DBE cache*, in solidDB) was 20% of the total database size. You can see that, by taking all the steps, the throughout can be increased almost 15 times, and the response time shortened to almost one tenth, compared to the worst case. Of all the steps shown, you might use only a few, but then, of course, there would also be fewer benefits. The subject of performance is further discussed in Chapter 6, "Performance and troubleshooting" on page 147, and Chapter 7, "Putting solidDB and the Universal Cache to good use" on page 219, including the description of the TATP benchmark.

1.2 Database caching with in-memory databases

In this section, we propose that using solidDB as a cache database to a larger enterprise database can alleviate the problems caused by the database growth.

1.2.1 Databases are growing

Enterprise databases can grow to very large sizes. The application load on the database system can grow to high levels too. That can lead to a situation where the database system becomes a bottleneck. In such an overloaded system, the response times may become painfully long.

One solution might be to scale the system up (to a more powerful computer) or out (to a cluster of computers). Both alternatives are costly and may expose scalability problems.

1.2.2 Database caching off-loads the enterprise server

You can off-load the enterprise database server by moving the data out of the server and close to where the applications are running. The solution is called *database caching* if the database interface in the applications can be preserved.

With database caching, all the data is still stored within the enterprise server. However, it is also replicated to smaller *cache databases*, typically residing in computer nodes where the applications are running. An in-memory database is an ideal candidate for a cache database because of the performance and response time advantages

1.2.3 IBM solidDB Universal Cache

IBM solidDB Universal Cache is a product enabling cache databases for popular enterprise database products such as Oracle, DB2, IDS, Sybase ASE, and Microsoft SQL Server. The solidDB in-memory database server maintains a cache database that contains a subset of data stored in the *back-end database* (being run by any of the systems previously mentioned). The applications connect to the cache database. Additionally, the product includes an elaborate replication mechanism known as InfoSphere Change Data Capture (InfoSphere CDC) available also as a separate product from IBM. The InfoSphere CDC components are responsible for carrying the data between the back-end database and the cache database--both ways.

The replication can take the form of one-time refreshes of cached data or continuous update propagation between the systems. The continuous replication is done in an asynchronous fashion: the data is first committed on the local system (the source) and then propagated to the other system (the target). The delay is usually within a second.

Because of the asynchronous nature of the continuous replication, the replication does not delay or block the transactions at the source side. However, there is one exception: if the cache database produces more updates that can be applied to the back-end database, the replication buffers fill-up, and a mechanism called *transaction throttling* is enacted. With transaction throttling, the cache database transactions are slowed down to keep up with the pace of data replication.

The advantage of an in-memory cache database is magnified with new access methods available with solidDB. By using linked library access or shared memory access, you can link the application with the server code and avoid any interprocess communications. With those approaches, the resulting response times can be an order of magnitude (or more) shorter than those achieved with the network-based access to the back-end enterprise server.

To know more about solidDB Universal Cache, see Chapter 3, “IBM solidDB Universal Cache details” on page 39 and Chapter 4, “Deploying solidDB and Universal Cache” on page 67.

1.3 Applications, competition, and the marketplace

Both solidDB and solidDB Universal Cache offer a new opportunity to certain applications characterized by the possibility to store the hottest part of the database in memory (or cache it from a back-end database) and a possibility to trade the increased memory cost for increased performance. Such applications exist in many sectors of the economy, such as telecommunications, real-time trading, online retail and booking, and online gaming, for example. See Chapter 7, “Putting solidDB and the Universal Cache to good use” on page 219 for more about applications.

When considering deployment of solidDB, the first competing solution can come from an unexpected direction, which is in-house development. In fact, application developers have been setting up application-specific in-memory stores and caches for a long time. Such solutions can be attractive from the performance perspective. However, in addition to increased performance, solidDB also has the following advantages:

- Total application life cycle management
- Data independence

  This is a core notion in databases, making the data and the application isolated from each other, with the purpose of allowing for change and growth.
- Other essential database characteristics, such as:
  - Date persistency
  - Transactional behavior
  - Recovery and high availability features.

These advantages are all productized in solidDB in a generalized way to also be immediately usable in other different applications. The situation is similar to what happened years ago when generalized disk-based data management systems overtook custom storage solutions.

Somewhere between an in-house data cache and the database are the concepts of *object store* and *object cache*. They both represent an object-oriented view on the data and are popular within Java™ community. Some of those solutions are commercialized and the vendors claim ease of use and fast access. This approach can be done, for example, if the vendors have removed the *impedance mismatch* between an object language, such as Java, and the relational model. They can also be fast after the data is cached. However, for any write-oriented activity having the purpose of producing persistent and consistent data, a DBMS is needed to take care of concurrency control, logging and recovery. Therefore, in those cases, it might be better to consider a database in the first place.

When speaking about solidDB as a product, it is not the only one available in the marketplace. That is, other vendors exist in the field of in-memory databases and database caching. However, we believe solidDB stands out from the other products in the following ways:

- As an in-memory database
  - With solidDB, a fully persistent and durable in-memory database is possible. Only a few other products are capable of doing that.
  - solidDB is a hybrid (on-disk and in-memory) database system. That can be achieved only in a few other products.
  - solidDB is equipped with low-latency access methods (direct linking). Only a few other products can provide that.
  - solidDB has rich, high functionality SQL, including stored procedures, triggers and events. Most in-memory products do not support those capabilities.
  - solidDB is equipped with a fully automatic high-availability (HA) solution with sub-second failover times. We believe that this is a level of data safeness and transparency that is unique to solidDB.
- As a cache database
  - solidDB Universal Cache is *universal* in the sense that it can be used with other models of back-end DBMSs. We believe this is a capability that is unique to solidDB.

To summarize, we believe that solidDB is the only product available today that has all of the capabilities and advantages we have just described.

Many of the topics in this chapter are described in more detail throughout the book.

2

IBM solidDB details

This chapter describes the architecture, data storage, table types, transactionality, structured query language (SQL) extensions, and database administration.

2.1 Introduction

IBM solidDB is a relational database server that combines the high performance of in-memory tables with the nearly unlimited capacity of disk-based tables. Pure in-memory databases are fast, but strictly limited by the size of memory. Pure disk-based databases allow nearly unlimited amounts of storage, but their performance is dominated by disk access. Even if the computer has enough memory to store the entire database in memory buffers, database servers designed for disk-based tables can be slow because the data structures that are optimal for disk-based tables are far from being optimal for in-memory tables.

The solidDB solution is to provide a single hybrid database server that contains two optimized engines inside it:

- The disk-based engine (DBE) is optimized for disk-based access.
- The main-memory engine (MME) is optimized for in-memory access.

Both engines coexist inside the same server process, and a single SQL statement may access data from both engines. The architecture for the solidDB hybrid server is depicted in Figure 2-1.

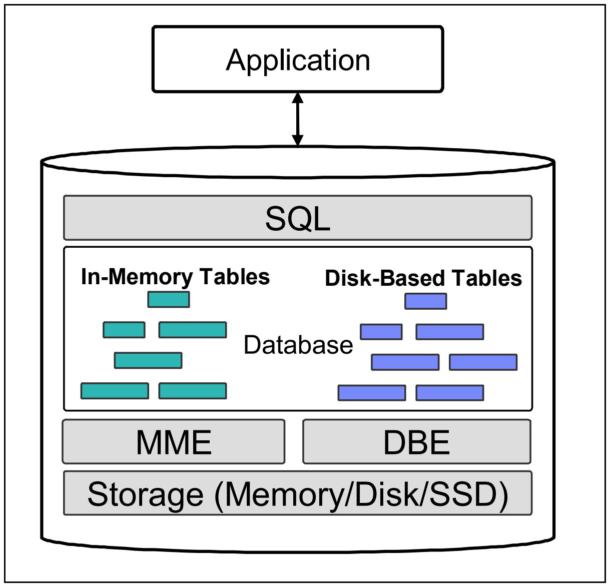


Figure 2-1 IBM solidDB hybrid server architecture

2.2 Server architecture

The solidDB server architecture is based on a client/server model. Typically, a solidDB configuration consists of cooperating server and client processes. The server process manages the database files, accepts connections to the database from client applications, and carries out actions on the database as requested by the clients.

The client process is used to pass the required tasks (through the server process) to the database. There can be several client types: a client could be a command-line tool, a graphical application, or a database management tool. Typically, separate applications act as clients to connect to solidDB.

The client and the server can be located on separate hosts (nodes), in which case they communicate over a network. IBM solidDB provides simultaneous support for multiple network protocols and connection types. Both the database server and the client applications can be simultaneously connected to multiple sites using multiple different network protocols.

IBM solidDB can also run within the application process. This capability is provided by solidDB shared memory access (SMA) and linked library access (LLA). In those cases, the application is linked to a function library that is provided with the product. The linked application communicates with the server by using direct function calls, thus skipping the overhead required when the client and server communicate through network protocols such as the TCP/IP. By replacing the network connection with local function calls, performance is improved significantly.

The high-level architecture of IBM solidDB is shown in the Figure 2-2.

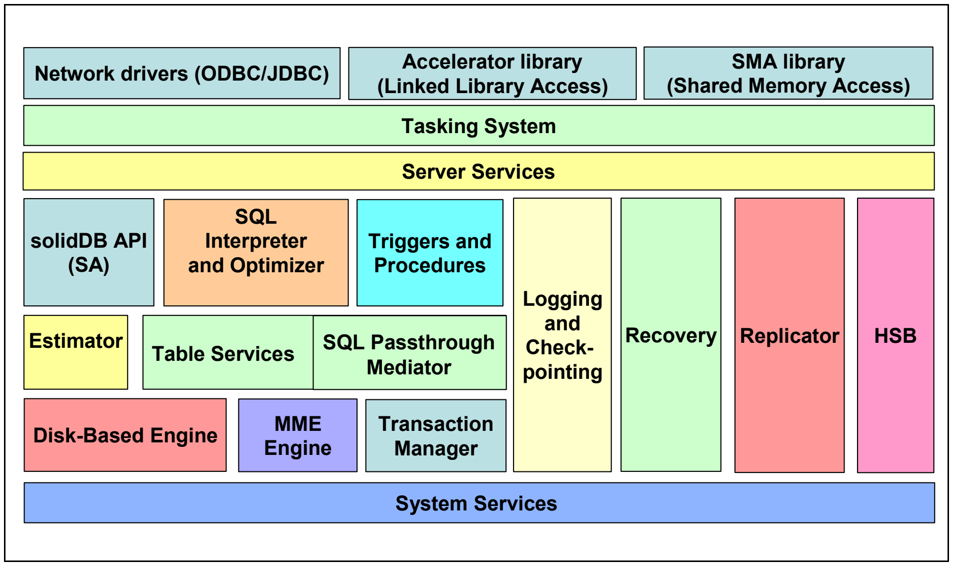


Figure 2-2 IBM solidDB architecture

2.2.1 Database access methods and network drivers

Applications can connect to the solidDB server using network drivers or by linking to the server directly. In network based access methods, the applications and the solidDB server are separate programs, typically communicating using the solidDB ODBC Driver or solidDB JDBC Driver.

Direct linking is provided through linked library access (LLA) and shared memory access (SMA). SMA and LLA are implemented as library files that contain a complete copy of the solidDB server in a library form.

The SMA and LLA servers can also handle requests from remote applications which connect to the server through communications protocols such as TCP/IP. The remote applications see the SMA or LLA server as similar to any other solidDB server; the local SMA and LLA applications see a faster, more precisely controllable version of the solidDB server.

The network drivers component contains support for both ODBC and JDBC API.

solidDB ODBC Driver

The solidDB ODBC Driver conforms to the Microsoft ODBC 3.5.1 API standard. It is distributed in the form of a library.

The solidDB ODBC Driver supported functions are accessed with solidDB ODBC API, a Call Level Interface (CLI) for solidDB databases, which is compliant with ANSI X3H2 SQL CLI. The solidDB implementation of the ODBC API supports a rich set of database access operations sufficient for creating robust database applications:

- Allocating and de-allocating handles
- Getting and setting attributes
- Opening and closing database connections
- Accessing descriptors
- Executing SQL statements
- Accessing schema metadata
- Controlling transactions
- Accessing diagnostic information

Depending on the application's request, the solidDB ODBC Driver can automatically commit each SQL statement or wait for an explicit commit or rollback request. When the driver performs a commit or rollback operation, the driver resets all statement requests associated with the connection

solidDB JDBC Driver

The solidDB JDBC Driver 2.0 is a JDBC type 4 driver. Type 4 means that it is a 100% Pure Java implementation of the Java Database Connectivity (JDBC) 2.0 standard. The JDBC API defines Java classes to represent database connections, SQL statements, result sets, database metadata, and so on. It allows a Java programmer to issue SQL statements and process the results. JDBC is the primary API for database access in Java.

The solidDB JDBC Driver is written entirely in Java and it communicates directly with the solidDB server using the TCP/IP network protocol. The solidDB JDBC Driver does not require any additional database access libraries. The driver requires that a Java runtime environment (JRE) or Java developer kit is available.

The solidDB server can also participate in distributed transactions using the Java Transaction API (JTA) interface. solidDB supports the following interfaces, as described in the Java Transaction API Specification 1.1:

- XAResource Interface (javax.transaction.xa.XAResource)
- Xid Interface (javax.transaction.xa.Xid)

solidDB SA

The solidDB SA is a low level C-language client library to access solidDB database management products. the solidDB SA is a layer that resides internally in solidDB products. Normally, the use of an industry standards-based interface, such as ODBC or JDBC, is recommended. However, in environments with heavy write-load, such as batch inserts and updates, solidDB SA can provide a significant performance advantage.

Linked library access (LLA)

With the LLA, an application links to a static library or a dynamic library that contains the full database server functionality. This means solidDB runs in the same executable with the application, eliminating the need to transfer data through the network. The LLA library is sometimes called an accelerator library.

Shared memory access (SMA)

With SMA, multiple applications can be linked to a dynamic driver library that contains the full database server functionality. This means that the applications ODBC or JDBC requests are processed almost fully in the application process space, without a need for a context switch among processes. To facilitate the processing of a common database, the driver has access to a shared memory segment initialized by the server.

2.2.2 Server components

In the remaining sections, we describe server components:

- Tasking system

  The tasking system is a framework to abstract threads to a concept task. Tasking system implements concurrent execution of the tasks also in single threaded systems.

- Server services

  The server services component contains services and utilities to use components on the lower levels.

- SQL interpreter and optimizer

  The SQL interpreter and optimizer is responsible of SQL-clause parsing and optimization. solidDB uses SQL syntax based on the ANSI X3H2 and IEC/ISO 9075 SQL standards. The SQL-89 Level 2 standard is fully supported and SQL-92 Entry Level. Many features of full SQL-92, SQL-99, and SQL-2003 standards are also supported.

  solidDB contains a cost-based optimizer, which ensures that even complex queries can be run efficiently. The optimizer automatically maintains information about table sizes, the number of rows in tables, the available indexes, and the statistical distribution of the index values.

- Triggers and procedures

  The triggers and procedures component contains a mechanism for parsing and executing SQL-based stored triggers and procedures:

  – A trigger activates stored procedure code, which a solidDB server automatically executes when a user attempts to change the data in a table.

  – Stored procedures are simple programs, or procedures, that are executed in solidDB databases. You can create procedures that contain several SQL statements or whole transactions, and execute them with a single call statement. In addition to SQL statements, 3GL type control structures can be used enabling procedural control. In this way complex, data-bound transactions may be run on the server itself, thus reducing network traffic.

- Logging and checkpointing

  The logging and checkpointing component is responsible for maintaining persistency of transactions by write-ahead logging, consistency, and recoverability of the database by checkpointing. Various durability options are available.

  Reading the transaction log file as it is being written by the server is possible and is done with a special SQL-based interface called *Logreader API*.

- Recovery

  The recovery component is responsible for recovery from transaction log and database checkpoints.

- Replicator

  The replicator component provides support for the solidDB advanced replication feature. The advanced replication feature is used for asynchronous, pull-based replication between a master database and replica databases. A “master” database contains the master copy of the data. One or more replica databases contain full or partial copies of the master's data. A replica database, like any other database, may contain multiple tables. Some of those tables may contain only replicated data (copied from the master), some may contain local-only data (not copied from the master), and some may contain a mix of replicated data and local-only data. Replicas may submit updates to the master server, which then verifies the updates according to rules set by the application programmers. The verified data is then “published” and made available to all replicas.

- HotStandby

  The HotStandby (HSB) component enables a secondary server (a hot standby server) to run in parallel with the primary server and keep an up-to-date copy of the data in the primary server.

- Estimator

  The estimator component provides cost-based estimates for single table access based on projections and constraints. It executes a low-level execution plan generation using index selection and index range calculations.

- Table services

  The table services module contains interfaces for single-table access, data type support, transaction management interface and, table and index caches.

- Disk-based engine

  The disk-based engine component handles the storage of disk-based tables (D-tables) and indexes.

- Main-memory engine

  The main-memory engine component handles the storage of in-memory tables (M-tables) and indexes.

- Transaction manager

  The transaction manager component contains commit and rollback implementation and concurrency conflict checking and resolution.

- System services

  The system services component contains operating system abstraction layer, memory management, thread management, mutexing and file I/O services.

2.3 Data storage in solidDB

The main-memory engine that handles the in-memory tables (M-tables) and the disk-based engine that handles the storage of disk-based tables (D-tables) use different data storage architectures.

In-memory engine is designed for maximum performance. Knowing that data is always stored in main-memory allows for use of data structures and data access methods that are designed to minimize the computational (CPU) cost of retrieving or updating database records.

Disk-based engine, however, can reduce disk access. That is achieved by data structures and access methods that trade disk access for additional computational processing. Therefore, an in-memory engine has the potential to outperform a disk-based engine even when the latter has all the data cached in the memory buffer pools because it needs to consume fewer CPU cycles to access database records.

We describe the data storage principles of both engines in more detail.

2.3.1 Main-memory engine

In addition to the actual data, the indexes for M-tables are built in main memory also. solidDB uses a main-memory-optimized index technology, called *tries*, to implement the indexes.

The basic index structure in the in-memory engine is a VTrie (variable length trie) that is optimized variation of the trie. A *trie* (from retrieval), is a multi-way tree structure widely used for storing strings. The idea is that all strings sharing a common prefix hang off a common node. For example, when the strings are words over {a..z} alphabet, a node has at most 27 children: one for each letter plus a terminator. VTrie uses bitwise tree where individual bits compose a key allowing keys to be any supported data type. VTrie uses nodes of the capacity of 8 bits. Consequently, each node has at most 257 children, that is, the fan-out is 257 (256 for bits plus a terminator).

A simplified example of the VTrie structure with node capacity of 2 bits and fan-out of four is shown in Figure 2-3.

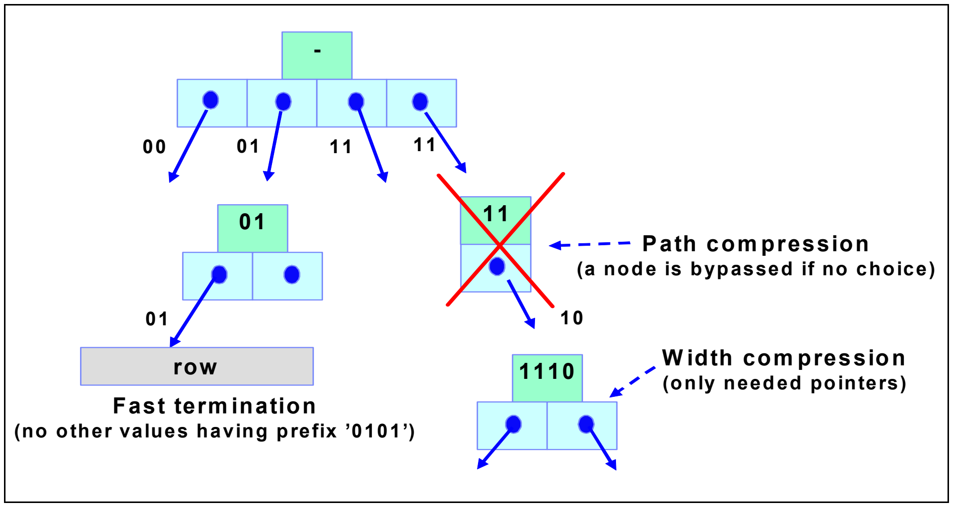


Figure 2-3 Simplified example of a VTrie structure

The elements in a string can be recovered using a scan from the root to the leaf nodes that ends a string. All strings in the trie can be recovered by a depth-first browse of the tree.

A competitive solution to VTrie would be a kind of a binary search tree. In a binary tree, the node fan-out is two. In each node you compare a full key value against a node separation value and then choose one of the two children to continue with.

The main advantages of VTries over binary search trees are as follows:

- Looking up keys is faster. Looking up a key of length m takes time that is proportional to m. A binary search tree requires $log2(n)$ comparisons of keys where n is the number of elements in the tree. The total search time is proportional to $m\ log2(n)$. The advantage of VTrie is because no value comparisons are needed. Each part of a key (a "letter") is applied as an array index to a pointer array of a child node. Contrary to a value comparison, array lookup is a fast operation if the array is cached in processor caches.
- Tries can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common prefix.

Several optimizations are used in Vtrie to speed up retrieval when the key value space is not fully exhausted, as illustrated in Figure 2-3 on page 21. These are *path compression*, *width compression*, and *fast termination*:

- In path compression all internal nodes with only one child are removed and a common prefix is stored in the remaining node.
- In width compression, only the needed pointers are stored in the nodes and every node contains a bitmap storing the information which pointers are present in the node.
- In fast termination, a pointer to the data record is elevated to a node representing a prefix that is not shared among the key values.

2.3.2 Disk-based engine

The main data structure used to store D-tables is a B+tree variation called B+tree. The idea of a B+tree is illustrated in Figure 2-4, where two node levels of a B+tree are shown.

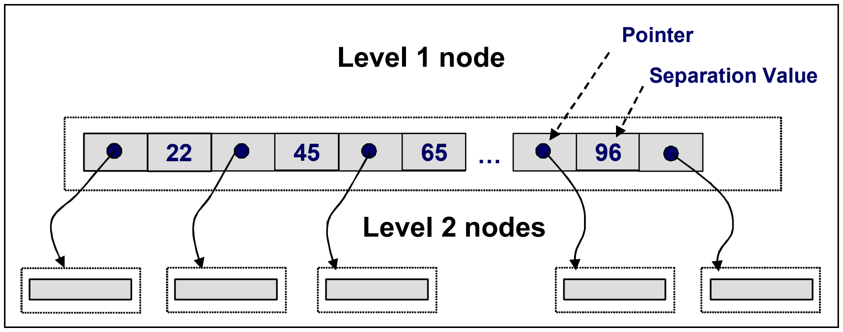


Figure 2-4 Illustration of two levels of a B-tree

Each node has a large set of value-pointer pairs. They normally fill a database page being a unit of data buffering. The page sizes vary from 4 to 32 kilobytes. Compared to VTrie, that makes the nodes much larger, resulting in a wide, or *bushy* tree. The key value is compared against the *separation values* in the node and, if the key value falls between two separation values, the corresponding pointer is followed to a similar node on the next level. Thanks to a large node size, the number of disk accesses is minimized, and that make B-tree fit for D-tables.

The server uses two incarnations of a B-tree: the main storage tree holds permanent data, and a differential index tree called *Bonsai Tree* stores new data temporarily until it is ready to be moved to the main storage tree.

In both B-tree structures, two space optimization methods are used. First, only the information that differentiates the key value from the previous key value is saved. The key values are said to be *prefix-compressed*. Second, in the higher levels of the index tree, the key value borders are truncated from the end; that is, they are *suffix-compressed*.

The main storage tree contains all the data in the server, including tables and indexes. Internally, the server stores all data in indexes; there are no separate table stores. Each index contains either complete primary keys (all the data in a row) or secondary keys (what SQL refers to as *indexes*, which is just the column values that are part of the SQL index). There is no separate storage method for data rows, except for binary large objects (BLOB) and other long column values.

All the indexes are stored in a single tree, which is the main storage tree. Within that tree, indexes are separated from each other by a system-defined index identification that is inserted in front of every key value. This mechanism divides the index tree into several logical index subtrees where the key values of one index are clustered close to each other.

The Bonsai Tree is a small active index (data storage tree) that efficiently stores new data (deletes, inserts, updates) in central memory, maintaining multiversion information. Multiple versions of a row (old and new) can coexist in the Bonsai Tree. Both the old and new data are used for concurrency control and for ensuring consistent read levels for all transactions without any locking overhead. With the Bonsai Tree, the effort needed for concurrency control is significantly reduced.

When a transaction is started, it is given a sequential Transaction Start Number (TSN). The TSN is used as the "read level" of the transaction; all key values inserted later into the database from other connections are not visible to searches within the current transaction. This approach offers consistent index read levels that appear as though the read operation was performed at the time the transaction was started. This way guarantees read operations are presented with a consistent view of the data without the need for locks, which have higher overhead.

Old versions of rows (and the newer version or versions of those same rows) are kept in the Bonsai Tree for as long as transactions need to see those old versions. After the completion of all transactions that reference the old versions, the “old” versions of the data are discarded from the Bonsai Tree, and new committed data is moved from the Bonsai Tree to the main storage tree. The presorted key values are merged as a background operation concurrently with normal database operations. This way offers significant I/O optimization and load balancing. During the merge, the deleted key values are physically removed.

2.4 Table types

This section describes the table types that solidDB offers, highlighting the key differences you should consider when deciding what type of tables to use.

The table types are shown in Figure 2-5.

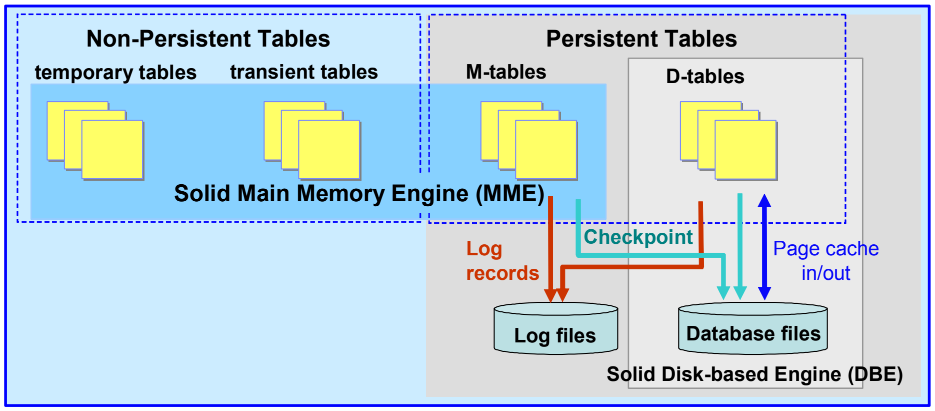


Figure 2-5 The solidDB table types

2.4.1 In-memory versus disk-based tables

If a table is designated as an in-memory table (M-table), the entire contents of that table are stored in memory so that the data can be accessed as quickly as possible. If a table is disk-based, (D-table), the data is stored primarily on disk, and usually the server copies only small pieces of data at a time into memory.

In many respects, in-memory tables are similar to disk-based tables:

- Both table types provide full persistence of data unless specified differently.
- You may perform the same types of queries on each of them.
- You can combine disk-based and in-memory tables in the same SQL query or transaction.
- Both table types can be used with indexes, triggers, stored procedures, and so on.
- Both table types allow constraints, including primary key and foreign key constraints.

The main difference between M-tables and D-tables is performance. M-tables provide better performance; they can provide the same durability and recoverability as D-tables. For example, read operations do not wait for disk access, even when the system is engaged in activities such as checkpointing and transaction logging.

2.4.2 Persistent versus non-persistent tables

The two basic types of M-tables are persistent tables and non-persistent tables. Persistent tables provide recoverability of data; non-persistent tables provide fast access. D-tables are always persistent tables.

Persistent tables

Persistent tables ensure recoverability of data through checkpointing and logging. *Checkpointing* means that committed transactions are copied from main memory to database files on disk during checkpoints. If the server fails between checkpoints, solidDB ensures that the disk has a consistent snapshot of the data. In-between checkpoints, solidDB writes committed transactions to a transaction log. After a system crash, solidDB uses the transaction log to recovers transactions that were committed since the last checkpoint.

By default, both M-tables and D-tables are created as persistent tables.

Non-persistent tables

Only M-tables can be created as non-persistent tables. Non-persistent tables are intended for use with temporary data that does not need to be recoverable. Data in non-persistent tables is never written to disk; therefore, any time that the server shuts down, whether normally or abnormally, the data is lost. Also, data in non-persistent tables is not logged or checkpointed. As a result, they are irrecoverable but remarkably faster than persistent tables.

The two types of non-persistent in-memory tables are transient tables and temporary tables[1]. Temporary tables are visible to a single connection; transient tables are visible to all connections (users) until the database shuts down. Because concurrent users cannot access data in temporary tables, temporary tables do not use concurrency control. Temporary tables are thus faster than transient tables.

Non-persistent tables cannot be used with solidDB HotStandby.

2.4.3 Choosing between different table types

The choice between the table types is typically a trade-off between performance and the following aspects:

- Amount of main memory available: M-tables or D-tables

  Ideally your system would have enough memory to store all of your tables in memory and thus benefit from the best possible performance for database transactions. If you cannot fit all tables in memory, try to put the most frequently used data in memory. Also, small, frequently-used tables should go into memory, and large, rarely-used tables can be left on disk.

- Recoverability of data: persistent or non-persistent tables

  Persistent tables provide full recoverability over performance. Non-persistent tables are faster as they require no logging or checkpointing.

- Access to temporary data: transient or temporary tables

  Transient tables allow multiple concurrent users to access the data over several connections, but require concurrency control (locking) to preserve consistency of data. Temporary tables are faster than transient tables but data is available only to a single user during one session.

[1] The solidDB implementation of temporary tables complies with the ANSI SQL:1999 standard for “Global Temporary Tables.” All solidDB temporary tables are global tables regardless of whether the keyword GLOBAL is specified. solidDB does not support “Local Temporary Tables” as defined by ANSI.

2.5 Transactionality

IBM solidDB guarantees reliable transactional processing by implementing a database server that satisfies all ACID (atomicity, consistency, isolation, durability) requirements.

- *Atomicity* requires that database modifications must follow an "all or nothing" rule. Each transaction is said to be atomic. If one part of the transaction fails, the entire transaction fails and the database state is left unchanged.
- *Consistency* ensures that any transaction that the database performs can take it from one consistent state to another.
- *Isolation* refers to the requirement that other operations cannot access data that has been modified during a transaction that has not yet completed. The question of isolation occurs in case of concurrent transactions (multiple transactions occurring at the same time).
- *Durability* is the ability of the DBMS to recover the committed transaction updates against any kind of system failure (hardware or software). Durability is the DBMS guarantee that after the user has been notified of a transaction's success, the transaction will not be lost.

2.5.1 Concurrency control and locking

The purpose of concurrency control is to prevent two users (or two connections by the same user) from trying to update the same data at the same time. Concurrency control can also prevent one user from seeing uncommitted (dirty) data while another user is in the process of updating it.

More generally, concurrency control is used to preserve the overall correctness of concurrent transaction executions. The ultimate form of that correctness is called *serializability*. A serializable execution of concurrent transactions produces a result that is identical to a case when all these transaction would be executed serially: one after another. Preserving generalized serializability for all possible cases is resource-consuming. Therefore, the actual correctness can be set with a parameter called *isolation level* that can be adjusted as needed, even dynamically.

IBM solidDB offers two concurrency control mechanisms, pessimistic concurrency control and optimistic concurrency control:

- *Pessimistic concurrency control* mechanism is based on locking. A lock is a mechanism for limiting other users' access to a piece of data. When one user has a lock on a record, the lock prevents other users from changing (and in some cases reading) that record.
- *Optimistic concurrency control* mechanism does not place locks but prevents the overwriting of data by using timestamps.

D-tables are by default optimistic; M-tables are always pessimistic. In D-tables, you can override optimistic concurrency and specify pessimistic locking instead. You can do this at the level of individual tables. One table might follow the rules of optimistic concurrency while another table follows the rules of pessimistic locking. Both tables can be used within the same transaction and even the same statement; solidDB handles this internally.

Pessimistic concurrency control

Pessimistic concurrency control (or pessimistic locking) is called *pessimistic* because the system assumes the worst; it assumes that two or more users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are.

The locks are placed as soon as any piece of the row is accessed, making it impossible for two or more users to update the row at the same time. Depending on the lock mode (shared, exclusive, or update), other users might be able to read the data although a lock has been placed.

Optimistic concurrency control

Optimistic concurrency control assumes that although conflicts are possible, they will be rare. Instead of locking every record every time that it is used, the system merely looks for indications that two users actually did try to update the same record at the same time. If that evidence is found, then one user’s updates are discarded and the user is informed. The step of checking whether a transaction can commit is called *transaction validation*. Typically, the validation is performed at the commit time but solidDB uses, by default, a modified method called *early validation*. With early validation, the data being read (the *read-set*) and written (*write-set*) are checked against other transactions at each database operation, without waiting for commit. If the data in the read-set and write-set has changed since the beginning of the transaction, the transaction is considered to be violating the data consistency and is aborted. The details of checking rules for read-sets and write-sets depend on the isolation level that will be discussed shortly.

Optimistic concurrency control is available for disk-based tables (D-tables) only.

Locking and performance

Optimistic concurrency allows fast performance and high concurrency (access by multiple users), at the cost of occasionally refusing to write data that was initially accepted but was found at the last second to conflict with another user's changes.

Pessimistic locking requires overhead for every operation, whether or not two or more users are actually trying to access the same record. The overhead is small but adds up because every row that is updated requires a lock. Furthermore, every time that a user tries to access a row, the system must also check whether the requested row or rows are already locked by another user or connection.

When using M-tables, best performance in reference to locking is achieved with temporary tables; because concurrent users cannot access data in temporary tables, temporary tables do not use concurrency control.

When using D-tables, optimistic concurrency control provides the best performance, if the possibility for conflicts is low. For example, there are many records but relatively few users, or few updates and mostly read-type operations. However, in workloads that expose more updates, like in typical online transaction processing (OLTP) applications, pessimistic locking is actually more beneficial.

2.5.2 Isolation levels

solidDB supports the isolation levels defined in the SQL-92 standard, except for the READ UNCOMMITTED level[2]. The isolation level can be set per session or per statement.

The three supported isolation levels are explained in the following sections.

READ COMMITTED

This isolation level allows a transaction to read only committed data. However, the view of the database may change in the middle of a transaction when other transactions commit their changes.

In solidDB HotStandby configurations, the isolation level of the Secondary server is always READ COMMITTED.

[2] Uncommitted, or dirty, reads violate transactional paradigm as modified data becomes visible before the transaction performing the update completes. Hence the READ UNCOMMITTED isolation level is rarely, if ever, used in production database systems.

REPEATABLE READ

This isolation level allows a transaction to read only committed data and guarantees that read data will not change until the transaction terminates. With optimistic D-tables, solidDB, in fact, maintains an isolation model called *snapshot isolation*. It ensures that the transaction sees a consistent view of the database. When using optimistic concurrency control, conflicts between transactions are detected by using only the transaction write-set validation. For example, if a transaction involves one read and one update, solidDB validates that no one has updated the same row in between the read operation and the update operation. In this way, lost updates are detected, but the read is not validated. With transaction write-set validation only, transactions are not serializable in many cases. Additionally, phantoms may occur. Phantoms are table rows that appear (are seen) in the course of the transaction although were not seen in the beginning. Such rows may result from insert and update activities by other concurrent transactions.

In M-tables, the repeatable read level is implemented in a more traditional level—by way of locks. Shared locks are kept on all the data items read until the transaction commit. That preserves other transactions from changing the read-set. The correctness of the write-set is assured with exclusive locks on all the items written. Because both read-set a d write-set are protected, transactions running on M-tables, in the repeatable read mode, are serializable with the exceptions of phantoms.

SERIALIZABLE

This isolation level allows a transaction to read only committed data with a consistent view of the database. Additionally, no other transaction may change the values read by the transaction before it is committed because, otherwise, the execution of transactions cannot be serialized in the general case.

With D-tables, solidDB can provide serializable transactions by detecting all conflicts between transactions. It does this by using both write-set and read-set validations. Because no locks are used, all concurrency control anomalies are avoided, including the phantoms.

The SERIALIZABLE isolation level is not supported with M-tables.

2.5.3 Durability levels

The durability level controls how solidDB handles transaction logging. The solidDB server supports three durability levels: strict, relaxed, and adaptive. Relaxed durability yields best performance; strict durability minimizes loss of transactions. The adaptive durability level is available only in HotStandby configurations.

The durability level can be set as a server default, per session, or per transaction.

Strict durability: synchronous logging

With strict durability, transaction logging is synchronous: the transaction is written to the transaction logs as soon as the transaction is committed.

Relaxed durability: asynchronous logging

With relaxed durability, transaction logging is asynchronous: solidDB is permitted to defer the transaction write until the server is less busy, or until it can write multiple transactions together.

In a server that is not part of a HotStandby pair, using relaxed durability means that you risk losing the most recent few transactions if the server terminates abnormally. If the server is part of a HotStandby pair, a copy of the transaction is on the other server (the Secondary); even if the Primary server fails before logging the transaction, the transaction is not lost. Thus, when relaxed durability is used with HSB, relaxed durability causes little reduction in safety. On the other hand, relaxed durability can improve the performance of the system, especially in situations where the server load consists of a large number of small write transactions.

Adaptive durability

Adaptive durability applies only to HotStandby Primary servers. Adaptive durability means that if the server is in Primary Active state (sending transactions to the Secondary), it will use relaxed durability. In any other state it will use strict durability. This gives you high performance (with little loss of safety) when HSB is active, yet maintains high safety if only one server is operating. Adaptive durability is effective only when the HotStandby has been set to use a 2-safe replication: the Primary server does not tell the client that the transaction has been successfully committed until the Primary receives acknowledgement that the Secondary has the transaction.

2.6 solidDB SQL extensions

The SQL support in solidDB is comparable to any advanced SQL-based system; solidDB offers the most commonly expected features and a set of useful extensions employing solidDB-specific (nonstandard) SQL syntax. Additionally, procedural SQL extensions such as stored procedures and triggers enable moving parts of the application logic into the database. These extensions help reduce network traffic, thus improving performance.

2.6.1 solidDB SQL standard compliance

No commercial relational DBMS fully supports the SQL standard beyond the SQL-92 Entry Level, and solidDB is no exception. The full standards known as SQL-92, SQL-99, and SQL 2003 are too broad to be implemented in a cost-efficient manner.

solidDB supports the SQL-92 Entry Level fully and has adapted selected features from the broader standards. An example of advanced standard features is the possibility to manage table constraints dynamically by using the ALTER TABLE syntax.

In addition to standard features, solidDB also borrows suitable, nonstandard solutions from other proprietary products. Examples are as follows:

- `START WITH ... CONNECT BY` syntax for calculating hierarchical queries
- LIMIT clause for limiting the size of the result set

2.6.2 Stored procedures

Stored procedures are simple programs, or procedures, that are compiled and parsed after and then stored in the database for future execution. Because stored procedures are stored and executed directly in the server, usage of stored procedures reduces network traffic and can thus improve performance. For example, complex, data-bound transactions may be run on the server itself.

You can create a procedure that contains several SQL statements or a whole transaction and execute it with a single call statement. In addition to SQL statements, 3GL type control structures can be used enabling procedural control. You can also create nested stored procedures where one procedure is executed from within another.

Stored procedures can also be used for controlling access rights and database operations. Granting execute rights on a stored procedure automatically invokes the necessary access rights to all database objects used in the procedure. Therefore, administering database access rights may be greatly simplified by allowing access to critical data through procedures.

Stored procedures are created and called using SQL statements.

The three calling methods for the stored procedures are local, remote and deferred stored procedures:

- Local procedures are executed on a local database server.
- Remote procedures are procedures that are stored on one server and called by another. Remote stored procedures are applicable only to advanced replication setups.
- Deferred procedures are procedures that are called after commit has been processed.

2.6.3 Triggers

A *trigger* is a mechanism for executing a series of SQL statements when a particular action (an INSERT, UPDATE, or DELETE) occurs. The trigger contains SQL statement that need to be executed when the trigger is invoked. Triggers are created using solidDB proprietary stored procedure syntax.

You can create one or more triggers on a table, with each trigger defined to activate on a specific INSERT, UPDATE, or DELETE command. When a user modifies data within the table, the trigger that corresponds to the command is activated.

You can use only inline SQL or stored procedures with triggers. If you use a stored procedure in the trigger, the procedure must be created with the CREATE PROCEDURE command. A procedure invoked from a trigger body can invoke other triggers.

Triggers enable you to perform the following tasks:

- Implement special integrity constraints, such as checking that certain conditions are maintained, for example, to prevent users from making incorrect or inconsistent data changes.
- Take action based on the value of a row before or after modification.
- Transfer much of the logic processing to the back end, reducing the amount of work that your application needs to do and reducing network traffic.

2.6.4 Sequences

Sequences are objects that are used to get sequence numbers in an efficient manner. Sequence objects can be used, for example, to generate primary key numbers. The advantage of using a sequence object instead of a separate table is that the sequence object is specifically fine-tuned for fast execution and requires less overhead than normal update statements.

By default, solidDB sequences are sparse. Being *sparse* means that there is no guarantee that the generated sequence numbers are consecutive (they are, however, unique). Another possibility is a *dense* sequence. In that case the generated sequence numbers follow each other. The penalty of dense sequences is that they are locked by the transactions incrementing them, so no two transactions can increment the same sequence in the same time. One of the transactions must wait until the other transaction commits or aborts. Sparse sequences are more performant because they are not locked by the incrementing transactions.

Sequence objects are created with the CREATE SEQUENCE or CREATE DENSE SEQUENCE statement. Sequence values can be incremented and used within SQL statements using the sequence_name.CURRVAL and sequence_name.NEXTVAL constructs. Sequences can also be used inside stored procedures.

2.6.5 Events

Event are database objects that are used to signal events in solidDB databases. Together with stored procedures, events can be used for automating administrative tasks. You can make your application use event alerts instead of polling, which uses more resources.

The events mechanism is based on one connection waiting on an event until another connection posts that event. More than one connection may wait on the same event. If multiple connections wait on the same event, all waiting connections are notified when the event is posted. A connection may also wait on multiple events, in which case it will be notified when any of those events are posted.

In addition to system events, solidDB supports also user-defined events. However, user-defined events can only be used within stored procedures; system events can also be used without stored procedures. The events are managed using SQL statements.

2.6.6 Replication

IBM solidDB is equipped with three data replication technologies:

- Advanced replication

  This method to disseminates parts of a master database to remote locations called replicas. With extended SQL syntax, a master user can define publications (with CREATE PUBLICATION) being views of a database. Users at replicas can subscribe to those publications and request data refreshes. Propagation of updates from the replicas to the masters is also possible. Because permanent connections between the masters and replicas are not required, advanced replication is suitable for applications operating in loosely connected networks, for example with mobile replica devices.

- HotStandby replication

  The solidDB HA product called solidDB HotStandby applies continues transactional replication between the active node and the standby node. A user can make no choices about the replicated data: the whole database is always replicated. However, controls (both configuration parameters and ADMIN COMMANDs) exist so that the user can start and stop the replication, and change the characteristics. Both synchronous and asynchronous replication modes are possible. The solidDB HA solution is described in Chapter 5, “IBM solidDB high availability” on page 109.

- Logreader API

  If none of the these replication methods fit the user’s needs, one can develop a custom replication solution using the *logreader*. Logreader is a component in the server, externalizing the contents of the transaction log. With the log reader, all changes made to the database can be read from the log and transferred elsewhere. The interface is in the form of a SELECT statement reading from a virtual (non-existing physically) table called SYS_LOG. The log reading is done through a standard ODBC/JDBC interface and thus it can be executed both locally and remotely. The replication solution used in solidDB Universal Cache is based on the logreader.

2.7 Database administration

This section describes the main principles of database administration with solidDB.

2.7.1 Configuration settings

Most solidDB configuration settings are defined using configuration parameters that are stored in a `solid.ini` configuration file. The `solid.ini` file is not mandatory; if no configuration file exists, the factory values are used. Also, all parameters do not need to be present in the `solid.ini` file; if a parameter is not present in the `solid.ini` file or if the value for a particular parameter is not set, the factory value is used.

Generally, the factory values offer good performance and operability but in some cases modifying some parameter values can improve performance. You might also need to set configuration parameters to enable or disable certain functionality.

You can set the configuration parameter values by editing the `solid.ini` file manually or, in most cases, using ADMIN COMMANDs, a set of solidDB proprietary SQL statements.

Parameters are defined as parameter name value pairs. The parameters are grouped according to section categories. In the `solid.ini` file, each section category starts with a section name inside square brackets, for example:

```
[Logging]
LogEnabled=yes
```

Tip: In documentation, parameters are typically referred to in the format `section.parameter`, for example, `Logging.LogEnabled`.

Some parameter settings, such as durability level, can also be overridden per session or per transaction by using the SQL commands SET or SET TRANSACTION, or by defining the settings per connection with the ODBC connection attributes or JDBC connection properties. The precedence hierarchy is as follows (from high precedence to low):

- SET TRANSACTION: transaction-level settings
- SET: session-level settings
- ODBC connection attributes and JDBC connection properties
- Parameter settings specified by the value in the solid.ini configuration file
- Factory value for the parameter

Additionally, you can control some solidDB server operations with the following options:

- solidDB command line options at solidDB startup
- environment variables
- ODBC client connect string arguments

2.7.2 ADMIN COMMAND

The ADMIN COMMAND is a SQL extension specific to solidDB and that executes administrative commands.

The ADMIN COMMANDs are used for operations such as creating backups of the database, invoking performance monitoring, or displaying information about users connected to the database. The ADMIN COMMANDs can also be used for changing certain configuration settings dynamically.

2.7.3 Data management tools

IBM solidDB provides a set of utilities for performing various database tasks.

solidDB SQL Editor (solsql)

solidDB SQL Editor (`solsql`) is a console tool used to issue SQL statements and solidDB ADMIN COMMANDs at the command prompt, or by executing a script file that contains the SQL statements.

Tip: When using `solsql`, ADMIN COMMANDs and SQL statements must be terminated with a semicolon (;) character. Note that if you are not using `solsql`, terminating SQL statements with a semicolon leads to a syntax error.

solidDB Console (solcon)

solidDB Console (`solcon`) is a console tool used to issue solidDB ADMIN COMMANDs at the command prompt, or by executing a script file that contains the commands. Only users with administrator rights can access `solcon`; if only `solcon` is deployed at a production site, the administrators cannot accidentally execute SQL statements that could change the data.

Tools for exporting and loading data

solidDB provides the following tools for exporting and loading data:

- solidDB Speed Loader (**`solloado`** or **`solload`**) loads data from external files into a solidDB database.
- solidDB Export (**`solexp`**) exports data from a solidDB database to files. It also creates control files used by solidDB Speed Loader (solloado or solload) to perform data load operations.
- solidDB Data Dictionary (**`soldd`**) exports the data dictionary of a database. It produces an SQL script that contains data definition statements that describe the structure of the database.

2.7.4 Database object hierarchy

solidDB uses *catalogs* and *schemas* to organize data. solidDB's use of schemas conforms to the SQL standard but solidDB's use of catalogs is an extension to the SQL standard.

The solidDB syntax for database object hierarchy is as follows:

```
catalog_name.schema_name.database_object
```

Catalogs are the highest (broadest) level of the hierarchy. A catalog can be seen as a logical database, and two or more catalogs can be used in the same time with the help fully qualified table names. Schema names are the mid-level of the hierarchy; specific database objects, such as tables, are the lowest (narrowest) level. Thus, a single catalog may contain multiple schemas, and each of those schemas may contain multiple tables.

Object names must be unique within a catalog, but they do not have to be unique across catalogs.

The default catalog name is the system catalog name that was specified during database creation. The default schema name is the user name. Objects can be created without specifying the catalog and schema name; by default, the server uses the system catalog and the user name of the object creator to determine which object to use.

3

IBM solidDB Universal Cache details

This chapter describes the inner workings of IBM solidDB Universal Cache. Consequently, the chapter is intended for a more technical audience.

Specifically, the chapter details components that comprise the IBM solidDB Universal Cache product and describe the architecture and operation of the overall product.

The chapter also addresses common usage patterns for IBM solidDB Universal Cache, enumerating key considerations for how to best use the solution to effect performance gains.

3.1 Architecture

The architecture of solidDB Universal Cache is based on three main components: the solidDB in-memory database (the cache database), the relational database server (the *back end*), and the data synchronization software that copies data to and from the cache and the back end. The replication method is asynchronous, ensuring fast response times.

3.1.1 Architecture and key components

The architecture and key components of a typical configuration of the solidDB Universal Cache is shown in Figure 3-1.

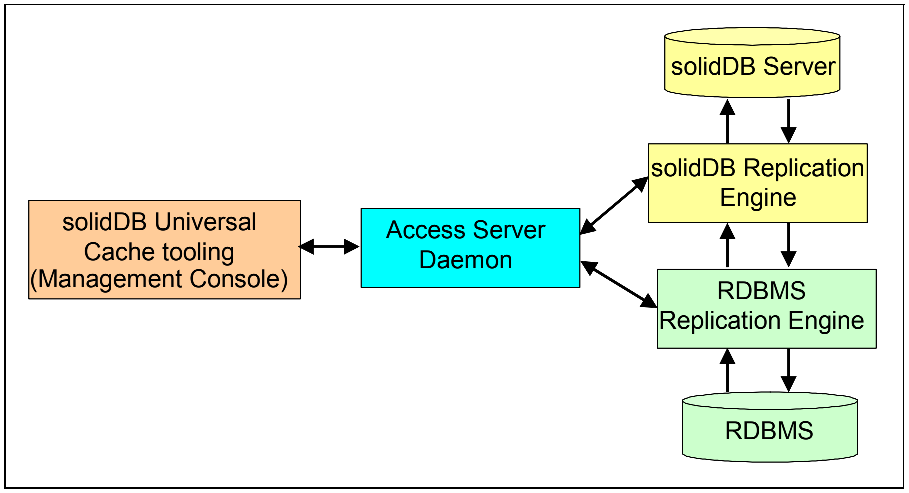

solidDB Server
solidDB Replication Engine
solidDB Universal Cache tooling (Management Console)
Access Server Daemon
RDBMS Replication Engine
RDBMS

Figure 3-1 Sample IBM solidDB deployment

IBM solidDB: cache database

The solidDB server implements the cache database (or front end) in the IBM solidDB Universal Cache solution. The cache database benefits from various solidDB features, such as HotStandby that provides high availability and failover, or shared memory access (SMA) that enables collocating of data with the application.

Relational database server (RDBMS): back end

The RDBMS is a relational, disk-based data server that contain the data to be cached.

Replication engines

The InfoSphere Change Data Capture (CDC) replication software ensures that as changes are made to the cache database, the back-end database is updated, and vice versa. The replication engines run typically on the same hosts as the data servers.

The replication engines are configured using a graphical user interface (GUI) or command-line based Configuration Tool (**`dmconfigurets`**). A set of commands (dm-commands) is available and can be used to control the replication engine instances.

Access Server

InfoSphere CDC Access Server is a process that manages a solidDB Universal Cache deployment. It is typically executed as a daemon.

Configuration tools such as Management Console communicate with the Access Server to allow deployments to be configured.

Access Server controls access to the replication environment; only users who have been granted the relevant rights can modify configurations. However, after the replication environment has been configured, Access Server is not needed for the replication to be operational: only the InfoSphere CDC replication engines need to be running.

Configuration tools

InfoSphere CDC Management Console is a graphical application that allows authorized users to access, configure and monitor their InfoSphere CDC deployments. It does so by communicating with the Access Server.

Similar functionality is available for command-line users. This functionality is realized through the **`dminstancemanager`** and **`dmsubscriptionmanager`** utilities, which are included the InfoSphere CDC for solidDB package.

3.1.2 Principles of operation

To use solidDB Universal Cache, you must first identify the data you want to cache and configure the environment accordingly. The data can then be loaded from the back-end database to the cache, so that when applications run against the cache database, they can take advantage of high performance and low latency of solidDB. (With the SQL pass-through functionality, some statements can also be passed to the back-end database.) As changes are made to the data, the InfoSphere CDC replication technology synchronizes data between the cache database and the back-end database.

Log-scraping

InfoSphere CDC uses log-scraping technologies, triggers, or both to capture databases changes. The front-end replication engine accesses the solidDB transaction log to capture data changes and transmits these changes to the back-end replication engine, which copies the changes to the back-end database.

Similarly, the back-end replication engine accesses the log (or uses triggers) to capture data changes in the back end and transmits these changes to the front-end replication engine, which copies the changes to the back-end database.

Asynchronous replication

The InfoSphere CDC replication method is asynchronous in nature. This means that as applications write, for example, to the cache database, control is returned to the application as soon as the write completes; the application does not block, waiting for these updates to be successfully applied to the back end.

Updates to the back end are not performed until the following tasks are completed:

1. The transaction has been committed.
2. The entries for the transaction are scraped from the log.

In a solidDB Universal Cache environment, asynchronous replication benefits applications by reducing the round-trip time required to access data. Instead of potentially incurring an expensive network hop and writing to the back-end database, applications can write directly to the solidDB in-memory database.

Asynchronous replication means also that applications cannot assume that the back-end database has been written to at the same time as the front-end, which can have ramifications for error recovery.

Mirroring and refresh

The two manners in which data can be copied between the cache database and the back end are mirroring and refresh.

Mirroring involves actively scanning a *source* database to see if any changes have been made, then applying these changes to a *target* database. This step is accomplished by using the asynchronous replication mechanism. The mirroring process may be thought of as *active caching*.

Refresh involves taking a snapshot of the source database and writing it directly to the target. Refresh can thus be utilized to initialize or rebuild a database.

Communication between components

The InfoSphere CDC replication components communicate with each other using TCP/IP. To collocate the data in the cache database with the application, the solidDB server can be configured as a shared memory access (SMA) server, so that both the application and the InfoSphere CDC for solidDB replication engine connect to solidDB using SMA. TCP/IP protocol can be used with solidDB too.

The inter-component communication in the solidDB Universal Cache environment is shown in Figure 3-2.

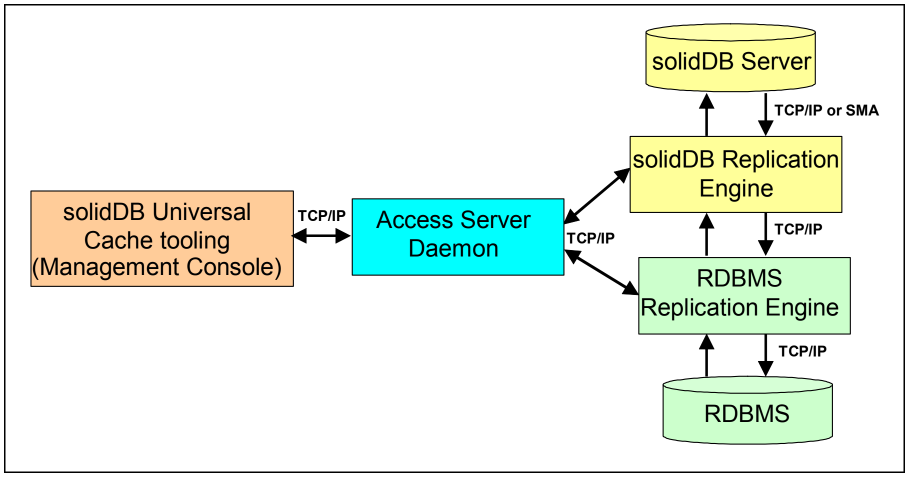
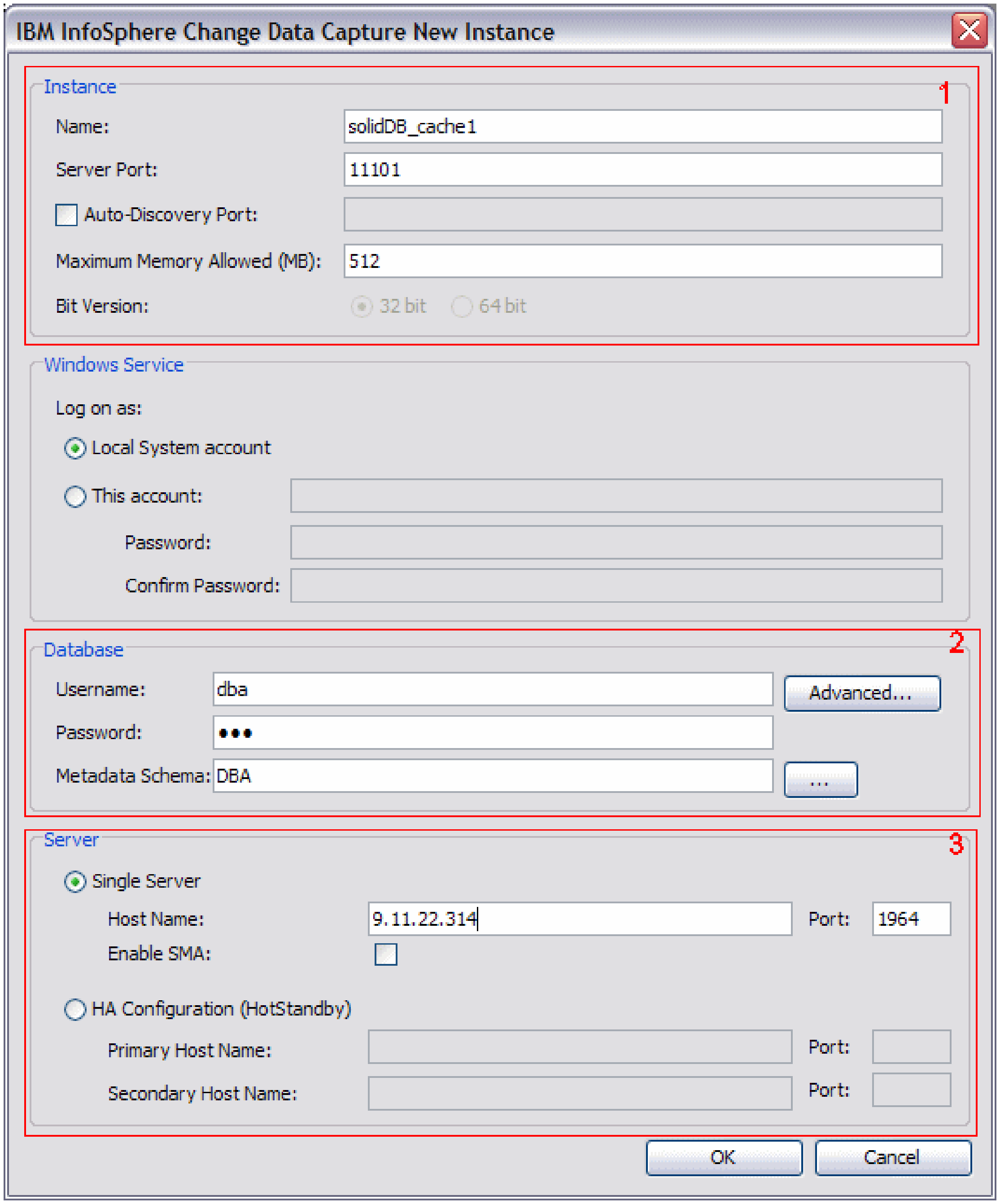


Figure 3-2 solidDB Universal Cache inter-component communication

The Access Server is configured as a TCP/IP server and it listens on one or more ports. On UNIX systems, it may be deployed as a daemon service (inetd). All communication between the Access Server and tooling also uses TCP/IP.

Each replication engine instance must use a unique port number to connect to the Access Server; the port number is defined when creating the InfoSphere CDC instances. Figure 3-3 on page 44 shows the configuration dialog for the InfoSphere CDC for solidDB replication engine.


IBM InfoSphere Change Data Capture New Instance

Instance 1

Name: solidDB_cache1

Server Port: 11101

Auto-Discovery Port:

Maximum Memory Allowed (MB): 512

Bit Version: [✓ 32 bit] 64 bit

Windows Service

Log on as:

[✓ Local System account]

This account:

Password:

Confirm Password:

Database 2

Username: dba

Password: •••

Metadata Schema: DBA

Server 3

[✓ Single Server]

Host Name: 9.11.22.314 Port: 1964

Enable SMA:

HA Configuration (HotStandby)

Primary Host Name: Port:

Secondary Host Name: Port:

Figure 3-3 InfoSphere CDC for solidDB configuration

The following areas are highlighted in Figure 3-3 on page 44:

1. Instance area: Server Port defines the port number, which InfoSphere CDC instance uses for communication with Access Server and other replication engines.
2. Database area: Defines the user account to access the database that contains the tables for replication, in this case, the solidDB database.
3. Server area: Defines the connection information to the database that contains the tables for replication, in this case, a stand-alone solidDB server.

3.2 Deployment models

The solidDB Universal Cache architecture affords much flexibility. For example, different cache instances can be configured to maintain identical copies of the same data, to facilitate load balancing for read or read-write access. Alternatively, large tables in the back-end database can be partitioned, where each data partition can be hosted by a dedicated in-memory cache instance, with read or read-write access.

Depending on the application needs, solidDB Universal Cache can be deployed as a read-only cache or as a read-write cache.

Read-only cache

When configured as a read-only cache, the data is owned by the back-end database. This “ownership” means that the data stored in the cache cannot be modified by the application. In this configuration, applications can modify the data directly in the back-end database and changes can be synchronized to the in-memory cache, transaction by transaction, automatically or on-demand. This configuration is ideal for applications that require fast access to data that changes occasionally, such as price lists, or reference or lookup data.

Read-write cache

There are two deployment options for read-write cache, depending on the ownership of data.

When configured as a read-write cache, where the data is owned by the cache, applications can read, add, modify, or delete data in the cache, but not in the back-end database. Changes are propagated from the in-memory cache to the back-end database, transaction by transaction, automatically, or on-demand. This configuration is useful for applications that have stringent service level agreements that demand short response times, for a variety of data intensive operations.

When configured as a read-write cache where the data ownership is shared, applications can update the same data in both the cache and in the back-end database at the same time. In this case, changes to the data can be propagated automatically in both directions. Conflicts are detected and resolved by using predefined conflict resolution methods. This configuration is especially useful when applications need to update the data in the back-end database while the data is also cached for read-write access.

3.3 Configuration alternatives

This section describes the configuration alternatives for the deployment options.

3.3.1 Typical configuration

A simple solidDB Universal Cache deployment might involve caching a single back-end database to a single cache database. The cache database related components and the back-end related components are installed on separated nodes, as are the Access Server and tooling. A typical node configuration is shown in Figure 3-4.

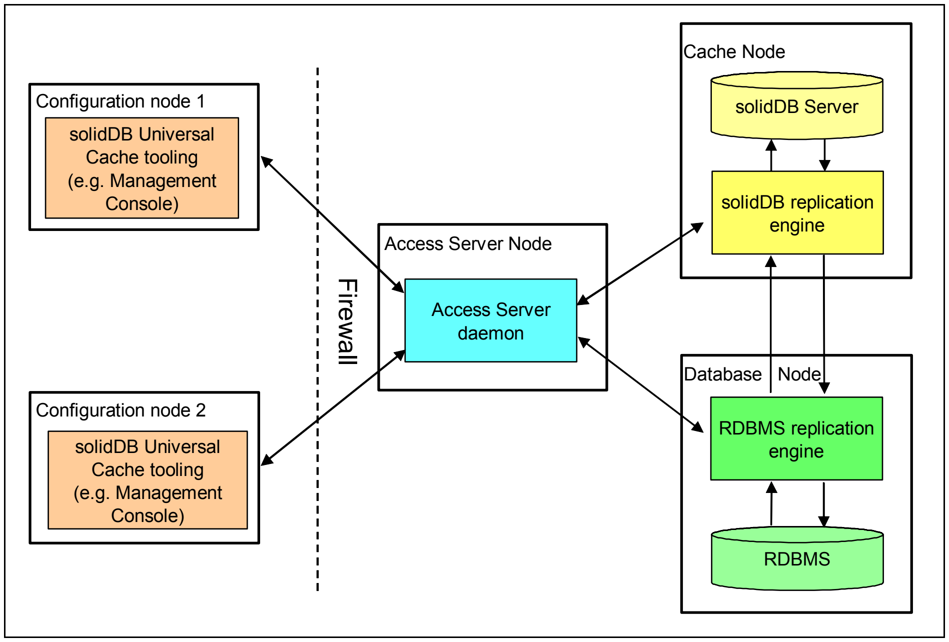


Figure 3-4 A simple solidDB Universal Cache deployment with a single cache node

Note the following information:

- Single cache node

  The solidDB server and InfoSphere CDC for solidDB products are typically installed and configured on the same machine (“cache node”). This machine is often “closer” to the applications that use the data.

  Collocation of the two servers minimizes the overhead when scraping the logs (solidDB as source) or applying updates (solidDB as target).

- Single database node

  The InfoSphere CDC for back end is typically installed and configured on the same machine on which the back-end RDBMS is running (“database node”).

  This approach helps to minimize the overhead of communications between the replication engine and the database.

- Single access node

  The Access Server is typically deployed on a separate machine (“access node”). The advantage to installing it on a separate machine from “cache” and “database” nodes is to more easily configure the firewall, because solidDB Universal Cache tooling communicates with the Access Server, not with the InfoSphere CDC replication engines.

  InfoSphere CDC Access Server is only required during the configuration of solidDB Universal Cache, or during the starting or stopping of caching (that is, subscriptions).

- Configuration nodes

  Any node from which solidDB Universal Cache tooling (Management Console, **`dmsubscriptionmanager`**, Access Server tools) is run can be considered a configuration node.

3.3.2 Multiple cache nodes

Multiple solidDB servers (cache nodes) can be used, for example, for partitioning back-end data so that each cache node has only the data that is relevant to it. However, in such deployments, each solidDB server is autonomous and processes the application requests without accessing data in any of the other solidDB servers.

3.3.3 SMA for collocation of data

The shared memory access (SMA) feature of solidDB Universal Cache can boost application performance when accessing solidDB data. In place of costly network-based communication, such as TCP/IP, SMA uses direct function calls to the solidDB server code. In the same time, the in-memory database is located in a shared memory segment available to all applications connected to the server with SMA. Multiple concurrent applications running in separate processes can utilize that access method to reduce significantly response times.

In a SMA setup, the application, the solidDB SMA server, and the InfoSphere CDC replication engine are located on the same node. In the setup phase, the following steps must be considered:

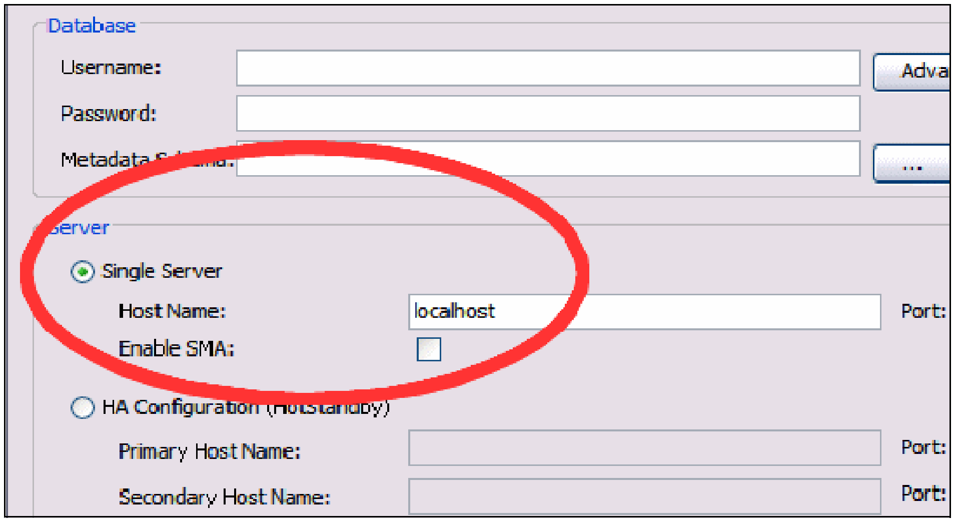
1. Configuring solidDB server to run as a SMA server
2. Configuring InfoSphere CDC for solidDB to use SMA for communication with solidDB server
3. (Optional) Configuring user applications to use SMA for communication with solidDB server

Configuring solidDB server to run as a SMA server

To use solidDB with SMA, the solidDB server is started with the **`solidsma`** executable, instead of the **`solid`** executable.

Configuring InfoSphere CDC to support SMA

SMA must be enabled during the creation of an InfoSphere CDC for solidDB instance. For example, in the GUI tool, the **Enable SMA** check box must be selected; see Figure 3-5.



Username:
Password:
Metadata Schema:
Single Server [✓]
Host Name: localhost
Enable SMA: ☐
Primary Host Name:
Secondary Host Name:

Figure 3-5 Enabling SMA on an InfoSphere CDC for solidDB instance

Configuring user application to support SMA

The SMA feature does not require any code changes to the applications themselves, except for ensuring that the solidDB connection is configured for SMA. The SMA connection is defined within the ODBC connection string or JDBC connection property. For example, when using ODBC, instead of connecting to solidDB using the connection string `‘tcp 2315’`, the SMA connection is specified with the string `‘sma tcp 2315’` string. When using JDBC, the following connection property is used:

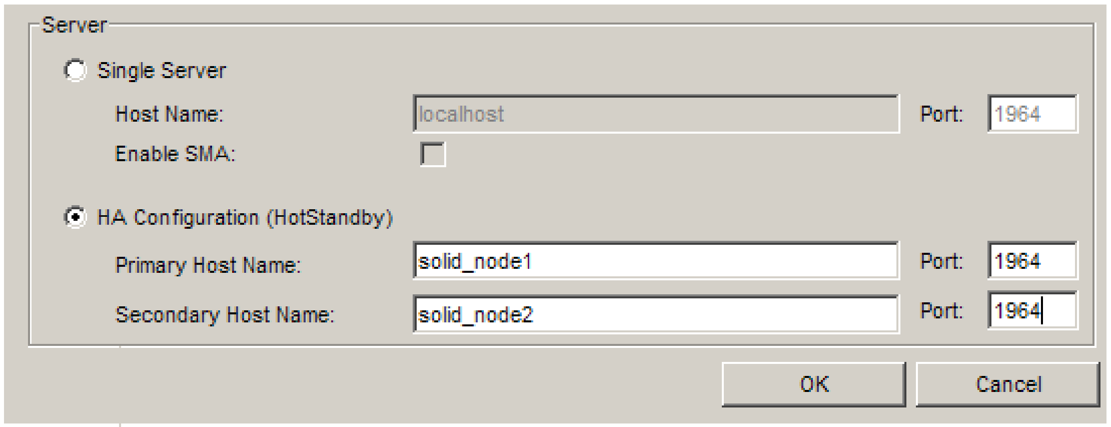
```
solid_shared_memory=yes
```

3.3.4 solidDB HSB servers for high availability

The solidDB HotStandby (HSB) solution allows for redundancy to be incorporated at each individual cache node, thus providing reliable access to data stored in the cache database.

Reliability in the cache database requires that the data pathways to and from the cache are capable of handling HSB failovers. The InfoSphere CDC for solidDB can be made aware of HSB deployments transparently, so that replication to and from the cache remains operational, after the primary solidDB server is down. If the primary solidDB server does go down, InfoSphere CDC simply redirects active subscriptions to use the new primary solidDB server and replication continues as normal.

The HSB support must be enabled explicitly when configuring the InfoSphere CDC for solidDB instance by defining the connection to the primary and secondary servers, as shown in Figure 3-6.


Server

Single Server
Host Name: localhost Port: 1964
Enable SMA: ☐

[✓ HA Configuration (HotStandby)]
Primary Host Name: solid_node1 Port: 1964
Secondary Host Name: solid_node2 Port: 1964

Figure 3-6 Enabling HotStandby in InfoSphere CDC for solidDB

3.4 Key aspects of cache setup

The usage of solidDB Universal Cache solution requires the implementation of replication *subscriptions* between the cache and the back-end database.

A subscription defines the replication direction and various replication rules. The subscriptions also maintain the state of replication, indicating whether or not replication is in progress.

The application and deployment needs dictate the direction of the subscriptions between a *source* and *target* data store. In the InfoSphere CDC replication solution, a *data store* is the representation of a database and the related InfoSphere CDC instance.

The cache and back end can act as both source and target data stores in different subscriptions. There can also be several subscriptions between two data stores; multiple subscriptions can be used to partition the data and workload.

Data stores and subscriptions are created and managed with the Management Console or the **`dmcreatedatastore`** and **`dmsubscriptionmanager`** command-line tools.

3.4.1 Deciding on the replication model

Before creating subscriptions, consider the following information:

- Ownership of data

  Does the master copy of the data reside in the back-end database, as is typically the case, or does the master copy of the data reside in the cache?

- Read-only or read-write cache

  Do you want changes made to the cache to be reflected in the back-end database, or is the cache read-only?

Typically, the back-end database represent the master copy of the data and data must be cached in read-only mode.

For such setups, only a single subscription is required. The back-end (RDBMS) data store should be used as the subscription source and the cache data store (solidDB) should be used as the subscription target. This configuration ensures that any changes made to the back end can be replicated to the cache.

A list of common subscription configurations is shown in Table 3-1. The Procedure column also refers to the necessary recursion prevention method, which is discussed in more detail.

Table 3-1 Creating subscriptions

Cache type	Behavior	Procedure
Back-end owned, read-only cache	Typical scenario. Changes made to back-end database are reflected in cache.	Create a single subscription using the back-end data store as source and the cache data store as target.
Back-end owned, read-write cache	Changes made to back-end database are reflected in cache; changes made to cache are reflected in back end.	Create a subscription using the back-end data store as source and the cache data store as target. Enable prevent recursion. Specify SOURCE wins as conflict resolution option. Create another subscription using the cache data store as source and the back-end data store as target. Enable prevent recursion. Specify TARGET wins as conflict resolution option.
Cache owned, archival	Changes made to cache are archived to back end.	Create a single subscription using the cache data store as source and the back-end data store as target.
Cache owned, read-write cache	Changes made to back-end database are reflected in cache; changes made to cache are reflected in back end.	Create a subscription using the cache data store as source and the back-end data store as target. Enable prevent recursion. Specify SOURCE wins as conflict resolution option. Create another subscription using the back-end data store as source and the cache data store as target. Enable prevent recursion. Specify TARGET wins as conflict resolution option.

3.4.2 Defining what to replicate

Each subscription must contain table mappings that define the table subsets that are to be replicated from the source data store to the target. Sample table mappings are shown in Table 3-2.

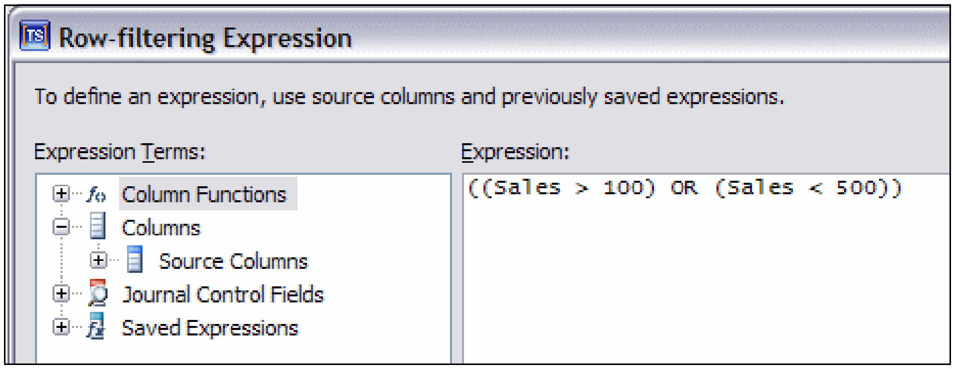
The table mappings are created and managed with the Management Console or the **`dmsubscriptionmanager`** command-line tool.

Table 3-2 Example of table subsets that can be used when defining subscriptions

Description	**Example**	**Behavior**
Complete table	Sample.EMPLOYEES → DBA.EMPS	Specifies that table Sample.Employees in the source database should be replicated to the target table DBA.EMPS.
Table with column filters	Sample.EMPLOYEES → DBA.EMPS Column Filter = COUNTRY	Specifies that the Sample.Employees table should be replicated, but that the COUNTRY column should be excluded.
Table with row filters	Sample.EMPLOYEES → DBA.EMPS Row Filter = “((COUNTRY=’IE’) OR (COUNTRY=’FI’))”	Specifies that table Sample.Employees in the source database should be replicated to the target table DBA.EMPS. Only rows with a country value of ‘IE’ or ‘FI’ are replicated.
Tables with row and column filters	Sample.EMPLOYEES → DBA.EMPS Row Filter = “((COUNTRY=’IE’) OR (COUNTRY=’FI’))” Column Filter = AGE	Specifies that table Sample.Employees should be replicated, but that the AGE column should be omitted and only rows where country is ‘IE’ or ‘FI’ should be selected.

Defining subset of data with row and column filters

A table mapping may define column filters, row filters, or both that restrict the amount of data that is replicated to the target database. An example of a row filter is depicted in Figure 3-7.

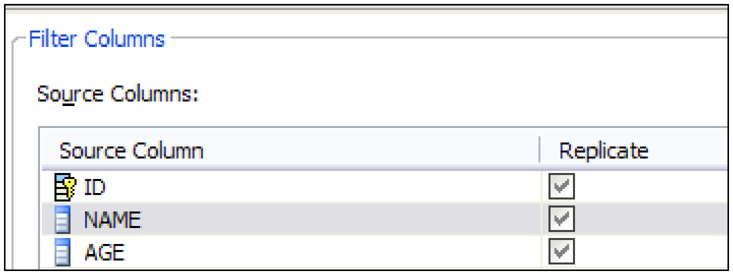


To define an expression, use source columns and previously saved expressions.

```
((Sales > 100) OR (Sales < 500))
```

Figure 3-7 Specifying a row filter

An example of a column filter is depicted in Figure 3-8.



Source Column	Replicate
ID	✓
NAME	✓
AGE	✓

Figure 3-8 Specifying a column filter

Ensuring consistency of data

This section describes the capabilities for ensuring data consistency.

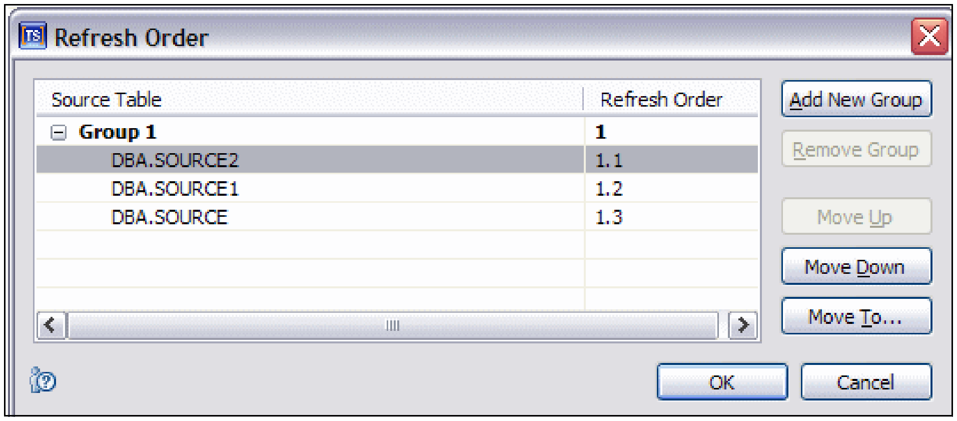
Referential integrity

Referential integrity is an important concept in databases that deals with table relationships and how tables should refer to the same underlying data. These relationships are described using primary keys and foreign keys and ensure that data needs to be defined only once to the system.

When using solidDB Universal Cache, asynchronous replication is used to copy data between cache database and back-end database, or vice versa. It is important that data is copied in the correct order, such that referenced records are copied before referencing records.

In solidDB Universal Cache, referential integrity associations are to be confined within subscription; foreign keys cannot point to tables outside the subscription.

Also, if you intend to use the Refresh operation to synchronize data between the cache and back end, the subscriptions with referential integrity constraints must define a refresh order; the refresh order specifies that referenced tables are listed first and referencing tables listed last (as depicted in Figure 3-9). The referential integrity must also be enforced on the solidDB data stores by setting an InfoSphere CDC for solidDB system parameter (refresh_with_referential_integrity) using the Management Console.



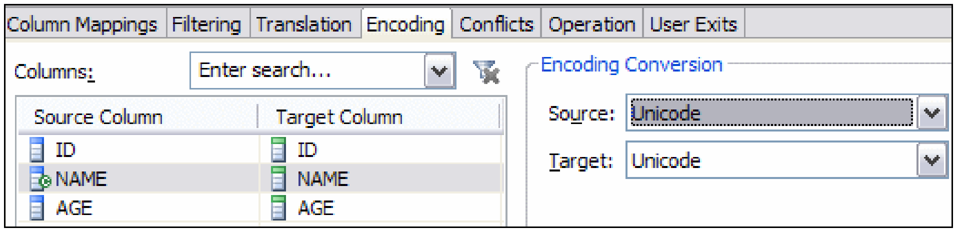
Source Table	Refresh Order
Group 1	**1**
DBA.SOURCE2	1.1
DBA.SOURCE1	1.2
DBA.SOURCE	1.3

Figure 3-9 Specifying a refresh order

Encoding and replication of character data

The cache and the back-end database might use different character encoding for data. InfoSphere CDC can replicate character data among a wide variety of encodings and can automatically convert the data from the column encoding detected on the source to the column encoding detected on the target.

In some cases, you might need to define the encoding conversions manually. You can specify character encoding at the column level for subscriptions using the Management Console (as shown in Figure 3-10).



Source Column	Target Column
ID	ID
NAME	NAME
AGE	AGE

Source: Unicode
Target: Unicode

Figure 3-10 Specifying encodings to use during caching

In solidDB, the encoding of character data depends on the database mode; a solidDB database is created either in Unicode mode or partial Unicode mode (default).

When a new instance of InfoSphere CDC for solidDB is created, the partial Unicode mode is assumed; default encoding for columns of CHAR type is set to ISOLatin1 and for WCHAR type it is set to UTF-16BE.

Note the following information:

- If the encoding of character data in your partial Unicode database is not ISO Latin1, you must select the correct encoding to reflect the nature of data stored in character columns.
- If your solidDB database mode is Unicode, you must specify the encoding of character columns as UTF-8.

Conflict resolution

In read-write cache setups, both the cache database and back-end database could be modified at the same time, resulting in conflicting operations. InfoSphere CDC has the capability of detecting conflicts and resolving them according to user defined logic. Conflicts are detected and resolved on a table basis at the target node of a subscription.

The most simple type of conflict resolution method that can be employed is a *Source Wins* or *Target Wins* rule. However, no matter which conflict resolution method is chosen, the basic premise is always the same: the rule makes a decision about which version of the data to keep, thus resolving the conflict.

For Source Wins and Target Wins, the logic is as follows:

- Source Wins: The incoming change from the subscription’s source database is to be maintained. Changes made to the subscription’s target database is overridden.
- Target Wins: The current data in the subscription’s target database is to remain unchanged. The incoming change from the subscription’s source database is to be ignored.

By using these two rules, you can implement a simple precedence scheme, whereby you decide to always keep either the cache or database version of the data when conflicts occur.

In addition to Source Wins and Target Wins rules, InfoSphere CDC also offers comparative based rules such as Largest Value Wins. More complex (user programmed) rules can also be specified through the User Exit mechanism, enabling domain specific business logic to be applied to the resolution process. For example, in addition to resolving the conflict, User Exit rules can be used to also log details of the conflict for later auditing.

3.4.3 Starting replication

After a subscription has been configured, refresh or continuous mirroring operation may be performed on it:

- *Refresh* operation takes a full snapshot of the source data store and copies it to the target data store. It is a one-time off operation and runs to completion. The refresh operation is used to initialize the target data store and may also be used when and where a full rebuild of the target is required.
- *Continuous mirroring* operation listens for changes that are made to a source data store and copies these to the target, so long as the data is not being filtered through a row or column filter. Continuous mirroring is an active process.

3.5 Additional functionality for cache operations

This section describes key operational aspects of solidDB Universal Cache.

3.5.1 SQL pass-through

The SQL pass-through feature of solidDB Universal Cache allows applications to access the front-end database (the cache) and back-end RDBMS database with a single connection. In other words, applications can access both cached and non-cached data, negating the need to maintain an explicit connection to the back-end database.

The implementation of pass-through relies on solidDB being configured to use an ODBC driver to communicate with the back-end database. The solidDB server then uses this driver to execute pass-through statements directly against the back-end RDBMS database, as shown in Figure 3-11 on page 57.

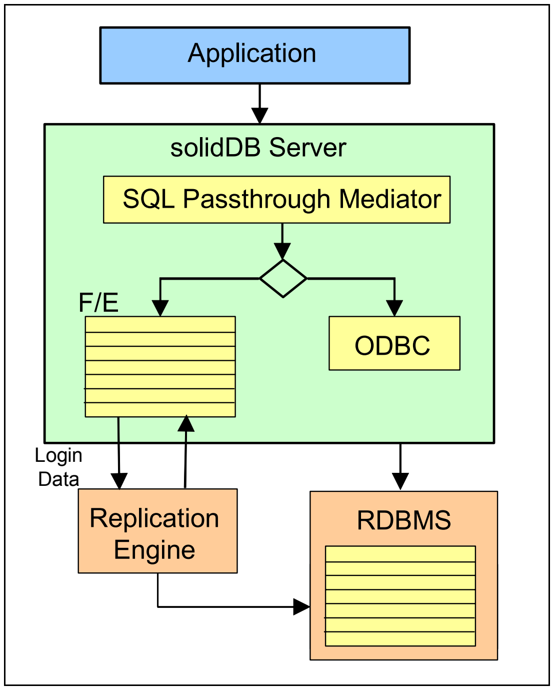


Figure 3-11 Pass-through architecture

A component of the solidDB server, called the *SQL pass-through* mediator, is responsible for handling the pass-through of SQL statements to the back-end database. It determines where a statement should be executed, based on pass-through settings.

The SQL pass-through capability can be enabled at a session or a transaction level and can be changed dynamically at run time, thus allowing the application a wide degree of flexibility and control.

The SQL pass-through feature can be configured against the following items:

- READ/WRITE

  Pass-through settings may be configured independently for read and write SQL statements.
- TRANSACTION/SESSION

  Pass-through may be enabled at the transaction or session level.
- MODE

  The pass-through mode defines whether statements are executed always in front end, always in the back end, or conditionally in the front end or back end. The conditional pass-through mode is based on, for example, the availability of data (if a statement cannot be handled locally by the cached tables, it is passed to the back end) or complexity of the query as defined by the user.

The pass-through feature of solidDB Universal Cache maintains the consistency of data in the back-end database. This task is accomplished by adopting an isolation level that is at least as strong as the level used for the back end.

To ensure consistency of write operations, if pass-through is enabled, the writes are always forced to either the front end or the back end. Also, distributed queries are not allowed; individual statements must execute fully on either the cache database or the back-end database.

3.5.2 Aging

The *data aging* feature enables solidDB Universal Cache to optimize the amount of memory allocated to cache, ensuring that the data closest to the application is also the most relevant and active. With solidDB Universal Cache, the application layer has full control over the aging of data. The purpose of data aging allows an application to remove, or age, outdated or otherwise obsolete data from the cache while still preserving it in the back end.

Data aging is useful in situations where data is considered to be owned by the back end (where the master copy of data resides in the RDBMS and not in solidDB).

Operation

The application can perform data aging through simple SQL statements specifying which data is to remove from the cached tables. The application can specify the aging to occur at a transaction level, or at a session level. Specifically, the act of enabling data aging only affects the connection upon which it is performed. The enablement of data aging does not affect other connections, nor does it affect the operation of HotStandby or normal transaction logging.

Alternatively, data aging can also run automatically with the help of stored procedures. The solidDB Universal Cache can continue bidirectional replication with a back-end database; necessary steps are taken to ensure that data removed from the cache is not replicated back into the cache from the back-end database.

When data aging is enabled on a cache database, a user can prune data from the front end. When complete, aging is disabled. See Figure 3-12.

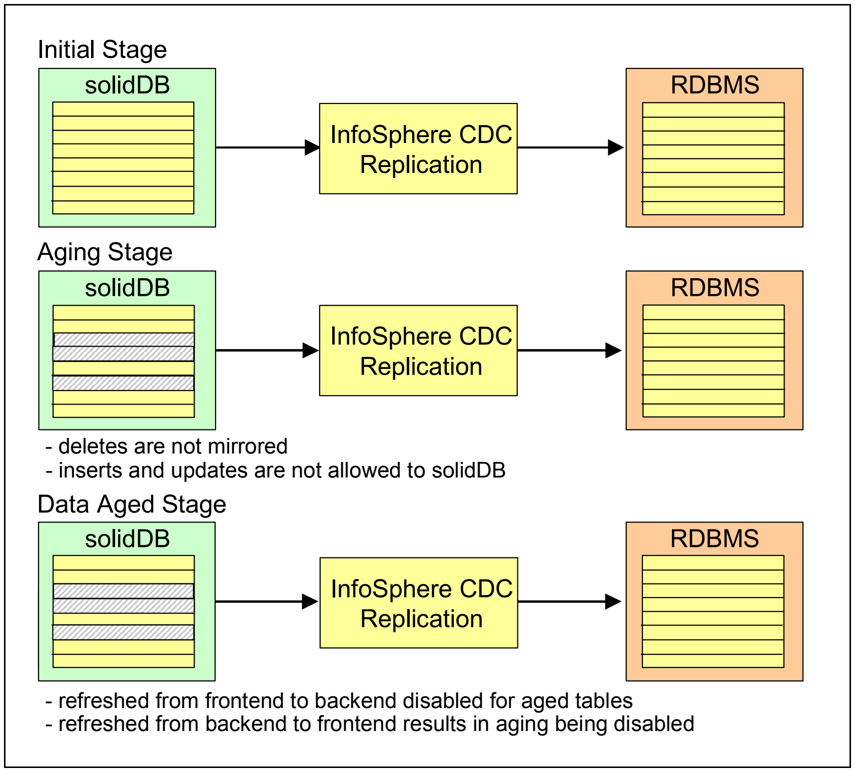

Initial Stage
solidDB
InfoSphere CDC Replication
RDBMS
Aging Stage
solidDB
InfoSphere CDC Replication
RDBMS
- deletes are not mirrored
- inserts and updates are not allowed to solidDB
Data Aged Stage
solidDB
InfoSphere CDC Replication
RDBMS
- refreshed from frontend to backend disabled for aged tables
- refreshed from backend to frontend results in aging being disabled

Figure 3-12 Aging in action

When data aging is enabled, SQL inserts and updates are not allowed against the solidDB database, and result in an SQL error. This restriction also applies to triggers and stored procedures and is imposed to preserve the integrity of data across the front-end database and the back-end database.

When data aging is enabled, SQL deletes are allowed against the solidDB database, but are not propagated to the back-end database.

When data aging is enabled, the InfoSphere CDC operations, such as refresh, are disabled or must be executed with care. The reason is to prevent a situation where data that is aged from the front-end database (cache) is also deleted from the back-end database.

3.5.3 Improving performance with parallelism

The performance gains from the solidDB Universal Cache solution can be greatly improved by spreading the cache's workload over multiple nodes. With InfoSphere CDC replication, you can closely control the flow of information emanating from the back-end database and make this information available at multiple solidDB nodes. By using multiple solidDB nodes, the workload of server applications can be effectively spread across multiple databases, resulting in increased data throughput and decreased query times.

The two main strategies for implementing parallelism are *duplication* and *partitioning*. These two strategies are not necessarily mutually exclusive and may be combined to varying extents within the context of a single solidDB Universal Cache deployment.

Duplication

With *duplication*, data from the back-end database is copied across multiple solidDB nodes on purpose. The rationale behind duplication is simple: it can reduce the contention on database (and network) resources by providing a local copy of data at multiple nodes.

Duplication works especially well to create a local cache for data that does not change often, and whereby most queries involving that data are read-only.

To illustrate where duplication may be useful, consider the case of an online global retailer. The retailer presents a basic web interface to its users where they can view details of listed items before purchasing. The number of pages served to customers world-wide can be enormous (for example, one page per item view). Consequently the retailer might decide to split page requests along territorial boundaries to keep response times acceptable. The actual data describing each item would be mostly immutable, except for occasional updates to pricing or item descriptions.

Using InfoSphere CDC replication, the retailer could transport the data to multiple solidDB nodes in separate territories, thereby allowing page requests to be processed locally in the country of origin. This approach reduces network latency, and also enables request workload to be spread across multiple nodes.

Data duplication is illustrated in Figure 3-13.

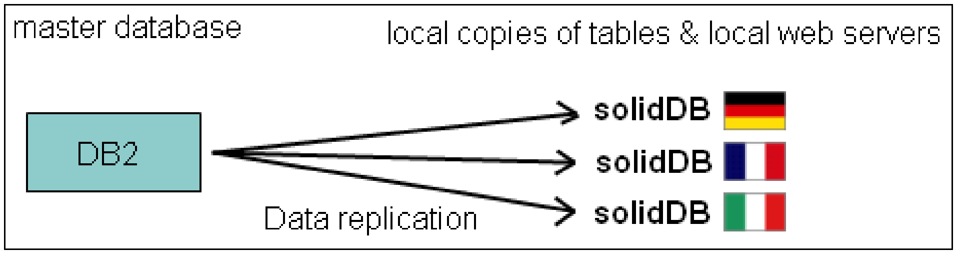


Figure 3-13 Data duplication

Because in this scenario the data is mostly immutable and will not be modified by customers, it is sufficient to employ unidirectional replication. That is, changes are replicated only to the tables in the master database; changes to the cache database are not replicated to the master database.

If local changes at each cache are required, bidirectional (two-way) replication would need to be employed, so that changes made at each cache can propagate back to the master database. Additionally, conflict resolution rules on InfoSphere CDC subscriptions would also be required to handle any conflicting updates that can be performed independently at each cache. However, a better alternative for handling potential conflicts in mutable data is to use a clearly defined partitioning scheme so that conflicting modifications can be avoided entirely in the first place.

Partitioning

With *partitioning*, you do not need to keep all of the data at every single node. Instead, you can distribute it across several nodes, perhaps evenly. This approach enables the burden of queries to be shared across many nodes, and also avoids unnecessary data duplication because only a unique subset of the data is stored at each node. Most important, however, a partitioning scheme allows for safe, non-conflicting modification of data at each node; the uniqueness of the data at each node ensures that contradictory changes to the same data (at separate nodes) does not occur.

To illustrate where partitioning might be useful, consider the case of a bank that stores account details for its customers. The bank stores details of each customer account in a table, Accounts, that uses the AccountID as the primary key.

Because the clientele of the bank has grown substantially (along with the number of transactions), the bank decides to split account data across four separate cache nodes, to ensure transactional workload is kept manageable. The partitioning rules are based on the remainder of the integer primary key AccountID when it is divided by 4. For example, if AccoundID divided by 4 is 0, the record is stored in Cache Node 0; if AccoundID divided by 4 is 1, the record is stored in Cache Node 1; and so on.

Figure 3-14 shows an example of a solidDB Universal Cache deployment that has four cache nodes with eight subscriptions (two per cache, one in each way). We also assume that the bank wants to be able to update the account details also in the back-end database, for example, for administrative purposes.

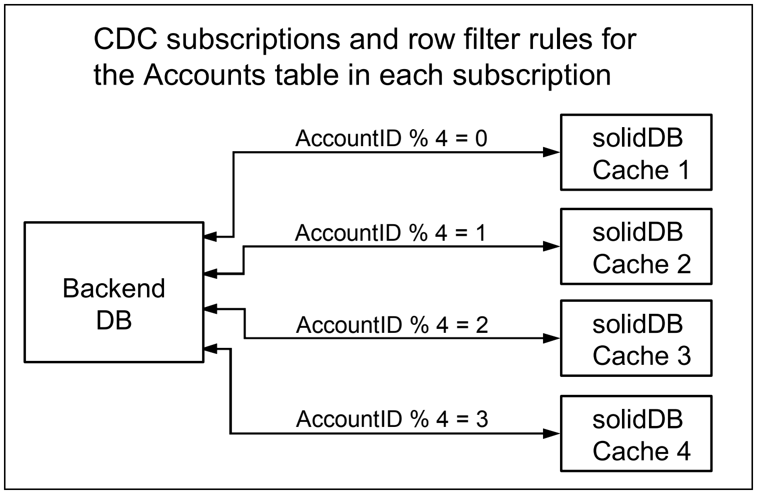


Figure 3-14 Example of partitioning with solidDB Universal Cache

The arrows show the direction of replication for each subscription. There are row filters on the subscriptions replicating from the back end to the cache database; this way ensures that only the desired subset of data reaches each cache. If new records are inserted at a cache node (rather than at the back-end database), the inserting application needs to respect the partitioning rules and choose the correct cache node to place the data within.

In this scenario, we always assume that the subset of data at each cache node will match the expected subset, as defined by the partitioning rules. Because of this assumption, subscriptions replicating from one of the caches to the database do not require any row filter rules, because the data already conforms to the partitioning rules.

This type of partitioning architecture enables concurrent modifications to records in separate caches. In the bank example, the bank can process monetary transactions on customer accounts concurrently, in each of the individual cache nodes. Because of the partition rules, transactions do not conflict with simultaneous transactions in separate cache nodes. A partitioned Universal Cache architecture can therefore be desirable in cases where data is potentially mutable or where duplication is too costly in terms of storage but where concurrency is still required.

3.6 Increasing scale of applications

This section describes large-scale strategic mechanisms to scale applications, their characteristics, and a comparison of them to the usage of a cache as a scaling mechanism. It also describes generalized application classes that benefit from the use of a cache.

3.6.1 Scaling strategies

Applications can be scaled to handle greater and greater loads using the following common strategies:

- Adding servers
- Redesigning applications
- Adding database cache

Adding servers to increase capacity

Adding new hardware and virtual machines can be a simple way to increase capacity of caching user web sessions, execute business logic, process transactions, and so on. However, each additional hardware component carries capital and operational costs. Also, if the number of application instances or application servers is increased, the database will in most cases eventually become the bottleneck. Databases can be redesigned to scale up or out, but it is often costly to do so in terms of re implementation of the application.

Cloud computing is also an option; although cloud computing can deliver server instances on demand, it cannot substitute for redesign of the application code and data to take advantage of additional resources.

Redesigning applications to increase capacity

Applications can be redesigned to allow for greater capacity, for example, by adding stateless paradigms, improving the layering and separation of responsibilities, introducing or adding asynchronicity, or introducing application or object caches. Such strategies are typically lengthy, expensive, and risky.

Adding cache databases to increase capacity

Another way to increase capacity is to add a relational cache database such as solidDB Universal Cache to the existing application/system architecture. In relative terms, a cache database can be an inexpensive addition to the application, requiring few if any application changes. The cache database can alleviate database bottlenecks and provide easier scaling especially in read-only scenarios. Moreover, availability of the solution is also not of serious concern because most caches including solidDB Universal Cache have mature HA and HSB mechanisms. Relational cache databases also fit easily into existing architectures and systems and allow scaling at a lower cost than adding hardware or redesigning the applications.

In 3.6.2, “Examples of cache database applications” on page 64, we discuss application architectures that typically benefit from cache deployment.

3.6.2 Examples of cache database applications

This section has examples of application domains that are good candidates for acceleration by cache databases.

Web session management

Web session management refers to the process of keeping track of user states while interacting with an application over a number of sessions. It is typified in a web application whose context is dependant on knowing the user’s current state and previous session information. The application sessions typically need to manage a large volume of sessions where each session is small in nature but sensitive to the speed of retrieval and update of the state information. A cache database can have a big impact by eliminating network and database operation latency.

Reference and common lookups

Reference lookups are ideal for cache deployment. Whether they are terrorist list lookups or product pricing, the nature of lookups is typically of the type read-only or seldom-updated.

Time-sensitive transaction processing

Applications that are sensitive to latency can gain large benefits from solidDB Universal Cache through the elimination of the network and the faster servicing of database requests by solidDB as opposed to disk based databases.

3.7 Enterprise infrastructure effects of the solidDB Universal Cache

This section describes how the introduction of the cache into an existing environment can lead to efficiencies in various infrastructure components within the existing IT assets of an enterprise. The infrastructure pain points described in this section can also be used as a guide to differentiate between new product development using cache architecture and more traditional database patterns.

The general pattern for the discussion is the comparison between an existing application where the database tier resides on a distinct hardware component accessed through the enterprise network infrastructure and a solidDB Universal Cache setup where a subset of the data required by the application is cached and collocated with the application.

3.7.1 Network latency and traffic

The existence of a cache can lead to the reduction of the network traffic between the application and the remote back-end database machine. This way has the dual advantage of both reducing the load on the network (and thus the overall load on the enterprise network assets) and reducing the overall latency of the database operations through the elimination of the network hop required by a more traditional architecture.

Two facets must be considered when you decide what type of data should reside in the cache:

- Consider the volume of data to be transported over the network and if the network has the capacity to efficiently transport this data. Most commonly, if the application can make use of an operational (or hot) data set that is a subset of the overall data, this case leads to a reduction on the throughput load on the network.
- If the application is sensitive to database operation latency, it is appropriate to cache the data that corresponds to the sensitive operations. In such a case, the amount of individual operations going across the network is reduced which leads to the reduction in the dependence of the application on the consistent response of the network.

3.7.2 Back-end machine load

The cost of hardware used to host enterprise database systems can be high in both capital and operation terms. If you remove the load from the back-end machine through the use of solidDB Universal Cache by handling database queries in the application tier, you are both delaying the need for an enterprise to upgrade or replace existing hardware infrastructure, and also reducing the operation expense of such systems, thus reducing the overall cost to an enterprise of an application.

By taking away CPU cycles from the back-end hardware, you can free hardware resources for use by other applications, whether resident on the machine or simply using the shared back-end database. Again, upgrades can be temporally delayed to some time in the future. The trade-off is the possible reallocation of hardware resources or funding to the front end or application tier; however, such hardware is typically commodity-based and less expensive in nature. Primarily, the hardware enhancements on the front end require increase of main memory, which is becoming progressively less expensive over time, particularly on commodity hardware.

3.7.3 Database operation execution

In this section, we consider two facets of the introduction of a cache database:

- The reduction of the latency of database operations
- The increased availability of enterprise database resources to other applications within the enterprise.

The response time of a database operation is defined as the *round-trip* time required to return a result to an application. The introduction of solidDB Universal Cache to the application tier of the enterprise can have two advantages:

- The database response time itself can be improved
- The transport layer between the application tier hardware and the back-end database hardware can be the eliminated.

The result of these advantages has the effect of accelerating the application through the raw speedup in response times.

Through the elimination of a percentage of database operations and interactions from the back-end database, the resources available to the back-end database can be substantially increased, which in turn leads to improvements in the availability and response time of the database to other applications using the database resource.

However, consider that existing disk-based enterprise databases are better suited to certain database operations than solidDB would be, such as queries with large result sets. This consideration is important when trying to quantify the benefit that solidDB Universal Cache can have on the increased availability of the back-end database to the enterprise.

4

Deploying solidDB and Universal Cache

In this chapter, we discuss application development for when you use IBM solidDB products. We emphasize the effect that using a cache database has on application design as compared to other database applications. IBM solidDB is a standards-compliant relational database supporting SQL and the standard ODBC and JDBC programming interfaces. Most application design, programming, data model design and system administration paradigms used with other database systems are directly applicable with IBM solidDB products also. A multitude of literature is available about all these topics. Therefore, they are only briefly described in this chapter, highlighting solidDB-specific details when necessary. We assume that you known SQL and its basic concepts.

4.1 Change and consideration

Change from a single database system to a system of multiple databases is always a major step in architecture design, significantly increasing the number of considerations for deployment. A far more straightforward approach is to consider the issues when designing a new application that will use a cached database and back-end database as compared to extending an existing RDBMS system to use a front-end database cache. We look at both of these scenarios for deployment of a cached database system.

When moving a system to production, several things related to system initialization and administration must be considered. As examples, in the system initialization section, we focus on installation and initial data load. Then, in the administration section, we focus on monitoring the system health, illustrating how the system can be recovered from various disasters, and prepare for several types of upgrade operations, such as hardware upgrades, IBM solidDB software upgrades, and application software upgrades.

4.2 How to develop applications that use solidDB

The development of an application that will use solidDB as a stand-alone database conceptually and architecturally resembles application development on any other type of relational client/server database. Almost all concepts and methods work should be applied similarly, with some relatively minor things to consider when using direct linking models or the high availability architecture of solidDB. We briefly review the basics of database application development in this chapter. For more detailed information, see any available literature and examples, most of which is applicable for solidDB.

4.2.1 Application program structure

Basically, application development on relational databases is about creating and executing programming commands along with SQL statements inside the host language, according to host language-specific programming paradigms. Generally, this task is done by linking a database driver (which is a component provided as part of IBM solidDB product package) to an application program. This driver contains a set of functions (based in C-language ODBC drivers) or a set of classes and methods (in a Java-based JDBC driver) to be called by the application. ODBC and JDBC standards specify the names and parameters of these functions in detail to enable changes to the database by simply changing the ODBC or JDBC Driver. This task can be done either in a linker's file list or a call to Driver Manager, which is a system component that picks up an appropriate driver, based on the database connection string or URL.

All relational database application programs are structured as shown in Figure 4-1. The connection must first be initialized. Then, the application program executes one or several statements within the connection, and ultimately either explicitly or implicitly (application program termination) terminates the connection.

In the interfaces provided by IBM solidDB, there is always an implicit current transaction related to each connection. Hence, the Begin Transaction statement is always implicit. The transaction statements are executed according to the structure shown in Figure 4-1. For statements that are expected to be executed multiple times, using prepared statements is the best approach to avoid having the SQL Interpreter be activated for each execution. This way can improve performance. Naturally, the statements that are expected to return data will have to process the returned data in one way or another; statements only writing data do not process results.

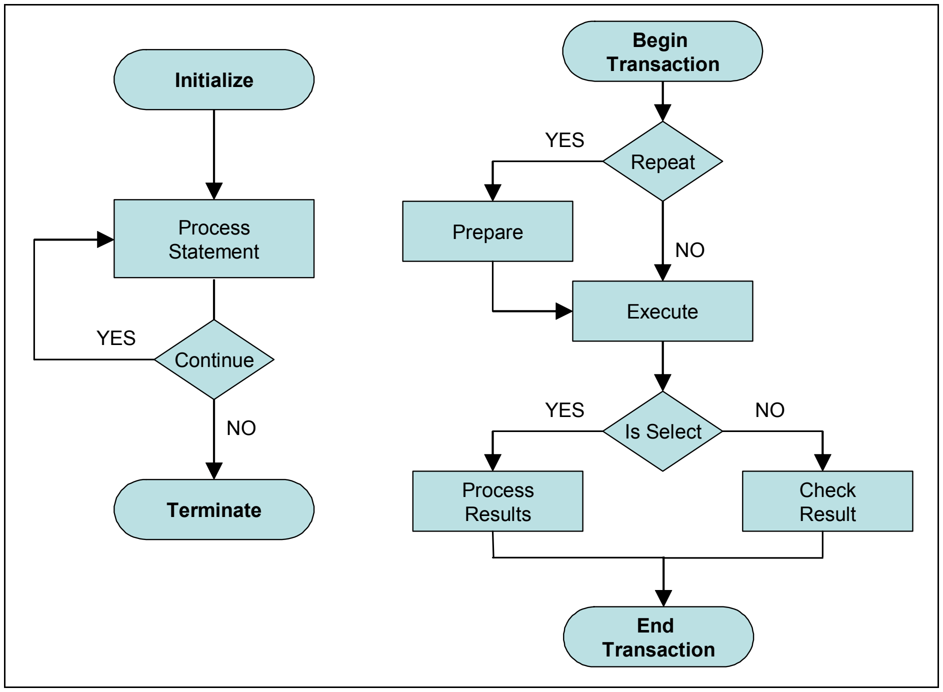


Figure 4-1 Application structure

4.2.2 ODBC

This section introduces and helps you get started with ODBC on solidDB.

Introduction

ODBC provides a standard for accessing relational databases. ODBC is defined as a set of strictly defined C-language functions implemented by database manufacturers. to develop a native ODBC-application. The programmer must include the header file that contains ODBC definitions and link an appropriate driver that is provided by the database manufacturer. The application can use a separate database simply by linking to a separate driver.

A key benefit of ODBC is the capability of accessing several databases at the same time. To avoid naming conflicts between similar functions in separate drivers, the application links to the ODBC Driver Manager instead of individual drivers. The driver manager then routes the call to the appropriate driver. For applications that require access to only one type of database, the use of Driver Manager is not required. From a linking perspective, the ODBC Driver Manager is only another library to link. It provides exactly the same function interface as the ODBC drivers.

ODBC is widely used, especially in Windows environments, as the database access layer for applications that are not written in the C-language. In those cases, the application middleware translates the application code that is not based in C language to appropriate C-based calls to ODBC (or more commonly, to the ODBC Driver Manager).

The left side of Figure 4-2 shows the basic ODBC functions that are required to prepare to connect to the database and establish the actual connection. The right side of the figure shows the actual functions related to statement preparation, execution and transaction handling.

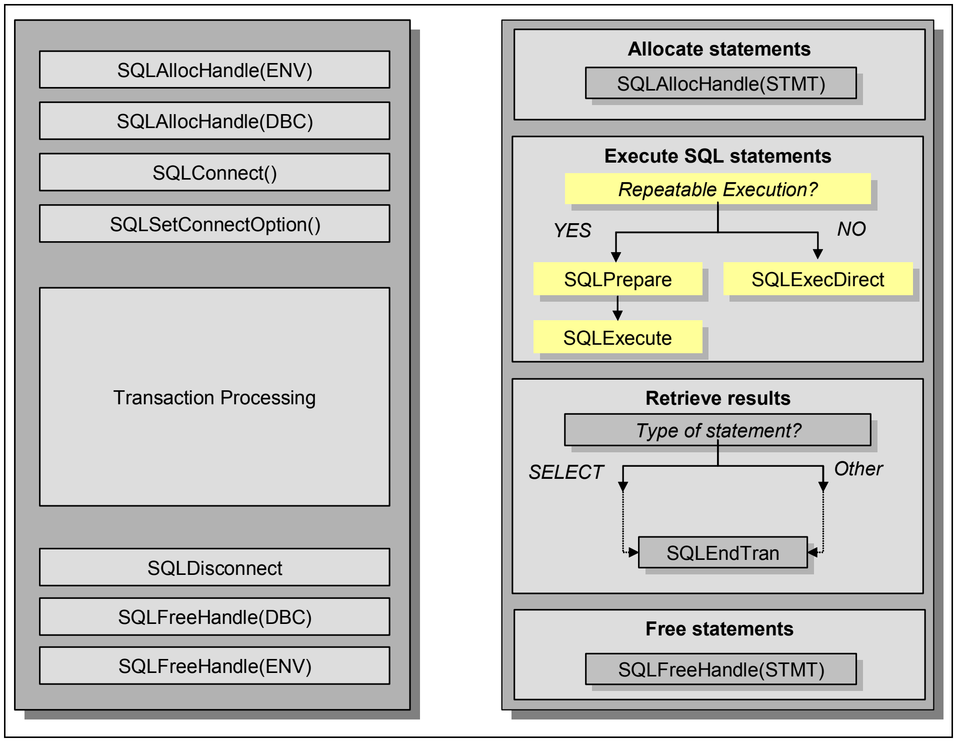


Figure 4-2 ODBC functions for database connection

For insert, delete, and update operations, there is no need to process results in any other way than to validate that SQLExecute does not return errors.

Figure 4-3 lists the ODBC functions related to processing the result set in the case of running a Select statement.

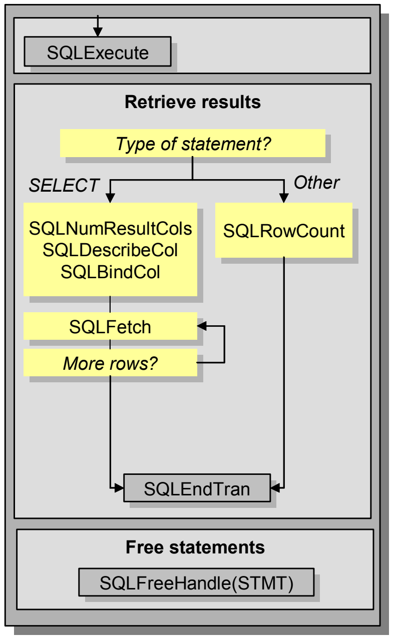


Figure 4-3 ODBC functions for processing a result set

Example 4-1 on page 73 shows an ODBC-application that performs all the steps to allocate resources, connect the database, perform a query, and process the retrieved results. Note that checking return code validity has been removed for simplicity.

Example 4-1 A simple, but complete ODBC application

```
// section 1: Allocating Handles
rc = SQLAllocEnv(&henv);
rc = SQLAllocConnect(henv, &hdbc);

// section 2: Establish connection
rc = SQLConnect(hdbc, “tcp 1313”, SQL_NTS, “dba”, SQL_NTS,  “dba”,
SQL_NTS);

// section 3: Prepare Statement for execution
rc = SQLAllocStmt(hdbc, &hstmt);
rc = SQLExecDirect(hstmt, (SQLCHAR *)"SELECT ID,NAME FROM NAMEID",
                           SQL_NTS);
// section 4: Define Variables in C
rc = SQLBindCol(hstmt, 1, SQL_C_SLONG, &id, 0, NULL);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, name,
                        (SDWORD)sizeof(name), &namelen);

// section 5: Run a loop until rows run out. After each SQLFetch
// call the contents of variables id and name change to match
// row contents
rc = SQLFetch(hstmt);
 (rc == SQL_SUCCESS)
{
   printf("A row found (%d, %s).\n", id, name);
   rc = SQLFetch(hstmt);
}

rc = SQLFreeStmt(hstmt, SQL_DROP);

// section 6: Release the statement handle, disconnect and release
// the environment handles
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
```

Getting started with ODBC on solidDB

The solidDB product package contains a set of ODBC sample programs and an operating system (OS)-specific build environment for the particular sample. The samples can be located in the product package in the following directory:

```
./samples/odbc
```

The directory contains two simple C-language programs and the **`makefile`** that is required to build and run the programs. A C-development environment (compiler and make utility) are assumed to exist.

Tip: Details of the ODBC Interface

ODBC Function Interface:

`http://msdn.microsoft.com/en-us/library/ms714562(VS.85).aspx`

ODBC Header File name: sql.h (located in the `./include` product package directory)

The solidDB ODBC Driver names are as follows:

- Depends on platform: socw3265.dll on 32-bit Windows for dynamic linking
- Direct linking stub: solidimpodbcu.lib on 32-bit Windows

4.2.3 JDBC

This section introduces and helps you get started with JDBC on solidDB.

Introduction

JDBC is direct counterpart to ODBC in the Java language. Because Java is an object-oriented language, the standard is specified as a set of classes and their methods, instead of function calls. The underlying principles are exactly the same as with ODBC. That is, every ODBC operation has a direct counterpart in JDBC. In Java, instead of including header files as in C, the JDBC interface is imported to Java application code by import clause. Instead of linking the driver, in JDBC, the driver's classes are loaded when a connection is requested from the JDBC Driver Manager. As an application interface, JDBC is slightly simpler to use than ODBC in the sense of generally having fewer parameters in function calls and being (because it is Java-based) less vulnerable to typical C-language development-time problems, such as loose pointers and uninitialized memory.

The JDBC Driver manager is the JDBC counterpart to the ODBC Driver Manager. It is included in Java runtime environment, so circumventing the Driver Manager in JDBC is not practical, as it is in ODBC.

The left side of Figure 4-4 shows the JDBC methods that register the driver with JDBC Driver Manager and establish a connection using a registered driver. The right side of Figure 4-4 shows the methods to execute the statement either by using a Statement object for one-time execution or a PreparedStatement object for multiple executions.

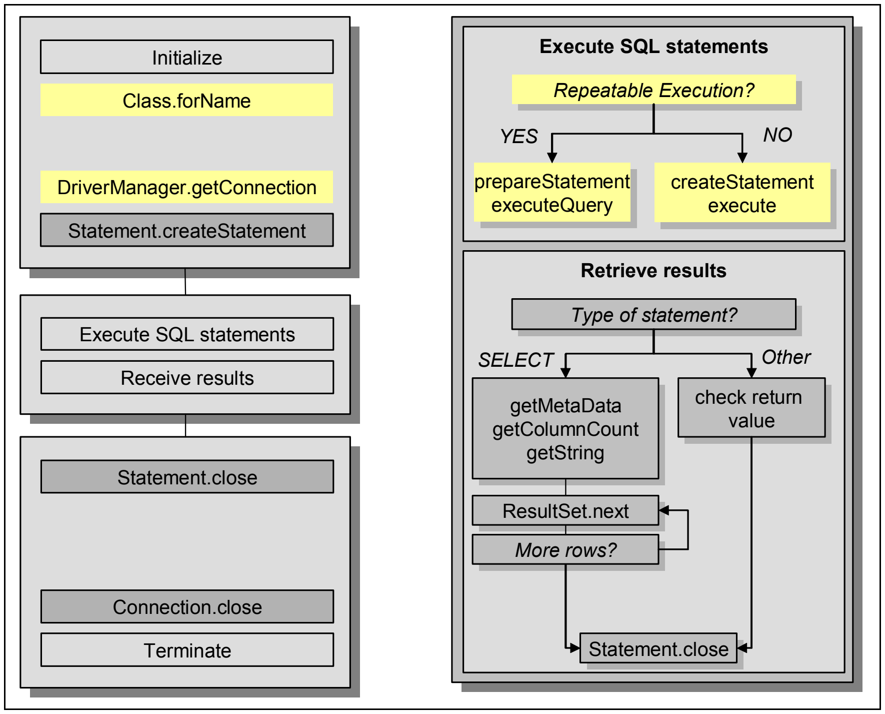


Figure 4-4   JDBC process to register the driver with the Driver Manager

Similar to ODBC, multiple ways exist of developing applications with a slightly higher abstraction level than writing direct JDBC calls. Java Application Servers such as WebSphere®, WebLogic, and JBoss, and Object-Relational mappers such as Hibernate, are based on classes or templates for application classes that call the appropriate JDBC methods without the application developer having to see that part of code at all.

Example 4-2 shows a JDBC version of the application in Example 4-1 on page 73. It contains counterparts for resource allocation, connection establishment, query execution and result set processing.

Example 4-2 A simple JDBC-application

```
import java.sql.*;

public class jdbcsample
{
  public static void main (String args[]) throws Exception
  {
    // section 1: Register driver for JDBC Driver Manager
    Driver d = (java.sql.Driver)Class.forName
               ("solid.jdbc.SolidDriver").newInstance();

    // section 2: Establish a JDBC connection from JDBC Driver Manager
    Connection conn = java.sql.DriverManager.getConnection
                      ("jdbc:solid://localhost:2315/dba/dba");

    // section 3: Define Statement for Execution and Execute
    Statement sta = conn.createStatement();
    ResultSet rs = sta.executeQuery("SELECT ID,NAME FROM NAMEID");

    // section 5: Run a loop and retrive rows until they run out.
     (rs.next())
    {
      System.out.println("A row found (" + rs.getString(1) + ", " +
                         rs.getString(2) + ")");
    }

    // section 6: Close appropriate resources
    rs.close();
    sta.close();
    conn.close();

  }
}
```

Getting started with JDBC on solidDB

As with ODBC, the solidDB product package for JDBC contains a set of JDBC sample-programs and the environment used build and run them. The samples are located in the product package in the `.samples/jdbc` directory. The directory contains four simple Java programs illustrating the basic JDBC operations. A Java developer kit is expected to be installed to run these programs.

Tip: Details on JDBC Interface

JDBC interface classes and methods are described in the following location (by looking at the `java.sql` package):

`http://download.oracle.com/javase/6/docs/api/`

JDBC Classes can be imported by the following statement:

```
import java.sql.*;
```

The solidDB JDBC Driver name is: `SolidDriver2.0.jar`

4.2.4 Stored procedures

IBM solidDB provides a third way of executing SQL statements and processing the results, and that is in the form of stored procedures.

The standard SQL does not contain definitions for a stored procedure language. Therefore, stored procedure languages used in the various databases are generally not compatible with each other, although conceptually they might strongly resemble each other.

Architecturally, all stored procedure execution takes place inside the server process. Therefore, no network communication is needed within procedure code. Depending on a linking model, a network message might be needed when the application process calls the procedure and receives results.

Example 4-3 shows how to create a stored procedure.

Example 4-3 Creating a stored procedure

```
"create procedure proc returns (str varchar)
begin

  -- section 1: Declare variables to store the retrieved data
  declare id integer;
  declare name varchar;

  -- section 2: Declare trivial error handler
  exec sql whenever sqlerror rollback, abort;

  -- section 3: Prepare and execute the piece of sql
  exec sql prepare sel select id, name from nameid;
  exec sql execute sel into (id, name);
  exec sql fetch sel;

  -- section 4: Run a loop and retrieve all the rows. Instead of
  -- printing, return the rows to the caller
   sqlsuccess loop

    str := convert_varchar(id) + ' ' + name;

    return row;

    exec sql fetch sel;
  end loop;


  -- section 5: close and drop the cursor
  exec sql close sel;
  exec sql drop sel;


end";
commit work;
```

To get started with solidDB stored procedures, the IBM solidDB product package contains a set of samples of solidDB stored procedures, which are in the following directories:

- `./samples/sql`
- `./samples/sql/course_exercises`

Tip: Details about the Stored Procedure Interface

No special component is needed to enable or run solidDB stored procedures, because the procedure engine is included in the server process.

The solidDB procedure language syntax is proprietary to solidDB. That syntax is described in *IBM solidDB Manual: SQL Guide*, SC23-9871, Chapter 3, “Stored procedures, events, triggers and sequences.”

4.2.5 Special considerations

Several special situations must be considered in application code when developing a client application for IBM solidDB. We provide a brief description of those situations in this section.

Linking models

The database part of application development is about creating a string of SQL statements within the host language, calling the appropriate function or method to send the string to database server, and the use of appropriate functions or methods to process the results retrieved by the database.

In a traditional client/server linking model, the application and database servers have been running in separate processes either in the same machine or in separate machines. In this case, the database driver has been hiding the fact and arranging the communication between separate processes by network messages or interprocess communication inside same machine. This fact is true with IBM solidDB and regular client/server connections also. Additionally however, a mechanism to combine the application and database server processes and to bypass all messaging is provided. In some cases, this way is beneficial for response times and performance, and is depicted in Figure 4-5 on page 80.

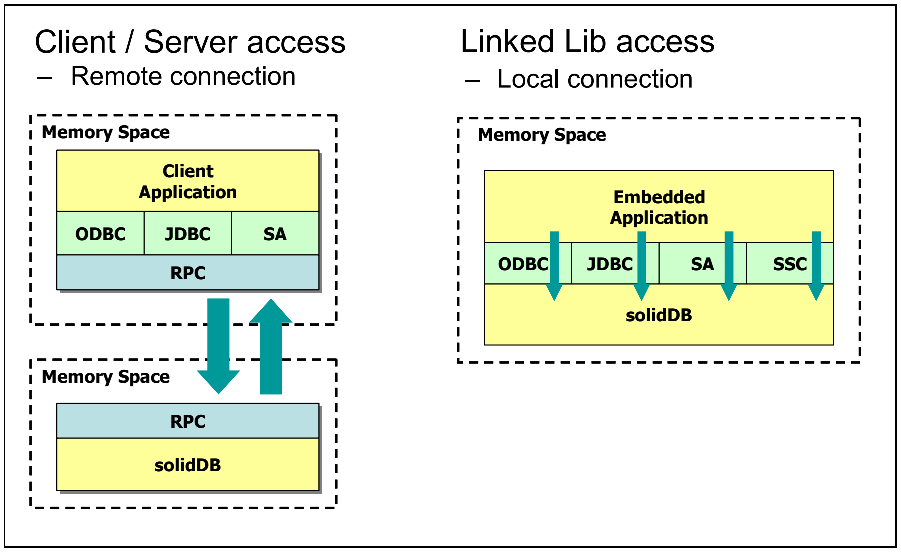


Figure 4-5 Linking models

Figure 4-5 illustrates the difference between client/server and linked lib access methods. No other differences exist, from a programming perspective, than the need to avoid application crashes, because the server is vulnerable to application side crashes also.

Note the following information about models:

- Traditional client/server

  With this default linking model, the application process and database server are running as separate processes. They can run either in the same machine or separate machines, with their process images and memory spaces being separated either by machine boundary, operating system, or both. There are no special considerations for application development when running this default linking model.

- Linked Lib access

  With this model, instead of linking to a database driver and sending network or interprocess messages to a server process, the application links to a database driver containing the full server functionality. All communication between application and server take place in same process. The application and database server share the memory space. All loose pointers in application code can refer to the database server's memory, overwrite it and create a crash in server code. We strongly suggest using the Client/Server linking model in development and functionality testing, and only moving to Linked Lib drivers for performance testing and deployment. For deploying immature applications (that are known, or expected, to crash), deployment with Linked Lib drivers is not suggested. In some environments, deploying the final application and database servers in same process is seen beneficial because, from systems management perspective, the number of processes to manage will be smaller.

- Shared memory access (SMA)

  This model provides the benefits of both the Client/Server model (memory protection, enabling several application process instances sharing the same performance) and the Linked Lib model (added performance). The server process is also protected against application process crashes.

Solid HotStandby

IBM solidDB HotStandby feature is basically transparent to application code. A regular application works without modifications with solidDB high availability feature. Both ODBC and JDBC Drivers contain a transparent failover functionality that hides the two underlaying connections to primary and secondary databases to appear as one JDBC or ODBC connection. Making the application aware of two separate connections is possible, but it increases application complexity substantially.

A failover or role change in HotStandby does not cause loss of committed data in the database. However, when failover, or role change occurs, the current transaction is terminated. That is, the ODBC call fails with appropriate error code or JDBC call throws an appropriate SQLException. To handle the situation properly, the application needs to call the rollback function or method to start a new transaction. Transparent failover automatically directs the connection to the new primary database. The application is responsible for capturing the ODBC and JDBC error caused by roleswitch or failover, rolling back the current transaction and rerunning the transaction. In practice, this technique means that error codes caused by failover/rollback need to be handled differently in applications as compared to fatal errors (syntax error, table missing, user rights missing) or other errors having different recovery policies.

Running applications on multiple separate databases

SQL, ODBC, and JDBC are mature standards supported by multiple database vendors. Conceptually, an application that has been implemented based on these standards should be portable and run on any standard compliant database with minimum effort.

In practice, it seldom works this way, for the following reasons:

- Databases typically have different extensions to the standards. However, if applications carefully avoid using all extensions, this issue is less of a problem.
- Standards are somewhat loose on minor details such as exact column length (for example, decimals allowed for timestamps), and sorting and unique behavior with null data, allowing cursors to outlive transactions.
- Certain crucial elements in database architecture have not been included in the standardization effort. As examples, these elements include stored procedures, triggers, and sequencers.
- The standards specify an error code, *SQLState*, that is supposed to be returned in all error cases. For proper error handling, however, the applications generally need to access the *native error code*. These native error codes are not standardized at all and vary from database to database.
- Occasionally, a substantial part of application is implemented in database scripts that are executed by particular database utilities. However, the scripting languages and utilities are not standardized.

These problems can be most easily addressed separately in the design phase of software, for example, by the following methods:

- Implementing wrapper layers to hide the difference between databases, thereby making the application database agnostic. This means mapping the native database error codes to error codes meaningful for the particular application. As examples, fatal error, recoverable by retry, and recoverable by reconnect.
- Avoiding the use of non-standardized elements of databases, such as SQL extensions for nonstandard data types.
- Preparing to write several versions of code for elements that differ but cannot be avoided. For example, having a version of a stored procedure for each supported database.
- Using interfaces that are available on all considered databases.

Having the capability to run the same application code on separate database brands is essential when deploying solidDB Universal Cache (UC). The same code must process data, error codes, and transactions from both the solidDB front-end database and back-end database, whether it is DB2, IDS, Oracle, Sybase or other supported back-end database brands. The application must be built to run seamlessly on multiple database brands.

Most high level programming techniques, such as those using application servers or object-relational mappers, that hide the database-level coding usually make the applications more portable.

Other programming environments

IBM solidDB's supported interfaces of ODBC and JDBC enable application developers to write applications with C or Java languages calling ODBC functions or JDBC methods and writing or generating the SQL string at the application level. This common way of developing application logic is not the only one.

Multiple ways exist to raise the abstraction level from the ODBC/JDBC level. It can be done either by enabling database access from various (usually higher level programming or scripting languages, such as Visual Basic, Perl, and PHP) or enabling database access directly through application level objects that are able to load or save themselves without the application programmer having to be aware of database connections, transactions, or even SQL Strings.

Database access from higher level programming is usually based on some middleware component translating the higher level language calls to regular ODBC or JDBC calls. In these conditions, the middleware component is seen as an application from the database perspective. Usually the middleware components do a good job in hiding the difference between database brands.

IBM solidDB is a relatively new product, and as such not all middleware vendors explicitly list it among the supported database products. In those cases, there is usually an option to have a generic ODBC database or generic JDBC database that works with IBM solidDB drivers.

Certain programming environments do not have a direct counterpart in IBM solidDB applications, such as Embedded SQL or Java-based stored procedures. Applications designed to run on these programming environments must be redesigned to fit IBM solidDB.

4.3 New application development on solidDB UC

Database application architecture built on cache database or back-end database instead of a single database becomes more complicated. In a high-level conceptual diagram, the legacy back-end database is simply replaced with a cache database that sits between the back-end database and application making the database appear faster from an application perspective. There are no changes in the database interface layer. This concept is illustrated in Figure 4-6.

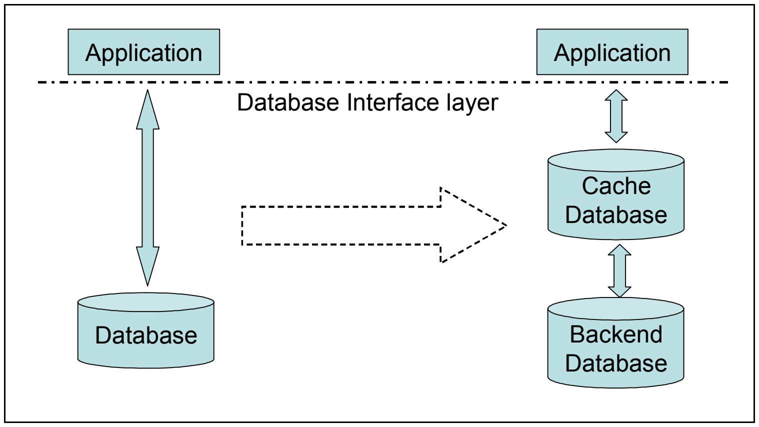


Figure 4-6 Database interface layer

In reality, the conversion from single database system to a cache database system is not quite so straightforward. Consider the following issues, among others, in the application codes:

- The application must be aware of the properties of two database connections, one to the cache database and the another to the back-end database. SQL pass-through can mask the two connections to one ODBC or JDBC connection but will require cache awareness in error processing.
- Transactions combining data from the cache database and a back-end database are not supported.
- Queries combining data from front-end and back-end database are not supported.
- A combination of back-end and front-end database is not fully transactional although both individual components are transactional databases.
- Support is limited for stored procedures.

Knowing these limitations or conditions of a cached database system enables taking them into account and avoiding them in the design phase.

Based on the conditions, the architectural diagram becomes more complicated, as we have illustrated in Figure 4-7

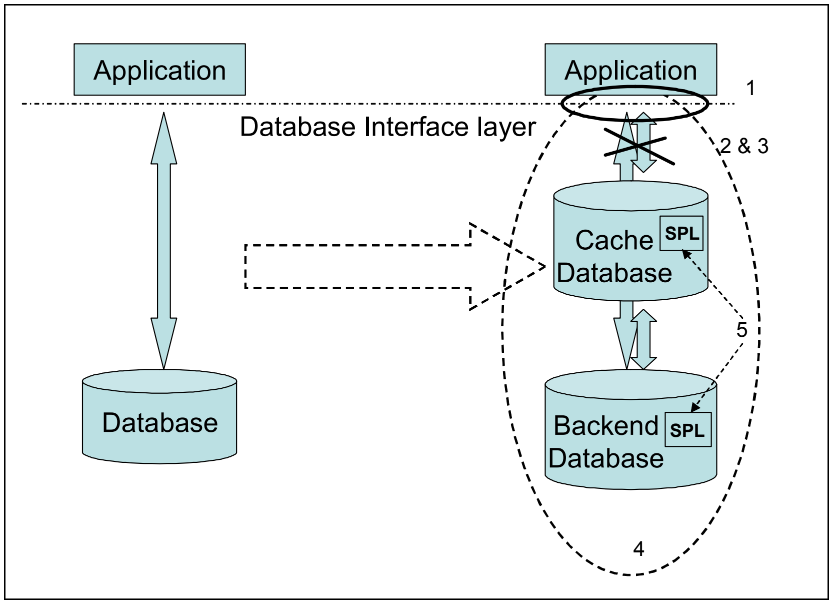


Figure 4-7 Database interface layer with a cached database

Certain changes are required in the interface between application and cached database as compared to an application with similar logic accessing only a single database.

4.3.1 Awareness of separate database connections

A regular single-database application sees only one database and can handle everything with one database connection. All transactions that have been successfully committed to the single database automatically have the ACID (atomicity, consistency, isolation, durability) properties.

A cached database system has two physical databases, a front end and a back end. Certain performance-critical data has been moved to front-end database; volume data remains in the back-end database. These databases are synchronized by Universal Cache's Changed Data Capture (CDC) replication but they still act as individual databases.

The application can access these two databases by two strategies:

- Opening and controlling separate connections to the two databases
- Using SQL pass-through to route all queries to the back-end database using the front end

Opening separate connections

The application can open two database connections to the two databases and retain and monitor these connections constantly. This strategy provides the application full control on which queries to route to the front end and which to the back end. This is rather laborious but provides flexibility for distribution strategies.

SQL pass-through functionality

SQL pass-through functionality provided by the solidDB Universal Cache product can be used. SQL pass-through assumes that all statements are first run at the front-end database. If any error takes place, the statement is run at the back end. Errors are assumed to be caused by tables not being in place at the front end. The application sees only one connection but the front-end and back-end databases are still separate and individual databases. The key challenges with SQL pass-through are as follows:

- The set of two databases is not transactional. For example, writing something that is routed to the front end is not synchronously written to the back end. If a transaction writes something to a front-end table and in the next statement executes a join that combines data from the same table and another table that only resides in back end, the statement will be routed to back end. The recently written data will not be visible until the asynchronous replication is completed.
- Cross-database queries are not supported, so joining data from a front-end table and back-end table is not possible. These queries are always automatically fully executed at the back-end database.
- For large result sets, SQL pass-through can present a performance bottleneck. All rows must be first transferred from the back-end database to the front-end database, and then from front end to the application. The front-end database ends up processing all the rows and potentially performing type conversions for all columns. The impact of this challenge is directly proportional to size of result set. For smallish result sets it is not measurable.
- SQL pass-through is built to route queries between the front-end and back-end databases on assumption that the routing can be done based on table name. SQL pass-through does not provide a mechanism for situations where a fraction of a table is stored on the front end and the whole table at back end.

4.3.2 Combining data from separate databases in a transaction

Although both front-end and back-end databases are individually transactional databases, the two transactions taking place in two different databases do not constitute a transaction that would meet the ACID requirements.

Using the default asynchronous replication mechanism does not enable building a transactional combined database, because some compromises are always implicitly included in this architecture.

Creating a transactional combination of two or more databases, using Distributed Transactions, is possible. A Distributed Transaction is a set of database operations where two or more database servers are involved. The database servers provide transactional resources. Additionally, a Transaction Manager is required to create and manage the global transaction that runs on all databases.

4.3.3 Combining data from different databases in a query

Joining data from two or more tables by one query is one of the benefits of relational database and SQL. This is easily possible in the Universal Cache architecture as long as all tables participating the join reside in the same (either front-end or back-end) database. If this is not the case, several ways are available to work around the limitation:

- Generally, the easiest way is to run all the joins of this kind in the back end. Typically, all tables would be stored at the back end, but the most recent changes to the tables that reside at the front end also might not have been replicated to the back end yet. If there is no timeliness requirement and if there is no performance benefit visible based on running the query at the front end, this approach is a good one.
- Because there is no statement-level joins available between two separate databases, the only way to execute the join between two databases is to define a stored procedure that runs in the front end and executes an application-level join by running queries in the front-end and back-end databases as needed. All join logic will be controlled by the procedure. From the application perspective, the procedure is still called by executing a single SQL statement.
- Application-level joins can also be executed outside the database by the application, but they cannot be made to appear as the execution of single statement in any way.

4.3.4 Transactionality with Universal Cache

Transactionality is a typical requirement for a database. In Universal Cache architecture, both individual databases, the front end and the back end, are transactional as individual databases. The combination of the two databases, however, does not behave as a transactional database. Consider, the following example with two tables. The table FRONT is stored in front end and replicated asynchronously to table BACK on the back end. BACK stays only at BACKEND.

1. The application writes something to FRONT and commits. The data is visible for queries such as `SELECT * FROM FRONT`, which are fully executed at the front end. Asynchronous replication to the back end is free to move the data to BACK, but it does not complete the replication instantly.
2. The application runs a `join SELECT * from FRONT, BACK` query. Because the data written at step 1 is still being synchronously moved from the front end, the data is not yet there in the back end, and recently written rows are not visible. Therefore, the *consistency* requirement for transactionality is violated.

Other scenarios are either variants of these or also violate the transactionality requirements in other ways.

4.3.5 Stored procedures in Universal Cache architectures

For some applications, implementing part of application logic in database stored procedures is practical design decision. However, procedure languages in different databases are not compatible. Using stored procedures at all in Universal Cache architectures is not suggested unless it cannot be avoided.

If data can be split to the back-end and front-end segment, having two totally separate sets of procedures for the front-end and back-end databases might be possible. In these cases, having two procedure languages might be acceptable. The procedure at the front end would only be accessing data in front-end tables and the procedure at the back end would only be accessing data in back-end database tables.

If the back-end database is solidDB also, the procedures in the front-end and back-end database are code-compatible.

Stored procedures running in the front-end database can use SQL pass-through to access data in back-end database, similar to applications running SQL statements when Pass-through is turned on. This is one additional way for making the front-end/back-end database architecture invisible to the application.

4.4 Integrate an existing application to work with solidDB UC

The solidDB Universal Cache is seen as a way for speeding up existing systems that are running on top of disk-based enterprise systems. This way is true if the workload is favorable. See Chapter 6, “Performance and troubleshooting” on page 147 for further details.

Feasibility of retrofitting a cache database between back-end database and existing application depends on how well the issues we listed are handled in the existing application and whether the existing application can live with the implicit compromises. Feasibility and effort required for retrofitting might not always be obvious. A thorough analysis might be required. In this section, we outline the process for the analysis and present several workarounds to typical problems.

IBM solidDB supports the standard Java Transaction API (JTA), through providing a set of XA (see entry XA in “Abbreviations and acronyms” on page 277) classes. JTA methods enable the Transaction manager to control solidDB as one of the transactional resources in a global transaction.

4.4.1 Programming interfaces used by the application

Converting an application using a legacy database to using a Universal Cache database between legacy database and the cache database can be a relatively simple effort or a major project, depending on how the application has been designed and implemented.

Generally, the applications that have been implemented directly using JDBC or ODBC APIs, or a middleware running on top of those APIs, might require no conversion at all. If no extensions to the SQL Standard are used, the applications are expected to work with minor modifications.

Because stored procedure languages are not compatible with each other, a rewrite for stored procedures will be required if they are used in the application. This can be automated to some level but a separate project is necessary for stored procedure conversion.

If APIs, access methods or programming paradigms that are not supported by IBM solidDB (such as embedded SQL) are used, and there is no ODBC or JDBC-based middleware available to act as a gateway, this part of the application must be rewritten altogether.

4.4.2 Handling two database connections instead of one

Existing applications have most likely been written to handle connections to only one database, which would have all the required tables and data. Implementing the logic to an existing application that is capable for routing the queries to an appropriate database will not be a project without risks.

SQL pass-through provides a mechanism to combine the two database connections to one connection that routes every query to the front-end server at first. The front-end server might end up routing the query to the back-end based on table existence.

The change of transactional model included in moving from one physical database to a combination of front-end and back-end databases might prove to be a challenge to several applications. The combination of two databases is no longer strictly transactional, which might prevent migration the two database architecture altogether.

The strictness of the transactionality requirement should be estimated among the first things in technical feasibility assessment of moving to a front-end/back-end-based architecture. If the requirement is strict, the effort for implementing a transaction manager as part of a migration process (4.3.2, “Combining data from separate databases in a transaction” on page 87) must be included in the estimates.

Various applications rely heavily on a database’s capability of running complicated queries that combine data from several tables in one query. IBM solidDB does not have a capability of joining data residing in a front-end database and back-end database. The standard fallback mechanism is to direct all these kinds of queries to the back-end database. Although this approach works, it does not enable the application to take advantage of the performance benefits of main memory-based data management.

Based on analyzing the questions in this section, creating an estimate of the retrofitting effort for Universal Cache database is possible. It should be estimated in parallel to estimating expected performance and potential for other benefits of the product. The bigger the retrofitting effort is, the more substantial the benefits would need to be.

4.5 Data model design

All IBM solidDB architectures described in this section are based on IBM solidDB being a relational database product. To successfully design, implement, and deploy a relational database system to be able to handle even moderate data volumes and query loads, a number of basic relational database principles must be mastered. In this section, we briefly review the basic principles related to data model design with IBM solidDB products and emphasize the extra nuances implied by IBM solidDB internal implementation and Universal Cache architecture. We also review the aspects of running IBM solidDB in hybrid mode where some data is stored into in-memory tables and some to IBM solidDB's own disk-based tables. Finally, we look at aspects related to running IBM solidDB in a Universal Cache configuration where the performance-critical part of the data is defined to reside in front-end database and the rest will remain in the back-end database.

4.5.1 Data model design principles

Most database design methodology used in data model design is directly applicable with IBM solidDB products. These principles are as follows:

- Having unique primary keys for rows. As in most databases this approach is not enforced. If the schema does not contain primary key definition, a generated rowid is used instead. In IBM solidDB disk-based tables, the primary key defines the physical order of the rows in B-tree structure. With in-memory tables, the primary key is implemented only as an additional index. As a result, in using disk-based tables, the capability of writing and reading data in primary key order can have a substantial positive impact on performance.
- In IBM solidDB's main memory tables, images for indexes inside memory are created dynamically only at start-up and when running the database. Indexes do not consume disk space but do increase memory footprint and startup time. With disk-based tables, the indexes are stored on disk to a B-tree structure similar to other databases.
- Indexes will be required to speed up queries in both main-memory and disk-based databases. For bigger tables, full table scans are costly operations and also with in-memory databases. Maintaining several indexes can slow the speed of write operations in both in-memory and disk-based tables.
- Query optimization with IBM solidDB (with in-memory and on-disk tables) is a similar problem as query optimization is with any other relational database. IBM solidDB has a cost-based optimizer, diagnostic feature EXPLAIN PLAN FOR, and optimizer hints that are conceptually similar to other databases.

- FOREIGN KEY and UNIQUE definitions implicitly cause an index to be defined.
- IBM solidDB can use only one index per query per table. For example, if the query needs to find the rows according to index 1 and provide them in proper order according to index 2, only one index is used. The decision on which index will be used is determined by the optimizer.
- Large objects (both binary large objects and character large objects) are handled with separate algorithms and must be designed accordingly.
- Unlike most other databases, IBM solidDB is optimized for dynamic data sizes. Hence, there is capacity or performance benefit for pre-defining column sizes. For example, VARCHAR(20) has no other benefits on performance or capacity size than preventing too long strings to be inserted.

Implementation of IBM solidDB disk-based tables are relatively similar to other disk-based relational databases. In-memory tables inside solidDB have a similar performance edge over solidDB's disk-based tables as they have over other disk-based database product tables. The functional difference with IBM solidDB in-memory tables is not that substantial either. They key differences are as follows:

- Indexes do not increase disk file size.
- Different implementation exists for primary key.

The IBM solidDB product family does not contain a specific data modeling tool. The table, index, procedure and trigger creation is done by Data Definition Language (DDL) SQL Statements such as **`CREATE TABLE`**, **`DROP TABLE`**, and **`CREATE INDEX`**. Most modeling tools that support generic SQL Database through ODBC or JDBC Interface can work with IBM solidDB also.

4.5.2 Running in-memory and disk-based tables inside solidDB

The IBM solidDB database executable contains two database server engines. One engine is storing the data in memory using main memory-based algorithm; the other one is a disk-based algorithm to store the data on the disk. Like most other databases, IBM solidDB has an optimized buffer pooling mechanism to avoid unnecessary disk head movement.

IBM solidDB's hybrid nature is a powerful feature because it provides all the performance benefits of main memory database technology inside the same server with volume scalability benefits of disk-based database.

The hybrid server works based on the following principles:

- The address of the table is defined in CREATE TABLE statement, for example, with either STORE DISK or STORE MEMORY, combined with CREATE TABLE.
- The address of the table is fully transparent. All IBM solidDB SQL is fully supported in both engines and all data types are similarly supported.
- The addresses of the tables are fully transparent in transactions also. Transactions can combine write operations at disk-based and in-memory databases retaining full ACID properties. Also, individual statements can join data from both disk-based and in-memory tables.
- Although checkpoint algorithms between in-memory and disk-based tables differ, the database file (or files) and transaction log files are fully shared between the database engines. Therefore, checkpointing, database backup and restore procedures, HotStandby replication, and IBM solidDB Universal Cache functionality all work similarly regardless of whether tables inside solidDB engine are configured to be main-memory tables or disk-based tables.

Data model design for hybrid database with IBM solidDB creates an additional element to a regular data model design process. For each table, there is a decision whether to define the table to be an in-memory table or a disk-based table. Usually the amount of available RAM will set a hard limit to the overall amount of data that can be stored into all main-memory tables. The remaining tables will have to be disk-based tables. In most practical cases, picking up the right set of tables to reside in main-memory tables is not a trivial task.

When picking up the tables to be stored in the main-memory engine, consider the following aspects:

- The queries that will benefit from main-memory algorithm are the ones processing small amounts (one to five, ten, or more) of data inside one query. For larger result sets (hundreds, thousands or more rows), the performance benefit will be lost and a main-memory database will be close to a well-optimized fully buffer-pooled disk-based database.
- Finding the queries that are frequent enough and that benefit from main-memory technology performance might be sizable reverse engineering effort requiring specialized knowledge about the application. If possible, practical measurement is often the fastest way for reliable results.
- Because of application transparency, finding the optimal configuration between disk-based and in-memory tables can be iteratively experimental through a relatively simple process. Because of full transparency at the application level, the performance and capacity measurements can be done

with moderate effort by running and measuring the test application with different configurations.

In addition to simply choosing the right tables to be stored in the in-memory database engine, some logical changes in the data model can be beneficial (in certain instances). Design patterns include the following possibilities:

- Splitting the table vertically. For a table with large number of columns, it is possible to create a version of a table with key values and few performance critical columns in main-memory and a version with all columns on disk. This works well if there is a substantial performance-critical set of queries that only needs some columns. If possible, the application can combine the data from two tables (views work here also) and perform write operations to two tables (triggers can be used to automate this).
- In some cases, logically splitting the table horizontally (that is, performance is required to be faster for some particular rows inside a table regular disk-based performance is good for others) is required. This task can be done by creating a copy of the table with performance critical rows in a main-memory table and leaving the rest to disk-based tables. Changes on the application side are required to handle two tables. Similar to vertical splitting, these changes can be made easier by views.

4.5.3 Data model design for solidDB UC configurations

From a data model perspective, designing the data model to a Universal Cache installation is a problem that has some similarity to defining a data model to be used in solidDB hybrid installation. Similarities are based on the following items:

- In both cases, similar queries benefit from being executed by faster algorithms. These can be either inside main-memory tables, inside hybrid solidDB installation, or by main memory algorithms in the front-end database in solid UC installation.
- In both cases, the queries that benefit are similar in nature.
- In both cases, there is a hard upper limit, which is the memory capacity for the data that can be stored to main-memory tables.

Generally, picking some tables to be cached, as would be picked for main-memory tables from IBM solidDB disk-based stand-alone installation, can be a good first approximation.

Although picking the right tables for the front-end application is based on the same fundaments, the process is substantially more tedious to execute because the front-end and back-end databases are two separate databases.

As a result, note the following information:

- Creating applications that would automatically run on both databases might not be a trivial effort, based on the premise that potentially incompatible SQL must process two connections, transactional challenges, and other issues related to creating an application based on Universal Cache.
- In running the iterative tests for finding the right tables, the effort and time required for each iteration might be considerably bigger.
- The application behavior is expected to change based on changes in table location policies.
- Queries that are supposed to join data between the front-end and back-end databases will not work.
- A number of table selections that would be practical and available for in-memory tables in a IBM solidDB stand-alone installation are not feasible as cached tables in a solidDB Universal Cache installation.

The problems are considerably bigger when doing the effort for a system that was originally not designed to host a UC database because of potential unexpected compromises on transactionality and visibility of two database connections.

Splitting individual tables vertically or horizontally to front-end and back-end sections is conceptually possible with Universal Cache installations also. Masking the split in a database view, however, is not possible. Therefore, the easiest approach is to route all the queries that need to access the back-end segment of the data to the back-end database only, either by the SQL pass-through feature or directly.

4.6 Data migration

Real production database systems are seldom isolated systems without interfaces to other database systems. Interfaces for both reading the data from outside world or writing the data to other systems might be required either in the initial setup of a system, on regular basis with sparse intervals or on an almost continuous basis.

A set of typical data migration options are illustrated in Figure 4-8.

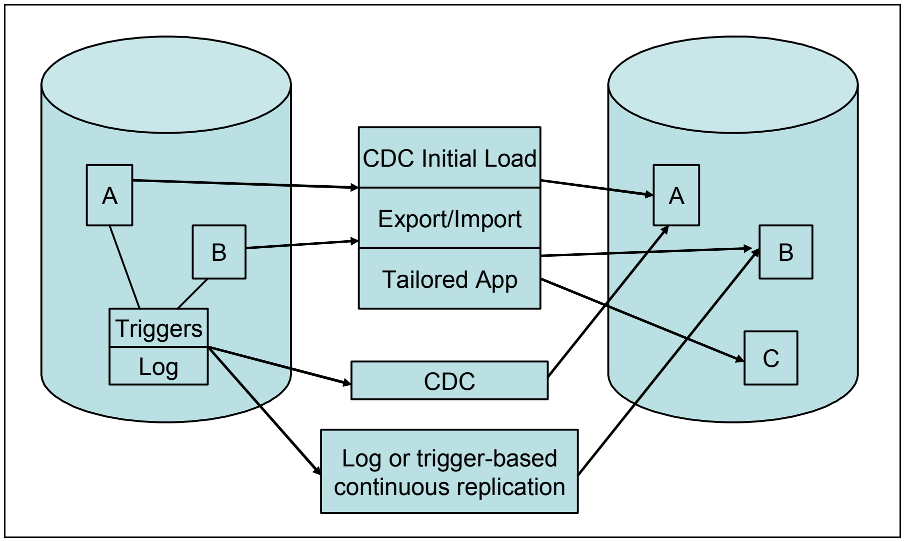


Figure 4-8 Typical data migration options with external database systems

For data transfer between separate databases, IBM solidDB provides the following capabilities:

- CDC replication provided as part of the Universal Cache product provides a subscription-based mechanism to enable initial data synchronization of any table configured to be part of a subscription and provide a mechanism for continuous near real-time synchronization of table data. This mechanism is available with all databases supported by CDC, such as DB2, IDS, Oracle, Sybase, and SQL Server.
- IBM solidDB has a set of simple ASCII-based export and import utilities to create ascii files on data content of individual tables and load them to a solidDB database. The ASCII files can be used in interfacing with other database servers.

- In certain cases with specific requirements for timeliness and transactionality of the data interface, the most practical approach is to design an interface application for reading the data from one database and writing it to the other. The interface application can be based on several mechanisms, such as the following mechanisms:
  - Running regular selects on the source database and write operations on the target database
  - Waiting for triggers to fire on the source database upon write operations, receive notifications and write the operations on the target database.
  - Using an API to access database transaction log and write the log entries to another database. In IBM solidDB this mechanism is possible through read operations in virtual table SYS_LOG.

Additional mechanisms exist for data transfers between IBM solidDB instances, such as copying the entire database file either by backup or a HotStandby netcopy operation or by using IBM solidDB Advanced replication for subscription-based data transfer between databases. However, discussion of these mechanisms are outside the scope of the chapter.

Choosing the right migration mechanism depends on several criteria:

- One-time operation versus a regular or continuous basis operation

  For one-time operations, the simplicity of implementation is crucial for avoiding costs. Performance and recoverability have usually limited importance in those cases. Some clients have decided to use CDC only for initial data load, without the need for continuous synchronization. Transferring individual tables through ASCII-based import and export tools is simple, but the complexity and vulnerability to manual errors increases when the number of tables increases. Implementing a specialized application is usually too costly for one-time operations.

  For continuous operations, that need to migrate data on a near real-time basis, the only practical options are CDC-based replication or specialized interface application based on triggers or Log API.

- Data volumes and performance criteria

  For high data volumes or situations where the data must be quickly transferred between databases, CDC-based solutions are too slow because configuring them to work on parallel threads is not possible. It is possible to run parallel ASCII-based import and export tasks. Also, implementing a specialized application can enable optimization for exact requirements.

- Need for recoverability during data transfer

  Often having relaxed requirements for the data durability during data transfer is possible. This is particularly true in the case of initial data loads, when the entire task can be re-executed from the beginning. The relaxed requirements, such as running the target system without transactional logging, can help in speeding up all import mechanisms.

- Need for transactionality

  In some cases, transactionality for migrated data is a must requirement. Usually, this means that the data arriving at the target system is expected to follow the transaction boundaries as they were when the data was originally written to the source system. Generally, this requirement is impossible to follow with table-based mechanism such as import or export. CDC-based replication and tailored interface applications running on triggers or Log API can enable transactional data migration.

- Simplicity of implementation and deployment

  The simplest mechanisms are ASCII-based export and import tools and CDC-based replication, assuming CDC is already installed.

- Need to process data migration

  In cases where table structures in source and target databases are not completely similar, some data processing is required for converting the data to fit the table structure of the target database. This task often requires implementing a specialized application to some level of the architecture or the clever use of views in the target database. The following list describes typical ways for doing the conversion:

  - Define a set of views in the source database to match the table structure of the target database. Export the data in views and run a regular import.
  - Write a specialized application that runs selects in the source database and populates the target database accordingly.
  - Create a set of working tables in the target database that follow the table definitions in source database and transfer the data in the most applicable method. Implement a specialized application that reads the working tables to populate the target tables. This option is logically similar to the previous option, but the application can be implemented in procedure language and can be made fully transactional.

4.7 Administration

When deploying a production system, plan for several policies regarding system administration. Requirements for administration policies might vary depending on system size, value, mission criticality, recoverability and other factors. Generally, the administration plans should cover at least the following system aspects:

- Regular health monitoring
- Recovery plan for expected failures and disasters
- Upgrade plan for expected hardware and software upgrades

This section describes the capabilities that are provided by IBM solidDB for all three aspects and describe practical solutions that have been used with solidDB in real production systems.

4.7.1 Regular administration operations

For production database systems, there are almost always some planned regular administrative operations that must be completed on a regular basis. Because of solidDB's history in the original equipment manufacturer (OEM) market where database administration must be automated, most of these operations take place automatically without configuration or administration intervention. Optimizer statistics collection and rebuilding of indexes belong to this group.

For persistent database systems, backups are an operation that must be configured to take place at appropriate times. Also, successful completion of backup needs to be validated.

4.7.2 Information to collect

For validating the health of a database system, be sure to regularly collect a number of measurement values of system behavior. Failures and disasters can be prevented by reacting to abnormal or alarming values before the disaster actually occurs.

Table 4-1 on page 100 lists the typical measurable values to systematically monitor to detect the system health. For capacity type values, a hard limit usually exists, which cannot be exceeded, such as running out of disk space or memory. If this hard limit is exceeded, the server is expected to crash. Be sure to have a mechanism to detect the capacity growth and react either automatically, semi-automatically, or manually.

Sporadic or continuous increases or decreases in load intensity values might be expected or unexpected, and caused by undetected problems elsewhere in system. Abnormal peaks or growth trends must be detected and reacted to. Because there is no obvious limit for acceptable or healthy values, similar to disk space limits, the typical approach is to collect and monitor the values, and detect anomalies either automatically or by visual monitoring.

Table 4-1 Measurable values for monitoring

Category	What to monitor	Monitoring technique
Capacity		
Disk Usage	Database file size Database log file size Free disk space	pmon 'DB size' pmon 'DB free size' OS level commands
Memory footprint	SolidDB memory footprint	pmon 'Mem size'
Load Intensity		
Database intensity	SQL Operations Table level operations	pmon 'SQL execute' pmon 'DBE insert' pmon 'DBE delete' pmon 'DBE update'
Disk usage intensity	I/O load caused by database server	OS level commands
CPU usage intensity	CPU load caused by database server	OS level commands

In addition to collecting numerical values of system behavior, systematic collection of other information, such as successful completion of backups, unexpected error messages, HotStandby role changes, abnormal amount of failed login attempts, and regular validation of database file consistency, are often considered necessary. The `solmsg.out` and `solerror.out` plain language files contain information about the phenomena. Collecting the files (possibly with proper filtering) can enable detection of the phenomena. Validating the file consistency is possible with the **`-x testindex`** and **`-x testblocks`** command file options.

4.7.3 Procedures to plan in advance

For a system expected to run in production for longer periods of time, many types of upgrades are expected to take place during the life cycle of the system. To be able to execute these upgrades smoothly and successfully and without excessive interference with production use, the operations need to be planned in advance.

In addition to preparing for various types of upgrades during the life cycle of a system, you must prepare for types of failures in the system to avoid down time or an excessive amount of down time in the system.

Upgrades (especially in-service upgrades) and fault tolerance are both fields of science by their own rights. This section describes the basic techniques, required and enabled by the IBM solidDB product, to manage several regular upgrade and failure scenarios:

The upgrade scenarios that we describe are as follows:

- SolidDB version upgrade
- Hardware upgrade
- Application upgrades including schema upgrade
- CDC upgrades
- Architecture upgrades, moving from non-HotStandby to HotStandby

The failure scenarios that we describe are as follows:

- Hardware failures causing database failures
- Software failures causing database failures
- Other failures

SolidDB version upgrade

As a minimum, upgrading from one IBM solidDB version to another involves shutting down the server, replacing the server executable in the appropriate directory, and restarting. The server versions are certified to be able to automatically open database files created by older versions of the product, occasionally needing a specific `-x convert` startup option. This way is certified to work between two major version levels. After a database has been converted to a newer version, opening with an older version is no longer possible. The solidDB HotStandby feature can facilitate solidDB version upgrades without system downtime through the following process:

Note: only an upgrade through one major version is certified to work.

1. Switch the role of the server to be upgraded to Secondary.
2. Shut down the secondary server process. The application continues to run on primary server.
3. Upgrade solidDB executable version to the new one at the server that was just shut down.
4. Restart the upgraded server.

5. Connect the server with the primary either with HSB CONNECT command or with HSB NETCOPY sequence if the servers were disconnected for a long period of time.
6. Repeat the process with the other server.

Hardware upgrade

Upgrading the hardware solidDB process involves getting the server executable, database file (or files), configuration file, and license file to appropriate directories in the new hardware, and restarting the system. In environments not having HotStandby in place, this process usually consists of shutting down the server and either copying the files to a new location or simply moving the disk. For HotStandby environments, hardware upgrade is possible without system downtime with a process that closely resembles solidDB version upgrade. The process is as follows:

1. Switch the role of the server to be upgraded to Secondary.
2. Shut solidDB process at the secondary server machine. The application continues to run on primary server.
3. Move the database file (or files) of the server down to the new hardware. The server executable, and configuration and license file must be in place also but they might have been moved before.
4. Restart the upgraded server machine and start solidDB process at the upgraded server.
5. Connect the server with the primary either with HSB CONNECT command or with HSB NETCOPY sequence if the servers were disconnected for a long period of time.
6. Repeat the process with the other server.

Application upgrades including schema upgrade

Most real production applications are built assuming that application can be upgraded during the life of the database. From the database server's perspective, this way usually involves changing the application software version. In a simple case, without schema change, the old version of the application is simply shut down and a new one is started. To avoid the time needed for application shutdown and restart, the database sets no limitation for new and old versions of the application to be concurrently connected to the database. Application changes without a database schema upgrade can be considered trivial from a database perspective.

Online application upgrades requiring changes in database schema are more complicated. To do this process successfully, the operation must already be considered in the design phase of the application. Similar to most other databases, IBM solidDB offers the following features:

- Discourages against using `SELECT *` command in the application. When the column structure changes, so does the result set.
- Enables adding and removing columns by the ALTER TABLE statement. Adding a column is potentially a complex operation if a default value needs to be populated to a large number of rows.
- Provides views and stored procedures that can be used as tools to hide the database structure from the application. By using these tools it might be possible to make some schema upgrades totally invisible to the application.

For a hybrid database installation (a schema upgrade that involves a disk-based table becoming a main-memory table or vice versa) doing the operation with ALTER TABLE is only possible for tables *without* data. For tables *with* data, the operation must be done in several steps, as follows (which moves a disk-based table called MY_TABLE, to main-memory):

1. Create a copy of the disk-based table with similar columns:
   ```
   CREATE MY_TABLE_COPY ... STORE MEMORY
   ```
2. Copy the data from original table:
   ```
   INSERT INTO COPY MY_TABLE_COPY VALUES (SELECT * FROM MY_TABLE)
   ```
3. Drop old table:
   ```
   DROP TABLE MY_TABLE
   ```
4. Rename new table:
   ```
   ALTER TABLE MY_TABLE_COPY TABLE_NAME MY_TABLE;
   ```

Observe that foreign key constraints referring to MY_TABLE in other tables might have to be dropped before the process and re-created after the process. For larger tables, consider the time required. The time needed is directly proportional to the number of rows. The speed of mass insertion to the same table on the same hardware and database server configuration provides a rough estimate of the table upgrade speed.

SolidDB Universal Cache does not directly support schema change replication. If DDL changes are required, then any associated subscriptions must be stopped, and both the source and target tables updated (for example, by running ALTER TABLE at both databases). Then, the table mappings in Management Console must be updated for both the source and target tables (by right-clicking the table in the Table Mappings view). The subscription must then be remapped (again by right-clicking in the Table Mappings view in Management Console).

Finally, the subscription can be restarted. The best approach is to explicitly start by doing a REFRESH operation to ensure that the contents of Source and Target tables are up-to-date. Mirroring can then be started (if needed).

CDC upgrades

If Change Data Capture (CDC) has already been installed and the replication is running, an upgrade of CDC can be installed on top of an existing one CDC. Uninstallation is not necessary because the upgrade can occur on an installed system. That said, a good approach is to stop the CDC components before upgrading. When the upgrade is executed, you are asked whether to upgrade the existing installation. If you choose to upgrade the existing installation, all configuration settings are preserved.

Architecture upgrades

Architecture upgrades with solidDB stand-alone products are similar to hardware upgrade operations in every respect with the exception of requiring a separate solidDB executable to run on new architecture in place. The solidDB database files and transaction log files are binary files with binary compatibility on all platforms supported by the product. An ASCII-based configuration file might require some environment-specific changes, such as exact directory paths in the new environment.

Solid HotStandby technology can be used to enable an architecture upgrade similar to what "Hardware upgrade" on page 102 describes.

Upgrade from a non-HotStandby solidDB installation to a HotStandby-based installation is a relatively simple operation, which includes the following steps:

1. Installation of second machine, with a server executable, and license file in place.
2. Configuration of the two servers to be HotStandby-enabled and aware of each other.
3. Running the `HSB SET PRIMARY` alone and `HSB NETCOPY` commands.
4. Waiting for the old database to be copied to the new server.
5. Running the `HSB CONNECT` command.

solidDB HotStandby is transparent to the application, so no application changes are required.

Hardware failures causing database failures

The solidDB HotStandby feature provides a mechanism to prevent committed data from being lost in the case of a single hardware failure. If the machine (or a crucial component in the machine, such as hard drive or network card) that is running the primary database fails, a secondary database has all the data already running and available for queries. All this is automatic and addressed in Chapter 5, "IBM solidDB high availability" on page 109.

When designing and configuring a HotStandby environment, consider the following information:

- What kind of high availability mechanism is used (High Availability Controller, Watchdog, other, none)?
- Are the applications expected to fail at the same time with the database? Can the applications fail over seamlessly?
- How will the hardware failure be automatically detected?
- Is the application allowed to continue without changes when running in failover mode?

For environments without HotStandby, IBM solidDB can survive some hardware failures without loss of data by restoring a database backup. If there are three separate physical hard drives, configuring separate drives for database files, transaction log files, and backup files, loss of any single disk drive can be tolerated without loss of committed transactions. Restoration of the database might involve manual work in moving the files to the replacement environment and starting the process, but no loss of data is implied. A separate plan for the recovery process is suggested for production systems.

Disk failures can cause the database file or transaction log file to be physically corrupted. With in-memory tables, the corruption usually prevents the server from starting; corruption in the disk-based tables is detected only when the corrupted disk block is read from the disk.

Software failures causing database failures

It is possible for the server process to go down without a hardware failure. Typical reasons might be that the operating system is out of resources (memory, file handles, other), an unhandled signal or deliberate process kill, or other problem.

The potential recovery processes can be as follows:

1. Restart the database process. If a problem exists with OS resources, they might not have been released when the database stopped, and so a restart might fail.
2. Boot the server and restart the database process. If another process was using all the OS resources, these resources might have been released when OS was booted.

Although restarting the database or booting the server machine might enable restarting the database process, the reason for the unexpected database failure must always be investigated. Collecting diagnostic information about operating system and database level before the failure can be immensely helpful in this process.

Other failures

Preparing for all types of failures outside of full database failure, is part of good systems design. Although full discussion of the topic is beyond the scope of this book, we briefly list failure types that you must be prepared for:

- Database-related application errors

  Errors in application code or failures in administration tasks (such as schema upgrades having failed) can lead to database error codes visible at the application level. All unexpected errors must be captured and logged for further analysis.

- Replication failures

  When replication fails, the databases are fully responsive and appear healthy on all database-level diagnostics. However, the data between front-end and back-end databases are out of sync and can lead to severe application problems. Replication must be monitored.

- Database overload or long response times

  In certain systems, long response times or decreased throughput are as serious as database failures. To manage the problems, monitor both throughput and response times. IBM solidDB's diagnostic features (described in 4.7.2, "Information to collect" on page 99) provide good tools for throughput monitoring. Monitoring response time is most practical to implement on an application level.

4.7.4 Automation of administration by scripts

In IBM solidDB, all administrative commands can be performed by solidDB SQL Extensions through the ADMIN COMMAND API. These SQL Extensions can be executed in several ways:

- Manually from solsql or another SQL Editor
- Part of a script run by **`solsql`** or **`solcon`**

  The **`solcon`** tool is a subset of **`solsql`** in the sense of being able to execute only ADMIN COMMANDs. If only **`solcon`** is deployed at production site, the administrators are unable to accidentally access the data.

- As SQL statements executed by a specific administration application through ODBC or JDBC

Typically, ADMIN COMMAND returns a regular SQL result set that can be viewed or processed by the application, just as regular data. Some admin commands, however, create a result file in the working directory of the server. For regular monitoring purposes, these files must be collected. The common monitoring commands that create the output to file are SQL Tracing (ADMIN COMMAND **`mon on`**), systematic collection of pmon counters (ADMIN COMMAND **`pmon diff`**) and detailed server variable dump (ADMIN COMMAND **`report`**).

5

IBM solidDB high availability

IBM solidDB high availability (HA) database is presented in this chapter. We describe the principles of HA, illustrate how the solidDB HA solution works, and how it can be used to its full potential.

Although the requirements for high level of HA are typical, for example in carrier-grade telecommunication networks and systems, IBM solidDB and its HotStandby technology (solidDB HSB) is used also in many other areas, such as railway systems, airplanes, defense applications, TV broadcasting, banking and trading applications, and so on, all the way to web applications and Point-of-Sale systems.

Any system preferring to guarantee its uptime in case of failures should consider making the system more redundant, and more tolerant against hardware and software problems, and also against human errors. In short, highly available.

When *thinking HA*, it is important to recognize that HA is more than a replication solution. HA thinking must include, among many other things, how to handle HA management and how to execute automated recovery after failure situations.

5.1 High availability (HA) in databases

The goal of HA systems is to make system failures tolerable. The extent failures are tolerable is specified with the availability measure `A` that is equal to the ratio of the time a service is operational to the total time the service is supposed to be operational. Availability may be derived from the maximum duration of an outage (equal to mean time to repair, MTTR) and the frequency of outages (represented with mean time between failures, MTBF), by using the following formula:

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

The value of `A` in the formula can be measured over a longer time or be calculated based on some estimates. The higher the value of `A`, the better is the availability of a system. When the required value of `A` is close to 1, the value of `MTTR` becomes small. For example, for `A` being 0.99999 (referred to as the five 9's availability), the total yearly `MTTR` is approximately 5 minutes. In a six 9's system, it is close to 30 seconds. Given the fact that there can be more than one failure per year (that depending on the reliability of the system) the time left to a single repair is a fraction of the yearly `MTTR`. It is not difficult to guess that the action of repair cannot be left to humans in systems aspiring to more 9's than four. It has to be performed by an automated system. An HA system is synonymous with a system having components designated to detect failures and deal with them automatically.

To deal with failures, an HA system embodies redundancy both in hardware and software. Redundant system parts are used to mask the failures. Various *redundancy models* can be applied. In the simplest redundancy model, called *2N*, or *hot standby*, the two units, *active* and *standby*, make up a mated pair. If a failure occurs, the failed active unit (hardware or software) is quickly replaced with a corresponding standby unit. That operation is called a *failover*. The purpose of failover is to maintain the required availability level, in the presence of failures. Other possible 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. In this chapter, the focus is on the principles and usage of solidDB hot-standby DBMS.

5.2 IBM solidDB HotStandby

IBM solidDB HotStandby (HSB) is an HA solution offered with solidDB. The normal solidDB product package image contains all the necessary components needed to enact the HA configuration. For example, the same server binary is used for both stand-alone and HotStandby operation modes. The latter one is enabled with configuration parameters. There are also other HA components that are described in this section.

5.2.1 Architecture

In solidDB HSB, a stream of transactions is continuously sent from the Primary (active) server to the Secondary (standby) server, by way of a replication protocol, as depicted in Figure 5-1. The connection between the servers is called an *HSB link*.

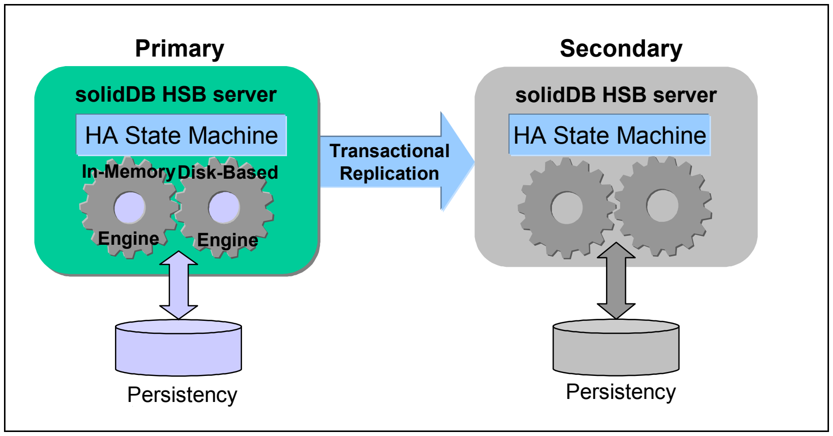


Figure 5-1 Hot-standby database

Figure 5-1 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. Note that, even in the case of an in-memory database, there is persistent storage, allowing for recovery of the database. In a shared-nothing system, you are also protected against media failures. On the contrary, in a shared-disk (or shared-storage) system, the assumption is that the common storage does not ever fail.

What differentiates an HA DBMS from one that is non-HA is the existence of an HA state machine, in the database server. The HA state machine makes the server aware of the HA state. The importance of the HA state machine is in preserving 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 will be disabled.

If any failure on the Primary site occurs, the failover takes place as depicted in Figure 5-2.

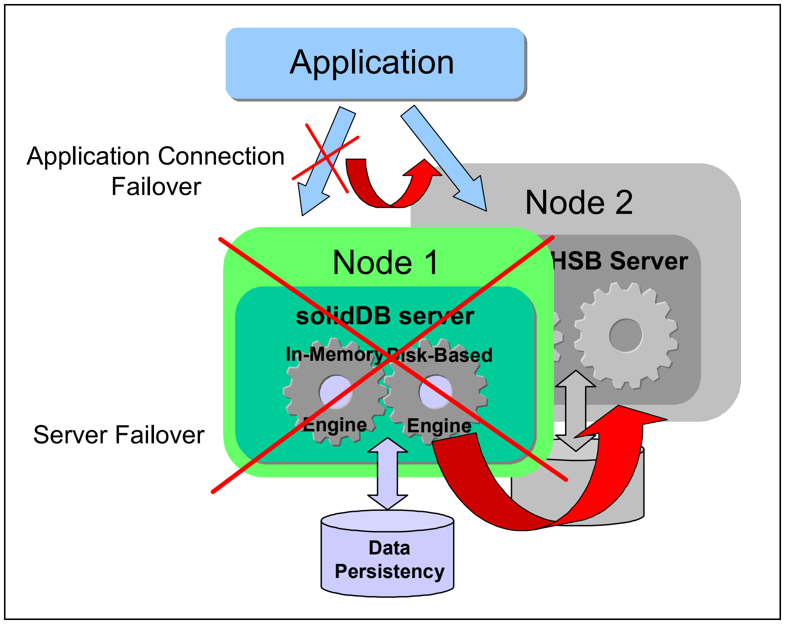


Figure 5-2 Failover

In a failover, two events happen:

1. The Secondary server takes over as a new Primary.
2. The applications are reconnected to the new Primary.

The way this happens is described in the subsequent sections.

5.2.2 State behavior of solidDB HSB

IBM 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 handing and recovery actions.

The HA state machine of solidDB HSB is illustrated with a simplified state diagram, as shown in Figure 5-3 on page 113. The operational hot-standby states are shown on the right side of the diagram: PRIMARY ACTIVE (active) and SECONDARY ACTIVE (standby). Other states come into the picture when taking care of various failure, startup, and in reconfiguration scenarios.

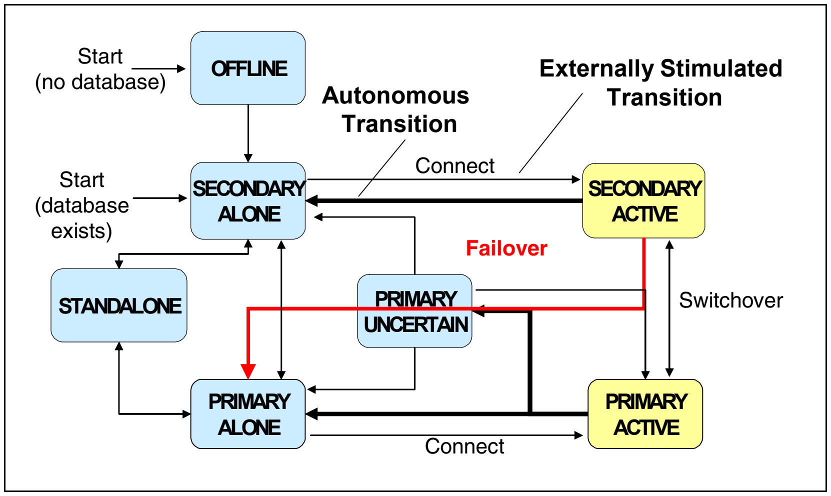


Figure 5-3 State transition diagram of solidDB HSB

The state behavior is externalized in the form of commands for invoking state transition and querying the state. The commands are available to applications (or a dedicated HA controller) as extensions of the SQL language (for a full description of the states and transitions, see the *IBM solidDB: High Availability User Guide*, SC23-9873). The transitions shown in bold are executed autonomously by the database server process. They have to do with 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 possible thanks to a built-in heartbeat function. All other transitions are invoked with administration commands.

Thus, 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,`** that may 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, that is HA-aware in the sense that it involves collecting committed transactions to be delivered later to the Secondary, upon reconnect and the resulting *catchup* (that is, resynchronizing the database state).

If a failure occurs, the situation we are dealing with is such that 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 may 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.

Note: The PRIMARY UNCERTAIN state is not mandatory, it may be by-passed with a configuration parameter setting.

In addition to the transitions dealing with failures, a role switch may be performed for maintenance purposes. It is invoked with dedicated admin commands **`hsb switch [to] primary`** and **`hsb switch [to] secondary`**. That operation is called a *switchover*.

5.2.3 solidDB HSB replication and transaction logging

The purpose of the HSB replication protocol is to carry the transaction results safely from the Primary and Secondary, over the HSB link. When delivered, the data serves the purposes of allowing for the following events:

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

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

Transaction Logging

Both the replication and transaction logging move the committed transactions out of the volatile memory, as depicted in Figure 5-4 on page 115.

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 way of a replication protocol. The Secondary DB may 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 in 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 represent some point in the past. In solidDB, the database file always contains a valid checkpoint.

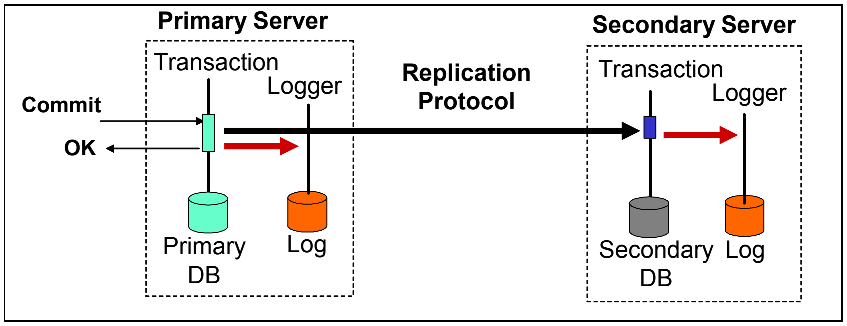


Figure 5-4 Replication and logging in solidDB HSB

In the recovery process, the Log is used to bring the database to the latest consistent state, by performing the following actions:

1. Removing the effects of uncommitted transactions
2. Re-executing 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 shut downs:

- Atomicity means that no partial transaction results are persistently stored in the database (nor 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 way of a replication protocol. To preserve the database consistency in the presence of failovers, the replication protocol is built much 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.

Similar to log writing, the replication protocol may 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 5-5.

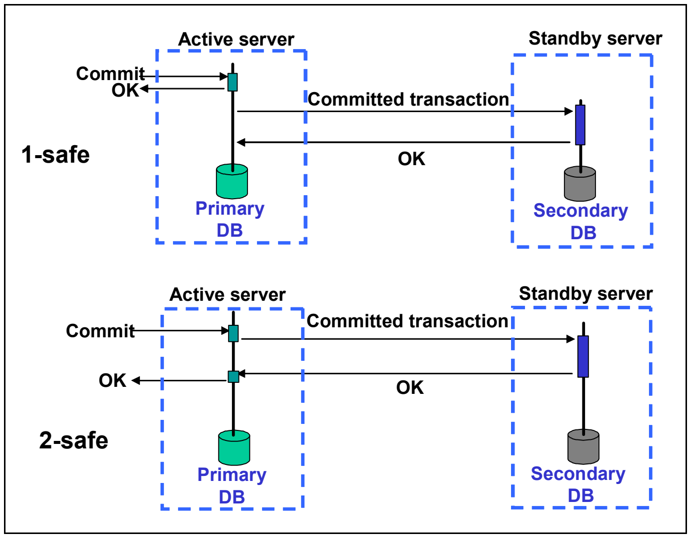
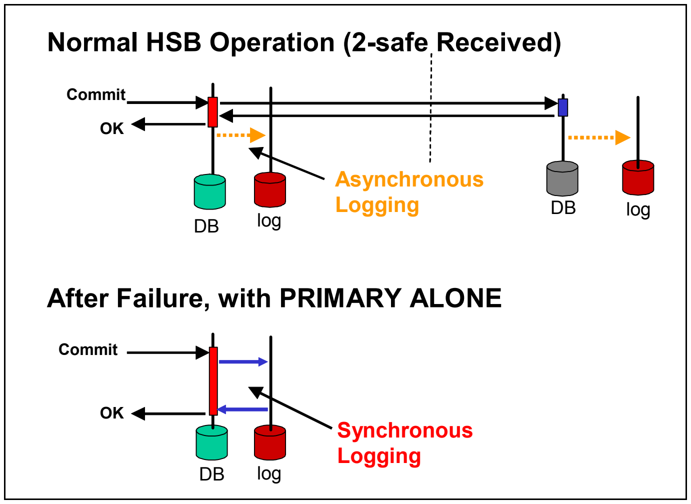


Figure 5-5 Illustration of the 1-safe and 2-safe replication protocol

You may see that the benefit of 1-safe replication is similar to that of relaxed durability. That is, the transaction response time is improved, and the throughput may be expected to be higher, too. 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 loosing 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 5-6). With adaptive durability, the Active server's log writing is automatically switched to strict if a node starts to operate without a standby. Otherwise, the Active server operates with relaxed durability.


Normal HSB Operation (2-safe Received)
Commit
OK
Asynchronous Logging
DB
log
DB
log
After Failure, with PRIMARY ALONE
Commit
OK
Synchronous Logging
DB
log
Figure 5-6 Adaptive durability

The possibility to transfer the log writing responsibility from the disk to the network is tempting because, by a common perception, a message round-trip travel over a high-speed network might be almost an order of magnitude faster than writing synchronously to the disk.

That perception was verified with a performance test. In the tests, both Primary and Secondary were dual CPU (Intel Xeon® E5410 2.33 GHz) systems each having a total of eight cores, two SATA disks and 16 GB of memory. The operating system was Linux RHEL 5.2. The load generator (TATP) was connected to the solidDB in-memory database by way of the shared memory access (SMA) driver library. The test used eight concurrent load generation threads.

With solidDB HSB running 2-safe protocol, the log-based durability was switched between strict and relaxed. The results are shown in Figure 5-7, for two read/write mix ratios being 80/20 and 20/80. The results were obtained with the Telecom Application Transaction Processing (TATP) Benchmark[1].

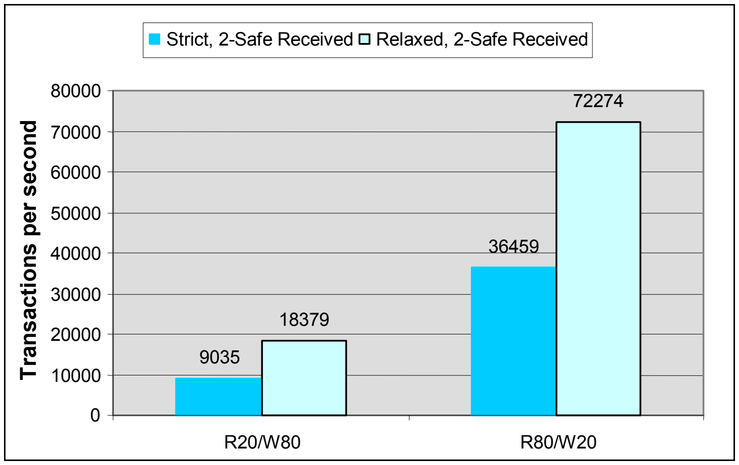


Figure 5-7 Impact of logging synchrony on performance

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.

[1] http://tatpbenchmark.sourceforge.net

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 5-8.

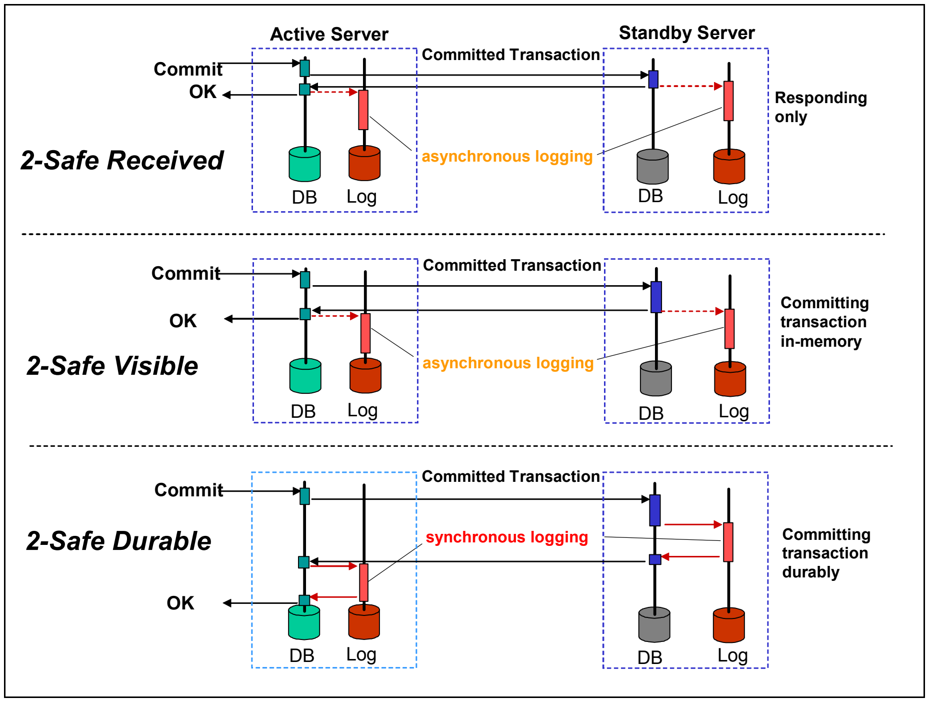


Figure 5-8 2-safe replication: policy levels

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 the 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 5-9.

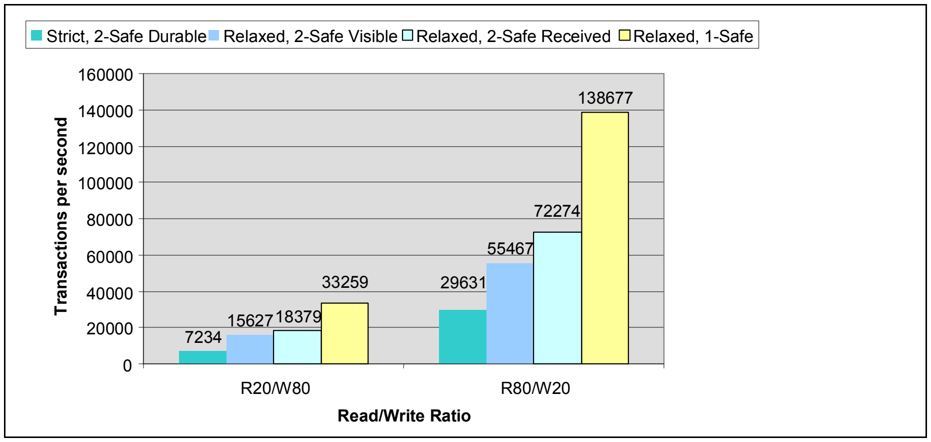


Figure 5-9 Impact of protocol synchrony on performance

You can see that the more asynchronous is the protocol, 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.

5.2.4 Uninterruptable system maintenance and rolling upgrades

To be able to run a database system for a longer period of time, there must be means to do necessary configuration changes, satisfy ad-hoc monitoring needs, and perform software updates without ever needing to shut down the system. These possibilities are available in solidDB.

Dynamic reconfiguration

A number of configuration parameters are defined in the configuration file, typically named *solid.ini*.

A need to change some of those parameters might appear because of changes in the application load or run environment. Many of solidDB configuration parameters can be changed online and immediately take effect. They are indicated with the *access mode* being *RW*, in the documentation. In particular, all crucial HSB configuration parameter are of that 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 scripts, or a program.

On-line monitoring

When a system is running 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, see Chapter 6, “Performance and troubleshooting” on page 147.

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. IBM 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.

5.3 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.

5.3.1 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 5-10.

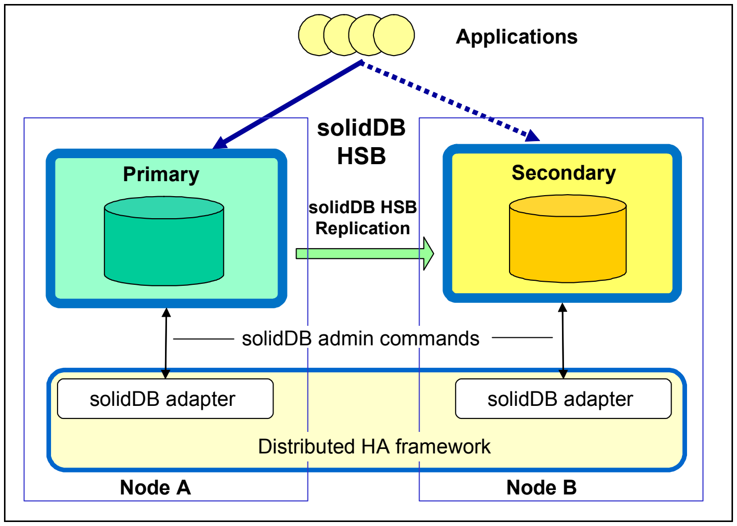


Figure 5-10 Using and external HA framework

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 implement 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 tasks 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.

5.3.2 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 5-11.

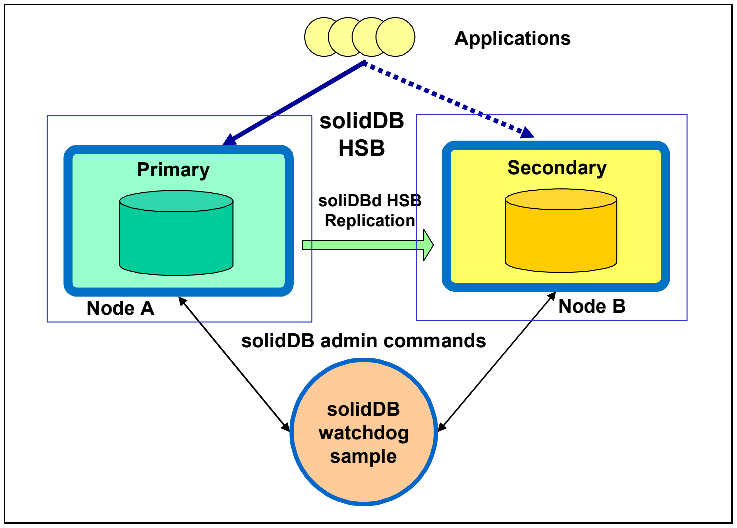


Figure 5-11 Using the Watchdog sample program

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.

5.3.3 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 5-12.

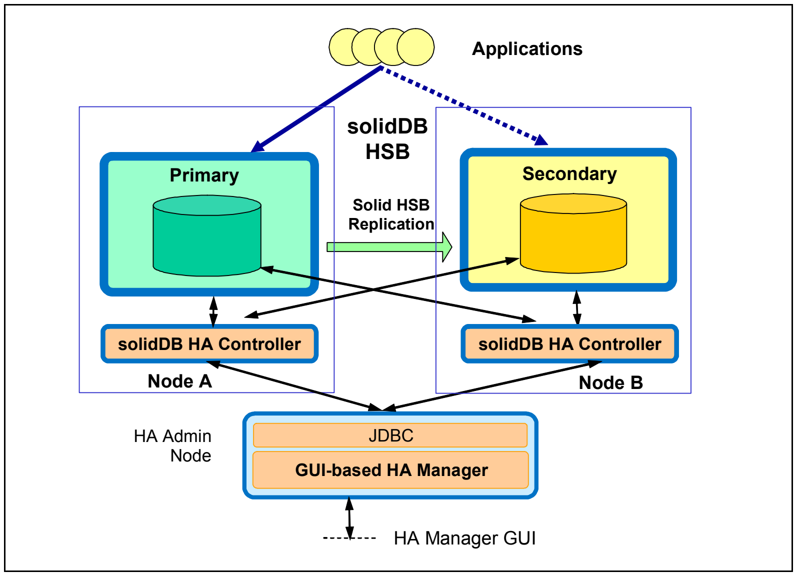


Figure 5-12 Using solidDB HA Controller

Failure detection in HAC is based on solidDB events. A surviving server will notify 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 5.3.4, “Preventing Dual Primaries and Split-Brain scenarios” on page 128.

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 5-13.

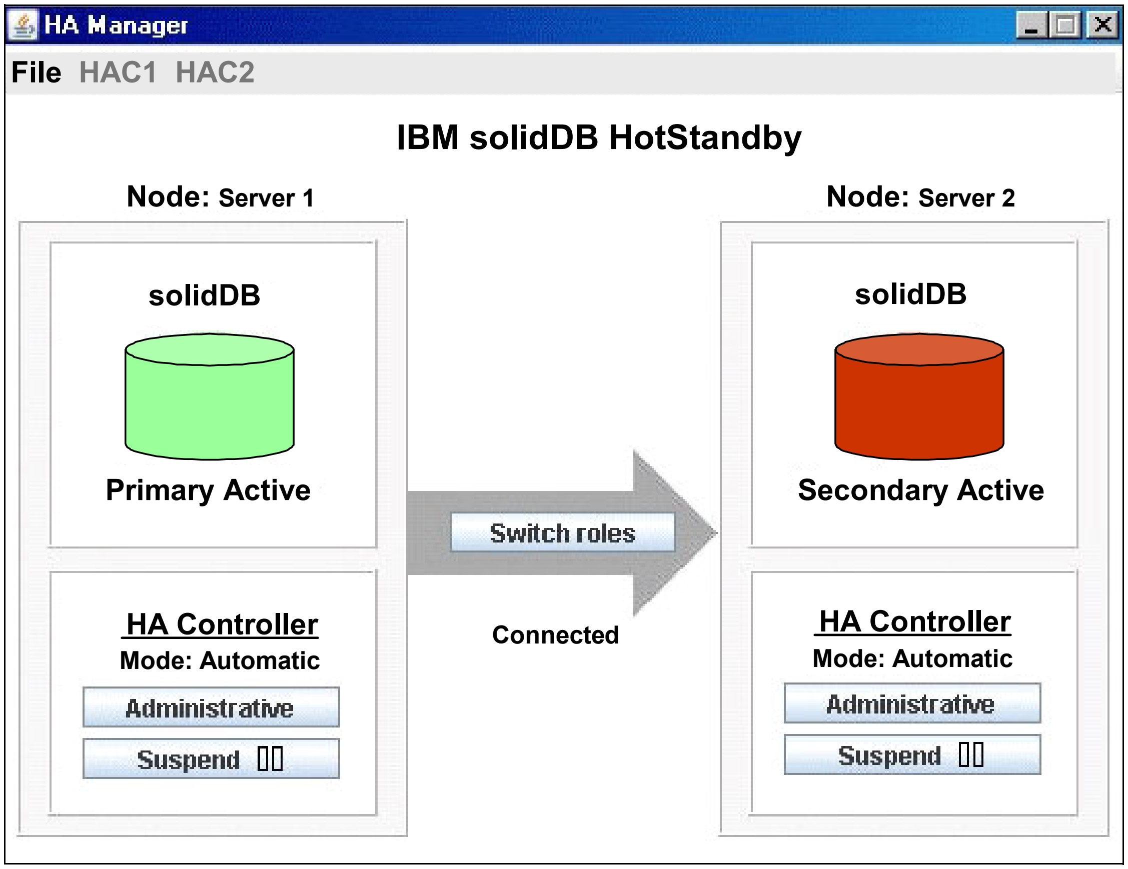


Figure 5-13 HA Manager GUI

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 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.

5.3.4 Preventing Dual Primaries and Split-Brain scenarios

A HAC configuration shown in Figure 5-12 on page 126 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 5-14.

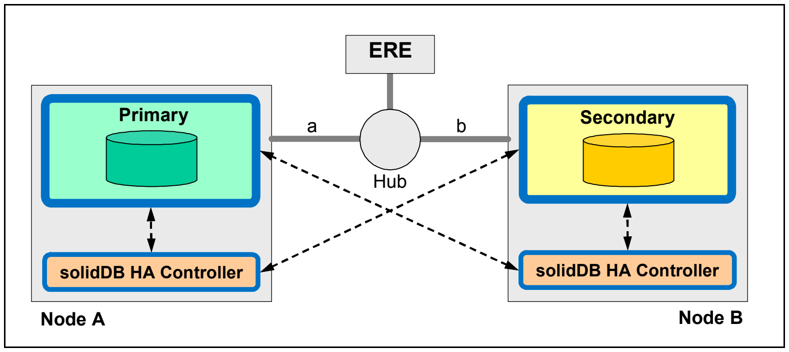


Figure 5-14 HA Controller used with the External Reference Entity

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 to become a Primary (Alone) database at the same time, preventing from 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 5-14, 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`.

5.4 Use of solidDB HSB in applications

In this section, we discuss how the applications use the underlying database pair. As examples, which database connect 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 IBM 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 IBM 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.

5.4.1 Location of applications in the system

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

In the first one, the application runs on a node separate from the database (or databases), as illustrated in Figure 5-15.

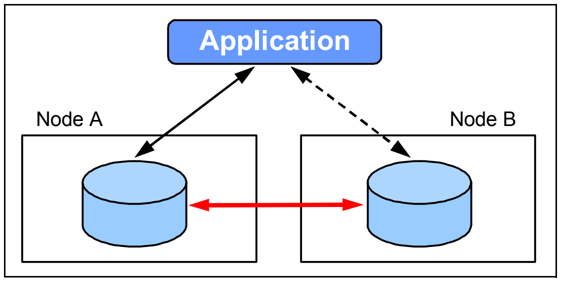


Figure 5-15 Application runs in a separate node from the databases

With this topology, the application normally continues to run regardless if one of the database nodes fail, 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 5-16.

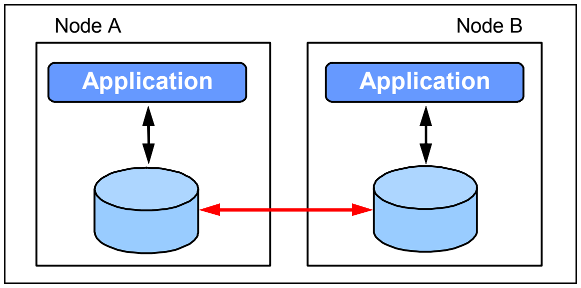


Figure 5-16 Application runs on the same node as the database

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 5-17, both applications can be active and using 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.

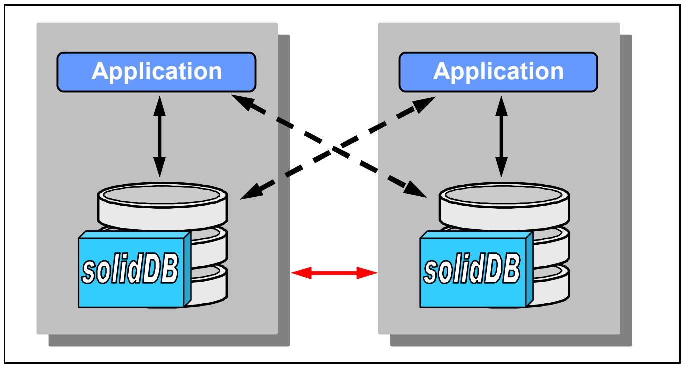


Figure 5-17 Two-node service cluster

5.4.2 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 IBM 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.

The *IBM solidDB High Availability User Guide*, SC23-9873, contains more details about the supported functionality, and also samples of the connection strings and URLs to use. The solidDB installation package contains sample C and Java program code for both ODBC and JDBC applications using the transparent failover handling.

5.4.3 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 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 IBM 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 *IBM solidDB: High Availability User Guide*, SC23-9873 and the *IBM solidDB: Programmer Guide*, SC23-9870 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.

5.4.4 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.

5.5 Usage guidelines, use cases

IBM 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.

5.5.1 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, 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.

5.5.2 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, that 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 effects performance, and hence they are discussed in a combined fashion in the following sections.

5.5.3 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.

5.5.4 Using default solidDB HA setup

The default solidDB HotStandby configuration is using Adaptive Durability (see “Adaptive durability” on page 117). 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?)*, several sample answers are available. Compared to a single database persisting all transactions properly and using synchronous log writing, this HA setup is, surprisingly enough, a faster mode. The explanation, however, is simple. The network I/O is faster than disk I/O. Sending the write operations to Secondary database is faster than flushing the data onto hard disk.

5.5.5 The solidDB HA setup for best data safeness

Finally, the most safe and also the slowest configuration, is using 2-safe durable hot standby configuration, which means using synchronous replication and synchronous log writing in both database nodes. In this case, 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 at the same level as with a single (stand-alone) server using synchronous (strict) logging.

Applications needing maximum safety for the write transactions should consider using the 2-safe durable solidDB HotStandby protocol configuration.

5.5.6 Failover time considerations

In terms of the differences regarding the failover time, in all configurations the error detection time is typically the same (a health check timeout, or similar). After the error has been detected and concluded to be 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 failed, the Secondary can be switched to 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 of 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 effects the speed also. If the requirement is rather to maintain the speed, even with the risk of loosing few transactions, then keeping the logging as relaxed also in the single node state (PRIMARY ALONE) should be considered.

5.5.7 Recovery time considerations

The recovery time depends on how many changes occurred during the downtime (the *delta*) and how much time is spent restarting the database. Also, the database checkpoint interval has an effect on how far back in time the restarted database has to go to find a proper sync point, from which the delta needs to be applied between the databases (the *catchup* phase).

Normally, making relatively frequent checkpoints is a good approach for minimizing the recovery time. This approach makes the database startup time faster, and also makes the catchup phase faster.

Several exceptions exist however, for example, when the database is of a small size, and the entire content changes quickly because of high speed writes. In this case, it may be actually be faster to copy the entire database over (using **`hsb netcopy`**) than 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.

5.5.8 Example situation

In this section, we illustrate a sample 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.

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

- Node A `solid.ini` configuration:

```
[Com]
Listen = tcp 1315

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

- Node B `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, including schema changes. You can now try creating a table, inserting rows (remember to commit), and then connect to Secondary database and read the data from there.

Note that in the initial situation (before executing **`hsb set primary alone`**) both databases have a secondary role, and any write attempts to either database will fail. This result is from having 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.

5.5.9 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 5-15 on page 129, 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 collocated with the server as shown in Figure 5-16 on page 130, 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.

5.6 HA in Universal Cache

A system for database caching, like IBM solidDB Universal Cache might be required of some HA characteristics also. In this section, we describe the possibilities and ramifications of Universal Cache HA configurations.

5.6.1 Universal Cache HA architecture

When Universal Cache (UC) is configured for high availability, hardware and software redundancy can be seen both in the front-end and back-end tiers, or in either one. The choice of where to apply the redundancy depends on the required availability level of the total system and expectations of the availability of stand-alone components. For example, if operation breaks (measured as MTTR) in the cache database are not allowed to be longer than it takes to restart a database server (and that might be long, for an in-memory database), the solidDB HSB configuration is needed.

If the UC application load is totally confined within the cache database (the pass-through is not used), the operation breaks of the back end are tolerable as long as the front-end transactions can be buffered in the cache database for later transfer to the back end (catchup). In that case, a stand-alone back-end server might be sufficient. If pass-through is used, an HA solution in the back end can be required.

The full HA configuration of Universal Cache is shown in Figure 5-18.

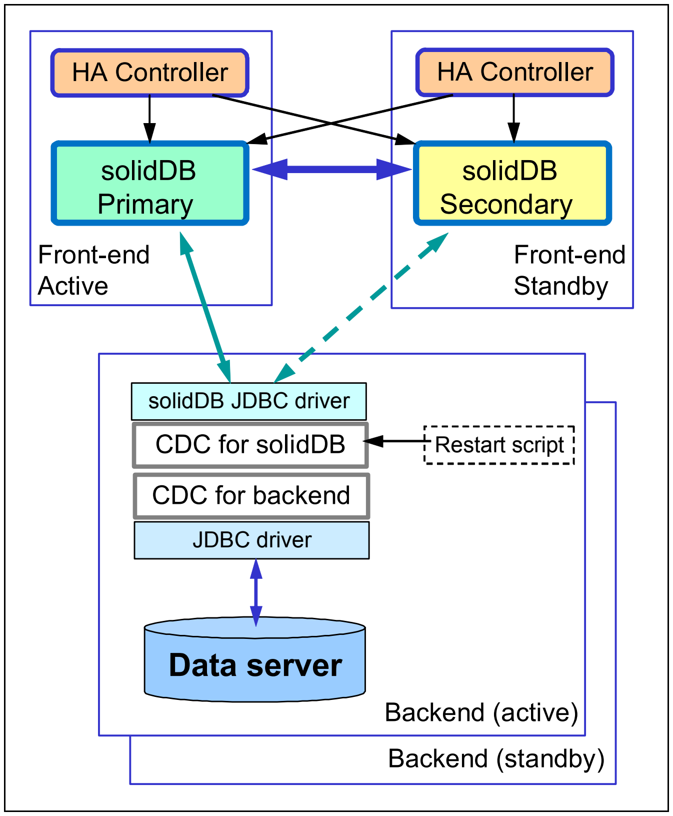


Figure 5-18 HA setup for Universal Cache

The front-end tier is configured in the same way as the solidDB-only HSB setup, involving HA controllers and, possibly, an ERE.

Regarding the InfoSphere CDC components, the InfoSphere CDC replication engine for solidDB (InfoSphere CDC for solidDB) is deployed on the back-end system instead of the front-end system, as was the case with the basic Universal Cache configuration. The reason is because the replication engine must survive solidDB server failovers. In some cases, the engine seamlessly continues the operation over the failover.

5.6.2 UC failure types and remedies

In a system such as the one shown in Figure 5-18 on page 143, various components can fail. In this section, failure types and recovery methods are described.

solidDB failures

Failures of solidDB servers in the front-end tier are dealt with in the normal ways that are available in the solidDB HSB solution. If the Primary server fails, a server failover is executed. If the Secondary server fails, the Primary continues to operate.

The InfoSphere CDC engine for solidDB has some resiliency to front-end failovers. All the subscriptions that use solidDB as a source continue mirroring normally, over a failover. However the subscriptions that use solidDB as a target stop the mirroring. They must be restarted using the InfoSphere CDC command-line interface. What we describe in the following text, must be implemented on an individual deployment basis. In practice, a shell script restarts the subscription. Also, a failover detection method must be implemented to initiate the script. It might be done with a component similar to the Watchdog that monitors the HSB server states, and act upon a detected failover.

InfoSphere CDC failures, InfoSphere CDC resilience

InfoSphere CDC replication engines can fail, causing the mirroring to stop. InfoSphere CDC does not have any built-in redundancy or failover methods. That means the InfoSphere CDC components must be explicitly restarted. That too can be achieved with shell scripts and a method to detect InfoSphere CDC component failures.

Temporary dysfunction of InfoSphere CDC components does not stop the cache database to serve the applications. Pay attention to restarting the InfoSphere CDC components fast enough for them to be able to catch up with the application load in the cache database. The actual time span allowed for the InfoSphere CDC replication engines to be non-functioning depends on the volume of updates in the cache database and the buffer size settings. The time can vary from minutes to hours.

Back-end failures

The back-end failures are dealt with in the ways typical for a given DBMS product and configuration. The solidDB Universal Cache product does not include any components to deal with those failures.

If the back-end database runs as a stand-alone system, normal restart and recovery is needed (to be initiated manually or automatically). Additionally, upon restart, the InfoSphere CDC subscriptions (or engines) must also be restarted.

If the back-end database is run in an HA configuration, the corresponding product-specific methods for failovers are used. Additionally, the InfoSphere CDC components have to be migrated, reconfigured, and restarted on a new active site. The process can be done with the InfoSphere CDC command-line interface and shell scripts.

During the unavailability of the back-end database, the cache database serves the applications. For the back-end database to be able to catch up with the load in the cache database, the back-end database and the InfoSphere CDC replication have to be reinstated in due time. If the back-end downtime is too long for catching up to be possible, the refreshing of the subscriptions must be done.

6

Performance and troubleshooting

Performance and troubleshooting is a vast and widespread topic that is also one of the most important to ensure the system runs as smoothly and as optimally as possible. In this chapter, we describe several valuable tools that can help with analyzing the performance of the solidDB server, InfoSphere Change Data Capture (CDC), and a user's application. The chapter also covers several tools and methods that can be used to troubleshoot situations such as an abnormal termination or a hang.

Application developers, database administrators, and system administrators can obtain practical and valuable information from this chapter and that can help ease their jobs.

6.1 Performance

This section provides an overview of the various tools available in the solidDB server and InfoSphere CDC to aid in performance analysis and monitoring. It then describes several common performance issues as observed from the application's perspective, and what can be done to address them. We provide pointers about which tools are best suited to help resolve issues.

6.1.1 Tools available in the solidDB server

At the heart of a database application is the solidDB server. The server is a highly complex piece of software that provides the functionality an application needs to retrieve and store data in a plethora of ways. When the speed at which an application performs does not meet expectations, one of the first places to look for performance monitoring data is the solidDB server.

In this section we cover the most valuable performance monitoring tools available in the solidDB server. We discuss these tools in a practical manner that can be used to directly relate to observations that may be seen in the applications and in the server.

Performance Monitoring (pmon) counters

Many pmon counters are available in the server. The counters provide a view into what the various components inside the server are doing, which can be directly correlated to observed application performance.

Methods of gathering pmons

The solidDB documentation describes the methods of gathering pmon data. The documentation is available at the following location:

http://publib.boulder.ibm.com/infocenter/soliddb/v6r5/topic/com.ibm.swg.im.soliddb.admin.doc/doc/monitoring.soliddb.html

In general, the most useful method is the continuous performance monitoring report ('pmon diff'), which produces a set of data for the time that the monitoring was enabled. This data can be considered to be similar to a *flight recorder* that contains critical information about the internal operations of the server during that time period.

If, for example, your application or the server exhibits slow or unexpected behavior during certain operations, gathering pmon diff data during this time is an excellent way to help determine the problem.

Example 6-1 shows commands to use within solidDB SQL Editor (solsql) to start and stop the utility.

Example 6-1 Starting and stopping the continuous monitoring report

```
Starting the continuous monitoring report:
admin command ‘pmon diff start /home/solid/pmondiff.txt 1000’

Stopping the continuous monitoring report:
admin command ‘pmon diff stop’
```

The interval specified is 1000 milliseconds, which is sufficient for most cases. Gathering data for too small an interval can add too much overhead and extra load on the disk subsystem, so use care.

Performance Monitoring counter details

Many counters are available; describing each of them in detail is impractical in this book. Furthermore, some counters only have meaning to solidDB support analysts and developers; others have been added to troubleshoot specific issues that rarely surface in general usage. A more valuable approach is to describe the more meaningful and useful pmon counters to help more quickly and easily troubleshoot performance problems. Therefore, the pmons are divided into the following categories:

- Low-level internal counters
- Internal structure counters
- SQL level counters
- Stored procedure/trigger counters
- Transaction level counters
- Logging and checkpointing counters
- HotStandby counters
- Lock-related counters
- Memory-table-specific counters
- SQL pass-through counters
- solidDB Universal Cache-specific counters

For each counter described in the tables, a *Quick diagnosis* column is provided to help you quickly understand under what externally visible conditions the counter is likely to be involved. A *Detailed description* column lists more background about what the counter means, what the values, or lack of values, in it means, and what interactions it might have with other pmon counters.

Low-level internal counters

These counters measure interactions typically at the level between the operating system and the solidDB server. Table 6-1 describes these counters.

Table 6-1 Low-level internal counters

Counter name	Quick diagnosis	Detailed description
File read / File write	High numbers can indicate excessive disk reads or writes.	Internally in the solidDB server, wrapper functions are around the read() and write() system calls. Each time these wrapper functions are called to read or write from a file, the respective pmon counter is incremented. Therefore if you see large numbers in these counters and you expect all of your tables to be in memory, you might have a problem that must be investigated. Note that other database operations, such as checkpointing, logging, and diagnostics (tracing and message logs) require file reads and writes. Make consideration for such events when analyzing these counters.
File open	High numbers can indicate excessive disk I/O.	Internally in the solidDB server, a wrapper function is around the fopen() system call. Each time a file is opened, this counter is incremented. Opening a file is an expensive operation in itself but also indicates that a read or a write to a file is required. This counter should be low when using only in-memory tables. Note, however, that other database operations such as checkpointing, logging, diagnostics (tracing and log messages) require file opens so this counter will likely not always be zero.
Mem size	This counter is primarily for informational purposes.	This is the current amount of memory that the server has allocated from the operating system, in kilobytes (KB). Every time the server requests memory from the operating system through malloc, and so forth, this counter is increased. This counter should correlate to the virtual memory size reported by operating system level tools such as *TOP* (which is a command in UNIX and Linux environments).
Thread count	Growing values can indicate connections are not being properly closed.	This counter is the total number of threads running internally in the server. It can be useful in tracking the number of connections to the database over time as there is one thread created per connection. Watch this counter for excessive values or for growing numbers. Growing numbers can indicate that connections are not properly disconnecting upon completion.

Internal structure counters

The pmon counters in this section provide information about internal server structures and tasks such as the database cache, the Bonsai Tree, merging, sorting, and table cursors. Table 6-2 describes these counters.

Table 6-2 Internal structure counters

Counter name	Quick diagnosis	Detailed description
Cache find	Low numbers can indicate ineffective database cache.	Each time something is found in the database cache, this counter is incremented. This counter applies only to disk-based tables, not to in-memory tables. If you use disk-based tables, you want to see large numbers in this counter. Compare this number to *Cache read* counter to determine the effectiveness of the database cache.
Cache read	High numbers can indicate excessive disk I/O.	Number of blocks (of database block size) that must be read from disk because of a database cache miss. It applies only to disk-based tables. You want this number to be as low as possible after the database cache is *warmed up,* meaning that the cache is full, or as full as it can be from normal database operation. Any time a cache miss occurs and data must be read from disk, performance is affected. If the total size of the database is larger than the amount of memory that can be assigned to database cache, numbers in this counter are likely to be seen depending on the workload and data access patterns.
Cache prefetch	High numbers can indicate table scans are being done.	Number of database blocks being prefetched into the database cache. This counter applies only to disk-based tables. High values in this counter indicate that the internal prefetching algorithm is detecting that large sequential reads are being performed. At the SQL level, this number can mean that table scans are being performed, which can affect performance and can be a result of a bad query plan or a missing index.
Cache prefetch wait	High numbers can indicate insufficient prefetching.	Number of waits that occurred because of prefetching while attempting a read. When a read is attempted, if prefetching is in the process of bringing the data into the database cache, the read must wait for the prefetch to complete. High numbers can indicate that prefetching is not performing appropriately for the type of reading that is being done. Consider increasing the IndexFile.ReadAhead configuration parameter.

Counter name	Quick diagnosis	Detailed description
Cache preflush	Excessive values can indicate insufficient database cache.	Number of preflush operations done by the preflush thread. A *preflush* is the asynchronous writing of dirty page data from the database cache to disk. This counter applies only to disk-based tables. Values in this counter indicate normal database operation when writes are occurring (meaning data is changing). If the number seems excessive, it might be an indication that the database cache is configured too small. It could also be an indication that the database cache is being used for other database operations and cache, which might or might not be expected.
Cache LRU write	High numbers can indicate insufficient database cache or insufficient preflushing.	Incremented when a dirty page in the database cache must be written to disk before a page being requested can be used. Values in this counter indicate that the database cache size is insufficient or that preflushing is not performing adequately. If preflush performance is suspected, considering modifying the IndexFile.PreFlushPercent configuration parameter.
Cache write bonsai leaf	High numbers can indicate cache is too small or applications not properly closing transactions.	Number of times that a Bonsai Tree leaf has been written to disk from the database cache. This applies only to disk-based tables. For best performance, the Bonsai Tree should remain in the cache. If cache does not have enough space, parts of the Bonsai Tree may be written out to disk to make room. This way affects performance negatively, which means either the cache is too small or the Bonsai Tree is too large. A large Bonsai Tree can be caused by applications not properly closing transactions.
Cache write bonsai index	High numbers could indicate unnecessary indexes exist.	Similar to *Cache write bonsai leaf* except that this counter indicates that a high volume of writes to a table, or tables, that have many associated indexes is occurring. If you see high numbers for this counter, examine your index design to determine whether any unnecessary indexes exist.
RPC messages	High numbers can indicate large result sets being returned over the network.	Number of RPC messages sent between client and server. Note that for SMA and Accelerator connections no RPC messages are sent. An SQLPrepare and an SQLExecute each send at least one RPC message. You can compare this number to the *SQL Prepare* and *SQL Execute* counters. If the value of this counter minus *SQL Prepare* is significantly larger than the number of *SQL Execute*, this can indicate that large result sets are being returned in the SQL queries. Note that *HSB packet count* may also be counted in this counter.
DBE inserts	This counter is mostly informative only.	This counter is for DataBase Engine (DBE) inserts and applies to both disk-based and in-memory tables. It is incremented when an insert to a table occurs.

Counter name	Quick diagnosis	Detailed description
DBE delete	This counter is primarily informative.	This counter is for DataBase Engine deletes and applies to both disk-based and in-memory tables. It is incremented when a delete to a table occurs.
DBE update	This counter is primarily informative.	This counter is for DataBase Engine updates and applies to both disk-based and in-memory tables. It is incremented when an update to a table occurs.
DBE fetch	High numbers relative to *SQL fetch* can indicate unnecessary table scans.	(This counter is not the same as *SQL fetch*.) Counts the number of rows that are fetched internally, which might or might not be returned back to the application. Compare this counter to *SQL fetch* to see how many rows are being read internally for each row returned to the application (rows read/rows returned ratio). If that ratio is high, you might have unnecessary table scans occurring and should investigate index usage.
DBE dd operation	This counter is mostly informative only.	Number of data dictionary operations that the server has executed. The data dictionary (or system catalog) stores all the definitions for tables, indexes, and so forth. This number should correlate to the expected number of data dictionary changes.
Ind write	Higher than expected values can indicate unnecessary indexes.	When a write operation is made to a table, this counter is incremented for each index that needs to be updated. Higher than expected values for this counter could indicate redundant indexes exist.
Ind nomrg write	High or growing values means the Bonsai Tree is too large.	Number of non-merged rows in the Bonsai Tree. This counter applies only to disk-based tables. This counter is essentially the size of the Bonsai Tree. If this counter is large or is constantly growing performance is affected because the Bonsai Tree consumes more database cache, which leaves less room for data caching and might ultimately cause disk paging.
Search active	High values result in more memory being used. Growing values can indicate a potential handle leak in the application.	Number of cursors currently open within the server. This important counter is used to determine memory growth in the server. Many applications are written to do their prepares up front, which is good for performance but can negatively affect memory use. If this number is high, consider reducing the number of concurrent connections or the number of prepares done up front. If this number is constantly growing it could indicate a handle leak in the application.
Merge nomrg write	Values higher than *Ind nomrg write* can indicate merging is not keeping up.	Number of index entries currently waiting for merge. In normal operation, this number is similar to *Ind nomrg write*. If the value of this counter is larger, it is an indication that merging is not keeping up and further investigation is required.

Counter name	Quick diagnosis	Detailed description
Merge level	Values not keeping up with *Trans read level* indicate merge is not keeping up.	Current merge level. Correlate this value with the *Trans read level* value. If the value of this counter is not keeping up, that is another indication that merge is not keeping up with cleaning the Bonsai Tree.
Sorter start sort	High values indicate excessive use of the external sorter.	Number of external sorts started. The external sorter is invoked when space required for the sort exceeds the memory available for the internal sorter. Because the external sorter spills to disk, excessive use of it negatively affects performance. Consider increasing the sort array size configuration parameter to avoid the external sorter.
Sorter add row	See *Sorter fetch row* for more information.	Number of rows being added to the external sorter. See *Sorter fetch row* for more information.
Sorter fetch row	If *Sorter add row* is incrementing faster than this, the external sorter is congested.	Number of rows per second that are fetched out of the external sorter. After a row is fetched, the memory is released. The *Sorter add row* counter incrementing faster than this one is a symptom of external sorter congestion, which can lead to unsatisfactory query performance. Consider increasing the memory used by the internal sorter with the SQL.SortArraySize parameter. Also consider reducing the number of sorts performed in the application. If external sorting is still required, try to speed up the external sorter by ensuring the underlying disk is as fast as possible. For example, use Solid State Disk or a RAMDRIVE.
Tabcur create	Significantly and constantly lower values than *Search active* could indicate too many unused statement prepares.	A table cursor is an active instance of a *Search active* cursor that is counted when a statement is actually executed. This way can be loosely correlated to SQL Execute times the number of cursors per statement. Use this counter with *Search active* to see what percentage of internal cursors that are created during statement preparation actually are used during statement execution.

SQL level counters

The pmon counters in this category keep statistics that are useful and meaningful to the application developer, as they are at the SQL level. Table 6-3 describes these counters.

Table 6-3 SQL level counters

Counter name	Quick diagnosis	Detailed description
SQL prepare	High numbers can indicate excessive SQLPrepare operations or insufficient statement cache.	Incremented every time a statement is prepared. Note that prepare operations done in stored procedures are counted in the *Proc SQL prepare* counter. Prepare operations are expensive; avoid them as much as possible. Also, a statement cache stores prepared statements. If a prepare is done for a statement that is saved in the statement cache, this counter is no incremented. If you see high numbers for this counter, consider doing fewer SQLPrepares or increasing the statement cache. Note also that a prepare operation is done implicitly when SQLExecDirect is used.
SQL execute	High numbers could indicate a hidden application problem.	Incremented every time a statement is executed. Note that execute operations done in a stored procedure are counted separately under the *Proc SQL execute* counter. If you see values for this counter that do not match your expectations, your application might have a problem.
SQL fetch	Numbers lower than *SQL execute* can indicate few rows being returned.	Number of rows returned to applications. An important note is that this counter is also incremented for things such as admin commands, LIST commands, and EXPLAIN PLAN FOR commands, as rows are returned to the user in all cases. This would explain cases where the value of this counter is less than the value of the *SQL execute* counter. Another explanation could be that return result sets often have zero rows.

Stored procedure/trigger counters

In solidDB, stored procedures and triggers are similar, so they are grouped together in this pmon category. Table 6-4 describes these counters.

Table 6-4 Stored procedure/trigger counters

Counter name	Quick diagnosis	Detailed description
Proc/Trig compile	This counter is primarily informative.	Incremented when a stored procedure or trigger is compiled. It occurs during the CREATE PROCEDURE or CREATE TRIGGER statement.
Proc/Trig exec	This counter is primarily informative.	Incremented when a CALL PROCEDURE statement is executed or when a trigger is fired, and can include nested stored procedure calls also. If you see growing values or higher than expected values, a stored procedure or trigger might be getting called (nested or not) more than expected.
Proc/Trig SQL prepare	This counter is primarily informative.	Incremented when an EXEC SQL PREPARE is done within a stored procedure or trigger. This counter is also incremented when an SQL EXECDIRECT is done, because a prepare operation is implicitly done.
Proc/Trig SQL execute	This counter is primarily informative.	Incremented when an EXEC SQL EXECUTE or EXEC SQL EXECDIRECT is done within a stored procedure or trigger.
Proc/Trig SQL fetch	This counter is primarily informative.	Incremented when an EXEC SQL FETCH is done within a stored procedure or trigger.

Transaction level counters

This pmon category describes counters specific to transaction levels. Table 6-5 describes these counters.

Table 6-5 Transaction level counters

Counter name	Quick diagnosis	Detailed description
Trans commit	Higher numbers than expected can mean autocommit is on unknowingly.	Number of transaction commits that have occurred, including commits done in application code, stored procedures, and in solsql (explicitly or with autocommit on).
Trans abort	Non-zero numbers can indicate a connectivity issue.	Number of transactions that have been aborted because of timeout or other issue. Values in this counter can indicate a possible connectivity issue where a client is able to connect and start a transaction, but then times out. The timeout period is configurable through the Srv.AbortTimeOut parameter and set at 120 minutes by default.

Counter name	Quick diagnosis	Detailed description
Trans rollback	This counter is primarily informative.	Number of explicit transaction rollbacks that have occurred.
Trans readonly	This counter is primarily informative.	Number of transactions that have been committed, rolled back, or aborted that contain read only operations.
Trans buf	High numbers indicate transactions are consuming a lot of memory. Growing numbers can indicate a long running query creating many rows and consuming significant resources.	Current number of rows that are in the transaction buffer. This number is essentially the working memory for a transaction and it gives you an idea of how much memory your transactions are consuming inside the server. If this number increases over several pmon intervals, this could indicate that one or more long running transactions are creating a large number of rows. Such a transaction can consume a significant amount of resources and might require further investigation.
Trans buf cleanup	Quickly growing values can indicate transactions are creating large numbers of rows and consuming significant resources.	Total number of transaction buffer cleanup tasks that have executed in the server because of startup. This task is responsible for removing stale rows from transactions that have committed, rolled back, or aborted to make room for new transactions. An internal threshold value determines when this task executes. If you see this value increasing rapidly, it should also be in conjunction with *Trans buf* having large values.
Trans buf removed	Quickly growing numbers indicate transactions are consuming significant resources.	Incremented every time a row is removed from the transaction buffer by the cleanup task. This counter is a supplement to the *Trans buf* and *Trans buf cleanup* counters.
Trans active	High values can indicate infrequent commits or roll backs. Growing values can indicate transactions are running slower over time.	Current number of active transactions in the system. Higher than expected values can indicate that transactions are not committing or rolling back frequently enough. Growing values can indicate either that workload is increasing or that transactions are running slower over time because of more resources being consuming for some other task for example.
Trans read level	Non-growing values can indicate one or more long running transactions blocking other transactions.	Current transaction read level. Write transactions cause this value to be incremented. If concurrent write operations are running but this value is not increasing, there might be one or more long running transactions in the system that should be investigated. Note that this is a 32-bit integer and can wrap to appear as a negative value.

Logging and checkpointing counters

This pmon category describes the counters that are specific to the logging and checkpointing tasks of the solidDB server. Table 6-6 describes these counters.

Table 6-6 Logging and checkpointing counters

Counter name	**Quick diagnosis**	**Detailed description**
Log write	Correlate with *Log file write* to see effectiveness of log writing.	Number of records being submitted to the internal logging component of the server per second. Correlate this counter with the *Log file write* counter to determine whether writing the log records to disk is keeping up with the number of records in this counter.
Log file write	Decreasing values can indicate logging is disk bound.	Actual number of blocks written to disk per second by the logger. When blocks are written is dependant on the Logging.DurabilityLevel setting. If the values in this counter are constant, it is likely a sign of healthy logging that is not disk-bound. A decrease in the values might indicate that logging is becoming disk bound.
Log nocp write	Can indicate the impact of the next checkpoint operation.	Number of pending log records because of the last checkpoint. This is an indication of how big and involved the next checkpoint operation will be. If these values are consistently large, increasing the frequency of checkpointing might be needed.
Checkpoint file write	Lower than expected values can indicate a disk bottleneck.	Number of file writes per second done by the checkpoint task. Use this counter to monitor the performance of checkpointing and the disk subsystem. Low values can indicate a disk bottleneck.

HotStandby counters

This pmon category describes several of the most important counters for HotStandby. Table 6-7 describes these counters.

Table 6-7 HotStandby counters

Counter name	Quick diagnosis	Detailed description
HSB cached bytes	Constantly growing values could indicate an internal problem.	Current size of the in-memory log reader, in bytes. The value of this counter should be relatively constant. An increasing value could indicate an internal problem.
HSB catchup reqcnt	Non-zero values indicate catchup is still in progress.	Indicates when the HSB catchup operation has completed. Non-zero values indicate that catchup is still in progress. This is given in requests per second.
HSB catchup freespc	Zero means that catchup cannot continue.	Number of log operations for which there is room during catchup. If this number becomes zero, catchup cannot continue.

Lock-related counters

This category of pmons describes the counters available to troubleshoot application locking issues. Table 6-8 describes these counters.

Table 6-8 Lock related counters

Counter name	Quick diagnosis	Detailed description
Lock ok	This counter is primarily informative.	Applies to both disk and in-memory engines (or main-memory engines, MME). It is the number of times an internal check for a lock indicated that no lock already existed, so execution could proceed.
Lock timeout	Increasing values can indicate an application locking issue.	Applies to both disk and in-memory engines. It is a count of the number of lock timeouts per second. This occurs when execution was blocked while waiting for another lock. Values that are constantly growing indicate that more and more operations are being blocked on locks, and a locking issue might exist.
Lock deadlock	Increasing values can indicate an application locking issue.	Applies to both disk and in-memory engines. A count of the number of deadlocks per second that have occurred.
Lock deadlock check	This counter is primarily informative.	Applies to both disk and in-memory engines. A count of the number of deadlock checks done by the lock manager.
Lock wait	Increasing values can indicate an application locking issue.	Applies to both disk and in-memory engines. A count of the number of times per second that a lock-wait occurred.

Counter name	Quick diagnosis	Detailed description
Lock count	Increasing values can indicate an application locking issue.	Applies to both disk and in-memory engines. A count of the total number of locks that exist in the system at a given time. A continuously increasing value can be an indication of a lock issue. Monitor this counter in conjunction with *Lock timeout*, *Lock wait*, and *Lock deadlocks* to watch for locking issues with your applications.
MME cur num of locks	Increasing values can indicate an application locking issue.	Locking for the in-memory engine is handled separately to locking in the disk-based engine. This is the current number of locks for in-memory tables.
MME max num of locks	Increasing values can indicate an application locking issue.	High-water mark for in-memory engine locks. Use this in conjunction with *MME cur num of locks* to watch for excessive number of locks.
MME cur num of lock chains	A large number of additional hash entries will degrade performance.	A lock chain is when the lock hash table has a conflict and additional locks are added to the hash entry as additional entries. This addition can slow down lock lookup because it has to first find the chain entry in the hash table, then parse the subsequent entries. Subtracting the value of this counter from *MME cur num of locks* gives the number of additional hash entries in the hash table. The larger that this number is, the more expensive lock processing is. In that case, consider increasing the MME.LockHashSize parameter.

Memory-table-specific counters

This pmons in this category are specific to the main memory engine (MME) of solidDB. Table 6-9 describes these counters.

Table 6-9 Memory-table-specific counters

Counter name	Quick diagnosis	Detailed description
MME mem used by tuples	This counter is primarily informative.	Tuples are an internal version of MME rows. This counter is the amount of memory in kilobytes (KB) needed to store all the rows of the in-memory tables. This value can be correlated to the total number of rows to determine the average row size (including overhead).
MME mem used by indexes	Higher than expected values could indicate unnecessary indexes.	Total memory in KB used by MME indexes.
MME mem used by page structs	This counter is primarily informative.	A page struct is the overhead needed for storing MME table and index data.

Counter name	Quick diagnosis	Detailed description
MME index key deletes	This counter is primarily informative.	Total number of delete operations on in-memory tables. Note that an UPDATE operation translates to delete then insert in MME tables.
MME index key inserts	This counter is primarily informative.	Total number of insert operations on in-memory tables. Note that an UPDATE operation translates to delete then insert in MME tables.
MME vtrie mutex collisions	Large values can indicate hot sections of data exist.	An increase in this counter means that several threads tried to access the same section of the trie at the same time. If you see values here, it means that the threads are trying to update the same information and are getting blocked.
MME vtrie version colls	High values could indicate hot sections of data.	For an update operation, a vtrie node gets a version update structure added to it. When another application views the same data that is currently being updated, the vtrie update structure is read, which results in a “vtrie version collision.” A high number here might mean that a “hot” section of data is being updated and read often. Specifically, values in the thousands or tens of thousands along with a significantly smaller number of updates can indicate a problem that most likely must be addressed in the application, perhaps with more frequent commits.

SQL pass-through counters

The counters in this category are specific to the SQL pass-through feature. Table 6-10 describes these counters.

Table 6-10 SQL pass-through counters

Counter name	Quick diagnosis	Detailed description
Passthrough rollbacks	This counter is primarily informative.	Incremented every time an explicit rollback is issued in the front end, or if the client disconnects from the front end. Note that the counter is incremented twice every time the client disconnects from the front end.
Passthrough result cnv	High values indicate lots of data conversion happening which can be expensive.	Incremented every time a result set value (retrieved from the back end) is being converted to a different internal data type in the front end. This conversion can be expensive so care should be take to avoid it if at all possible.

Counter name	Quick diagnosis	Detailed description
Passthrough param cnv	High values indicate lots of data conversion happening which can be expensive.	Similar to *Passthrough result cnv* except that it applies to SQL input parameters. For example, if the incoming parameter is a different type than the back-end column type it is addressing, conversion will occur.
Passthrough failures	Values indicate a problem that should be investigated.	Incremented every time either SQLPrepare fails (in the back-end) or a failure in type conversion between the back-end parameter/result set column and the corresponding types in the front end.

solidDB Universal Cache-specific counters

The pmon counters in this category are specifically related to solidDB Universal Cache implementations and do not apply to stand-alone solidDB. Table 6-11 describes these counters.

Table 6-11 solidDB Universal Cache-specific counters

Counter name	Quick diagnosis	Detailed description
Logreader spm freespc	Consistent values of 0 indicate write throttling is occurring in the solidDB server because of InfoSphere CDC replication not keeping up.	This counter, in conjunction with *Logreader spm waitct,* is important in determining whether write throttling is happening inside the solidDB server. Write throttling occurs when no space is available in the logreader buffer because of InfoSphere CDC replication not replicating the data to the back-end database quickly enough. Depending on your workload, occasional occurrences of this can be acceptable if not many write operations are having to wait; see the *Logreader spm waitct* counter. If this value is consistently 0, consider increasing the Logreader.Maxspace configuration parameter. If that does not help, investigate InfoSphere CDC performance; see 6.1.2, "Tools available in InfoSphere CDC" on page 180.
Logreader spm waitct	Non-zero values indicate write operations in the solidDB server are waiting for logbuffer space to be made available.	When the logreader buffer is full (*Logreader spm waitct* shows 0), write operations get throttled and must wait. Each time a wait occurs, this counter is incremented. The higher this value is, the more negative the write throttling effect is. See the suggestions in the *Logreader spm waitct* section.
Passthrough rollbacks	This counter is primarily informative.	Incremented every time an explicit rollback is issued in the front end, or if the client disconnects from the front end. Note that the counter is incremented twice every time the client disconnects from the front-end.

Counter name	Quick diagnosis	Detailed description
Passthrough result cnv	High values indicate lots of data conversion happening which can be expensive.	Incremented every time a result set value (retrieved from the back-end) is being converted to another internal data type in the front end. This conversion can be expensive so use care to avoid it if possible.
Passthrough param cnv	High values indicate lots of data conversion happening which can be expensive.	This counter is similar to *Passthrough result cnv* except that it applies to SQL input parameters. For example, if the incoming parameter is a different type than the back-end column type it is addressing, conversion will occur.
Passthrough failures	Values indicate a problem that should be investigated.	Incremented every time either SQLPrepare fails (in the back-end) or a failure in type conversion between the back-end parameter/result set column and the corresponding types in the front end.

Using the Monitor facility to monitor SQL statements

It is important to be able to analyze and tune performance at the individual SQL statement level. The solidDB server provides two useful tools to help with that effort:

- Monitor facility
- SQL Trace facility

The two are similar but provide slightly different data (see Table 6-12 on page 168 for a comparison). In this section, we describe the Monitor facility.

To enable the Monitor facility, execute the command shown in Example 6-2, in solsql.

Example 6-2 Enabling the Monitor facility in solsql

```
admin command 'monitor on';
       RC TEXT
       -- ----
        0 Monitor set to on
1 rows fetched.
```

After executing various SQL statements, `soltrace.out` file is created in the solidDB working directory. The contents of it is similar to Example 6-3.

Example 6-3 Default sample Monitor facility output

```
------------------------------------------------------------
2010-09-30 08:47:32
Version: 6.5.0.3 Build 2010-10-04
Operating system: Linux 2.6.18 AMD64 64bit MT
IBM solidDB Universal Cache 6.5
2010-09-30 08:47:33 50:15:exec rowcount 1
2010-09-30 08:47:33 7:13:execute Select D_NEXT_O_ID, D_TAX from
DISTRICT where D_W_ID = ? and D_ID = ? for update
2010-09-30 08:47:33 43:6:fetch next, 1 rows, total 1
2010-09-30 08:47:33 42:4:execute Insert into ORDERS values (?, ?,
?, ?, ?, ?, ?, ?)
2010-09-30 08:47:33 13:15:execute Update STOCK set S_QUANTITY =
?, S_YTD = ?, S_ORDER_CNT = ?, S_REMOTE_CNT = ? where S_W_ID = ?
and
S_I_ID = ?
2010-09-30 08:47:33 27:18:execute Update DISTRICT set D_NEXT_O_ID
= ? where D_W_ID = ? and D_ID = ?
2010-09-30 08:47:33 45:18:param 1:3721
2010-09-30 08:47:33 45:18:param 2:1
2010-09-30 08:47:33 48:18:execute Select S_QUANTITY, S_DIST_01,
S_DIST_02, S_DIST_03, S_DIST_04, S_DIST_05, S_DIST_06, S_DIST_07,
S_D
IST_08, S_DIST_09, S_DIST_10, S_YTD, S_ORDER_CNT, S_REMOTE_CNT,
S_DATA from STOCK where S_W_ID = ? and S_I_ID = ?
2010-09-30 08:47:33 48:18:param 1:4196
2010-09-30 08:47:33 48:18:param 2:471
2010-09-30 08:47:33 39:6:fetch next, 1 rows, total 1
2010-09-30 08:47:33 39:7:execute Select S_QUANTITY, S_DIST_01,
S_DIST_02, S_DIST_03, S_DIST_04, S_DIST_05, S_DIST_06, S_DIST_07,
S_DI
ST_08, S_DIST_09, S_DIST_10, S_YTD, S_ORDER_CNT, S_REMOTE_CNT,
S_DATA from STOCK where S_W_ID = ? and S_I_ID = ?
2010-09-30 08:47:33 39:7:param 1:282
2010-09-30 08:47:33 39:7:param 2:348
```

Note the time stamp at the beginning of each line and how all events appear to happen at the same time. The reason is because the default tracing timer resolution in solidDB is 1 second which is not practical for performance tuning purposes. Therefore be sure you add the following line to the `Srv` section of your `solid.ini` file and restart the server to enable millisecond trace timer resolution:

```
SRV.TraceSecDecimals=3
```

Example 6-4 shows the contents of `soltrace.out` from a solidDB server running a simplified transaction from an actual benchmark. The SQL statements that were run in the transaction are one SET command, two SELECT statements, and one UPDATE and one INSERT command.

Example 6-4 Sample Monitor facility output

```
------------------------------------------------------------
2010-11-17 09:21:22
Version: 6.5.0.3 Build 2010-10-04
Operating system: Linux 2.6.18 AMD64 64bit MT
IBM solidDB Universal Cache 6.5
2010-11-17 09:21:23 User 'DBA' connected, user id 23, machine id
coralxib02.torolab.ibm.com (127.0.0.1).
2010-11-17 09:21:23.676 23:0:opencursor SQL_CUR1 'SET PASSTHROUGH READ NONE
WRITE NONE'
2010-11-17 09:21:23.676 23:0:execute SET PASSTHROUGH READ NONE WRITE NONE
2010-11-17 09:21:23.676 23:0:exec rowcount 0
2010-11-17 09:21:23.677 23:1:opencursor SQL_CUR2 'Select C_LAST, C_CREDIT,
C_DISCOUNT, W_TAX from CUSTOMER, WAREHOUSE where C_W_ID = ? and C_D_ID = ? and
C_ID = ? and W_ID = ?'
2010-11-17 09:21:23.677 23:2:opencursor SQL_CUR3 'Select D_NEXT_O_ID, D_TAX
from DISTRICT where D_W_ID = ? and D_ID = ? for update'
2010-11-17 09:21:23.678 23:3:opencursor SQL_CUR4 'Update DISTRICT set
D_NEXT_O_ID = ? where D_W_ID = ? and D_ID = ?'
2010-11-17 09:21:23.678 23:4:opencursor SQL_CUR5 'Insert into ORDERS values (?,
?, ?, ?, ?, ?, ?, ?)'
2010-11-17 09:21:23.678 23:1:execute Select C_LAST, C_CREDIT, C_DISCOUNT, W_TAX
from CUSTOMER, WAREHOUSE where C_W_ID = ? and C_D_ID = ? and C_ID = ? and W_ID
= ?
2010-11-17 09:21:23.678 23:1:param 1:3838
2010-11-17 09:21:23.678 23:1:param 2:2
2010-11-17 09:21:23.678 23:1:param 3:23
2010-11-17 09:21:23.678 23:1:param 4:3838
2010-11-17 09:21:23.679 23:1:fetch next, 1 rows, total 1
2010-11-17 09:21:23.679 23:2:execute Select D_NEXT_O_ID, D_TAX from DISTRICT
where D_W_ID = ? and D_ID = ? for update
2010-11-17 09:21:23.679 23:2:param 1:3838
2010-11-17 09:21:23.679 23:2:param 2:2
2010-11-17 09:21:23.679 23:2:fetch next, 1 rows, total 1
2010-11-17 09:21:23.679 23:3:execute Update DISTRICT set D_NEXT_O_ID = ? where
D_W_ID = ? and D_ID = ?
2010-11-17 09:21:23.679 23:3:param 1:32
2010-11-17 09:21:23.679 23:3:param 2:3838
2010-11-17 09:21:23.679 23:3:param 3:2
2010-11-17 09:21:23.679 23:3:exec rowcount 1
2010-11-17 09:21:23.679 23:4:execute Insert into ORDERS values (?, ?, ?, ?, ?,
?, ?, ?)
2010-11-17 09:21:23.680 23:4:param 1:31
```

```
2010-11-17 09:21:23.680 23:4:param 2:23
2010-11-17 09:21:23.680 23:4:param 3:2
2010-11-17 09:21:23.680 23:4:param 4:3838
2010-11-17 09:21:23.680 23:4:param 5:2010-11-17 09:21:23
2010-11-17 09:21:23.680 23:4:param 6:NULL
2010-11-17 09:21:23.680 23:4:param 7:8
2010-11-17 09:21:23.680 23:4:param 8:1
2010-11-17 09:21:23.680 23:4:exec rowcount 1
2010-11-17 09:21:23.680 23:transopt commit (6)
2010-11-17 09:21:23.680 23:0:close
2010-11-17 09:21:23.680 23:1:close
2010-11-17 09:21:23.680 23:2:close
2010-11-17 09:21:23.681 23:3:close
2010-11-17 09:21:23.681 23:4:close
2010-11-17 09:21:23 User 'DBA' disconnected, user id 23, machine id
coralxib02.torolab.ibm.com (127.0.0.1).
```

To understand the output, you must first understand what each token in the output means:

- The first token is the time stamp. As we previously stated, ensure that millisecond resolution is enabled.
- The next token is the connection ID. This number uniquely identifies each client connection to the solidDB server. Example 6-4 on page 165 shows only one connection, which is represented by connection ID 23. The example also shows when the user connected and disconnected.
- The third token is either a statement ID or a transaction operation. Because Example 6-4 on page 165 represents output from five SQL statements that were run within one transaction, we can see statement IDs from 0 to 4. When a workload is running with more than one client that runs many SQL statements, the combination of connection ID and statement ID can uniquely identify each entry in the trace output. Using **`grep`** or **`search`** facilities in your favorite file viewing utility, this combination can help you to quickly isolate and view one sequence of operations. Note also that the Monitor facility can be enabled for a specific user to have the server output less information, which can be easier to analyze. This step can be done with the following admin command:

  `monitor on user` *`username`*

  When the third token is not the statement ID, it is usually a transaction level operation, such as commit or rollback.
- Finally, after the connection ID and statement ID, the output shows the actual trace data for the operation. This information can be the actual SQL statement being prepared or executed, the parameters being used, or another statement level operation being performed.

Analysis of Monitor facility output

Various determinations can be made from analyzing Monitor facility output. One of the most important determinations relative to performance is seeing the length of time for statement operations to complete.

Using the output in Example 6-4 on page 165, analyze several operations:

- Focusing on statement ID 2 (CUR2), which is a SELECT statement, you can see that the start timestamp for the prepare, shown as `opencursor` followed by the internally assigned cursor identifier is 2010-11-17 09:21:23.677. The execute started at 2010-11-17 09:21:23.679 which means that prepare took about 2 milliseconds to complete.
- Next we can see that the fetch completed at 2010-11-17 09:21:23.679, therefore it appears to have taken 0 milliseconds. In actuality, this means that the execution completed in sub-milliseconds or microseconds, but because the timer resolution is not able to display microseconds we do not know exactly how many microseconds. This is also why we say that the operations take “approximately” a certain amount of time rather than exactly that amount of time.

  The fact that the prepare took about twice as long as the execute aligns with the known fact that preparing SQL statements is more expensive than executing them. For this reason, prepare statements as few times as possible.
- We can also see from the output that statement ID 2 fetched a total of 1 row. This is important information to know, because, often the more rows that are being fetched, the longer the statement execution takes and the less advantage an in-memory database has over traditional disk-based database management systems.
- Now we look at another example, statement ID 4, which is an INSERT statement. The prepare started at 2010-11-17 09:21:23.678, the execute started at 2010-11-17 09:21:23.679 and the exec completed at 2010-11-17 09:21:23.680. This means that the prepare took about 1 millisecond and the execution appears to have taken less than 1 millisecond.
- Another useful item that can be gleaned from the output is the time duration for a transaction to complete. We can see that the transaction executed by connection ID 23 started at about 2010-11-17 09:21:23.676. We can find the end of the transaction execution by the token `transopt commit (6)`. Note that the 6 in parentheses is the internal identifier, for a commit transaction operation. The timestamp associated with this token is 2010-11-17 09:21:23.680, therefore it took about 4 milliseconds to complete the transaction.

Because the parameter values for the dynamic SQL statements are also displayed in the Monitor facility output, it is also useful for being able to reconstruct the actual SQL statements that were executed. You could then execute the statements with the same parameters in solsql, for example, to further analyze the statement. You can also examine the statement's execution plan with the same parameters to determine whether the statement is fully optimized. More details about this information is in "Statement execution plans" on page 170.

The Monitor facility also provides the user the ability to perform various ad-hoc per-statement statistic calculations. The performance monitor counters, which are described in "Performance Monitoring (pmon) counters" on page 148, are all at a global level. With some searching (`grep`) and text parsing of the output file, you can get statement level counters. For example, you can see how many rows a specific SQL SELECT statement returns during a given period of time.

Using the SQL Trace Facility to trace SQL statements

The SQL Trace Facility is similar to the Monitor facility but does have key differences. Certain investigations require the information that provided by the Monitor facility, others require the SQL Trace Facility, and others might require a combination of the two. Table 6-12 illustrates various differences between the two facilities.

Table 6-12 Comparison of the Monitor facility and the SQL Trace facility

Description	**Monitor facility**	**SQL Trace facility**
Trace SQL statements executed in stored procedures	No	Yes
Dynamic SQL parameter values dumped	Yes	No
Statement row counts dumped	Yes	No
Actual commit return code dumped	No	Yes
Includes user connect and disconnect messages	Yes	No
"trans begin" dumped at the start of transactions	No	Yes
Correlate statement ID to `userlist` and `sqllist` admin commands	Yes	No

Enabling the SQL Trace Facility is similar to enabling the Monitor facility, as shown in Example 6-5.

Example 6-5 Enabling the SQL Trace Facility in solsql

```
admin command 'trace on sql';
       RC TEXT
       -- ----
        0 Trace sql set to on
1 rows fetched.
```

Example 6-6 provides output generated by the SQL Trace Facility for the same transaction that was run to generate the Monitor facility output in Example 6-4 on page 165.

Example 6-6 SQL Trace Facility output

```
2010-11-17 11:11:38.959 2:sql:161:prepare SET PASSTHROUGH READ NONE WRITE NONE
2010-11-17 11:11:38.959 2:sql:161:execute:SET PASSTHROUGH READ NONE WRITE NONE
2010-11-17 11:11:38.960 2:sql:163:prepare SELECT C_LAST, C_CREDIT, C_DISCOUNT,
W_TAX FROM CUSTOMER, WAREHOUSE WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ? AND
W_ID = ?
2010-11-17 11:11:38.961 2:sql:164:prepare SELECT D_NEXT_O_ID, D_TAX FROM
DISTRICT WHERE D_W_ID = ? AND D_ID = ? FOR UPDATE
2010-11-17 11:11:38.961 2:sql:165:prepare UPDATE DISTRICT SET D_NEXT_O_ID = ?
WHERE D_W_ID = ? AND D_ID = ?
2010-11-17 11:11:38.961 2:sql:166:prepare INSERT INTO ORDERS VALUES (?, ?, ?,
?, ?, ?, ?, ?)
2010-11-17 11:11:38.961 2:sql:trans begin
2010-11-17 11:11:38.961 2:sql:163:execute:SELECT C_LAST, C_CREDIT, C_DISCOUNT,
W_TAX FROM CUSTOMER, WAREHOUSE WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ? AND
W_ID = ?
2010-11-17 11:11:38.962 2:sql:163:fetch
2010-11-17 11:11:38.962 2:sql:164:execute:SELECT D_NEXT_O_ID, D_TAX FROM
DISTRICT WHERE D_W_ID = ? AND D_ID = ? FOR UPDATE
2010-11-17 11:11:38.962 2:sql:164:fetch
2010-11-17 11:11:38.962 2:sql:165:execute:UPDATE DISTRICT SET D_NEXT_O_ID = ?
WHERE D_W_ID = ? AND D_ID = ?
2010-11-17 11:11:38.962 2:sql:stmt commit (0)
2010-11-17 11:11:38.963 2:sql:166:execute:INSERT INTO ORDERS VALUES (?, ?, ?,
?, ?, ?, ?, ?)
2010-11-17 11:11:38.963 2:sql:stmt commit (0)
2010-11-17 11:11:38.963 2:sql:trans commit (0)
2010-11-17 11:11:38.963 2:sql:161:close
2010-11-17 11:11:38.963 2:sql:163:close
2010-11-17 11:11:38.963 2:sql:164:close
2010-11-17 11:11:38.963 2:sql:165:close
2010-11-17 11:11:38.963 2:sql:166:close
```

The timestamp format and the connection IDs are the same as for the Monitor facility. However, immediately after the connection ID token is an `sql` token. This token is used because the `soltrace.out` output file can contain trace information for other components also.

Following the `sql` token is either a transaction level token or the transaction ID. Note that the transaction ID differs from the statement ID that is dumped in the Monitor facility output. The transaction ID is an internal number assigned by the server to each transaction. The statement ID in the Monitor facility is more useful as it can be correlated to the output of the **`sqllist`** or **`userlist`** admin commands.

A useful feature in the SQL Trace Facility is that it dumps out a `trans begin` token when the transaction is started. Note that in solidDB, transactions are started during the first SQL statement execution. Prepares and most SET statements are not part of a transaction.

The SQL Trace Facility does not dump out dynamic SQL parameter values, which can be considered an advantage if your goal is not to reconstruct exact SQL execution and rather analyze the flow of execution.

Timing statements using the SQL Trace Facility output is not as easy as with the Monitor facility as nothing is dumped when an INSERT, UPDATE, or DELETE statement completes. Timing transactions, however, is easier with the SQL Trace Facility. For example, the timestamp for the `trans begin` token is 2010-11-17 11:11:38.961 and the timestamp for the `trans commit` token is 2010-11-17 11:11:38.963. Therefore, this transaction took approximately 2 milliseconds to complete. Also note that the value in parentheses after the `trans commit` token is the actual return code of the commit, unlike the output from the Monitor facility.

Statement execution plans

When SQL statements are not running as fast as you expect them to, be sure to examine the execution plan of the statement, which can be done using the **`EXPLAIN PLAN FOR`** command. Basically, this means having solidDB display what lower level operations it will perform to complete the statement. From this, you can determine problems such as a table scan being performed where an index should be used instead, a loop join being performed where a merge join should be done instead, and so forth. After you determine problems such as these, you can then proceed to attempting to resolve them.

Important: The execution plan that is displayed is the plan that would have been executed at that point in time (the statement is not actually executed). It is not a guarantee that this will always be the execution plan that will be followed. The reason for that is because over time the data and associated samples may change which could then change the optimizer decision for how to execute the statement.

Getting an execution plan is simply a matter of prepending the following line to the SQL statement and running that in solsql:

“`explain plan for `”

Often times however, the SQL statement might not be known or the parameters to dynamic SQL are not known. The best way to address this issue is to enable the Monitor facility as described in “Using the Monitor facility to monitor SQL statements” on page 163, and then view the resulting `soltrace.out` file.

Example 6-7 shows sample output from the Monitor facility, which we can cut and paste from to obtain an execution plan.

Example 6-7 Obtaining SQL statement and its parameters from Monitor facility output

```
2010-11-16 19:17:07.389 3:1:execute Select C_LAST, C_CREDIT, C_DISCOUNT, W_TAX
from CUSTOMER, WAREHOUSE where C_W_ID = ? and C_D_ID = ? and C_ID = ? and W_ID
= ?
2010-11-16 19:17:07.389 3:1:param 1:2318
2010-11-16 19:17:07.389 3:1:param 2:6
2010-11-16 19:17:07.389 3:1:param 3:5
2010-11-16 19:17:07.389 3:1:param 4:2318
```

In solsql, you have two options for entering the complete SQL statement:

- Manually substitute the parameter markers (?), with the parameter values listed.
- Submit the SQL statement as is with the parameter markers and let solsql prompt you for each parameter, as Example 6-8 shows.

Example 6-8 Obtaining an execution plan in solsql

```
explain plan for Select C_LAST, C_CREDIT, C_DISCOUNT, W_TAX from CUSTOMER,
WAREHOUSE where C_W_ID = ? and C_D_ID = ? and C_ID = ? and W_ID = ?;
Param 1:
2318;
Param 2:
6;
Param 3:
5;
```

```
Param 4:
2318;
      ID   UNIT_ID    PAR_ID JOIN_PATH UNIT_TYPE          INFO
      --   -------    ------ --------- ---------          ----
       1         1         0         2 JOIN               LOOP JOIN
       2         1         0         3
       3         2         1         0 TABLE              CUSTOMER
       4         2         1         0                    PRIMARY KEY
       5         2         1         0                    C_ID = 5
       6         2         1         0                    C_D_ID = 6
       7         2         1         0                    C_W_ID = 2318
       8         3         1         0 TABLE              WAREHOUSE
       9         3         1         0                    PRIMARY KEY
      10         3         1         0                    W_ID = 2318
10 rows fetched.
```

Note that end of line markers (;) must be used after each parameter value is entered.

The actual execution plan is presented in a table form. This table form can be used to construct an execution plan graph or flowchart which is easier to read and understand.

You must first understand what the columns mean. The solidDB Information Center documents the meanings at the following location:

http://publib.boulder.ibm.com/infocenter/soliddb/v6r5/topic/com.ibm.swg.im.soliddb.sql.doc/doc/the.explain.plan.for.statement.html

Fully understanding this information, however, can still be daunting. To help you understand, we draw a picture of the execution plan shown in Example 6-8 on page 171.

To draw the picture, first start with row 1 which is the row that has UNIT_ID = 1. This row is part of the top most Unit or flowchart object, as shown in Figure 6-1.

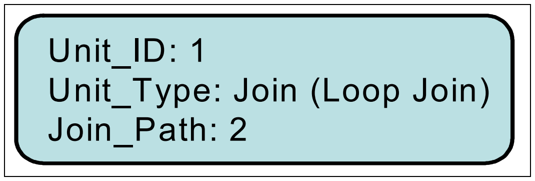


Figure 6-1 Start of the execution plan graph

Next, we see that the second row of the execution plan also has a UNIT_ID of 1 which means that the rest of the information in this row is associated with the same unit drawn in Figure 6-1. The only difference between this row and row 1, however, is the JOIN_PATH of 3 instead of 2. What these two JOIN_PATH values mean is that UNIT_ID 2 and UNIT_ID 3 are joined to this unit. Therefore, what we need to do next is add unit 2 and 3 to our graph, as shown in Figure 6-2.

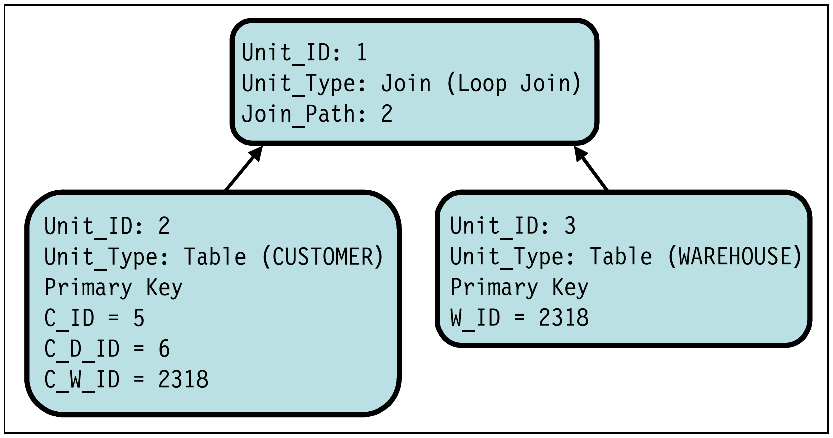


Figure 6-2 Final execution plan graph

Figure 6-2 is a relatively simple example, but it illustrates how to read, understand, and visualize the table form shown in Example 6-8 on page 171. After you become accustomed to looking at the simpler execution plans, you probably do not need to convert to or visualize it in a graph. However, some queries can become fairly complex; for those, you might want to draw them to help you visualize them in a graph form.

Using optimizer hints

The optimizer is a highly complex and accurate software component in the solidDB engine. It times however, it might not make a correct decision and you will have to suggest an alternative to it. This is where optimizer hints can be valuable. A hint is not to be confused with a directive. The optimizer overrides the hint if it has compelling evidence that the choice being made is correct. It is good practice to run your SQL statements through EXPLAIN PLAN FOR with and without the optimizer hint to see if the optimizer changed the plan.

The solidDB Information Center documents the optimizer hints:

http://publib.boulder.ibm.com/infocenter/soliddb/v6r5/topic/com.ibm.swg.im.soliddb.sql.doc/doc/using.optimizer.hints.html

However, we can walk through an actual example to see what effect a hint can have on a query. Example 6-9 on page 174 shows the query being used and the execution plan.

Example 6-9 Query with which to try an optimizer hint

```
EXPLAIN PLAN FOR
SELECT
                COUNT(DISTINCT S_I_ID)
FROM
                STOCK, ORDER_LINE
WHERE
                (S_W_ID = 2885 AND
                 S_QUANTITY < 18 AND
                 S_I_ID = OL_I_ID AND
                 S_W_ID = OL_W_ID AND
                 OL_D_ID = 9 AND
                 OL_O_ID BETWEEN 11 and 30);
       ID   UNIT_ID    PAR_ID JOIN_PATH UNIT_TYPE          INFO
       --   -------    ------ --------- ---------          ----
        1         1         0         0 GROUP
        2         2         1         0 ORDER              NO PARTIAL SORT
        3         3         2         4 JOIN               LOOP JOIN
        4         3         2         5
        5         4         3         0 TABLE              ORDER_LINE
        6         4         3         0                    PRIMARY KEY
        7         4         3         0                    OL_O_ID <= 30
        8         4         3         0                    OL_O_ID >= 11
        9         4         3         0                    OL_D_ID = 9
       10         4         3         0                    OL_W_ID = 2885
       11         5         3         0 TABLE              STOCK
       12         5         3         0                    PRIMARY KEY
       13         5         3         0                    S_W_ID = ...
       14         5         3         0                    S_I_ID = ...
       15         5         3         0                    S_QUANTITY < 18
       16         5         3         0                    S_W_ID = 2885
16 rows fetched.
```

As Example 6-9 shows, a LOOP JOIN is being performed to join the two tables. Without analyzing the size of the tables or any other statistic, assume for demonstration purposes that we think this query might benefit from doing a MERGE JOIN instead. To do a comparison however, we first must quantify the execution time of this query as it is.

The solsql query editor has the ability to time statements executed within it. To do this start **`solsql`** with the **`-t`** option. After every statement execution, the total time required to execute the statement will be displayed. If you use the **`-tt`** option, this functionality is enhanced further to show the timings needed for prepare, execute, and fetch separately as they occur. Example 6-10 on page 175 shows the execution of the sample query in **`solsql`** started with the **`-tt`** option.

Example 6-10 Sample query timed in solsql

```
SELECT
                COUNT(DISTINCT S_I_ID)
FROM
                STOCK, ORDER_LINE
WHERE
                (S_W_ID = 2885 AND
                 S_QUANTITY < 18 AND
                 S_I_ID = OL_I_ID AND
                 S_W_ID = OL_W_ID AND
                 OL_D_ID = 9 AND
                 OL_O_ID BETWEEN 11 and 30);
Prepare time 0.0004671 seconds.
Execute time 0.0044830 seconds.
COUNT(DISTINCT S_I_
-------------------
       12
Fetch time 0.0001431 seconds.
1 rows fetched.

Time 0.0051959 seconds.
```

As shown, executing the query took 0.0044830 seconds or 4.5 milliseconds.

Now, we analyze the query with a hint to use MERGE JOIN instead of LOOP JOIN. Example 6-11 shows the execution plan of the sample query with the optimizer hint to use MERGE JOIN. As we can see, UNIT_ID 3 is now a MERGE JOIN, so we know that the optimizer is using the hint.

Example 6-11 Execution plan of sample query with optimizer hint

```
EXPLAIN PLAN FOR
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED *)--
                COUNT(DISTINCT S_I_ID)
FROM
                STOCK, ORDER_LINE
WHERE
                (S_W_ID = 2885 AND
                 S_QUANTITY < 18 AND
                 S_I_ID = OL_I_ID AND
                 S_W_ID = OL_W_ID AND
                 OL_D_ID = 9 AND
                 OL_O_ID BETWEEN 11 and 30);
       ID   UNIT_ID    PAR_ID JOIN_PATH UNIT_TYPE          INFO
       --   -------    ------ --------- ---------          ----
```

```
         1         1         0         0 GROUP
         2         2         1         0 ORDER                 NO PARTIAL SORT
         3         3         2         4 JOIN                  MERGE JOIN
         4         3         2         6
         5         4         3         0 ORDER                 NO ORDERING REQUIRED
         6         5         4         0 TABLE                 STOCK
         7         5         4         0                       SCAN TABLE
         8         5         4         0                       S_QUANTITY < 18
         9         5         4         0                       S_W_ID = 2885
        10         6         3         0 CACHE
        11         7         6         0 ORDER                 NO PARTIAL SORT
        12         8         7         0 TABLE                 ORDER_LINE
        13         8         7         0                       PRIMARY KEY
        14         8         7         0                       OL_O_ID <= 30
        15         8         7         0                       OL_O_ID >= 11
        16         8         7         0                       OL_D_ID = 9
        17         8         7         0                       OL_W_ID = 2885
17 rows fetched.
```

Next, we time the actual execution of the query when it uses the hint. Example 6-12 shows the output.

Example 6-12 Timing the sample query which is now using MERGE JOIN

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED *)--
                COUNT(DISTINCT S_I_ID)
FROM
                STOCK, ORDER_LINE
WHERE
                (S_W_ID = 2885 AND
                 S_QUANTITY < 18 AND
                 S_I_ID = OL_I_ID AND
                 S_W_ID = OL_W_ID AND
                 OL_D_ID = 9 AND
                 OL_O_ID BETWEEN 11 and 30);
Prepare time 0.0005541 seconds.
Execute time 2.5068240 seconds.
COUNT(DISTINCT S_I_
-------------------
       12
Fetch time 0.0001600 seconds.
1 rows fetched.

Time 2.5075850 seconds.
```

As we can see, execution of the query took 2.5 seconds to complete, which is much longer than running without the hint to use MERGE JOIN. Therefore, in this case, changing the optimizer default execution plan decisions is not a good plan. This is normally the case, but as mentioned previously, in certain cases, using a hint can help.

Other useful admin command commands

Many other useful admin commands can be used to view and analyze current server performance. They are documented and can be searched for in the solidDB Information Center:

`http://publib.boulder.ibm.com/infocenter/soliddb/v6r5/index.jsp`

The most useful commands for performance analysis are as follows:

- `admin command status`

  Use this command to see a quick snapshot of the overall database status. Interesting items here are Cache hit rate to see the effectiveness of the database cache, memory usage to correlate with OS memory status, and total user counts.

- `admin command userlist -l`

  Use this command to get detailed information about each user currently connected to the database, and various associated transaction information.

- `admin command sqllist`

  Use this command to see a list of currently running SQL statements. This command is useful for watching any long running queries. You can correlate the statement ID in this output to the statement ID output from the Monitor facility.

Useful operating system utilities

In addition to using the various tools provided by solidDB to analyze server and application performance, monitoring performance at the operating system level is also important. If the solidDB server process is using too many system resources, that will definitely slow down overall performance. In this section we discuss some of the most important operating system tools and specifically what to look for. These tools are documented in many places and many other books, therefore, we describe only the most useful items. In our examples, we use the Linux operating system.

vmstat

As with any other database management system, monitoring the memory, CPU, and paging statistics of the operating system are important. The vmstat utility is one of the most important utilities for achieving monitoring. Because many problems occur without warning, run vmstat indefinitely with a 2 - 5 second interval, and save the output with a timestamp. Unfortunately vmstat output does not include a timestamp so Example 6-13 provides a sample script for achieving this.

Example 6-13 vmstat with timestamps

```
#!/bin/ksh

 [ 1 ]; do
   date
   vmstat 2 30 | awk '{ if (skipnext == 1) { \
                           skipnext=0 ; \
                           print "<<summary line omitted>>"; } \
                        else \
                           print $0 ; \
                        if (/swpd/) \
                           skipnext=1 }'
done
```

Because the first statistic line dumped out by vmstat is a summary since the last system restart, the awk utility is used to prune that line to ensure that there is no confusion in later analysis of the output. Example 6-14 shows sample vmstat data, using this script, collected during a solidDB benchmark run.

Example 6-14 Sample vmstat output

```
Tue Nov 23 13:04:00 EST 2010
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
<<summary line omitted>>
 0  0 467192  91760   4072 7829664    0    0  2588  2200 12490 24809 18  3 79  0  0
 1  0 467192  93580   4232 7826420    0    0  4707  8830 12982 28081 19  4 77  0  0
 3  0 467192  89616   4408 7825216    0    0  5187  2221 13059 28605 19  4 77  0  0
 1  0 467192  88964   4572 7825052    0    0  7166  2200 13605 32149 20  4 76  0  0
 9  0 467192  86944   4768 7820800    0    0  8140  2234 13843 33824 20  5 75  0  0
 5  0 467192  89488   4928 7817524    1    0  8671  2182 13928 34627 21  5 74  0  0
 9  0 467192  92188   5116 7814252    0    0 10701  2208 14500 37650 21  5 74  0  0
 4  0 467192  92192   5308 7809952    0    0 15434  8825 15716 45398 21  6 73  0  0
 0  0 467192  91004   5468 7805680    0    0 14493  2190 15370 43643 20  6 74  0  0
```

First, look at the `si` and `so` columns. If these values are anything other than 0 or consistently larger than approximately 500 - 1000, then system swapping is occurring. System swapping means that the operating system is running out of physical memory and needs to use the disk as virtual memory. This way is bad for performance and must be avoided. Example 6-14 on page 178, shows that no swapping is occurring. If swapping is occurring, examine solidDB memory usage. If the usage exceeds the total available physical RAM minus approximately 500 MB to 1 GB for operating system and other applications, consider reducing the usage by analyzing and appropriately setting one or more of the following configuration parameters:

- Srv.ProcessMemoryLimit
- MME.ImdbMemoryLimit
- SharedMemoryAccess.MaxSharedMemorySize
- IndexFile.CacheSize

Also in vmstat output, look at the CPU usage. The closer that values of `us` (user) and `sy` (system) columns are to 100, the more saturated the CPU is and the more bottlenecked the system is. If you see this situation and you think you should be able to run more statements, it could be that one or more SQL statements are running for a long time, consuming a lot of CPU usage. In this case, consider using the Monitor facility in conjunction with the **`sqllist admin`** command to identify the suspect query. After the suspect query is found, look at its execution plan to determine whether it is unnecessarily doing a table scan, using the external sorter, or doing some other non-optimal operation.

iostat

For in-memory tables, disk bottlenecks are not an issue as they are with traditional disk-based databases. However, solidDB is capable of disk-based tables also, so iostat is an important operating system monitoring utility. On Linux, **`iostat`** must always be run with the **`-x`** parameter to ensure that the extended disk statistics are gathered. Example 6-15 shows sample **`iostat -x`** output taken from a relatively idle system.

Example 6-15 Sample iostat -x output

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.12    0.00    0.02    0.03    0.00   99.84

Device:         rrqm/s   wrqm/s     r/s     w/s    rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00    43.00    0.00    5.50     0.00   396.00    72.00     0.04    7.27   3.64   2.00
sdb               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
sdc               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
sdd               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
sde               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
```

First, look at is the `%util` column to see what percentage of the disk is being used. As the example shows, the usage is low, which is expected because the machine is mostly idle. If values approach 100% here, that is a good indication of a disk bottleneck.

Next, look at `svctm` column, which lists the average number of milliseconds of time for the device to service an IO request. The lower the better, and anything over 7 or 8 starts to be a concern.

Also consider the `avgqu-sz` column metric, which reports the average queue length sent to the device. You typically want this to be in the single digits. Anything too low in conjunction with high disk usage can mean that the application is flushing to disk too often and not doing enough buffering, thus putting an extra load on the disk.

6.1.2 Tools available in InfoSphere CDC

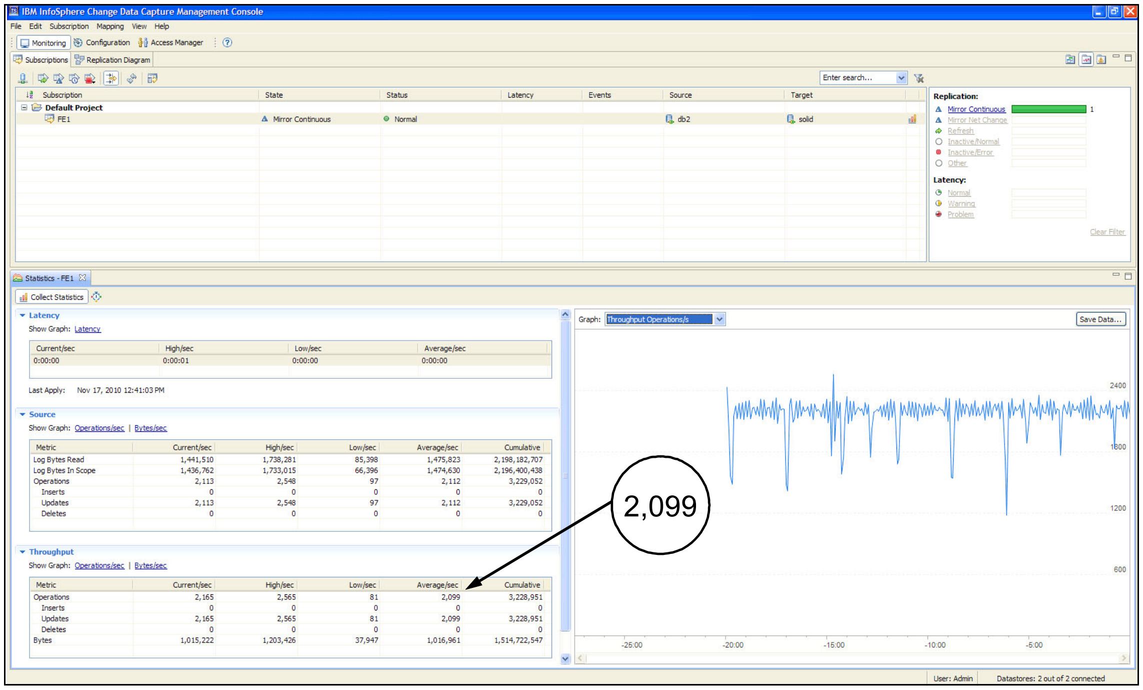
InfoSphere CDC is the component that handles the replication of data between the solidDB front-end database and the back-end database. It is also a stand-alone product with many features that are not necessary in a solidDB Universal Cache configuration. Therefore, in this section we focus on the most useful and easy-to-use performance monitoring tools.

Management Console statistics

The GUI Management Console is capable of capturing replication statistics and presenting them in table and graph form. You must first enable statistics collection on your subscriptions first, as follows:

1. In the Subscriptions tab of the Monitoring view, right-click a subscription and select **Show Statistics**. A tab opens at the bottom half of the management console showing Latency, Source, and Throughput statistics tables.
2. Click **Collect Statistics** at the top left corner of this tab to enable statistics collection. The statistics tables and graphs are populated in real-time.
3. The graphs can be exported to Excel format by clicking **Save Data** at the top right corner of the statistics tab.

Figure 6-3 gives you an idea of the Management Console with a statistics view and a graph of Throughput Operations per second. This figure is from an active workload that was running in solidDB Universal Cache.



Subscription	State	Status	Latency	Events	Source	Target
Default Project						
FE1	Mirror Continuous	Normal			db2	solid

Latency

Show Graph: Latency

Current/sec	High/sec	Low/sec	Average/sec
0:00:00	0:00:01	0:00:00	0:00:00

Last Apply: Nov 17, 2010 12:41:03 PM

Source

Show Graph: Operations/sec | Bytes/sec

Metric	Current/sec	High/sec	Low/sec	Average/sec	Cumulative
Log Bytes Read	1,441,510	1,738,281	85,398	1,475,823	2,198,182,707
Log Bytes In Scope	1,436,762	1,733,015	66,396	1,474,630	2,196,400,438
Operations	2,113	2,548	97	2,112	3,229,052
Inserts	0	0	0	0	0
Updates	2,113	2,548	97	2,112	3,229,052
Deletes	0	0	0	0	0

Throughput

Show Graph: Operations/sec | Bytes/sec

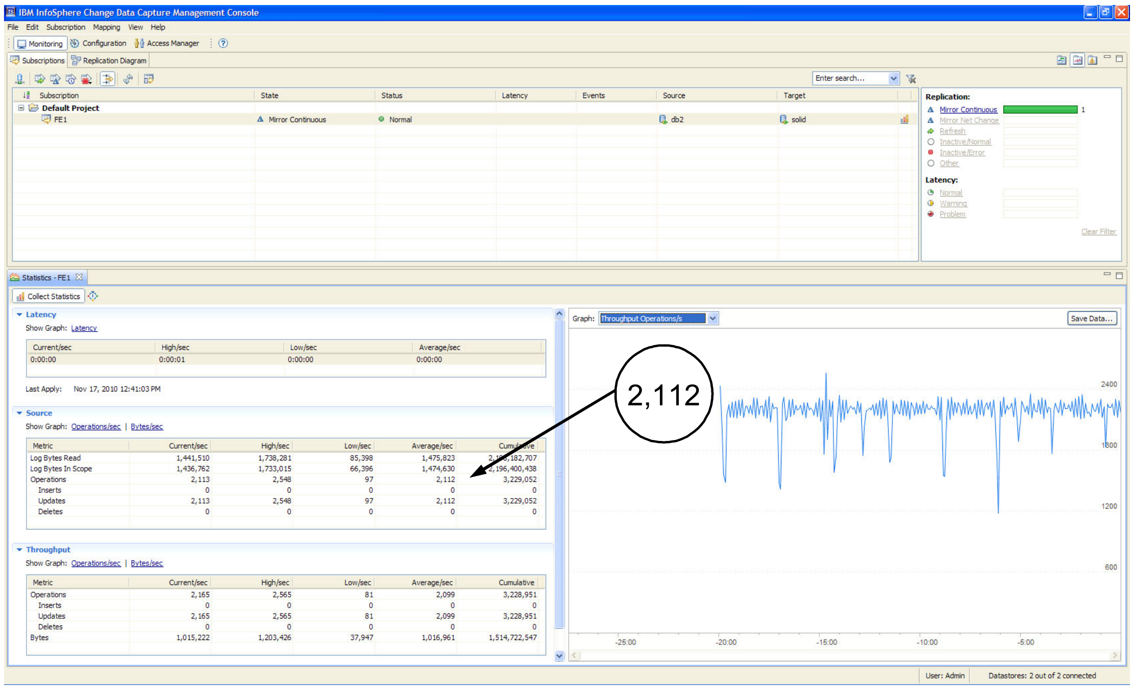
Metric	Current/sec	High/sec	Low/sec	Average/sec	Cumulative
Operations	2,165	2,565	81	2,099	3,228,951
Inserts	0	0	0	0	0
Updates	2,165	2,565	81	2,099	3,228,951
Deletes	0	0	0	0	0
Bytes	1,015,222	1,203,426	37,947	1,016,961	1,514,722,547

Figure 6-3 InfoSphere CDC Management Console Throughput Statistics

As Figure 6-3 shows, we were achieving 2,099 average operations per second. Our workload was not an exhaustive one so this number is by no means representative of the maximum operations per second that InfoSphere CDC is capable of replicating.

Figure 6-4 shows the same subscription and statistics view, except this time the live graph is showing log operations per second.

A number of live graphs are available to view the statistics.



Subscription	State	Status	Latency	Events	Source	Target
Default Project						
FE1	Mirror Continuous	Normal			db2	solid

Latency

Show Graph: Latency

Current/sec	High/sec	Low/sec	Average/sec
0:00:00	0:00:01	0:00:00	0:00:00

Last Apply: Nov 17, 2010 12:41:03 PM

Source

Show Graph: Operations/sec | Bytes/sec

Metric	Current/sec	High/sec	Low/sec	Average/sec	Cumulative
Log Bytes Read	1,441,510	1,738,281	85,398	1,475,823	2,1[illegible],182,707
Log Bytes In Scope	1,436,762	1,733,015	66,396	1,474,630	2,196,400,438
Operations	2,113	2,548	97	2,112	3,229,052
Inserts	0	0	0	0	0
Updates	2,113	2,548	97	2,112	3,229,052
Deletes	0	0	0	0	0

Throughput

Show Graph: Operations/sec | Bytes/sec

Metric	Current/sec	High/sec	Low/sec	Average/sec	Cumulative
Operations	2,165	2,565	81	2,099	3,228,951
Inserts	0	0	0	0	0
Updates	2,165	2,565	81	2,099	3,228,951
Deletes	0	0	0	0	0
Bytes	1,015,222	1,203,426	37,947	1,016,961	1,514,722,547

Figure 6-4 InfoSphere CDC Management Console Log Operations Statistics

6.1.3 Performance troubleshooting from the application perspective

In this section, we address typical performance problems that might be encountered. It is structured to address typical perceived performance problems from the application's perspective, such as the following problems:

- Database response time (latency) is too high
- Database throughput is too low
- Database resource consumption is too high
- Database performance degrades over time
- Database response times are unpredictable
- Special operations take too long

Database response time (latency) is too high

Particularly for online transaction processing (OLTP) applications, database response time is a critical aspect of performance. The quicker SQL statements and transactions can run the better the results.

Ideally, an application has timing capabilities built into it to ensure statements and transactions are running fast enough to meet or exceed service level agreements. This usually means that the operations that are performed during that timing window are significantly more than simply reading the data from a table for example. Therefore, many possible causes for slow response time exist, and many possible cures are available. To fully realize where and how response time can be negatively affected, you should understand the steps that occur during the processing of a typical statement:

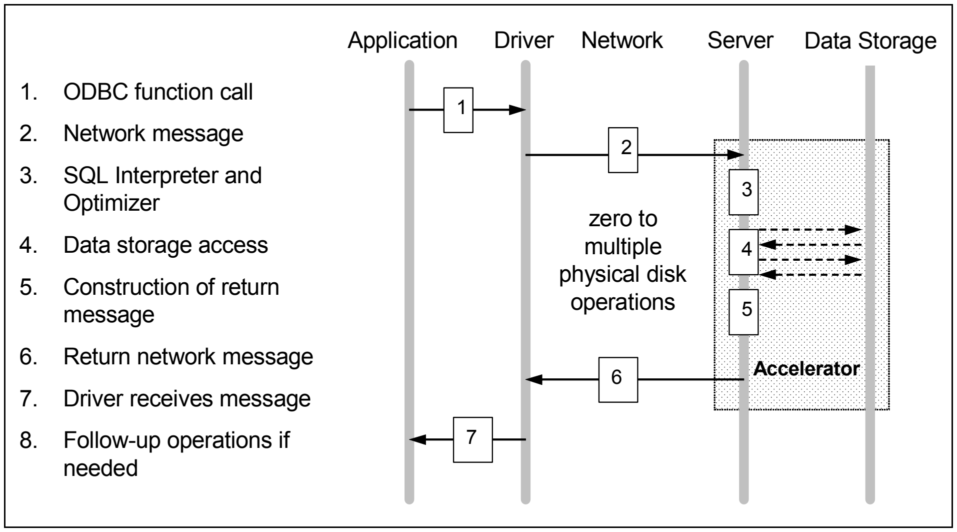
1. The function, for example SQLExecDirect(), is called by the application. Execution moves into the solidDB driver, but is still within the client application process. The driver builds a network message in a memory buffer consisting essentially of the SQL string and some session information.
2. The network message created by the driver is passed through the network to the solidDB server process. This is where network latencies can have a negative effect. Network messages can also be used within the same host machine also (local connections). solidDB supports a direct linking model called Accelerator and a shared memory access model (SMA), where the application contains all the server code also. Therefore, the following conditions are true:
   - No need to copy the host language variables to a buffer
   - No need to send the buffer in a network message
   - No need to process the buffer in the server
   - No context switches, that is, the query is processed within server code using the client application's thread context.

     Using the Accelerator or SMA model essentially removes steps 1, 2, 5, 6 and 7 from the process.
3. The server process captures the network message and starts to process it. The message goes through the SQL Interpreter (which can be bypassed by using prepared statements) and the SQL Optimizer (which can be partly or fully bypassed by Optimizer hints). The solidDB server process can be configured to write an entry to tracing facilities with a millisecond-level timestamp at this point.

4. The query is directed to the appropriate storage engines (MME, disk-based engine, or both) where the latencies can consist of multiple elements:
   - In the disk-based engine, the data can be found in the database cache. If it is, no disk head movements are required. If the data resides outside of cache, disk operations are needed before data can be accessed.
   - in The main-memory engine, the data is always found in memory.
   - Storage algorithms for the main memory engine and disk-based engine differ significantly and naturally have an impact on time spent in this stage.
   - For complicated queries, the latency will be impacted by the optimizer decisions made in step 3, such as the choice of index, join order, sorting decisions, and so forth.
   - For COMMIT operations (either executing COMMIT WORK or SQLTransEnd() by ODBC), a disk operation for the transaction log file is performed every time, unless transaction logging is turned off or relaxed logging is configured.
5. After statement execution is completed, a return message is created within the server process. For INSERT, DELETE, and UPDATE statements, the return message is always a single message that essentially contains success or failure information and a count of the number of rows affected. For SELECT statements, a result set is created inside the server to be retrieved in subsequent phases. Only the first two (configurable) rows are returned to the application in the first message. At this stage, an entry with a timestamp exists, written to the server-side `soltrace.out` file.
6. The network message created by the server is passed back to the driver through the network.
7. The driver captures the message, parses it and fills the appropriate return value variables before returning the function call back to driver application. The real response time calculation should end here.
8. Under strict definition, logical follow-up operations (for example retrieval of subsequent rows in result sets) should be considered as part of a separate latency measurement. In this chapter, however, we accept the situation where several ODBC function calls (for example SQLExecute() and loop of SQLFetch() calls) can be considered as a single operation for measurement.

Figure 6-5 on page 185 shows the eight steps of the statement processing flow. It suggests that latencies at the client depend on network response times and disk response times for all queries where disk activity is needed. Both response times are environment specific and cannot really be made much faster with any solidDB configuration changes. There are, however, ways to set up the solidDB architecture to prevent both network and disk access during regular usage.

Reducing the database latency is essentially all about either reducing times spent within individual steps by design, architectural or configuration changes or possibly eliminating the steps altogether.



Application
Driver
Network
Server
Data Storage
1. ODBC function call
2. Network message
3. SQL Interpreter and Optimizer
4. Data storage access
5. Construction of return message
6. Return network message
7. Driver receives message
8. Follow-up operations if needed
1
2
3
4
5
6
7
zero to multiple physical disk operations
Accelerator

Figure 6-5 Statement processing flow

The importance of the steps depends fully on the type of load going through the steps. Optimization effort is not practical before the load is well understood. Consider the following two examples:

- Example 1: The application tries to insert 1 million rows to a single table without indexes as fast as possible using prepared statements. Steps 1, 2, 3, 4, 5, 6, 7 are all small but they are executed a million times. An architectural change to use the directly linked accelerator or SMA model will help to bypass steps 1, 2, 5, 6, and 7, and will significantly speed up the operation.
- Example 2: The application runs a join across 3 tables of 1 million rows each. The join returns 0 or few rows. Steps 1, 2, 5, 6, and 7 are executed only once and are trivial and fast. Steps 3 and 4 are executed once but are significantly more time consuming than they were in Example 1. Almost all the time will be spent in step 4, and removing steps 1, 2, 5, 6, and 7 does not bring meaningful benefits.

Optimization of simple write operation latencies

By simple write operations, we mean inserts, deletes, or updates to one table (preferably having few indexes) that modify only a small number of rows at a time. All the steps previously described are involved and none of the steps are extensively heavy.

The main performance issues are as follows:

- If there is no need for persistence on disk for all operations, *one* of the following statements is true:
  - There might not be a need to commit every time.
  - Transaction logging could be turned off.
  - Relaxed logging might be acceptable (perhaps in conjunction with HotStandby).
- Simple write operations cause intensive messaging between client and server that can be optimized if use of the Accelerator linking or SMA model is possible.
- In database write operations, finding the location of the row is generally a substantial part of the effort. solidDB's main memory technology enables faster finding of individual rows. Hence, using main memory technology can potentially improve performance with simple write operations. In most practical cases, it will be fully effective only after the need for disk writes at every commit has been removed one way or the other.
- For small statements, running through the SQL interpreter is expensive. Hence, the use of prepared statements will improve performance.
- In all write operations, all indexes must be updated also. The more indexes there are, the heavier the write operations will be. In complicated systems, there is a tendency to have indexes that are never used. The effort involved in validating whether or not all indexes are really necessary can pay off with better performance for write operations.
- For simple write operation latencies, the effect of API selection (ODBC, JDBC, SA) is quite marginal. SA is the fastest in theory but the difference is typically less than 10%.
- In theory, avoiding data type conversions (such as using DATE, TIME, TIMESTAMP, NUMERIC and DECIMAL) can improve performance. However, because of small number of rows affected, this effect is also marginal.

Key diagnostics in simple insertion operations

Many pmon counters can monitor overall throughput: SQL Execute, DBE Insert (or DBE Update, DBE Delete), Log file write, File write, Trans commit, and several HotStandby counters if HotStandby is being used. See “Performance Monitoring (pmon) counters” on page 148 for details about each counter.

For simple insertion latencies, Monitoring and SQL Tracing are the only available diagnostic tools within the solidDB product. See “Using the Monitor facility to monitor SQL statements” on page 163 for more details.

Optimization of simple lookup latencies

In simple lookups, the database executes a query which is expected to return one row, or no rows at all. Also, index selection is considered to be trivial. That is, the where condition is expected to be directly resolvable by the primary key or one of the indexes.

Main memory technology was designed for applications running predominantly simple lookups. With these kinds of applications, performance improvements can be up to ten times better than databases using disk-based algorithms. If available RAM exists, using in-memory tables can be highly beneficial in these circumstances.

Similar to simple inserts, using prepared statements to avoid the SQL Interpreter being used for every statement execution and removing unnecessary network messages and context switches by using directly linked accelerator or SMA mode, can also improve performance.

Almost all discussions about database optimizers are related to bad optimizer decisions. With reasonably simple and optimized lookups in tables with several indexes, it is possible that the time used by the optimizer is substantial enough to be measurable. This time can be removed by using optimizer hints to avoid the optimization process altogether. See “Using optimizer hints” for more information.

Optimization of massive write operation latencies

Massive write operations differ from simple operations essentially by the number or rows being involved in a single operation. Both single update or delete statements affecting huge number of rows or a succession of insert statements executed consecutively are considered massive write operations in this section.

Massive write operations can conceptually have more of a throughput problem than a latency one however, we address them here mostly to aid in understanding and comparing these types of operations to others.

In solidDB’s disk-based tables, the primary key defines the physical order of the data in the disk blocks. In other words, the rows with consecutive primary key values reside next to each other (most likely in the same block) on the disk also. Therefore, by doing massive write operations in primary key order can dramatically reduce disk operations. With disk-based tables this factor is usually the strongest one affecting primary key design.

For example, consider a situation in which the disk block size is 16 KB and row size is 100 bytes, which means that 160 rows can fit in same block. We consider an insert operation where we add 1600 rows. If we can insert these 1600 rows in primary key order, the result is 10 disk block writes in the next checkpoint. If we are inserting the rows in random (with respect to the primary key) order, almost all 1600 rows will access separate disk blocks. Instead of 10 file-write operations, the server could be doing 1600.

When running the massive write operations by executing a statement for every row (that is, not running update or delete that affect millions of rows), using prepared statements is important.

When running massive insertions with strict logging enabled, the size of the commit block is an important factor. If every row is committed, the disk head must be moved for every row. An optimal size for a transaction depends on many factors, such as table and index structures, balances between CPU and disk speed. We suggest some iteration with real hardware and data. Common starting values with typical applications and hardware would be in the range of 2000 - 20000. Fortunately, the performance curve regarding medium level transaction size is reasonably flat. The key is to avoid extremes.

For maximum performance of massive insertions in the client/server model, the solidDB SA API using array insert can have an edge over ODBC or JDBC. This way is mostly based on providing the programmer full control on inserting multiple rows in the same network message. solidDB ODBC and JDBC drivers provide support for bulk operations as defined by the corresponding standards. The implementations, however, are built on calling individual statement executions for every row in the bulk.

Excessive growth of the solidDB Bonsai Tree is the single most common performance problem experienced with disk-based tables. The problem is caused by the database preparing to respond to all queries with data as it was when the transaction started. Because disk-based tables' optimistic locking allows other connections to continue modifying the data, duplicate versions of modified rows are needed. If a transaction lasts infinitely long, the Bonsai Tree grows infinitely large. With massive insertions, the problem can be caused by having one idle transaction open for a long time. The problem is relatively easy to detect and fix by closing the external connection.

Key diagnostics in massive insertions

The key diagnostics to follow are the pmon counters DBE Insert, DBE Delete, DBE Update, Trans commit, SQL Execute, File Write, and Ind nomrg write.

Optimization of complicated query latencies

In certain applications, most queries are in the *complicated query* category. Essentially this statement means that something else is required in addition to simply retrieving the rows, such as the following examples:

- Sorting the data
- Grouping the data
- Joining the data across multiple tables

The more complicated the query is, the more potential execution plans there are available for the optimizer to choose from when running the query. For example, there are many, many potential ways to execute a join of 10 tables.

Query optimization is a field of expertise all its own. solidDB, like all other major RDBMS products, has an optimizer to decide on the execution plan, which is essentially based on estimates of the data. Experience with query optimization for any other product is almost directly applicable when analyzing solidDB query optimization. For applications running complicated queries, preparation for bad optimizer decisions is an important step in system design. Even with low failure rates (say one in ten million), the impact of bad optimizer decisions might well transform a sub-second query to a query that will run for several hours.

The problem with bad optimizer decisions is that they are data specific and superficially random. For the decision, the optimizer selects a subset of randomly selected data from the tables and creates an estimate on the number of rows involved based on the random set. By definition, the random sets are not always similar and it is possible that a random set is sufficiently unlike the real data so as to mislead the optimizer.

With the randomness of the process, fully resolving the problem during the development and testing process is practically impossible. The application needs the capability of detecting the problems when they occur in the production system. When doing so, consider the following examples:

- Detecting unexpectedly long response times can be done fully on the application level or it can be done using solidDB diagnostics such as SQL Trace files or the admin command **`sqllist`**.
- To validate how incorrect the optimizer decision is, the bad execution plan must be captured also by running the EXPLAIN PLAN diagnostic of the query when the performance is bad. Running the query when performance is good does not prove anything. Building a mechanism into an application, which automatically collects EXPLAIN PLAN diagnostics for long lasting queries is suggested, but may not be trivial.
- Almost always, bad execution plans (and even more so, the disastrously bad ones) are combined with an excessive number of full table scans. We show an example in “Example: How to detect full table scans from pmon counters with disk-based tables” on page 190. It describes how to look for patterns in pmon counters to understand when a full table scan might be in progress even without collecting EXPLAIN PLAN output for all queries.

Key diagnostics related to complicated query latencies

Often, complicated and heavy queries are executed concurrently with massive amounts of other (usually well-behaving) load. Then, in addition to executing slowly, they interfere with the other load also. Identifying this situation is not straightforward without application diagnostics. Finding one individual long-lasting query among hundreds of thousands of fast queries usually requires time and effort.

Before starting the task of finding potentially heavy queries executing full table scans, assess whether the perceived problems are likely to be caused by individual heavy queries. There is no exact method for that, but pmon counters Cache Find, DBE Find, and File read can be used to detect exceptionally large low-level operations indicating full table scans. Suggestions for doing this are in “Example: How to detect full table scans from pmon counters with disk-based tables” on page 190.

For finding long lasting queries among a huge mass of well-behaving queries, the following methods are available:

- Use the admin command **`sqllist`** to list all queries currently in the server's memory.
- SQL tracing with time stamps (either admin command **`monitor on`** or admin command **`trace on sql`** produce a potentially very large file containing all SQL executions. Finding the long-lasting ones from the big file might be time consuming but it can be done. See the following sections for more information:
  - “Using the Monitor facility to monitor SQL statements” on page 163
  - “Using the SQL Trace Facility to trace SQL statements” on page 168

After the problematic queries have been found, the execution plan can be validated with the EXPLAIN PLAN utility. See “Statement execution plans” on page 170 for more information.

Example: How to detect full table scans from pmon counters with disk-based tables

In this example, we describe how to detect full table scans from pmon counters with disk-based tables. The pb_demo table has an index on column J but not on column K.

As expected, the execution plans shown in Example 6-16 illustrates an index based search and a full table scan.

Example 6-16 Sample explain plans showing an index scan and a table scan

```
explain plan for select * from pb_demo where j = 324562;
       ID   UNIT_ID    PAR_ID JOIN_PATH UNIT_TYPE          INFO
       --   -------    ------ --------- ---------          ----
        1         1         0         2 JOIN
        2         2         1         0 TABLE              PB_DEMO
        3         2         1         0                    INDEX PB_DEMO_J
        4         2         1         0                    J = 324562
4 rows fetched.

Time 0.03 seconds.
explain plan for select * from pb_demo where k = 324562;
       ID   UNIT_ID    PAR_ID JOIN_PATH UNIT_TYPE          INFO
       --   -------    ------ --------- ---------          ----
        1         1         0         2 JOIN
        2         2         1         0 TABLE              PB_DEMO
        3         2         1         0                    SCAN TABLE
        4         2         1         0                    K = 324562
4 rows fetched.

Time 0.03 seconds.
```

Also, as expected, indexed searching is faster, as shown in Example 6-17.

Example 6-17 Comparing index based search versus table scan based search

```
select * from pb_demo where k = 324562;
        I         J         K TXTDATA1           TXTDATA2
        -         -         - --------           --------
   319995    321229    324562 sample             data
1 rows fetched.

Time 1.30 seconds.
select * from pb_demo where j = 324562;
        I         J         K TXTDATA1           TXTDATA2
        -         -         - --------           --------
   323328    324562    327895 sample             data
1 rows fetched.

Time 0.02 seconds.
```

The pmon counters, also show a distinct pattern. In Example 6-18 the un-indexed search (with admin command **pmon** accounting for the second SQL Execute) was run during the first time slice (the left-most column of numbers), while the indexed search was run during the second last time slice of 31 (third column of numbers from the right). The execute during the last time slice of 13 seconds is the second execution of **pmon** admin command.

Example 6-18 pmon counters for indexed and un-indexed search

```
admin command 'pmon -c';
       RC TEXT
       -- ----
        0 Performance statistics:
        0 Time (sec)                     35     35     30     35     31     13    Total
        0 File read                 :   643      0      0      0      1      0    17473
        0 Cache find                :  2064      0      0      0      4    101 17428110
        0 Cache read                :   643      0      0      0      1      0    17115
        0 SQL execute               :     2      0      0      0      1      1      146
        0 DBE fetch                 :     1      0      0      0      1      0     3908
        0 Index search both         :     0      0      0      0      0      0  1002100
        0 Index search storage      :     1      0      0      0      2      0      213
        0 B-tree node search mismatch :  1957      0      0      0      6      1 29018247
        0 B-tree key read           : 500000      0      0      0      2      0  8476993
```

To find full-table scans, search for the number of SQL Executes being disproportionate to the numbers for File Read, Cache Find, B-Tree node search mismatch and B-tree read operations. We can see in this example that we are doing 643 Cache reads and 2064 Cache finds for 2 SQL Executes. Then a few minutes later we are doing 0 Cache reads and 101 Cache finds for 1 SQL Execute. This is good evidence of a table scan being done in the first execution.

Optimization of large result set latencies

Creating a large result or browsing through it within the server process set is not necessarily an extensively heavy operation. Transferring the data from the server process to the client application through the appropriate driver will, however, consume a significant amount of CPU resources and potentially lead to an exchange of large amount of network messages.

For large result sets, the easiest way to improve performance is to use the directly linked accelerator or SMA options. It removes the need for network messages, copying data from one buffer to another and context switches, almost entirely. There are also key diagnostics that can be used to analyze the latencies of large result sets. They are the pmon counters SQL fetch and SA fetch, as well as fetch termination in the `soltrace.out` file.

When processing large result sets, applications are always consuming some CPU to do something about the data just retrieved. To determine how much of the perceived performance problem is really caused by the database, we suggest the following approach:

- In a client/server architecture, look at how much of the CPU is used by the server process and how much by the application.
- In an accelerator or SMA model, it is occasionally suggested to re-link just to assess database server performance. Also, rerunning the same queries without any application data processing might be an option

The difference between a fully cached disk-based engine and in-memory tables in large result set retrieval is quite minimal. Theoretically, disk-based engines are actually slightly faster.

In large result sets the speed of data types becomes a factor, because the conversion needs to be done for every column in every row to be retrieved. Some data types (such as INTEGER, CHAR, and VARCHAR) require no conversion between the host language and binary format in the database others do (such as NUMERIC, DECIMAL, TIMESTAMP, and DATE).

Database throughput is too low

In some cases, all response times seem adequate but overall throughput is too low. *Throughput* is defined as the count of statement executions or transactions per time unit. In these types of situations, there is always a bottleneck on one or more resource (typically CPU, disk, or network) or some type of lock situation blocking the execution. Often, somewhat unexpected application behavior has been misdiagnosed as a database throughput problem, although the application is basically not pushing the database to its limits. This application behavior can be either the application consuming all of the CPU for processing the data on the application side, for example, or having only application-level locks (or lack of user input) to limit the throughput of the database.

In this section, we focus on what can be done in the database to maximize throughput.

Essentially, database throughput is usually limited by the performance of the CPU, disk, or network throughput. If the limiting resource is already known, focus only on the following principles.

- To optimize for CPU-use:
  - Avoid full table scans with proper index design.
  - With disk-based tables, optimize the IndexFile.BlockSize configuration parameter. Smaller block size tends to lead to less CPU usage with a price of slower checkpoints.
  - Assess whether MME technology can be expected to have an advantage.

  - Use prepared statements when possible.
  - Optimize commit block size.
- To optimize for minimal dependency on disk response times:
  - Use main memory tables when possible.
  - With disk-based tables, make sure your cache size is big enough (optimize the IndexFile.CacheSize configuration parameter).
  - Check whether some compromises on data durability would be acceptable. Consider turning transaction logging off altogether or using relaxed logging mode. These choices can be more acceptable if the HotStandby feature is used or there are other data recovery mechanisms.
- To optimize for the least dependency on network messages:
  - Optimize message filling with configuration parameters.
  - Consider API selection (especially for mass insertion).
  - Consider whether the volume of moved data can be reduced, for example by not always moving all columns.
  - Consider architectural changes (for example, move some functionality either to the same machine to run with local messages or into the same process through stored procedures or accelerator linking or shared memory access model).

In addition to straightforward resource shortages, certain scalability anomalies can limit database performance although the application does not have obvious bottlenecks.

Current releases of solidDB fully support multiple CPUs, which can greatly improve parallel processing within the engine.

solidDB can benefit from multiple physical disks to a certain level. It is beneficial to have separate physical disks for the following files:

- Database file (or files)
- Transaction log file (or files)
- Temporary file (or files) for the external sorter
- Backup file (or files)

For large databases, it is possible to distribute the database to several files residing in separate physical disks. However, solidDB does not have features that enable the forcing of certain tables to certain files or for forcing certain key values to certain files. In most cases, one of the files (commonly the first one) becomes a hot file that is accessed far more often than the other ones.

Analysis of existing systems

When encountering a system with clear database throughput problems it is usually reasonably straightforward to identify the resource bottleneck. In most cases, the limiting resource (CPU usage, disk I/O usage, or network throughput) is heavily exploited, and the exploitation is obviously visible using OS tools. If none of these resources are under heavy use, most likely no performance problem can be resolved by any database-related optimization.

CPU Bottleneck

Identifying CPU as being a bottleneck for database operations is relatively easy. Most operating systems that are supported by solidDB have good tools to monitor the CPU usage of each process. If numbers of solidDB processes are close to 100% utilization, CPU is obviously a bottleneck for database operations.

Note: In certain situations in multi-CPU environments it is possible to have one CPU running at 100% while others are idle. CPU monitoring tools do not always make this situation obvious.

The CPU capacity used by the database process is always caused by some identifiable operation, such as a direct external SQL call or background task. The analysis process is basically as follows:

1. Identify the operation(s) that takes most of the CPU resources
2. Assess whether the operation is really necessary
3. Determine whether CPU usage of the operation can be reduced in one way or another

Identifying the Operations

The admin command **`pmon counters`** give a good overview of what is happening in the database at any given moment. Look for high counter values at the time of high CPU usage. See “Performance Monitoring (pmon) counters” on page 148 for more details.

solidDB has an SQL Tracing feature that prints out all statements being executed into a server level trace file. Two slightly different variants (admin command **`monitor on`** and admin command **`trace on sql`**) have slightly different benefits. The **`monitor on`** command provides compact output but does not show server-end SQL (for example, statements being executed by stored procedures). The **`trace on sql`** command shows server-end SQL but clutters the output by printing each returned row separately. For more details, see the following sections:

- “Using the Monitor facility to monitor SQL statements” on page 163
- “Using the SQL Trace Facility to trace SQL statements” on page 168

You may also map the user thread with high CPU usage from operating system tools such as *top* to an actual user session in the database server with the admin command **`tid`**, or, in recent solidDB versions, the admin command **`report`**.

Unnecessary CPU load can typically be caused by several patterns, for example:

- Applications that are continuously running the same statement through the SQL Interpreter by not using prepared statements through ODBC's SQLPrepare() function or JDBC's PreparedStatement class.
- Applications that are establishing a database connection for each operation and logging out instantly, leading to potentially thousands of logins and logouts every minute

Avoiding these patterns might require extra coding but can be beneficial for performance.

Multiple potential methods are available for the reduction of CPU usage per statement execution. Table 6-13 summarizes many of them and briefly outlines the circumstances where real benefits might exist for implementing the method. Seldom are all of them applicable and practical for one single statement.

Table 6-13 Methods of identifying and reducing CPU usage in statement execution

CPU capacity is used for	How to optimize	When applicable
Running the SQL interpreter	Use SQLPrepare() (ODBC) or PreparedStatement (JDBC)	Similar statement is executed many times. Memory growth acceptable (no tens of thousands of prepared statements simultaneously in memory).
Executing full table scans either in main-memory engine or disk-based cache blocks	Proper index design. Ensure correct optimizer decision.	Queries do not require full table scans by nature.
Converting data types from one format to another	Use data types not needing conversion. Limit the number of rows or columns transferred.	Large result sets are being sorted.

CPU capacity is used for	How to optimize	When applicable
Sorting, by internal sorter and by external sorter	Both internal and external sorter are CPU-intensive operations using different algorithms. For bigger result sets (hundreds of thousands or more rows), the external sorter is more effective (uses less CPU and with normal disk latencies response times are faster).	Large result sets are being sorted.
Aggregate calculation	Consider using faster data types for columns that are used in aggregates. Avoid grouping when internal sorter is needed (i.e. add an index for group columns).	When aggregates over large amounts of data are involved. When sorting is needed for grouping.
Finding individual rows either within Main Memory Table or disk-based table by primary key or by index	Optimize IndexFile.BlockSize with disk-based tables.	Application is running mostly simple lookups.
Creating execution plans by the optimizer	Use optimizer hints.	Beneficial when queries, as such, are simple so that optimization time is comparable to query execution time. Scenario with single table query where multiple indexes are available.

Disk I/O being bottleneck

Traditionally, most database performance tuning has really been about reduction of disk I/O in one way or the other. Avoiding full table scans by successful index designs or making sure that the database cache-hit rate is good has really been about enabling the database to complete the current task without having to move the disk head.

This point is still extremely valid, although with today's technology it has lost some of its importance. Today, in some operating systems, the read caching on the file system level has become good. Although the database process executes file system reads there are no real disk head movements because of advanced file system level caching. However, this does not come without adverse side effects. First, the cache flushes might cause overloaded situations in the physical disk. Second, the disk I/O diagnostics built into the database server lose their relevance because actual disk head movements are no longer visible to the database server process.

The solidDB disk I/O consists of following elements:

- Writing the dirty data (modified areas of cache or MME table data structures) back to appropriate locations in the solidDB database files (`solid.db`) during checkpoint operations
- Reading all the data in main memory tables from the solidDB database file (`solid.db`) to main memory during startup
- Reading data outside cache for queries related to disk-based tables
- Reading contents of disk-based tables not in the cache for creating a backup
- Writing log entries to the end of the transaction log file under regular usage
- Reading log entries from the transaction log file under roll-forward recovery, HotStandby catchup or InfoSphere CDC replication catchup
- Writing the backup database file to the backup directory when performing a backup
- Writing to several, mostly configurable, trace files (`solmsg.out`, `solerror.out`, `soltrace.out`) in the working directory, by default
- Writing and reading the external sort files to the appropriate directory for large sort operations
- Reading the contents of the solid.ini and license file during startup

All these elements can be reduced by some actions, none of which come without a price. Which of the actions have real measurable value or which are practical to implement is specific to the application and environment. See Table 6-14 on page 199 for a summary.

To be able to exploit parallelism and not have server tasks block for disk I/O, be sure you have multiple physical disks. See "Database throughput is too low" on page 193 for more information.

Table 6-14 Methods to reduce disk I/O

Type of file I/O	Methods
Database file	
Writing the dirty data (modified areas of cache or MME table data structures) back to appropriate locations in checkpoints.	Optimize block size with parameter IndexFile.BlockSize. Optimize checkpoint execution by parameter General.CheckpointInterval.
Reading all the data in main memory tables from solidDB database file (solid.db) to main memory in startup.	Optimize MME.RestoreThreads.
Reading data outside cache for queries related to disk-based tables.	Make sure IndexFile.CacheSize is sufficiently large.
Reading contents of disk-based tables not in the cache for creating a backup.	If pmon Db free size is substantial, run reorganize. Check that all indexes are really needed.
Swapping contents of Bonsai Tree between main memory and disk when Bonsai Tree has grown too large in size and cleanup of Bonsai Tree by merge task when it is finally released.	Control size of Bonsai Tree by making sure transactions are closed appropriately.
Transaction logs	
Writing log entries to the end of transaction log file under regular usage.	Consider different block size by parameter Logging.BlockSize. Consider relaxed logging or no logging at all.
Reading log entries from transaction log file under roll-forward recovery, hot standby catchup or CDC replication catchup.	Optimize General.CheckpointInterval and General.MinCheckpointTime to reduce size of roll-forward recovery. Set Logging.FileNameTemplate to refer to different physical disk (to db file and temporary files) to avoid interference with other file operations.

Type of file I/O	Methods
Backup file	
Writing backup database file to backup directory when executing backup.	If pmon Db free size is substantial, run reorganize. Check that all indexes are really needed. Make sure that actual database file and backup file are on different physical disks.
Other files	
Writing to several, mostly configurable, trace files (solmsg.out, solerror.out, soltrace.out) into the working directory.	Avoid SQL Tracing.
Writing and reading the external sort files to appropriate directory for large sort operations.	Disable external sorter by Sorter.SortEnabled; Configure bigger SQL.SortArraySize parameter; Optimize for temporary file disk latencies by Sorter.BlockSize.
Reading contents of solid.ini and license file in the startup.	Minimal impact, therefore optimization is not necessary.

Network bottlenecks

Two possible aspects to network bottlenecks are as follows:

- The overall throughput of the network as measured in bytes/second is simply too low, perhaps even with messages set to have maximum buffer length.
- No real problems exist with the throughput, but the network latencies are unacceptably long. The application’s network message pattern would be based on short messages (for example, the application's database usage consists of simple inserts and simple lookups) and nothing can be done to pack the messages more. In this case, network would be the bottleneck only in the sense of latency, but not in the sense of throughput.

Not much can be done to cope with the network being a bottleneck. The methods are based on the strategies of either making sure that the network is exploited to its maximum or reducing the load caused by the database server in the network.

For maximum network exploitation, the following methods are available:

- Making sure network messages are of optimal size by adjusting the Com.MaxPhysMsgLen parameter.
- Making sure network messages are packed full of data in the following ways:
  - Adjust the Srv.RowsPerMessage and Srv.AdaptiveRowsPerMessage parameters.
  - Consider rewriting intensive write operations by using the SA API and SaArrayInsertion. This way provides the programmer good control with flushing the messages.
  - For insertion, consider using INSERT INTO T VALUES(1,1),(2,2),(3,3) syntax.

For reducing the amount of network traffic, several architectural changes are required. Consider the following options:

- Move part or all of the application to the same machine as solidDB and use local host connections, linked library access direct linking, or shared memory access (SMA) as the connection mechanism.
- Rewrite part or all of the application logic as stored procedures to be executed inside the database server.

As an example, consider the following situation:

- solidDB performance (both latency and throughput) with directly linked accelerator or shared memory access model is more than adequate.
- solidDB performance in client/server model is not good enough. CPU usage of the server is not a problem.

In this situation, server throughput is not a problem. If the problem is network latency and not throughput, overall system throughput can be increased by increasing the number of clients. Latencies for individual clients are not improved but system throughput will scale up with an increased number of clients.

Database resource consumption is too high

In certain cases, performance measurements are extended outside of the typical response time or throughput measurements. Resource consumption, such as items in the following list, are occasionally considered as measurements of performance:

- Size of memory footprint
- CPU load
- Size of disk file(s)
- Amount of disk I/O activity

Occasionally an upper limit exists for the amount of memory available for the database server process. Achieving a low memory footprint is almost always contradictory to meeting goals for fast response times or high throughput. Common ways to reduce memory footprint with solidDB are as follows:

- Using disk-based tables instead of main memory tables.
- Reducing the number of indexes with main memory tables.
- Reducing the size of the database cache for disk-based tables.
- Not using prepared statements excessively. Avoid creating large connection pools that have hundreds of prepared statements per connection because the memory required for the prepared statements alone can be very large.

IBM solidDB built-in diagnostics are good for monitoring the memory footprint (pmon counter *Mem size*) and assessing what the memory growth is based on (pmon counter *Search active*, several MME-specific pmon counters, and statement-specific memory counters in admin command **`report`**). The admin command **`indexusage`** allows assessment of whether the indexes defined in the database are really used or not.

Although disk storage continues to be less expensive, in several situations minimizing disk usage in a database installation is a requirement. solidDB is sold and supported in some real-time environments with limited disk capacity. In bigger systems, the size of files affects duration and management of backups. The solidDB key monitoring features related to disk file size are pmon counters *Db size* and *Db free size*. The server allocates disk blocks from the pool of free blocks with the file (indicated by Db free size) until the pool runs empty. Only after that will the actual file expand. To avoid file fragmentation, solidDB is not regularly releasing disk blocks back to the operating system while online. This can be done by starting solidDB with the startup option *-reorganize*.

The question of whether all the data in the database is really needed is mostly a question for the application. In disk-based tables, the indexes are stored on the disk also. The indexes that are never used will increase the file size, sometimes substantially. The admin command **`indexusage`** can be used to analyze whether or not the indexes have ever been used.

Occasionally, simple minimization of disk I/O activity is considered important although there are no measurable performance effects. This case might be true for example with unconventional mass memory devices such as flash. Regular optimization efforts to improve performance are generally applicable to minimize disk I/O. Pmon counters *File write* and *File read* are good tools for monitoring file system calls from the server process.

Database performance degrades over time

Occasionally, system performance is acceptable when the system is started but either instantly, or after running for a certain period of time, it starts to degrade as illustrated in Figure 6-6. Phenomenon of this type generally do not happen without a reason and seldom fix themselves without some kind of intervention.

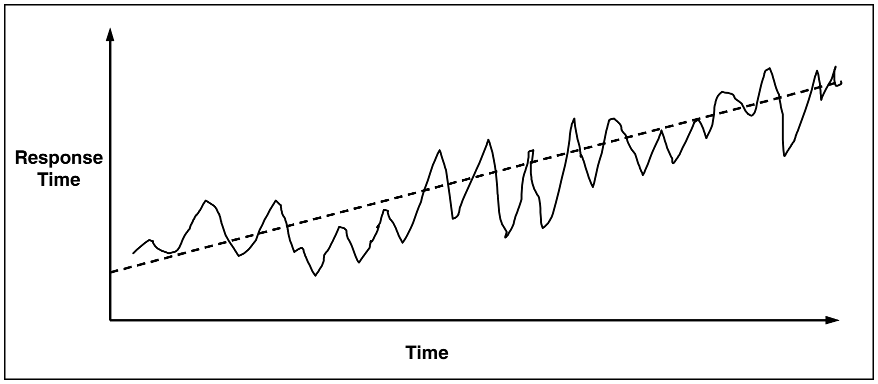


Figure 6-6 Data response time degrades over time

Typical reasons behind this kind of phenomenon are as follows:

- Missing index

  Certain tables grow, either expectedly or unexpectedly, which results in full table scans becoming progressively worse. The impact of managing a progressively larger database starts to reach the application only after a certain amount of time. In addition to thorough testing, it is possible to prepare for the problem by monitoring the row counts in the tables and monitoring the pmon counters (such as Cache find and File read) that are expected to show high values when full table scans take place.

- Memory footprint growth

  The memory footprint of the server process grows steadily for no apparent reason. This growth can lead to OS-level swapping (if enabled by operating system) and eventual emergency shutdown when the OS is no longer able to allocate more memory. Typically, the performance is not really impacted by a big footprint before swapping starts. The effect of swapping is, however, almost always dramatic. Usually the problem can be temporarily resolved by either killing and reopening all database connections or restarting the entire system. Usually these kinds of cures are only short-term solutions. Typical reasons for the problem are caused by unreleased resources in the application code. For example, the application opens cursors or connections but never closes them or never closes transactions.

- Disk fragmentation

  The butterfly-write pattern to the solidDB database file (`solid.db`) can cause disk fragmentation. For example, the disk blocks allocated for the `solid.db` file are progressively more randomly spread across the physical disk. This fragmentation means that response times from the disk become slower over time. Running OS-related defragmenting tools has proven to help with this problem.

Database response times are unpredictable

Relational database technology and related standards differ fundamentally from real-time programming. The standards have few ways to define acceptable response times for database operations. Mostly, the response times are not really mentioned at all and it is possible for the same queries with the same data within the same server to have huge variances in response times.

Varying response times can be explained partially by interference from other concurrent loads in the system. These other loads can be solidDB special operations such as backup, checkpointing, or replication. These are discussed later in further detail.

The interference can, however, be caused by phenomena that are not obviously controllable and might remain unknown even after careful analysis.

Database-specific reasons for sporadic long latencies are as follows:

- Optimizer statistics update

  When data in tables change, the samples must be updated periodically to ensure accurate estimates can be done.

- Merge operation

  This occurs when a sizable Bonsai Tree is cleaned. It can be controlled partly by the General.MergeInterval parameter.

- Massive release of resources

  An example is closing tens of thousands of statements when a pool of connections with many statements each, all complete at once.

In practice, the sporadic latency disturbances are caused by congestion of some system resource caused by other software sharing the same resource. Examples are as follows:

- Physical disk congestion

  Congestion is caused by extensive usage of the same disk by other software. Some database operations (log writes, checkpoint) might be totally blocked because of this kind of disturbance, bringing perceived database performance to a standstill. Also, various virus scanners might interfere with file systems, resulting in serious impact on database performance. solidDB does not contain diagnostics to analyze file system latencies directly. Most operating systems provide sufficient diagnostics to assess levels of disk I/O at the system level and each process. See “iostat” on page 179 for more details.

- Network congestion or failures

  Disturbances and uneven latencies in the network correlate directly with perceived database response times in a client server architecture. The solidDB diagnostic tools on network latencies are somewhat limited (that is, there are no automated latency measurements in regular ODBC, JDBC, and SA messaging). The solidDB ping facility provides a method of measuring network latencies. Go to the following address for more information:

  http://publib.boulder.ibm.com/infocenter/soliddb/v6r5/topic/com.ibm.swg.im.soliddb.admin.doc/doc/the.ping.facility.html

  It is possible to run the ping diagnostic in parallel to the actual application clients and correlate unexpected bad latencies with collected ping results. In addition to the ping diagnostic, be sure to use OS-level network diagnostics.

- CPU being used by other processes

  This means that CPU capacity is not available for solidDB, which does not have any diagnostics to collect absolute CPU usage statistics. Operating systems have ways of collecting CPU usage levels but their level of granularity does not always make a straightforward correlation with bad response times from the database.

Interference from special operations

In some cases, database performance is acceptable under regular load and circumstances. However, in special circumstances, interference of some special task has an excessive negative impact on database response times, throughput, or both. Disturbances of this kind can appear sporadic and unpredictable if the nature of the special tasks is unknown and their occurrence cannot be monitored.

Figure 6-7 illustrates response times in a pattern that shows interference from a special operation.

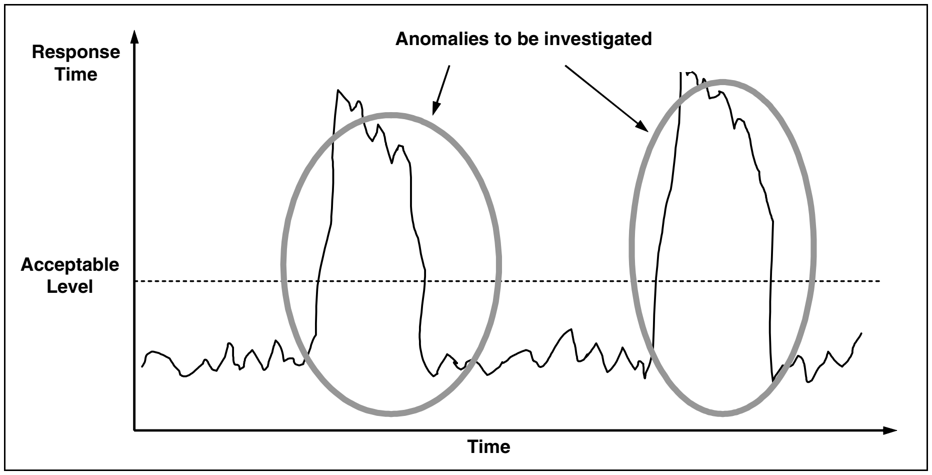


Figure 6-7 Query performance degrades for a period and returns back to normal

The special tasks can be database internal operations:

- Checkpoint
- Backup
- Smartflow (Advanced Replication) replication operations
- HotStandby netcopy or HotStandby catchup
- CDC replication operations
- Heavy DDL operations

Various aspects exist to optimizing the impact of special operations:

- By timing the special operation to occur outside most critical periods.
- By optimizing the size of the special task in one way or the other.
- In some cases, only simultaneous execution of two or more special tasks cause significantly worse interference than any of the special tasks alone.

The special tasks can also be application batch runs or other tasks that are not present all the time. Logically, application special tasks can be treated similarly to database-related special tasks.

Understand and apply the following aspects when optimizing a special task:

- How to control the special task's execution
- How to optimize task execution
- How to monitor task execution times

We examine each special operation in more detail:

- Checkpoint

  A *checkpoint* is a task that is executed automatically by the solidDB server process. The checkpoint task takes care of writing all the changed (*dirty*) blocks in memory back to the disk into the appropriate place in the solidDB database file (or files). The checkpoint task uses CPU to find the dirty blocks inside main memory and relatively heavy disk I/O to write the blocks to disk. Application queries that result in intensive disk I/O are heavily impacted by checkpoints.

  By default, a checkpoint is triggered by the write operations counter. Whenever the value of the General.CheckpointInterval parameter is exceeded, the server performs a checkpoint automatically.

  To avoid checkpoints entirely, set the General.CheckpointInterval parameter to an extremely large value. Do this only if persistence of data is not a concern.

  In heavy write operations, CheckpointInterval can be exceeded before the previous checkpoint completed. To avoid checkpoints being active constantly use the General.MinCheckpointTime parameter.

  Another possibility is to execute a checkpoint programmatically with the admin command **makecp**. This can be advantageous because the application may know about the workload pattern and can pick a more appropriate time to perform a checkpoint operation.

  Consider the following aspects:

  – How to optimize task execution

    By decreasing the General.CheckpointInterval parameter, the size of a checkpoint becomes smaller. The checkpoints, however, become more frequent. It might be that the impact of smaller checkpoints is tolerable; bigger checkpoints disturb the system measurably. In practice, modifications in checkpoint size seldom affect long-term throughput. Theoretically, overall effort in checkpoints is slightly reduced when checkpoints are bigger because of more rows that are possibly hitting the same blocks.

  – How to monitor the execution times

    The `solmsg.out` log file contains information about the start and end of each checkpoint. The last entries of `solmsg.out` can be displayed with the admin command **msgs**.

    The Checkpoint active pmon counter indicates whether a checkpoint is active at that point in time.

  The admin command **`status`** contains information about write counter values after last checkpoint and gives an indirect indication on whether the following checkpoint is about to occur soon or not.

- Backup

  In solidDB, backup is a task executed either automatically after setting the Srv.At configuration parameter, or manually/programmatically by the admin command **`backup`**. During backup, a full copy of the solidDB database file or files (`solid.db`) are created in the backup directory. With disk-based tables this essentially means extensive file reading of the source file and extensive file writing at the target file.

  Executing the task also requires CPU resources, but predominantly causes congestion in the file system. Running backup has a measurable impact on application performance and response times. It does not block the queries and does not cause disastrously bad performance.

  If possible, try to execute backup and application level batch operations consecutively rather than concurrently. This approach is especially true with batch operations that contain long lasting transactions.

  Consider the following aspects:

  – How to control

    Backup process can be started either automatically by timed commands or manually/programmatically by the admin command **`backup`**.

  – How to optimize task execution

    Backup speed depends on the file write speed of the target database. In solidDB, the backup is intended to be stored on different physical devices (to protect against physical failure of the device) than the source database file. Hence, the operation of reading the file and writing the file is not optimized for reading and writing from the same device concurrently.

    The most common reason for long lasting backups is the database file size being bigger than it should be, which can be caused by the following ways:

    • The database file consisting mostly of free blocks. In solidDB, the disk blocks are not automatically returned to the file system when they are no longer needed. To shrink the database file in size, use the reorganize startup parameter.

    • Unnecessary indexes or unnecessary data in the file. In many complicated systems, sizable indexes exist that are never used. Occasionally long backup durations are caused by application level data cleaning tasks having been inactive for long periods of time.

  – How to monitor the execution times

    Backup start and completion messages are displayed in the `solmsg.out` file. There is no direct way of assessing how far the backup task has proceeded. The backup speed can be, however, indirectly calculated by the Backup step pmon counter, based on each backup step passing one disk block of information.

    To determine whether a backup is currently active (to avoid starting concurrent batch operations), use the admin command **`status backup`** or examine the Backup active pmon counter.

- Advanced Replication operations

  Advanced Replication, solidDB's proprietary replication technology, is primarily based on building replication messages inside solidDB's system tables and passing the messages between different solidDB instances. Building and processing the messages can cause database operations related to these system tables.

  Finding rows to be returned from the master database to a replica is a similar operation to an indexed select on a potentially large table. If no rows have been changed, a comparable operation is an indexed select returning no rows. For environments with multiple replicas, the effort must be multiplied by the number of replicas.

  When moving data from replica databases to the master, the additional load in the replica consists of the following actions:

  – Double writing (in addition to actual writes, the data needs to be written in a propagation queue)

  – Reading the propagation queue and writing to system tables for message processing

  – Processing the message in system tables and sending to the master

  The master database processes the message in its own system tables (inserts, selects and deletes) and re-executes the statements in the master (essentially the same operations as in the replica).

  To fully understand Advanced Replication's impact on system performance, you should understand its internal architecture. See the *Advanced Replication User Guide* at the following location for more information:

  `http://publib.boulder.ibm.com/infocenter/soliddb/v6r5/nav/9_1`

Consider the following aspects:

- How to control

  All replication operations (that is, replica databases subscribing to data from master databases and replica databases propagating data to master databases) are controlled by commanding replication to take place with SQL extensions at the replica database. The application has full control of initiation of replication. Heaviness of each operation is naturally caused by the amount of changes in the data between replication operations. Advanced Replication enables defining the data to be replicated by publications. Publication is defined as a collection of one or several SQL Result Sets.

- How to optimize task execution

  The impact of replication depends on the number of rows to be replicated by a single effort. It is possible to make the impact of refreshing an individual publication smaller by breaking publications into smaller pieces as follows:

  - Creating several publications and defining different tables in different publications
  - Refreshing only part of a row (by key value) by individual subscribe effort

  These changes make the performance hits smaller but more frequent. The numbers of rows that are replicated are not affected. Timeliness of replicated data can suffer.

  For systems with a high number of replicas (with high number of tables to be replicated), the polling nature of Advanced Replication start to cause load at the master database. For each replica and each table to be replicated, an operation equivalent to selection can be executed in the master database even if there are no changes in data. Overhead that is caused by this phenomenon can be reduced by adjusting the replication frequency, which results in an adverse affect on the timeliness of the data.

- How to monitor the execution times

  The SQL commands that trigger the replication can be traced in the replica database's `soltrace.out` file like all the other SQL commands. The level of replication activity in both replica and master databases can be measured by a set of pmon counters. Pmon counters starting with the word *Sync* are related to Advanced Replication.

- HotStandby netcopy or HotStandby catchup operations

  In HotStandby, primary and secondary databases must run in Active state where every transaction is synchronously executed at the secondary database also. Before reaching active state, the secondary database must perform the following tasks:

- Receive the database file from the primary database
- Receive the transactions that have been executed since the databases were connected and process them

While the secondary database is in the process of receiving netcopy or processing catchup, it is not able to process requests at all. Also, sending the entire file or just the last transactions will cause load at the primary database and interfere with perceived performance of the primary database also.

HSB Netcopy is an operation quite analogous to a backup. It might lead to heavy disk read I/O at the primary database and interfere heavily with other intensive database file read I/Os.

HSB Catchup is essentially about rerunning the transaction log. If the transactions that were written during the time that the primary and secondary were disconnected exceed the buffer size, log file reads must be executed at the primary database. In reality, this interference seldom causes measurable problems.

Consider the following aspects:

- How to control

  Both HSB netcopy and catchup are operations that are executed by SQL at the primary database. This might be visible to the application or hidden by the Watchdog application or HAC module.

- How to optimize task execution

  The level of interference caused by these operations can be tuned by the HotStandby.CatchupSpeedRate configuration parameter.

- How to monitor the execution times?

  Both catchup and netcopy start and completion messages are logged in the solmsg.out file.

The database states of primary active and secondary active indicate that neither netcopy nor catchup are in progress. The states are visible in the HSB state pmon counter.

- CDC Replication operations

  CDC is a replication technology included as part of the solidDB Universal Cache product to facilitate data transfer and synchronization between solidDB and the back-end database.

  The CDC replication process can cause some load on the databases and that can potentially interfere with perceived application performance.

Essentially, CDC is based on two parts, capture and apply:

- Capture reads log entries from a virtual log table (in solidDB's case, SYS_LOG)
- Apply converts the log entries to SQL statements which get executed in the target database.

The reading from the virtual table SYS_LOG is implemented as reading from an in-memory buffer and, thus, can inflict a significant load. When the connection is broken and log operations occur, a catch-up situation results upon the next enablement of the solidDB InfoSphere CDC replication engine. In that situation, some I/O overhead may be observed until catchup completes.

CDC replication can lead to disastrous impact on performance when throttling is activated. This means that the target database has been too slow to accept applying the captured changes. To avoid the virtual log size growing too big, the server will slow down new write operations. The read-only load is unaffected by the throttling. Because the log reader operates in an asynchronous mode with respect to transactions (that is, it reads only committed transactions), no locking-related performance degradation is possible.

Consider the following aspects:

- How to control

  CDC configuration is performed with dedicated CDC tools. After the replication process is set, it can be controlled (started or stopped) both with GUI tools or with shell level commands, such as **dmts64** or **dmshutdown**. There are no solidDB controls to stop the replication. If there is a need to stop the replication, use the CDC instance configuration tool or the shutdown shell command. Any other way to stop the replication (such as stopping users) may damage the replication setup.

  The throttling is enacted when an in-memory buffer is filled up. The size of the buffer can be controlled with the configuration parameter LogReader.MaxSpace (in number of log records). The factory value is 100000.

- How to optimize

  Relatively few things must be done to optimize replication in the capturing database. In the database that receives the transactions being applied by CDC, you can use regular write optimization options such as:

  - Removal of unnecessary indexes. CDC does not force index structures in source and target databases to be identical.
  - Optimization in transaction logging mode.

  - How to monitor

    The status of throttling may be monitored with the Logreader spm freespc pmon counter. When it reaches zero, the throttling is enabled.

    Other CDC-based activity can be monitored by several other pmon counters. All of them are labeled *Logreader*.

    Write operations executed by CDC when applying the captured data are visible through regular pmon counters (such as SQL Execute and DBE Insert).

- Heavy DDL operations

  Data Definition Language (DDL) operations, such as ALTER TABLE and CREATE INDEX can cause interference as follows:

  - Potential heavy disk activity that interferes with concurrent operations that are related to other tables, which are not directly involved in the DDL operation
  - Blocking the related table (or tables) for the duration of the operation

  Consider the following aspects:

  - How to control

    Running DDL is entirely triggered by executing SQL components external to the database.

  - How to optimize task execution

    DDL operations are potentially heavy. Only limited means are available for optimization:

    - DDL-operations are executed as write operations, which includes a write to the transaction log file also. Disallowing new connections with the **`admin command close`** command, turning logging off (possibly along with other changes in configuration as suggested below), running the DDL, re-enabling logging, and opening the server back up to new connections with the admin command **`open`** can be faster than just running the DDL.
    - With disk-based tables, having a bigger cache (setting IndexFile.CacheSize) size can speed up most DDL operations.
    - With both main memory tables and disk-based tables, the disk block size (the IndexFile.BlockSize configuration parameter) affects checkpoint duration.

    Note: The optimal block size for minimizing the duration of index creation is not necessarily the optimal size for speed of index usage or regular application usage

The ALTER TABLE ADD COLUMN operation along with assigning a default value with the DEFAULT option is heavy for large tables. The operation will require a new value to be added to every row. If the new column will be NULL, there is no need to touch the rows.

– How to monitor the execution times

  SQL execution and transaction commits are usually done in scripting tools, therefore monitoring can be done at that level. The executions are also visible in SQL Trace files.

6.2 Troubleshooting

Two main categories of major problems can occur with any application, including the solidDB server:

- Crashes
- Hangs

This section describes what tools are available to help you try to understand what might have happened so that you can avoid it in the future, or so that you can provide information to IBM support to get to a faster resolution more efficiently.

Crashes

A server *crash* is when a programming error has occurred within the solidDB server resulting in the process abnormally ending. This can happen if an illegal memory location is dereferenced (segmentation fault), a misaligned memory access (bus error), or an illegal instruction is encountered during execution. Abnormal termination can also occur when the solidDB server encounters a condition that it does not expect and has no choice but to shut down the server to avoid any data corruption from occurring. Sometimes, this type is referred to as a *panic* or an *abort*. The following sections several available tools and facilities to help you with problem determination.

Server Stack Traces

As of solidDB version 6.5 fix pack 3, files with the naming convention of `ssstacktrace-<process_id>-<thread_id>.out` are created for each server thread in the solidDB working directory. Example 6-19 on page 215 shows a sample stack traceback file that was generated from sending the solidDB server process a SIGUSR1 signal, which instructs the server to create stack trace files for all currently running threads and then continue normal execution.

Example 6-19 Sample stack traceback file

```
Version info: 6.5.0.3 Build 2010-10-04
Timestamp: Thu Nov 18 14:03:37 2010
Signal name: SIGUSR1
Signal number: 10
Platform: Linux 2.6.18 AMD64 64bit MT
solid[0x846e3d]
/lib64/libpthread.so.0[0x3ac9a0eb10]
/lib64/libpthread.so.0(__nanosleep+0x41)[0x3ac9a0e1c1]
solid(SsThrSleep+0x51)[0x849c01]
solid(sqlsrv_thread_serve+0x12c)[0x4d548c]
solid[0x487487]
solid(ss_svc_main+0x106)[0x8466d6]
solid(ss_main_UTF8+0x102b)[0x48975b]
solid(main+0x45)[0x48ad25]
/lib64/libc.so.6(__libc_start_main+0xf4)[0x3ac8e1d994]
solid[0x484e7a]
Signal details (contents of siginfo_t):
Size of siginfo structure: 128 bytes
0A000000 00000000 00000000 00000000
6D140000 466A0000 00000000 00000000
00000000 00000000 40010000 00000000
482A15C9 3A000000 A0D3FFFF FF7F0000
00000000 00000000 21010000 00000000
10010000 00000000 04000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
Signal #10 (SIGUSR1); si_code: 0 (SI_USER: Sent by kill, sigsend, raise.)
```

In most cases, the actual stack does not mean much to anyone who does not have access to the solidDB source code. In certain cases, function names can give a clue about what was being done at the time the signal was received but usually its a job for IBM support and development personnel.

If a crash occurs and these stack trace files are created, be sure you save them to a safe location so that they can later be sent to IBM for support. The `solerror.out` file might also have important information about why the crash occurred so be sure to save that file also.

Core Files

When the solidDB server receives a signal that causes abnormal termination, UNIX and Linux operating systems have the ability to generate a core file, which saves the contents of memory to a file for later debugging. By default on Linux, the user limit (ulimit) for core files is set to 0, meaning that no core files will be generated. If you have enough disk space to hold a file the size of solidDB's memory use, a good practice is to allow core files to be generated in case a crash occurs. That way you will not be asked to reproduce the problem again after enabling core files.

To increase the core file limit, you may use the **`ulimit -c unlimited`** command in your shell. Any processes started thereafter will be allowed to create core files of any size, so be sure to start the solidDB server after your make this change.

If your server crashes and generates a core file, you will not really be able to get any usable information from it without access to the solidDB source code. Be sure you compress it and save it to a safe location so you can later send to IBM support and development.

Hangs

A *hang* is often a misdiagnosed condition. Most people say a process hangs if it becomes unresponsive. Understanding the difference between a true hang and something running slowly is important.

A true hang is when one or more threads are waiting for a resource that will never become available. Other threads in turn start waiting for resources that the first set of threads are holding, and so on. Usually in these cases, determining what the resource is that everything else is waiting for can be difficult. As a result, you usually have no choice other than to force a restart.

When something is running slowly it may appear to users that it is hung. The reason for running slowly in many cases can be because of some intermittent cause such as a network slowdown, extremely busy disks, a thread or process consuming all of the available CPU resources. If the user is patient enough, the issue causing the slow performance can be alleviated, perhaps by itself, and processing returns to normal. If the user it not patient enough, the server might be terminated unnecessarily resulting in the need for crash recovery and more lost time.

Using stack traces to determine a hang versus running slowly

In “Server Stack Traces” on page 214, we describe what server stack traces are and how they are created. Example 6-19 on page 215 shows a sample stack trace file that was generated when the user issued the following command:

```
kill -USR1 <server_pid>
```

The USR1 signal instructs solidDB to dump stack trace files for all currently running threads. Because stacks show the current paths of execution for a given thread, gathering multiple sets of stack trace files over a period of time can be used to determine whether processing with the solidDB engine is truly hanging as described in “Hangs” on page 216, or is actually running slowly.

Therefore, a strategy to determine whether the server is truly hanging or not is to generate two or three sets of stacks separated by approximately 10 seconds each. Because additional information is appended to the end of existing stack trace files, copying or moving the files between each stack trace file generation is unnecessary.

After you generate a set of stack trace files, analyze the stacks to determine whether they are changing between the three stack-generation operations. If they are not changing, then a true hang situation likely has occurred and the system must be forcefully restarted.

Using top to determine a Hang versus Running Slowly

Most UNIX and Linux operating systems include the **`top`** utility, which is useful for monitoring system and process resource usage.

If you suspect a hang, run **`top`** to determine whether the solidDB server process or application process if using the accelerator or shared memory access libraries is actively or periodically consuming CPU usage. If it does not appear to be consuming any CPU over about a 20-second period, then the server is likely hung and requires a hard restart.

7

Putting solidDB and the Universal Cache to good use

IBM solidDB and IBM solidDB Universal Cache use in-memory database technology, which can provide great throughput and response time advantages over traditional disk-based databases. However, such improvements cannot be achieved with all application types, and therefore not be taken for granted.

Performance of solidDB is sensitive to the overall system topology, application workloads, and database structures. In this book, we describe the usage characteristics, application paradigms, and workload profiles that are well-suited for solidDB in-memory technology.

Key requirements for the effective use of solidDB in-memory database is having enough physical main memory available in the system and willingness to trade these memory resources for the improved speed of database operations.

7.1 solidDB and Universal Cache sweet spots

The solidDB in-memory database is optimized for efficient access to data that is guaranteed to reside in the main memory. A number of access methods allow the in-memory database to outperform a standard disk-based database in a range of workloads, even if the disk-based database caches the relevant data set in main memory buffer pools. However, the advantage of solidDB is not unconditional; it depends on a number of preferred usage patterns: the *sweet spot*s.

A key performance advantage of the solidDB products comes from their ability to bring data closer to the application, all the way to the application memory space. This way significantly reduces the complete access path that the system must execute to serve the data to the application. Figure 7-1 and Figure 7-2 on page 221 illustrate the differences in access paths and data locations between the disk-based databases and the solidDB products.

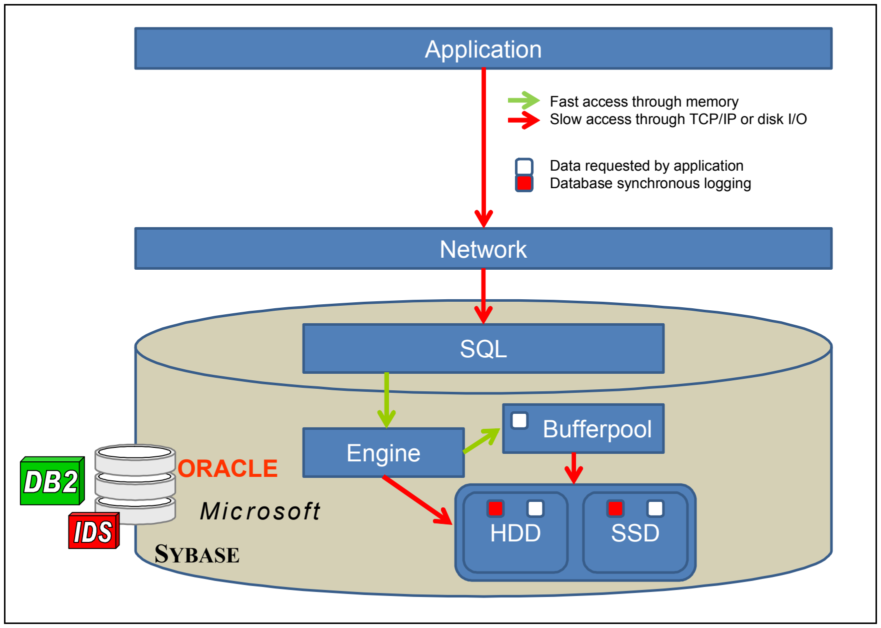
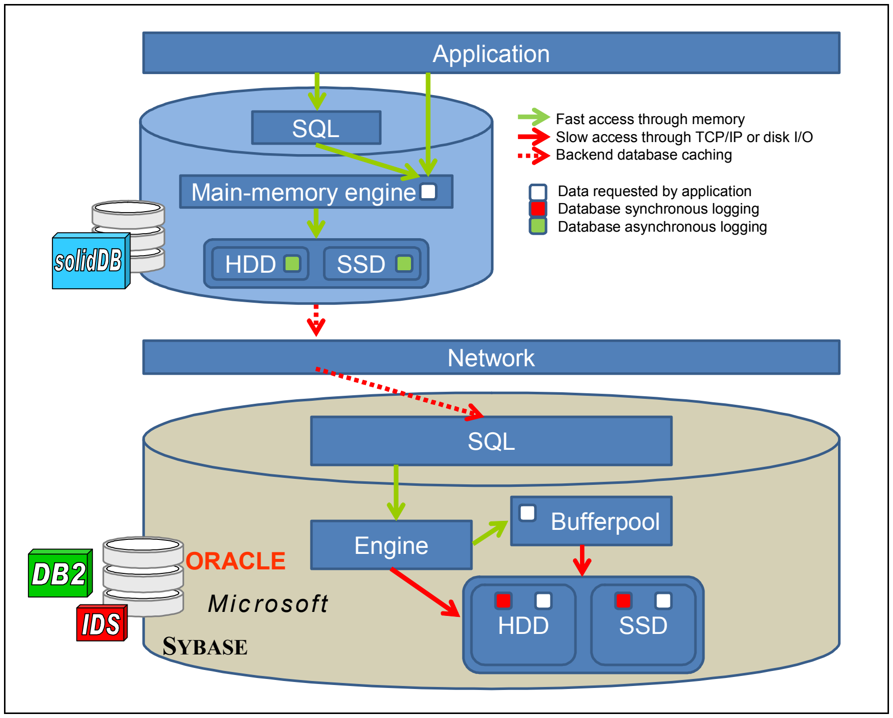

Application
Fast access through memory
Slow access through TCP/IP or disk I/O
Data requested by application
Database synchronous logging
Network
SQL
Engine
Bufferpool
HDD
SSD
DB2
IDS
ORACLE
Microsoft
SYBASE

Figure 7-1 Typical access path for disk-based databases

As shown in Figure 7-1, traditional disk-based databases are often accessed from separate client computers through a network, and data must be read from an external storage device (a hard disk drive or a solid state drive, for instance) before it can be accessed by the application. Although advanced caching algorithms exist to store frequently used data in the database main memory (often referred to as a *buffer pool*), there is no guarantee that the requested data page will be available in the buffer pool at access time, therefore, a disk I/O operation is needed. Moreover, database durability requirements often dictate that log records are synchronously written to the storage device prior to any database updates being committed, thus introducing additional performance impact on the transaction response time seen by the application.

As shown in Figure 7-2, solidDB and solidDB Universal Cache can collocate the data with the application. The combined cost of accessing data from the solidDB in-memory engine collocated with the application is significantly lower than accessing the data from the back-end database server. All expensive access paths (network and synchronous disk access) can be removed.



Application
SQL
Main-memory engine
HDD
SSD
solidDB
Fast access through memory
Slow access through TCP/IP or disk I/O
Backend database caching
Data requested by application
Database synchronous logging
Database asynchronous logging
Network
SQL
Engine
Bufferpool
HDD
SSD
DB2
IDS
ORACLE
Microsoft
SYBASE

Figure 7-2 Typical access path for solidDB and solidDB Universal Cache

Therefore, when we use the term *sweet spot*, we are referring to a collection of usage patterns that can emphasize and maximize the advantages that solidDB can bring to an application. The existence of such sweet spots can be explained by the way solidDB performs certain operations more efficiently (and thus faster) than a regular disk-based database management system (DBMS). However, there are also operations that solidDB is not optimal for, either because of their relative complexity, or because traditional disk-based databases already handle them efficiently.

The more sweet spots you can use, the more the probability that you gain an advantage from using solidDB, such as improved throughput and response times. The total advantage can be difficult to quantify because it depends on many factors. The rest of this section discusses each of these sweet spot aspects in more detail. Some basic guideline expectations are provided also.

7.1.1 Workload characteristics

The solidDB in-memory engine is optimized for workloads generally characterized by the following properties:

- Read dominant
- Unique key lookups
- Simple queries
- Small to medium row sizes
- Relaxed durability is tolerable

Read dominant

IBM solidDB and solidDB Universal Cache provide the greatest performance advantage for workloads where the number of read operations exceeds the number of write operations. As a general guideline, a mix of 80% reads and 20% writes has shown the best performance aspects with in-memory engine. With high-write workloads, aspects that are outside of the core engine might start dominating the throughput (such as transaction logging or synchronization with the back-end database in the Universal Cache case). Hence, the performance advantage over the traditional database systems is likely not significant.

Unique key lookups

Table lookups, especially on unique keys, can be extremely fast in solidDB for two reasons:

- The more obvious reason: The server never has to go to the disk for the data.
- The less obvious reason: The actual storage and index structure are optimized for an in-memory operation.

Access methods and data structures internal to the solidDB in-memory engine can take advantage of memory-resident data, and differ fundamentally from page and index structures that are used by traditional disk-based databases. The in-memory engine can reduce the number of processor operations needed to access the data. A disk-based engine, however, can reduce the number of external storage I/O operations needed to access the data. For example, the solidDB in-memory engine does not implement page-oriented indexes or data structures that would introduce inherent overhead of in-the-page processing.

Simple queries

Another area where solidDB offers an advantage is in improved interaction performance between the application and the database server. Subsequently, the less time a query spends in the server, the more an opportunity exists to realize this “interaction advantage.” Simple queries are fast to execute and thus they amplify the interaction speed advantage. On the contrary, complex queries (involving multi-table joins, non-indexed access, full table scans, aggregates and complex predicates) spend time in the server on scanning, moving, and transforming the data in a way similar to a disk-based system. For example, a full table scan is an operation that a disk-based database is ideally optimized for and therefore, in this case, an in-memory database engine would not bring about improvement.

Small to medium row sizes

Because a table row is a unit of query result processing, long rows induce more data copying and processing than short rows. Short rows reduce the time a query spends in the server and thus amplifies the advantage of fast interactions. Similar to the simple query example, if a large portion of the overall time is spent on an operation (such as memory copying or byte parsing), which is done equally efficiently in solidDB and traditional databases, the expected performance improvement of the in-memory engine becomes negligible compared to the overall execution cost.

Relaxed durability is tolerable

Traditionally, database systems maintain *strict durability*, which means that after a transaction is committed, it can always be recovered from the log files regardless of what happens to the server. Such a durability level requires that the transaction state is written synchronously to an external storage device before the commit call returns. This kind of log operation consumes resources and prolongs response times.

Another option is to run transaction processing in a relaxed durability mode. This mode writes transactions to the log asynchronously, therefore providing a higher level of overall transaction performance. This approach is enabled by solidDB, by default. The compromise is that when the server crashes in a stand-alone environment, some of the latest transactions can be lost. The actual delay of writing the transactions to the log depends heavily on the overall system characteristics.

In solidDB, the durability level can be set globally, per connection, or per transaction. Additional strict durability levels can be obtained through a mixture of relaxed logging and the solidDB HotStandby feature. With HotStandby, the secondary server is accessed through the network, making the transactions durable without the need for synchronous disk access.

7.1.2 System topology characteristics

The essence of any data caching is improving access response times by bringing the data closer to the consumer. In database caching, the solidDB product family can do this by removing overhead insinuated by network access, inter-process communication, and disk I/O.

Co-locate data with the application

In traditional databases using network based client access, there is a constant overhead involved in sending the requests to the DBMS and receiving the results. The cost and response time penalty is the highest with the remote application that accesses the database through the network, most often using the TCP/IP protocol. The reason is because network access involves multiple context switches, additional processing overhead, and the network travel. You can significantly reduce the response time by co-locating your application with the database server, or by bringing the data to the application node (Figure 7-2 on page 221). This way reduces the network travel and the associated overhead.

Link the application with the server

Even with a TCP/IP-based driver and collocated data, inter-process communication between the database server and the application will happen, involving multiple context switches on each interaction between the application and the server. The solidDB product family offers a possibility to avoid such context switches. In addition to the TCP/IP drivers, solidDB provides two drivers that allow the application to be directly linked with the server code and execute the application level requests within the same address space:

- The linked library access (LLA) driver
- The shared memory access (SMA) driver

The LLA driver allows for one linked application per server; the SMA driver allows many applications to access the database server at the same time. With both drivers, the context switches at the application-server interactions are avoided. To take the full advantage of solidDB shared memory data access protocol, use the direct linking methodology, using either LLA or SMA when possible.

7.1.3 Sweet spot summary

The solidDB product family can improve application performance in several ways:

- By removing the need for synchronous disk I/O for both data access and logging
- By removing the need for network or local TCP/IP data access
- By bringing the data into application main memory space for efficient access, using optimized algorithms and data structures

7.2 Return on investment (ROI) considerations

In many cases, the database performance can be directly tied with the revenue generated; the ability to serve more business requests is commonly connected with positive financial consequences of the volume growth. The profit can be increased if the additional transactional volume is at a lower relative cost.

The solidDB product family is well suited for such business growth scenarios because additional software and hardware costs associated with implementing the solidDB accelerating solution are lower than the costs of scaling the traditional database systems.

Moreover, shortening the response times can be easily tied with the organization’s ability to meet various service level agreements, and to gain competitive advantage and increase customer satisfaction. Examples could be the need to validate a mobile phone subscriber and establish the connection in under a few seconds, or the ability to quickly browse a travel company’s inventory based on an online request coming from a search engine.

The following sections provide examples that illustrate how financial gains can be achieved with solidDB product family solutions. The examples are based on a number of assumptions; as much as several assumptions might need to be modified to fit a particular real-life business case and thus individual results may vary, the logic we use to qualify and quantify the ROI is generally applicable.

The example calculations are based on the following common assumptions:

- The setup with the application running against the enterprise database server is profitable. We estimate the revenue of such a solution at twice the cost of the system. This estimate is conservative because most successful IT companies derive revenue many times larger than the cost of production. Larger revenue to IT cost ratio would increase the calculated solidDB ROI.
- Business revenue increases with overall application throughput but the returns for each additional transaction are diminishing. This assumption enforces the essential law of economics stating the marginal returns are always diminishing. Thus, revenue earned per transaction is smaller for solidDB and solidDB Universal Cache solutions because the number of processed transactions increases significantly. Total overall revenue of the solution still increases; a mathematical model is used to predict revenue growth with increased transactional throughput.
- Individual transaction response times have no direct impact on revenue.
- The ROI is calculated as the ratio of revenue increase to the cost increase.

7.2.1 solidDB Universal Cache stimulates business growth

In this example, we detail the potential solidDB Universal Cache ROI with the following scenario:

- solidDB Universal Cache is added to the system without modifying the hardware setup, hence there is no change in any of the HW costs.
- A $150,000 (U.S. dollars, or USD) in application porting costs is added to Universal Cache fixed cost to cover the development work needed to modify the existing application so that it runs against the Universal Cache and the necessary educational expenses; this equals roughly two person-years of skilled labor.
- Software costs are based on the current processor value pricing for the solidDB product family, the current processor value pricing of IBM enterprise disk-based databases, and a standard 20% support renewal charge. A 50% price discount is included.
- Overall costs and revenue are calculated for a three-year period, and all amounts are in thousands of USD.
- A workload that simulates an online retails store order entry system is used (see 7.4.5, “Retail” on page 248). Note that a different workload would result in different Universal Cache relative throughput improvements and thus in a different ROI of the overall solution.

The evaluation is done for two setups, one using commodity hardware and the other using enterprise hardware. The overall solution cost is heavily dependent on the choice between these two.

Case 1: Commodity hardware

The commodity hardware system consists of two IBM xSeries® servers (Xeon E5345 at 2.33 GHz on two chips, eight cores in total. One server is used for the standard disk-based database; the other server is used for remote database clients in the disk-based database stand-alone case, and for the solidDB cache in the Universal Cache case.

Hardware costs are often difficult to estimate because they include fixed cost of procurement amortized over a number of years, and ongoing cost of maintenance, power, cooling, floor space, and so on. Therefore, we are making a simple assumption that the cost of hardware equals the cost of software.

Table 7-1 lists the cost and revenue details for the ROI calculation of the commodity hardware case. The results are summarized in Table 7-2. In the table, K indicates thousand US dollars, and TPS indicates transactions per second.

Table 7-1 Estimated cost and revenue, commodity hardware case

Item	**Data server**	**solidDB Cache**
Fixed cost (HW, SW)	128 K	310 K
Operational cost (HW, SW)	96 K	120 K
Throughput	100 TPS	350 TPS
Cost per transaction	2.24 K	1.23 K

Table 7-2 solidDB Universal Cache ROI summary, commodity hardware case

Solution net earnings / ROI	**43 K / 121%**
Original revenue	448 K
New revenue	697 K
Added cost	206 K
Payback period	29.1 months
Revenue increased	1.6 times
Transaction cost decrease	45%

Case 2: Enterprise hardware

The enterprise hardware system consists of one IBM pSeries® server (P 750 MaxCore mode, 4 chips, 32 cores in total) and four IBM xSeries servers (Xeon E5345 at 2.33 GHz on two chips, eight cores in total. The pSeries server is used for the standard disk-based database. The xSeries servers are used for remote database clients in the disk-based database stand-alone case, and for the solidDB cache in the Universal Cache case.

The hardware costs are much more substantial for enterprise-level servers running in UNIX environments, therefore, we are making a simple assumption that the cost of hardware equals two times the cost of software.

The performance improvement brought by the solidDB Universal Cache is estimated to be about 100%, which is a conservative estimate given the results measured on commodity hardware.

Table 7-3 lists the cost and revenue details for the ROI calculation of the enterprise hardware case. The results are summarized in Table 7-4.

Table 7-3 Estimated cost and revenue, enterprise hardware case

Item	**Data server**	**solidDB Cache**
Fixed cost (HW, SW)	1536 K	1814 K
Operational cost (HW, SW)	1152 K	1248 K
Throughput	750 TPS	1490 TPS
Cost per transaction	3.58 K	2.06 K

Table 7-4 solidDB Universal Cache ROI summary, enterprise hardware case

Solution net earnings / ROI	**1402 K / 475%**
Original revenue	5376 K
New revenue	7152 K
Added cost	374 K
Payback period	6.0 months
Revenue increased	1.3 times
Transaction cost decrease	43%

7.2.2 solidDB server reduces cost of ownership

In this example, we detail the potential solidDB server ROI in the following scenario:

- The complete solution is implement using solidDB server instead of a standard disk-based database server. A single machine running both the solidDB server and the database is used instead of a standard client-server setup.
- Because a new system is being built, no application porting is included in the total cost. An application porting cost similar to what is described in 7.2.1, "solidDB Universal Cache stimulates business growth" on page 226 could be added to convert this scenario to a complete replacement of an existing solution with the solidDB server. A $50,000 educational cost is included in the calculation.
- Software costs are based on the current processor value pricing for the solidDB product family, the current processor value pricing of IBM enterprise disk-based databases, and a standard 20% support renewal charge. A 50% price discount is included.
- The overall costs and revenue are calculated for a three year period, and all amounts are in thousands of U.S. dollars.
- A workload that simulates a mobile carrier Home Location Register system is used (see 7.4.1, "Telecom (TATP)" on page 235). Note that a different workload would result in different Universal Cache relative throughput improvements and thus in different return on investment of the overall solution.
- The evaluation is done using commodity hardware, IBM xSeries servers (Xeon E5410 at 2.33 GHz on two chips, eight cores in total). Again, cost of hardware is estimated to be the same as the cost of software.

Table 7-5 shows the cost and revenue details for the ROI calculation. The results are summarized in Table 7-6 on page 230.

Table 7-5 Estimated cost and revenue, commodity hardware case

Item	Data server	solidDB
Fixed cost (HW, SW)	128 K	114 K
Operational cost (HW, SW)	96 K	48 K
Throughput	100 TPS	300 TPS
Cost per transaction	2.24 K	0.54 K

Table 7-6 solidDB ROI summary, commodity hardware case

Solution net earnings / ROI	**286 K / 415%**
Data server solution revenue	448 K
solidDB server solution revenue	672 K
Cost reduction	62 K
Revenue increased	1.5 times
Transaction cost decrease	76%

7.2.3 solidDB Universal Cache helps leverage enterprise DBMS

In this example, we show that when an application starts using solidDB Universal Cache to accelerate access to business critical data, part of the original workload is naturally off-loaded from the enterprise database server. As a number of database queries are then executed in the solidDB cache, additional processing capacity becomes available in the back-end database. Valuable resources like processor cycles, network bandwidth, and disk I/O bandwidth become available to either increase overall throughput to facilitate business growth, or to be used for other applications. The workload case study in 7.4.5, “Retail” on page 248 further illustrates such resource savings in a real system.

7.2.4 solidDB Universal Cache complements DB2 Connect

In this example, we demonstrate how IBM DB2 Connect™ makes your company’s host data directly available to your personal computer and LAN-based workstations. It connects desktop and palm-top applications to your company’s mainframe. DB2 Connect provides application enablement and robust, highly scalable communication infrastructure for connecting web applications, mobile applications, and applications running on Windows, UNIX, Linux systems to data on IBM z/OS® and IBM AS/400® systems.

IBM solidDB Universal Cache is a natural fit for an identical setup, where core data servers reside on IBM System z® or IBM System i® mainframes but the application accessing the databases is running in a distributed Linux, Unix, Windows environment. Although solidDB server does not run in native z/OS and AS/400 operating systems, caching of mainframe data is supported by the solidDB Universal Cache.

In this scenario, applications always have to access the data remotely through the network. Bringing a subset of the data to solidDB cache running on the Linux, UNIX, or Windows application server machine has great potential to improve overall performance and reduce critical data access response times. Moreover, as a portion of the transactions are now executed in the solidDB cache running off the mainframe, additional mainframe resources become available to either increase overall throughput to facilitate business growth, or to be used for other applications.

7.3 Application classes

IBM solidDB can be easily integrated in a number of application frameworks. These frameworks facilitate application development in Java or C programming languages by abstracting a number of database concepts from the application layer and accessing the database as a generic JDBC or ODBC data source. This simplifies the process of porting an application to use a different database server, because changes are needed only in the database connectivity layer that is managed by the framework, rather than in the application code itself.

An *application server* provides the infrastructure for executing applications that run your business. It insulates the infrastructure from hardware, operating system, and the network. An application server also serves as a platform to develop and deploy your web services and Enterprise JavaBeans (EJBs), and as a transaction and messaging engine while delivering business logic to users on a variety of client devices. The application server acts as middleware between back-end systems and clients. It provides a programming model, an infrastructure framework, and a set of standards for a consistent designed link between them.

Many applications written within these application development paradigms can benefit from low database transactional latency and improved database throughput resulting from the solidDB in-memory database technology and its ability to bring data close to the application, as discussed in 7.1, “solidDB and Universal Cache sweet spots” on page 220.

The solidDB SMA functionality can be used within these frameworks as long as the solidDB server runs on the same computer as the application server. This way provides optimal database access using shared memory only, as illustrated in Figure 7-2 on page 221.

The following sections introduce a set of frameworks that have been tested with IBM solidDB 6.5. More detailed setup instructions are available on the following IBM solidDB Support portal, and samples are provided in the solidDB installation package:

`http://www.ibm.com/software/data/soliddb/support`

7.3.1 WebSphere Application Server

IBM WebSphere Application Server is the IBM runtime environment for Java-based applications. WebSphere Application Server provides the environment to run your solutions and to integrate them with every platform and system as business application services that conform to the service-oriented architecture (SOA) reference architecture.

WebSphere Application Server is a key SOA building block. From the SOA perspective, with WebSphere Application Server you can perform the following functions:

- Build and deploy reusable application services quickly and easily
- Run services in a secure, scalable, highly available environment
- Connect software assets and extend their reach
- Manage applications effortlessly
- Grow as your needs evolve, reusing core skills and assets

WebSphere Application Server is available on a wide range of platforms and in multiple packages to meet specific business needs. By providing the application server that is required to run specific applications, it also serves as the base for other WebSphere products, such as IBM WebSphere Enterprise Service Bus, WebSphere Process Server, WebSphere Portal, and many other IBM software products.

More information about using solidDB with the WebSphere Application Server is provided in the “Configuring WebSphere Application Server with solidDB” article, available on the IBM solidDB Support portal:

`http://www.ibm.com/support/docview.wss?uid=swg21406956`

The article describes how to setup IBM WebSphere Application Server V7.0 with IBM solidDB V6.5 as a data store. A simple application provided with the solidDB package is used as an example. The article assumes basic familiarity with WebSphere Application Server, solidDB, and JDBC.

The task overview is as follows:

1. Start the solidDB server and the WebSphere Application Server.
2. Create solidDB JDBC providers and solidDB data sources.
3. Install the SolidTestEar application.
4. Run the SolidTestEar application.

7.3.2 WebLogic Application Server

WebLogic Application Server is an application server product, owned by the Oracle Corporation, is a part of the Oracle WebLogic Java EE platform product family.

More information about using solidDB with the WebLogic Application Server is provided in the "Configuring WebLogic Server for IBM solidDB" article, available on the IBM solidDB Support portal at:

http://www.ibm.com/support/docview.wss?uid=swg21439319

The article describes how to setup Oracle WebLogic Server with solidDB 6.5 as a data store. A simple WebLogic application provided with the solidDB package is used as an example. The article assumes basic familiarity with the WebLogic Server, solidDB, and JDBC.

The task overview is as follows:

1. Start the WebLogic Server.
2. Start the solidDB server.
3. Create the solidDB JDBC data source.
4. Set up the environment.
5. Deploy and run the sample application.

7.3.3 JBoss Application Server

JBoss Application Server (or JBoss AS) is an open-source Java-based application server product. It was originally developed by JBoss Inc, and is now owned by Red Hat.

More information about using solidDB with the JBoss Application Server is provided in the "Configuring JBoss Application Server for IBM solidDB" article, available on the IBM solidDB Support portal at:

http://www.ibm.com/support/docview.wss?uid=swg21452681

The article describes how to setup JBoss Application Server with solidDB 6.5 as a data store. A simple JBoss application provided with the solidDB package is used as an example. The article assumes basic familiarity with the JBoss Application Server, solidDB, and JDBC.

The task overview is as follows:

1. Set up the environment
2. Deploy the solidDB JDBC data source
3. Start the solidDB server
4. Start the WebLogic Server
5. Deploy and run the sample application

7.3.4 Hibernate

Hibernate is an open source persistence and query framework that provides object-relational mapping of Plain Old Java Objects (POJOs) to relational database tables, and data query and retrieval capabilities. Hibernate enables you to write database applications without writing SQL statements.

The mapping between objects and the solidDB database is facilitated with a solidDB dialect for Hibernate. The dialect enables the Hibernate library to communicate with solidDB. It contains information about the mapping of Java types to SQL types and the functions the solidDB database supports with Hibernate. In general, a Java class maps to a database table and a Java type maps to an SQL data type.

Hibernate eases migration between different databases: you can write an application for a database that will in principle work with all databases supported by Hibernate, that is, with any database that provides a dialect.

More information about using solidDB with Hibernate is provided in the "Hibernate and solidDB" article, available on the IBM solidDB Support portal at:

http://www.ibm.com/support/docview.wss?uid=swg21440246

The article describes how to get started using Hibernate with IBM solidDB. It also includes the solidDB dialect for Hibernate (`SolidSQLDialect.jar`), and instructions on how to build and run a sample application.

The task overview is as follows:

1. Configure your environment
2. Create mappings
3. Start the solidDB server
4. Run the sample application

7.3.5 WebSphere Message Broker

IBM WebSphere Message Broker provides universal connectivity, including web services and any-to-any data transformation. In addition, you can use such products as DataPower® and WebSphere Transformation Extender to extend the capabilities of the core enterprise service bus (ESB) products.

WebSphere Message Broker is a powerful information broker that allows business data, in the form of messages, to flow between disparate applications and across multiple hardware and software platforms. Rules can be applied to the data that is flowing through the message broker to route, store, retrieve, and transform the information. WebSphere Message Broker offers the following features:

- Universal connectivity
- Routing and transforming messages from anywhere, to anywhere
- Simple programming
- Operational management and performance
- Support for adapters and files

WebSphere Message Broker contains a choice of transports that enable secure business to be conducted at any time by providing powerful integration, message, and data transformations in a single place.

Official support for IBM solidDB was introduced in IBM WebSphere Message Broker V7.0.0.1. An example of solidDB working with the WebSphere Message Broker within an IBM financial services framework is presented in 7.4.3, “Banking Payments Framework” on page 243.

7.4 Examining specific industries

In this section we provide detailed discussion of solidDB applicability to workloads, frameworks, and use cases in several industries.

7.4.1 Telecom (TATP)

Telecom applications are prominent candidates for taking advantage of the solidDB sweet spots, especially in the area of service control. Service control applications are those that execute user services in a real-time environment. They usually operate on simple data structures and small amounts of data in each request. The key requirement is low latency, which is typically sub-millisecond. An example is a voice call setup in a mobile network, or a Voice over IP (VOIP) call setup.

The Telecom Application Transaction Processing (TATP) benchmark was built to represent a typical service control application. It originated from a network division of Nokia Corporation in the 1990s, and was at that time called Network Database Benchmark. Eventually, it made its way to the public use and is now available as open source software, in the form of a TATP distribution package[1].

TATP emulates the operations performed on the Home Location Register (HLR) database in a mobile telephone network switch. An example of the network architecture with HLR is shown in Figure 7-3.

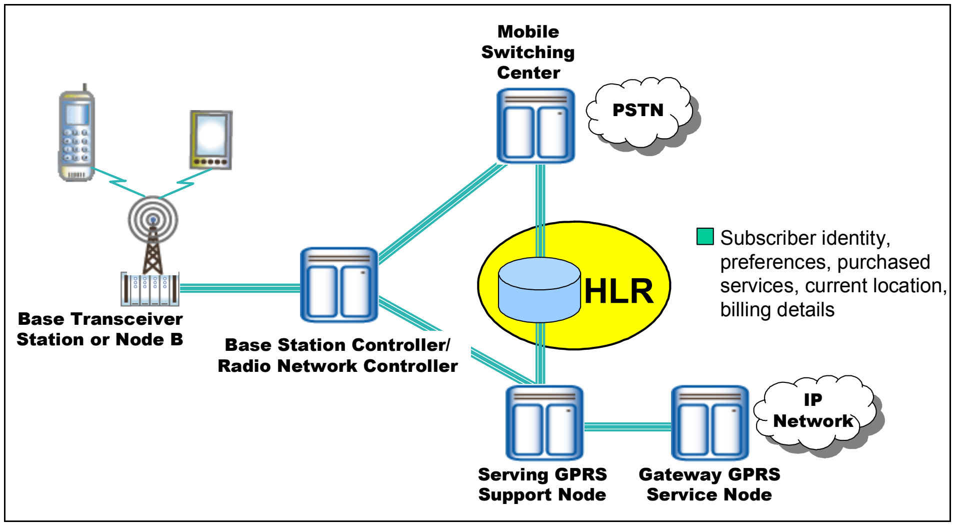

Mobile Switching Center
PSTN
HLR
🟩 Subscriber identity, preferences, purchased services, current location, billing details
Base Transceiver Station or Node B
Base Station Controller/ Radio Network Controller
IP Network
Serving GPRS Support Node
Gateway GPRS Service Node

Figure 7-3 Home Location Register database within a typical service architecture

HLR is a data repository holding essential subscriber information needed to set up a mobile call: the handset ID and subscriber ID (telephone number), the service profile including various access authorizations, service details including the call forwarding information, the current location of the handset, and so on. In TATP, only a subset of data structures and operations is used. The benchmark employs four tables and seven transactions to emulate the HLR load. A standard setup employs a transaction mix including 80% read transactions and 20% write transactions, and generates a load that represents the maximum server throughput.

[1] http://tatpbenchmark.sourceforge.net/

Benchmark description

The four tables used in TATP are as follows:

- Subscriber: basic subscriber information
- Access_info: subscriber's network access validation
- Special_facility: subscriber's service profile
- Call_forwarding: subscriber's call forwarding data

Detailed table descriptions and referential relationships between the four TATP tables are shown in Figure 7-4.

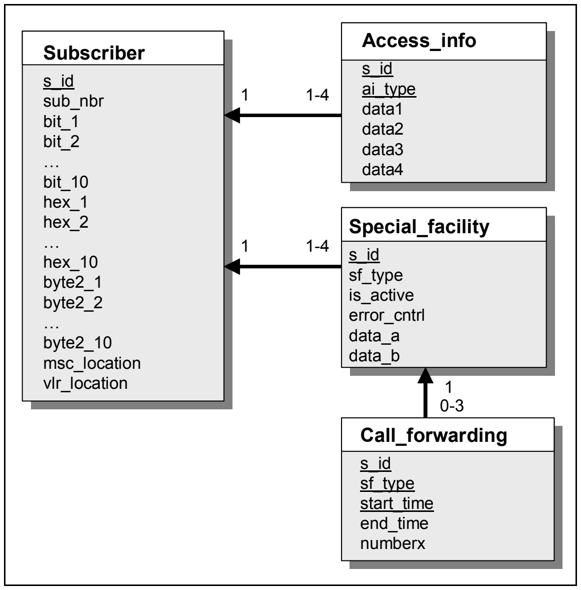


Figure 7-4 TATP schema

The database of a given size (expressed as the number of subscribers) is populated following predefined cardinality rules. For example, for each 10 rows in the Subscriber table, there are 25 rows in the Access_info table. Today, typical test database sizes start from one million subscribers and up. In solidDB, a one-million subscriber database has the physical size of about 1.5 GB.

The standard transaction mix consists of the following transactions (transaction types), with the percentage numbers reflecting the transaction's share in the total load:

- Read transactions (80%)
  - GET_SUBSCRIBER_DATA (35%)

    Look up one row in the SUBSCRIBER table, using the primary key, using one SELECT statement with a long select list.
  - GET_NEW_DESTINATION (10%)

    Retrieve the current call forwarding destination, using a SELECT statement with a two-table join and single-column select list.
  - GET_ACCESS_DATA (35%)

    Retrieve the access validation data, with a single-row SELECT using the primary key, with short select list.
- Write transactions (20%)
  - UPDATE_SUBSCRIBER_DATA (2%)

    Update the service profile data, using two UPDATE statements, with equality conditions on the primary keys.
  - UPDATE_LOCATION (14%)

    Change the location, using one UPDATE based on the primary key.
  - INSERT_CALL_FORWARDING (2%)

    Add new call forwarding information, using two single-table SELECTS and one INSERT.
  - DELETE_CALL_FORWARDING (2%)

    Remove the call forwarding information, using one primary-key based SELECT lookup and one DELETE based on the multi-column primary key.

During the load execution, the transactions to be run are picked up randomly, based on the specified distribution. The search keys are also generated randomly, following one of the two distributions: the uniform distribution across the key range, or a non-uniform one representing discrete hot spots. The hot-spots emulate subscribers that are more active than the others.

Running TATP

With TATP Benchmark distribution package, you can run the workload in various configurations and on separate products. TATP is implemented as a DBMS-agnostic program that can be run against any ODBC-enabled DBMS, over TCP/IP connections, by way of a driver manager and proprietary ODBC drivers. The software has been ported to all major platforms including Windows, Linux, HP-UX, Solaris, and AIX®.

The basic load generator component is called a *TATP client*. A single client represents a single thread of load execution and it establishes a connection to the target database. Clients can be configured to be run as threads in a process or as separate processes. Client processes can be configured to be distributed across several client nodes. In such cases, clients are controlled by a single node that also collects the result data. Additionally, separate clients can run against separate DBMS instances, both collocated or distributed. Moreover, separate clients can be set up to run on separate partitions of a database, residing in separate DBMS instances.

In addition to using driver managers (both for local and remote access), TATP can be built to be linked directly with solidDB drivers. The server code can also be linked with the application using the linked library access (LLA) and shared memory access (SMA) drivers. Also possible is to run the tests in multiple computer nodes in a coordinated way.

When setting up TATP, the basic test configuration unit is a *TATP test session*. A test session is a sequence of *test runs*, possible population steps, intermediate operations, and so on. Each test session definition is captured in a single file called TDF (test definition file). In the TDF, you can specify the following items for each test run:

- Number of clients
- Client distributions
- Database partitioning
- Test timing
- Transaction mixes (can differ from the standard read/write 80/20 mix).

In a session, each test run is a continuous execution of the load with one set of test parameters. A test run timing consists of the ramp-up (warm-up) time and the sampling (test) time. The test result data is collected in the sampling (test) time. Typically test runs are specified in a session to constitute a certain scalability experiment. For example, in a user load scalability (ULS) session, the number of clients is varied from a test run to another. In a read-intensity scalability (RIS) session, the read/write ratio of the transaction mix is varied.

The TATP distribution package contains the source code and binaries and the usage guidance information and sample files.

Collecting test results

TATP offers two ways of retrieving the result data. Summary-type test results are output in the console and TATP log files. More detailed results can be collected into the Test Input and Result Database (TIRDB). TIRDB is a pre-initialized database that is used as a total test respository. When TATP is running, it stores all the relevant data pertaining to test sessions and runs:

- Target hardware and software characteristics and versions
- DBMS configuration information
- Test session description
- Test run description

It also stores the following test results:

- Final throughput values expressed as mean qualified throughput (MQTh) in transactions per second
- Timeline throughput values with 1 second resolution (configurable)
- Response time histogram, for each transaction type
- Final response time values, per transaction type, expressed as 90-percentile response times, in microseconds. The 90-percentile response time is the shortest time that is bigger or equal to the response time of 90% of transactions executed during the test run.

With the existence of TIRDB, no separate result collection step is needed. All the results are stored persistently in an organized manner, and they are ready to use after each session execution. Because of the multidimensional data stored in TIRDB, various analyses across different dimensions (such as software versions, database sizes, test parameter values) are possible at any time.

Hardware and software considerations

The TATP benchmark is often used to measure the evolution in solidDB in-memory database performance between separate releases of the product, and on new hardware platforms. One example of such an effort is the close collaboration between Intel and IBM to showcase database performance improvements of new processor generations. A summary of the analysis and the results are published in support of Intel Xeon 5500 general availability announcement[2].

The following sections describe benchmarks results that compare the solidDB performance against TATP using several hardware and software combinations. The results demonstrate improved throughput between solidDB releases and improved throughput with newer hardware.

For the benchmarking tests, IBM solidDB is running with default settings, such as relaxed durability and read committed isolation level that are chosen for optimal performance. A TATP database simulating one million subscribers is used, with the default workload characteristics of 80% read transactions and 20% write

[2] http://download.intel.com/business/software/testimonials/downloads/xeon5500/ibm.pdf

transactions. To achieve maximum performance, the TATP workload application is accessing the solidDB database using the LLA method. To demonstrate database throughput scalability with the increased workload, the number of application client threads is varied.

Benchmark 1 results

The first benchmark compares performance of two solidDB product releases on two-socket Intel EP class hardware.

Both systems had 18 GB of RAM, two attached solid state disks, and were running SLES 10 SP2 operating system.

- Nehalem-EP: 2 x CPU Intel Xeon 5570 @ 2.93 GHz, total 8 cores
- Westmere-EP: 2 x CPU Intel Xeon 5680 @ 3.33 GHz, total 12 cores

The results are shown in Figure 7-5. The throughput peaks at more than half a million transactions per second.

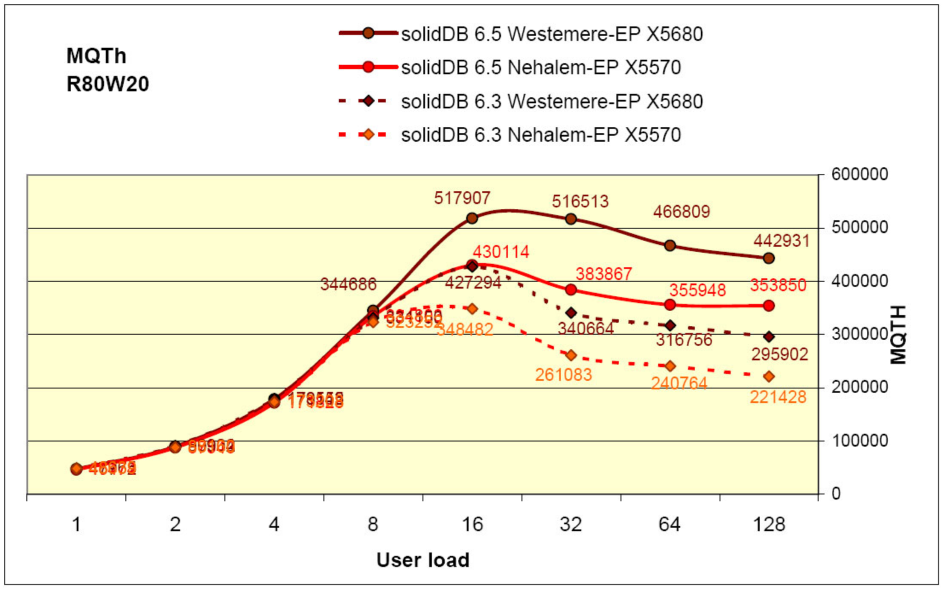


Figure 7-5 TATP Benchmark 1, throughput as a function of client load

Benchmark 2 results

The second benchmark compares performance of two solidDB product releases on four-socket Intel EX class hardware. Four solidDB database instances are running in parallel to take full advantage of the available processing power.

Both systems had 32 GB of RAM, four attached solid state disks, and were running RHEL 5.4 operating system:

- Dunnington Server: 4 x CPU Intel Xeon 7450 @ 2.4 GHz, total 24 cores
- Nehalem-EX Server: 4 x CPU Intel Xeon 7560 @ 2.27 GHz, total 32 cores

The results are shown in Figure 7-6. The throughput peaks at more than a million transactions per second.

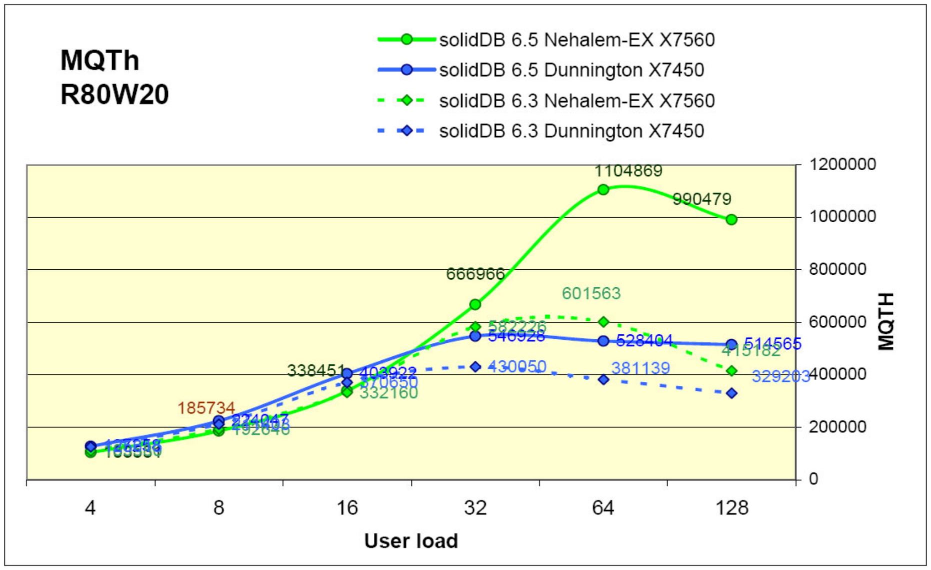


Figure 7-6 TATP Benchmark 2, throughput as a function of client load

7.4.2 Financial services

Financial systems are tremendously data-intensive and rely on speed in trading transactions that can result in huge profits and help exchanges compete and meet client demands. As market volatility continues to increase so does the risk of system failures that can lead to transaction delays, with a direct impact on the global financial system and the businesses and individuals that rely on it.

The broad applicability of solidDB product family in financial services sector is described in 7.4.3, "Banking Payments Framework" on page 243 and 7.4.4, "Securities Exchange Reference Architecture (SXRA)" on page 246. In financial markets, milliseconds can mean the difference between profit and loss. Database performance is often a critical factor in rapidly making a decision to trade, executing that trade, and reporting the trade. It is an even more critical factor considering that trading volumes are growing, and effective trading decisions require more complex analytics of more data.

7.4.3 Banking Payments Framework

In this section, we use a payments processing environment to demonstrate architectural and design patterns that can benefit from the use of both solidDB server and solidDB Universal Cache.

Introduction

The enterprise payments systems are characterized by the fact that all payment methods share a common set of data elements and processing steps. For example, checks and credit cards can seem different on the surface, but they are actually quite similar. Both have a source of funds, a security model, and a clearing and settlement network. All payment methods also require certain services, such as risk and fraud management. However, regardless of the similarities, in most banks the payment systems exist in silos, closely tied to particular retail and wholesale product lines.

The IBM Enterprise Payments Platform (EPP), also known as the Financial Transaction Directory, is a solution that addresses the common problems of the banking industry, simplifying payment systems by providing banks with a single platform to integrate new and existing payments systems regardless of where they reside.

The EPP is based on WebSphere Message Broker and WebSphere Process Server and DB2 and Oracle database technologies and software. With a service orientated architecture, the platform allows other banking applications to use componentized payment services in support of multiple lines of business. For example, with EPP, banks can purchase or develop one service to handle identity verification requirements and reuse it elsewhere to respond faster to changing regulatory requirements.

The solidDB Universal Cache can be integrated into the EPP. By using the solidDB in-memory cache, critical data can be collocated with the application, thus reducing latency by eliminating the network. The use of the in-memory engine can also speed database operations.

A schematic view of the EPP framework with solidDB Universal Cache is shown in Figure 7-7. To collocate data with the application, solidDB server can be implemented with SMA or LLA.

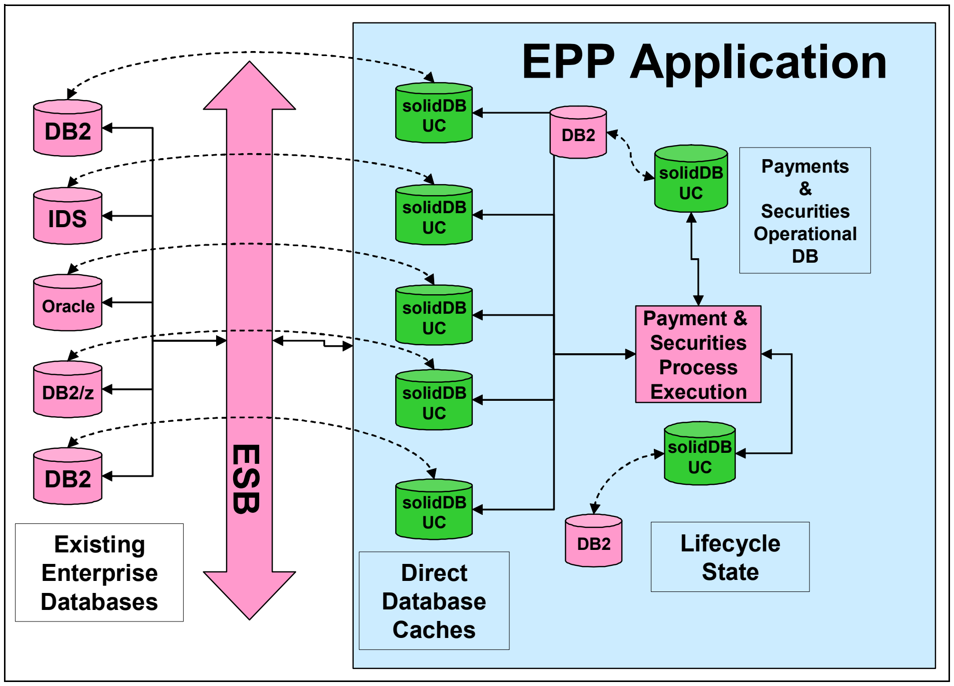


Figure 7-7 IBM Enterprise Payment Platform with solidDB Universal Cache

Benefits of solidDB Universal Cache in a payments system

Some of the key data in a payments system is by nature read-intensive, possibly requiring periodic updates. By caching such data into the in-memory cache, payment systems can benefit greatly from the solidDB Universal Cache.

The following sections describe the key payments system areas and the type of data that is suitable for caching with the solidDB Universal Cache.

Payments life cycle

Payment systems are mostly event-driven and asynchronous in nature. The life cycle of payments or batch of payments is thus typically specified through a state machine paradigm. The state machine definitions are rarely updated but they must be read frequently, returning single or small result sets. By caching the state machine definitions into the in-memory cache, the applications can read (and periodically update) the data with low latency.

Bulking and debulking of payment information

Typically payments are received in bulk. The batch of payments must be processed quickly so that a response on the validity of the batch can be returned to the financial institution's customer. The processing of the batch information involves the parsing and persistence of multiple payment instructions and the validation and construction of various object relationships in the database. This process is read- and write-intensive; the bulking and debulking operation is sensitive to latency. Again, the applications can benefit from caching the batch information data into the solidDB in-memory cache.

Payment operational data

Various decision points in the payments life cycle rely on processing relational transaction data, such as the value of the transaction, the destination of the transaction, the currency of the transaction. This type of data affects the processing path of the payment and hence needs to be accessed frequently with a low latency requirement.

Reference lookups

The payment processing includes several enrichment, legal, and processing data list lookups that are by definition read-intensive. For example, using the in-memory cache can accelerate access to the following types of lookup data:

- Fee management
- Billing
- Security
- Anti money laundering
- Account lookup
- Routing
- Risk scoring
- Negative databases
- Liquidity
- Exchange rates

The solidDB Universal Cache features that are useful to payments systems

In this section, we describe two solidDB Universal Cache features that can bring additional benefit to the payments systems setups: data aging and SQL pass-through.

Data aging

Data aging is the process by which data is removed from the cache but not from the back-end database. Reciprocally, it also enables only specific data to be moved into the cache initially. The main benefit of data aging is the reduction of the amount of memory that the cache requires.

To use the data aging, you must be able to define which data is operational and which data has aged. Within EPP, operational data is typically the data that the application requires to perform various tasks; it could be defined as all data that is associated with payments that have not completed processing. Because EPP is state-driven, the operational data set can be defined as all data that is not in the completed state. Data in the completed state, however, can then be aged, that is, removed from the cache.

SQL pass-through

SQL pass-through is the ability to have some database operations processed in the cache and some routed to the back-end database. Applications that use the cache typically cache a subset of the data required to accelerate the application. However, this way can lead to a situation in which some of the data is unavailable to the application in the cache. In such a case, the cache can determine that the data is not available in the cache and automatically route the request to the back-end database for servicing.

Within EPP, an example of data that does not need to be available in the cache could be the large transmissions of the raw data which is received or sent to the customer or the clearing system. Such data must be recorded but is seldom accessed subsequently.

7.4.4 Securities Exchange Reference Architecture (SXRA)

The IBM Financial Markets Framework enables the creation of highly-available and scalable infrastructures that help reduce costs and complexity, while achieving breakthrough productivity gains. The latest advances from IBM in engineering and software development have produced a Securities Exchange Reference Architecture (SXRA) lab benchmark that features low latency messaging. The solidDB in-memory database technology is used to store business critical data for monitoring purposes.

In a setup with solidDB, SXRA has been shown to achieve over five million orders per second and latency as low as 12 microseconds for the round trip from gateway to matching engine and back using a combination of InfiniBand and 10GbE. The performance results show a 70% increase in orders per second and 40% reduction in latency in comparison to previous results[3].

[3] http://www.a-teamgroup.com/article/ibm-pitches-financial-markets-framework-pre-integrates-hardware-software-for-trading-risk-and-more/

The exchange architecture is shown in Figure 7-8. The order generators (OMS) access the system through a series of gateway machines, WebSphere MQ Low Latency Messaging (LLM) pushes the orders to the parallel matching engines (ME) and stores the trade data in solidDB and a file system (IBM XIV® Storage System).

The matching engines process the orders and make the trades. Individual trade records are received by the LLM and pre-processed before they are written to the database. Simple data capture process is used to extract business critical information (mostly aggregates) from the raw data and record it in solidDB.

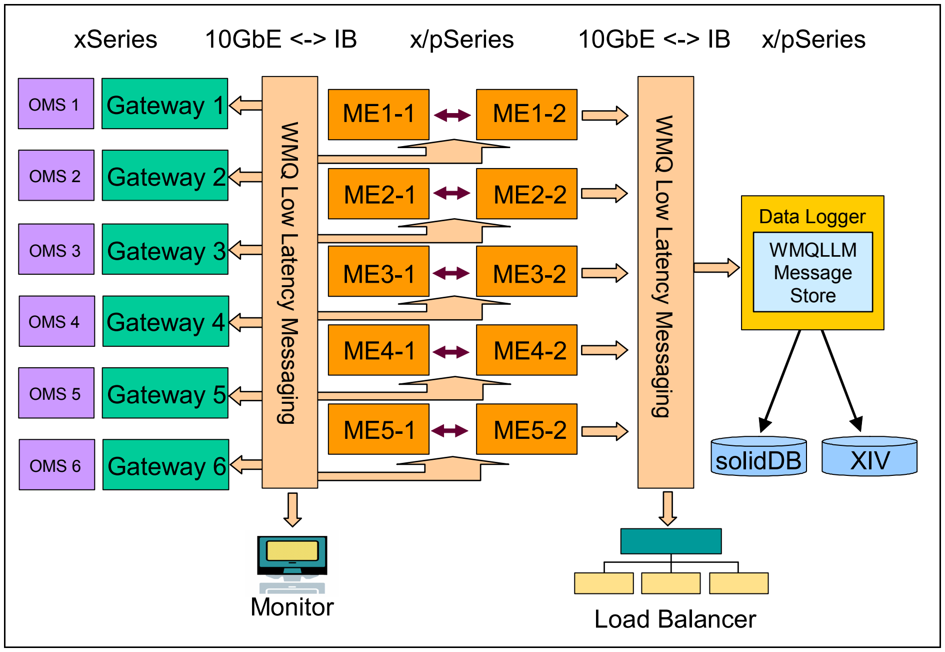


Figure 7-8 IBM Securities Exchange Reference Architecture

The figure also shows solidDB in-memory database used for trade monitoring.

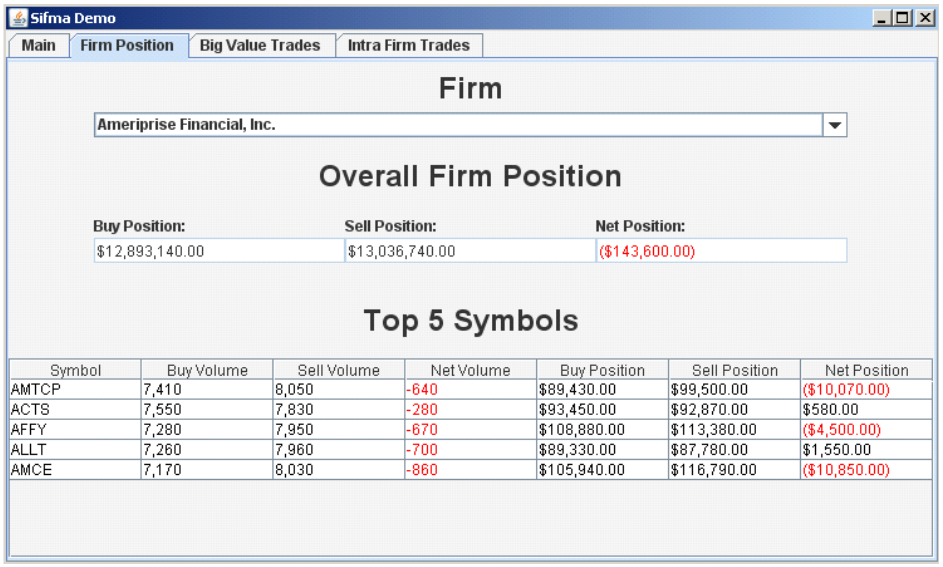
The solidDB database stores data that is used to identify information needed for further business decision-making; raw trade data stream is also stored to a high performance file system. For example, the following types of data can be stored in solidDB:

- Aggregate position of each firm
- Per-symbol position of each firm
- Total number of trades processed in last X minutes
- Total volume of trades processed in last X minutes

- Total value of trades processed in last X minutes
- Trades processed in the last X minutes with a value greater than Y
- Trades processed in the last X minutes with an amount greater than Y
- Trades processed in the last X minutes where the same firm is on opposite sides of the trade, with differing prices

With solidDB, the data can be queried in real time by using the solidDB SQL interface. An example report is shown in Figure 7-9.

Additional applications (such as automated order systems, real-time reporting facilities, or fraud detection) can also read from solidDB, triggering further actions based on the results.



Firm

Ameriprise Financial, Inc.

Overall Firm Position

Buy Position:	Sell Position:	Net Position:
$12,893,140.00	$13,036,740.00	($143,600.00)

Top 5 Symbols

Symbol	Buy Volume	Sell Volume	Net Volume	Buy Position	Sell Position	Net Position
AMTCP	7,410	8,050	-640	$89,430.00	$99,500.00	($10,070.00)
ACTS	7,550	7,830	-280	$93,450.00	$92,870.00	$580.00
AFFY	7,280	7,950	-670	$108,880.00	$113,380.00	($4,500.00)
ALLT	7,260	7,960	-700	$89,330.00	$87,780.00	$1,550.00
AMCE	7,170	8,030	-860	$105,940.00	$116,790.00	($10,850.00)

Figure 7-9 Example of real-time trade analysis data retrieved from solidDB

The figure shows an overall position of an individual trading firm and five most traded stocks.

7.4.5 Retail

In this section, we describe a case study in which a retail oriented workload, called Hybrid Database Transaction Workload (HDTW), is run against IBM solidDB Universal Cache. The HDTW workload and database is inspired by and derived from the TPC-C Benchmark Standard, created by the Transaction Processing Performance Council (TPC). However, the results presented in this section cannot be compared with any official results published by the TPC. The reason is because the HDTW includes significant alterations to the TPC-C Benchmark that have been made to more fully represent the wholesale supplier environment being simulated in this study.

Introduction

Database management systems and hardware are continuously improving to keep up with the ever increasing amounts of data and the need for faster access to it. Certain situations demand even more performance than a traditional disk-based database management system can provide, compelling a search for other technologies that can help.

This case study demonstrates how a medium complexity online transaction processing (OLTP) workload that gets good performance running on a conventional disk-based DBMS can receive a boost in response time and throughput when solidDB Universal Cache is integrated into the setup. We step through the phases involved in the process of designing and applying the in-memory cache into the system. This process includes identifying whether the solidDB Universal Cache can provide a tangible benefit, followed by a workload analysis and a planning phase, and finally an implementation and verification phase. This case study also describes best practices we learned during the implementation. This section assumes that you have basic understanding of the differences between traditional disk-based DBMSs and in-memory databases.

In today’s fast-paced economy, milliseconds can mean the difference between profit and loss. Database performance is often a critical factor in rapidly responding to customer requests, orders, and inquiries.

Database management systems optimize performance through effective use of disk storage and main memory. Because databases typically cannot fit entirely in memory, and memory transfer rates are much faster than disk, disk-based database management systems are designed to optimize I/O efficiency. In effect, disk-based database management systems get better overall performance than disk technology alone would suggest. This result is admirable and works well for many applications. However, considering the high stakes in various industries with respect to performance, IBM has continued to explore innovations that improve performance even further.

As a stand-alone database solution, solidDB can dramatically improve response time, throughput, or both, leading to significant competitive advantage. By presuming that all data fits in main memory, solidDB renders disk transfers moot (except for database recovery purposes). As a result, solidDB can use structures and access methods that optimize memory performance without regard for I/Os, resulting in better response time and higher throughput. Beyond a proven performance record, solidDB provides a comprehensive suite of features, and high reliability.

In this case study, the solidDB in-memory database is used with the solidDB Universal Cache solution. The solidDB database is referred to as the front end and a DB2 database is referred to as the back end. The IBM InfoSphere Change Data Capture (InfoSphere CDC) technology is responsible for replicating the data between the back end and front end to ensure that each database is performing transactions on the same data. Some data, but not all, is present in both the front end and back end. In cases where a transaction cannot be processed in the front end, the solidDB SQL pass-through feature is used to pass the transaction to the back end for processing. Transactions that can be run completely in the front end have the potential to benefit from faster processing in the solidDB database.

The back-end DB2 for Linux, UNIX, and, Windows also provides fast performance and has a complex optimizer that helps to provide exceptional performance even on the most complex queries. The choice between running your workload on solidDB versus on the DB2 database should be based on a number of factors, which this book describes. For example, although solidDB database is also capable of running a large variety of queries, the greatest speed benefit is observed with non-complex queries. Besides query complexity, the size of the tables that the query accesses can matter. A detailed discussion of how we determined which transactions should run in the front end is available in "Preparation and planning" on page 252.

Workload description

The HTDW workload is an OLTP workload which can simulate the functions performed by a wholesale supplier. However, the HDTW workload is not limited to the activity of a particular business segment, rather, it represents any industry that must manage, sell, or distribute a product or service. It is an order-entry processing simulation that can be generalized to just about any business segment.

The workload characteristics are as follows:

- The workload contains a database consisting of nine tables and a varied set of six medium to high complexity transactions that are executed on that database.
- The database schema contains information related to the retail business, such as a number of warehouses in various districts, stock availability for several sold items, customer, and orders.
- The transactions are modeled on the behavior of the retail firm managing warehouse inventories, executing orders, and allowing customers to browse the items.

- A browsing feature allows customers to see availability of a particular item in several stores when browsing, which simulates a standard feature of many existing web stores.
- The database table describing the individual sales items contains a moderately sized large object (LOB) column representing the product image which would, for instance, be displayed on the distributor's web page.
- The database workload is read-dominated, with 88% of operations not modifying the data. The remaining 12% is a combination of updates and inserts, with a small fraction of deletes.

Database schema

The database schema consists of nine tables with simple and complex data types including INTEGER, VARCHAR, TIMESTAMP, and BLOB. Indexes are also created on each table to eliminate table scans. The database tables can be grouped into three categories:

- Warehouse tables contain information about retail warehouses, including the items they carry, and the stock of each item.
- Order tables contain information about new, existing and completed orders issued by customers.
- Customer tables contain information about the store's customers, including their personal information and payment history.

Transaction model

The workload is driven by six transactions that simulate various functions of a retail supplier system. The workload is designed to be read-intensive with both complex and simple look-up queries in addition to simple insert, update, and delete queries. The transactions are as follows:

- The Browse-Item transaction simulates a customer browsing through the store's item catalogue which consists of several select queries.
- The Order-Entry transaction simulates an order being entered into the system resulting in the warehouse's stock being updated to reflect the number of items that the customer ordered. This transaction consists of single and multi-table lookup queries, and simple update and insert statements.
- The Order-Process transaction simulates the processing of an order entered through the order-entry transaction. A random new order is chosen for processing, and the customer who placed the order is charged an amount based on the price of the items and quantity they requested. This transaction consists of single-table select statements and simple updates.
- The Customer-Payment transaction simulates a customer making a payment on an account balance for orders that the customer issued. The customer's balance is adjusted, and the payment is recorded in the system's payment

history. This transaction consists of single-table select statements, and simple inserts and updates.

- The Order-Status transaction simulates the process of a customer checking on the status of the order. This transaction consists of single-table select statements.
- The Stock-Lookup transaction simulates a warehouse manager looking up the items for which stock is low. This transaction consists of single and multi-table select statements.

Each transaction is executed in a specific mix to create a standardized workload that can be easily reproduced.

The workload driver is a multi-process application that runs the transactions in a round-robin fashion to maintain the transaction mix. The application creates 32 processes that connect to the database server and communicate using ODBC calls. The application records a count of the number of successful executions of each transaction and the average response time for each transaction. The transaction counts and the response times are used as the performance metric for this workload. All transactions are sent to the front end; SQL pass-through is used for those transactions that need to be processed on the back end.

Preparation and planning

After the workload was tuned when running on DB2, we started the planning and preparation phase of integrating solidDB Universal Cache. Four key factors must be considered when you are deciding which queries should be run in the front end: table size, read versus write operations, query complexity, and workload data interactions.

Table size

The first factor is that in order for a query to fully run in the front end, all required tables must also be in the front end. If at least one table required by the query is not in the front end, the SQL pass-through feature of solidDB Universal Cache routes the query to the back-end database. When the total size of the database exceeds the amount of RAM available on the solidDB front-end server, you need to identify the tables with size in mind. To do this you can query the database’s system catalog to determine each table’s size on disk, which can be used as a rough guide as to how much memory will be required for the table to exist in solidDB.

Read versus write operations

Another important factor in determining which queries should run in the front end is the type of operation performed by the query. For any table that is cached in the front end, data changes that are made to that table, either in the front end or the back end, must be replicated to the other copy of the table to ensure that all queries are running against the same data regardless of where the query is run. To minimize the amount of replication required, it is advantageous to try to get as many read-only queries running in the front end as possible. To aid in this effort, we analyzed all queries run in the workload and characterized each by what operation was performed against what table or tables.

From this analysis, we could then easily see which tables had mostly reads, suggesting which might be good candidates for placement in the front end. Of the two tables we selected to be cached in the front end, one is purely read-only (the ITEM table) and the other has mostly reads done on it along with a few updates (the STOCK table). The updates to the stock tables are performed in the back end which are then replicated to the front end through InfoSphere CDC.

Accelerating some write operations in the front-end cache also helps the overall workload. Accelerating write operations in the front end requires that any changes made are replicated to the back end. If the amount of data to be replicated is high and continuous, a throttling effect could occur where the solidDB database must wait for InfoSphere CDC to replicate the data. Selecting which write queries run in the front end is an important exercise to do during the planning and design phase. The selection of queries should also be fine-tuned during implementation through trial-and-error experimentation. For this case study, no write statements execute in the front end. For this reason, we were able to disable logging and checkpointing in solidDB because the database could be easily recreated from the back end if required. This provided us with a performance boost of about 5%.

Query complexity

Another factor in choosing the queries to run in the front end is the complexity of the query. The types of queries that experience the best performance improvement in a solidDB database are non-complex, quick-running queries that do not do many table joins and do not return a large result set. Queries that are complex, have many table joins or return a large result set are better suited to run in the back end.

Workload data interactions

The final factor in determining which tables and queries to have in the front end is how the overall workload operates and how the various transactions interact. Changes in the data in the front end or the back end must be replicated to the other database. This replication takes a small amount of time, during which the two databases may not have identical information. You must have a thorough understanding of the transactions and which data they need to access and update.

For example, in this case study, the STOCK table is cached in the front end where all reads to it are performed. However, because updates are made to the STOCK table in the back end which then have to be replicated to the front end, there is a small window where a read to the STOCK table in the front end could get slightly old data. After this situation is identified, an assessment of how it could affect the workload needed to be done. We know that the nature of the updates to the STOCK table are increasing or decreasing the amount of an item's stock. Reads from the STOCK table in the front end are driven by the simulation of a user browsing an item and getting a report of the stock level of that item in different warehouses. Being liberal, if the replication of an update took a maximum of 100 milliseconds (ms) to complete, the 100 ms is then the maximum amount of time that the number of an item's stock can be out of date to the user. We deemed this to be acceptable.

Implementing and running the HDTW application

The implementation of HDTW on solidDB Universal Cache can be subdivided into three main steps: creating and loading the back-end DB2 database, creating the front-end solidDB cache, and configuring InfoSphere CDC to replicate data between the solidDB database and the DB2 database.

Creating the back-end DB2 database

The back-end DB2 database holds all the data of the workload therefore it is created and populated first. An in-house-developed kit is used to mostly automate the process of creating the nine tables and their associated indexes, generating all the random data, loading the data, and applying the various configuration changes. The resulting database is 10 GB in size.

Creating the front-end solidDB cache

The workload application is built to execute transactions on the solidDB front end through ODBC. Within each transaction, the SQL pass-through feature is used to route specific queries to the back end if they are unable to run in the front end. Before the solidDB server is started, SQL pass-through is enabled in the solid.ini file and the remote server information is provided. After the solidDB front-end server is running, a remote server definition is created with the login to the back-end DB2 database. The tables to be cached in the front end are then created in the solidDB front end along with the required indexes. The solidDB Speed Loader (solload) utility is then used to load the two tables with the same table that was loaded into the back-end database.

Configuring InfoSphere CDC to synchronize the data

To replicate data between DB2 and solidDB, separate InfoSphere CDC instances for both the solidDB database and the DB2 database must be created on each machine. After the InfoSphere CDC instance for the solidDB database, the instance for the DB2 database, and the InfoSphere CDC Access Server are running, replication can be configured using the InfoSphere CDC Management Console.

For this case study, because ITEM is a static table, no replication is necessary, therefore, we only need to create one InfoSphere CDC subscription for the STOCK table. The subscription is set to only replicate changes from the back-end DB2 database to the front-end solidDB database. After the subscription is put into a mirroring replication mode (changes are replicated continuously), the workload is ready to be run.

Figure 7-10 illustrates the entire database system.

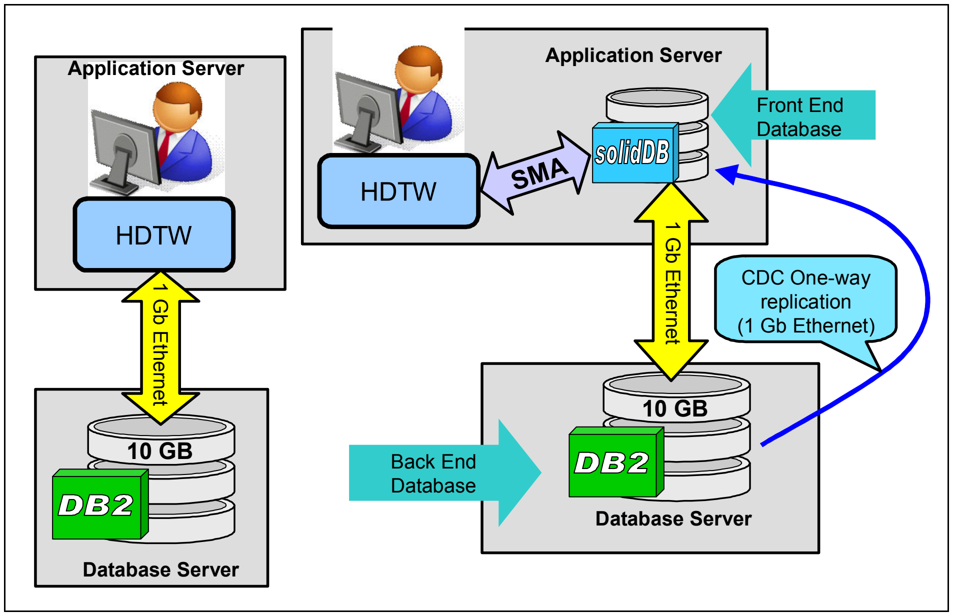


Figure 7-10 Old HDTW topology

In the figure, the application driver is accessing DB2 database directly (left). New HDTW topology with the application driver accessing the data locally through the solidDB shared memory access (right).

Running the HDTW application

The workload application is run through a command-line shell on the solidDB machine. The application creates 32 processes that connect to the solidDB server through SMA and performs database operations through ODBC calls. Whenever a statement is issued for the back-end database, an SQL pass-through connection for that process is used to pass the query to DB2. Because each process runs all transactions, there are at least 32 SQL pass-through connections to DB2.

Hardware and software considerations

Two separate systems are used in this workload to simulate a real environment with the application tier, generally utilizing some application server, which is separate from the database tier. One system contains the front-end solidDB server, and the other contains the back-end DB2 server. The two servers are connected by a private 1 Gb network. The systems had the following hardware and software configurations:

- Front-end solidDB system
  - solidDB Universal Cache 6.5 FP3
  - SUSE Linux Enterprise Server 10 SP1
  - IBM System x3650 (Intel Xeon E5345 – 2.33 GHz, 2-socket, 4-core)
  - 16 GB RAM, 2 GB allocated to solidDB
- Back-end DB2 system
  - DB2 9.7 FP2
  - SUSE Linux Enterprise Server 10 SP1
  - IBM System x3650 (Intel Xeon E5345 – 2.33 GHz, 2-socket, 4-core)
  - Externally attached storage: total of 1.8 TB over 60 disks using GPFS™
  - 16 GB RAM, 10 GB allocated to DB2

Results

The performance of the HDTW workload is measured using two metrics:

- Transaction throughput, measured in transactions per second
- Transaction latency, measured in milliseconds

The throughput is calculated by summing the total number of transactions executed and dividing this sum by the duration of the workload run. The response time is calculated by weighing each transaction response time based on the transaction mix and summing the result. The response time for each transaction is defined as the interval between the time the transaction is started by the application and the time the transaction commit has been executed by the application.

The results presented in Figure 7-11 show a greater than six times increase in throughput and almost five and half times reduction in latency with solidDB Universal Cache with a DB2 back end when compared to a stand-alone DB2 database.

In addition, the introduction of the solidDB Universal Cache reduces the network load by a factor of 10 because most of the read queries in a read dominated workload are now running locally against the solidDB cache and thus do not inflict any load on the network. Moreover, the solution reduces the disk I/O load in the back-end DB2 system by a factor of almost 73 because LOB data representing the product image is stored in the solidDB cache and does not have to be retrieved from the disk at every access time.

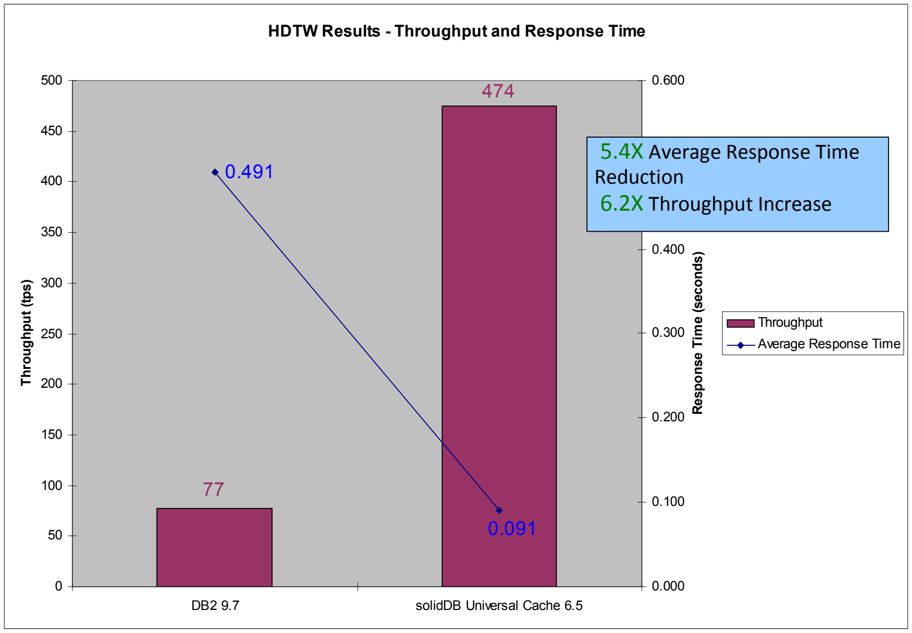


Figure 7-11 Performance impact of solidDB Universal Cache on the HDTW workload simulating an order entry processing system

In summary, this case study demonstrates how a demanding OLTP workload simulating an order-entry system running on solidDB Universal Cache with a DB2 back end favorably compares to the DB2 stand-alone. The Universal Cache solution brings the following increase, average, and reductions:

- 6.2X increase in transaction throughput
- 5.4X average transaction response time improvement
- 73X reduction in DB2 disk I/O
- 10X reduction in network I/O

Scaling out the workload

In time, no more hardware resources are available on the front-end server that can be used to process more transactions. A powerful available option to combat that situation is to add more front-end servers with the application and solidDB to process the workload in parallel. This procedure is commonly referred to as *scaling out*.

As previously discussed, the solidDB Universal Cache solution reduced DB2 disk usage by a factor of 73 and network usage by a factor of 10. This result, in turn, allowed for more processing to be performed on the back-end database server, which allows the addition of multiple solidDB front-end servers, and which results in increased transactions per second being executed.

Moreover, because our solidDB Universal Cache implementation consists of read-only operations in the front-end server, adding multiple front-end servers to the workload becomes easier, because each are basically replicas with the same data.

Figure 7-12 illustrates the architecture of the HDTW workload with five front-end servers.

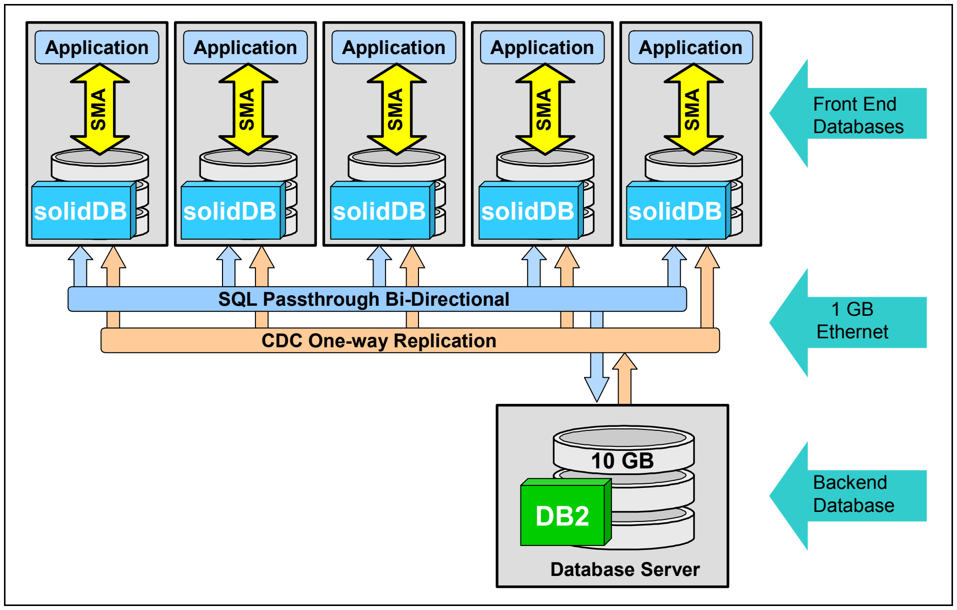


Figure 7-12 The HDTW workload scaled out by adding multiple solidDB front-end servers

Results

Results show that the additional performance gained with the addition of front-end server to the benchmark, scales performance linearly. Figure 7-13 shows that with five front-end servers, 2173 transactions per second were achieved which is 28 times the performance of a stand-alone DB2 configuration.

These results show that for minimal effort, throughput can be greatly increased by scaling out with multiple front-end servers.

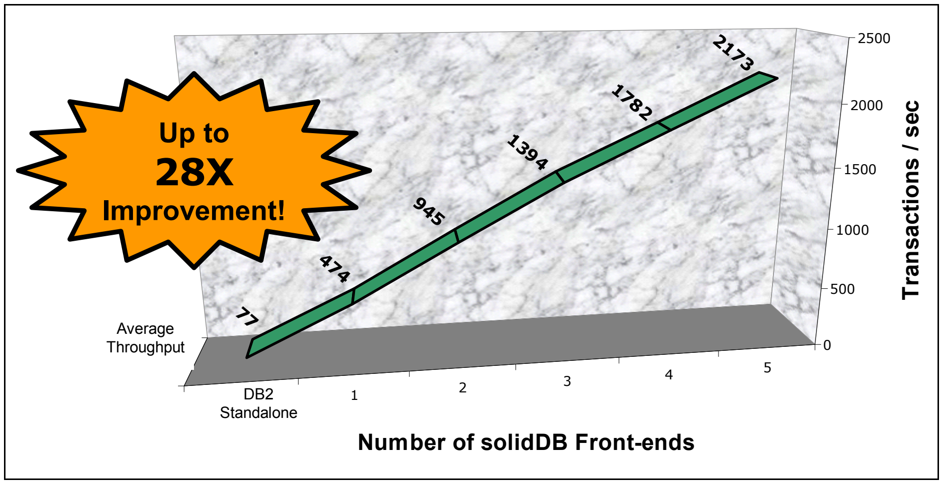


Figure 7-13 Performance scales linearly for each additional front-end server added

In summary, by adding multiple front-end solidDB caches, throughput can be scaled linearly. The throughput increases are as follows:

- 6.2X with 1 front-end cache
- 12.3X with 2 front-end caches
- 18.1X with 3 front-end caches
- 23.1X with 4 front-end caches
- 28.2X with 5 front-end caches

7.4.6 Online travel industry

Essentially, the three major roles in the online travel-reservation industry are travel suppliers, travel agents, and travel distributors:

- *Travel suppliers* are the hotels, airlines, and other companies that own the inventory being sold online. Travel suppliers typically have their own database on their own servers to store this inventory. They can sell that inventory through their own website, through travel agents, or through travel distributors. Several examples of travel suppliers include Hyatt, Choice Hotels International, and Air Canada.
- *Travel agents* make up the main distribution channel through which travel suppliers sell their inventory. Websites such as Expedia and Kayak are considered online travel agents. Travel agents connect to the servers of travel suppliers or travel distributors to retrieve inventory. Many online travel agents also maintain a cache of inventory on their own servers, but this cache has to be updated regularly from the source. Some examples of travel agents include Expedia, Inc., Travelocity, and Flight Center.
- *Travel distributors* are companies that centralize the inventory of multiple travel suppliers and supply that inventory to travel agents. Global Distributed System (GDS) companies such as Travelport Inc., and Amadeus install systems on travel supplier and travel agent servers, which then connect to a GDS server to facilitate data exchange. Some examples of travel distributors include Travelport Inc., Amadeus, and Sabre.

Clearly, the ability to effectively synchronize inventory data between the suppliers, agents, and distributors, and present it quickly to the online customer through a web interface is paramount to this industry. Again, bringing data closer to the user reduces the transaction times and improves the customer experience. Suppliers who are able to serve data faster can generally be listed higher in the search results, and data not retrieved within a preset time interval is often ignored.

Online Flight Reservation Workload

The Online Flight Reservation Workload is a custom-built workload that simulates an application load similar to the airline Computer Reservation System (CRS). Results are measured and presented using travel industry standard Passenger Name Records (PNRs).

PNRs contain the itinerary for a passenger, or group of passengers travelling together. The format and content of a PNR was defined by the International Air Transport Association (IATA) to standardize reservation information that is exchanged by airlines when passengers use more than one airline to get to their destination. Typically, when a passenger books an itinerary, the travel agent or travel website user will create a PNR in the CRS it uses. If the booking is made directly with an airline, the PNR can also be in the database of the airline's CRS. This use case simulates an airline CRS.

Although PNRs were originally introduced for air travel, they are now also being used for bookings of hotels, car rental, railways, and so on.

Database schema and workload

The database in the workload consists of 25 tables that make up all the information in a PNR. Those 25 tables can be grouped into three major types:

- *Historical passenger tables* contain historical PNR information about passengers. These tables record personal information supplied by passengers, special service requests (SSR) made (for example, special meals), tickets issued to the passengers, travel documents supplied by the passengers, and flight segments booked.
- *New passenger tables* contain PNR information about passengers for flights that have not occurred yet. These tables record information about SSRs, tickets, travel documents, and flight seats.
- *Reference tables* contain reference information that is not related to the passenger. This includes fares, flight segment information, and group PNR information.

Most of the tables accessed have indexes on the columns that are used in the queries of the workload.

The workload driver is a multi-threaded Java application that randomly executes a configurable number of use cases. There are two general types of use cases: PNR Data Retrieval and PNR Data Update. The workload application randomly executes the following database transactions using multiple database connections, with all queries accessing each of the 25 database tables and returning one or more rows in the result set:

- PNR Data Retrieval
  - ‘PNR Primary key’ search: find and read one PNR is 35%
  - ‘PNR Primary key’ search: PNR not found is 2%
  - ‘Passport number’ search: find and read one PNR is 35%
  - ‘Frequent flyer number’ search: find and read n PNR is 23%
  - ‘Group name’ PNR Report/List: find 100 rows is 5%
- PNR Data Update
  - Delete data from all 25 tables
  - Insert new PNR data to all 25 tables

Hardware and software considerations

The workload topology is shown on Figure 7-14. Four computers are used to generate the workload, and one computer is used to run a pair of solidDB HotStandby servers:

- Test workload system
  - Two machines: 4x Dual Core Intel Xeon Processor 5110 (1.6 GHz, 16 GB RAM)
  - Two machines: 8x Quad Core Intel Xeon Processor E5504 (2.0 GHz, 40 GB RAM)
- solidDB HSB system
  - solidDB 6.5.0.0
  - 16x Quad Core AMD Opteron Processor 8346 (1.8 GHz, 128 GB RAM)

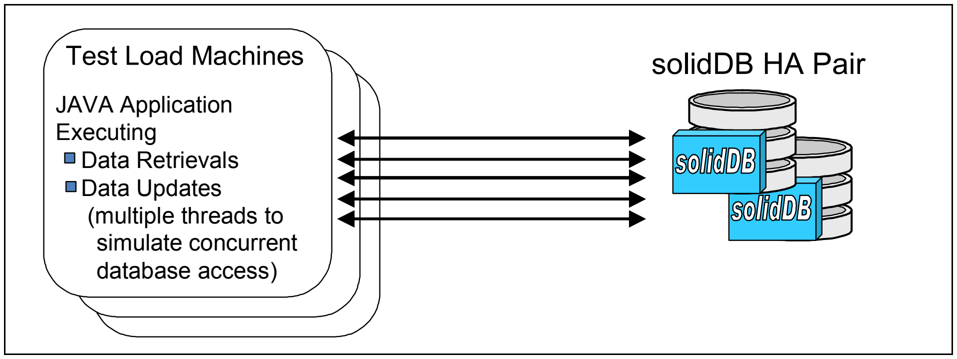

Test Load Machines
JAVA Application Executing
Data Retrievals
Data Updates
(multiple threads to simulate concurrent database access)
solidDB HA Pair
solidDB
solidDB

Figure 7-14 Online Flight Reservation Workload setup

Results

The results are presented in Figure 7-15 on page 263 and Figure 7-16 on page 264, showing transactional response times and total workload throughputs as a function of the increased number of database connections. Both are measured for individual transactions, with each transaction executing SQL many statements against all database tables.

The results show that the overall throughput increases as more clients connect to the database, demonstrating excellent solidDB database scalability. Moreover, transactional response times do not vary with the number of clients. The solidDB in-memory engine is able to deliver near constant and predictable response times even as the overall database workload is being increased to meet growing business needs or to manage anticipated peak times.

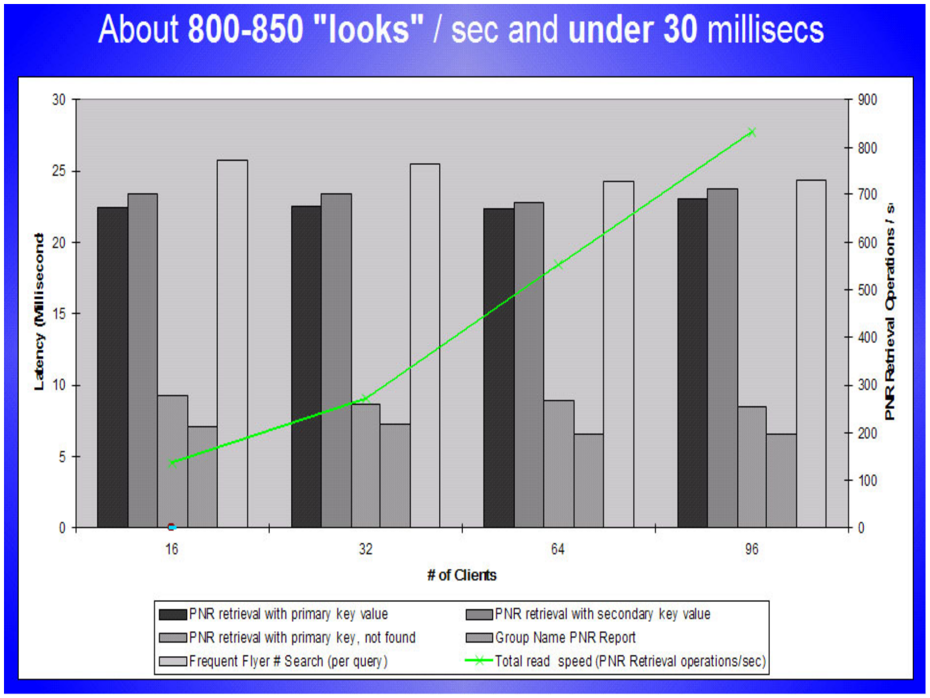


Figure 7-15 PNR data retrieval throughputs and response times

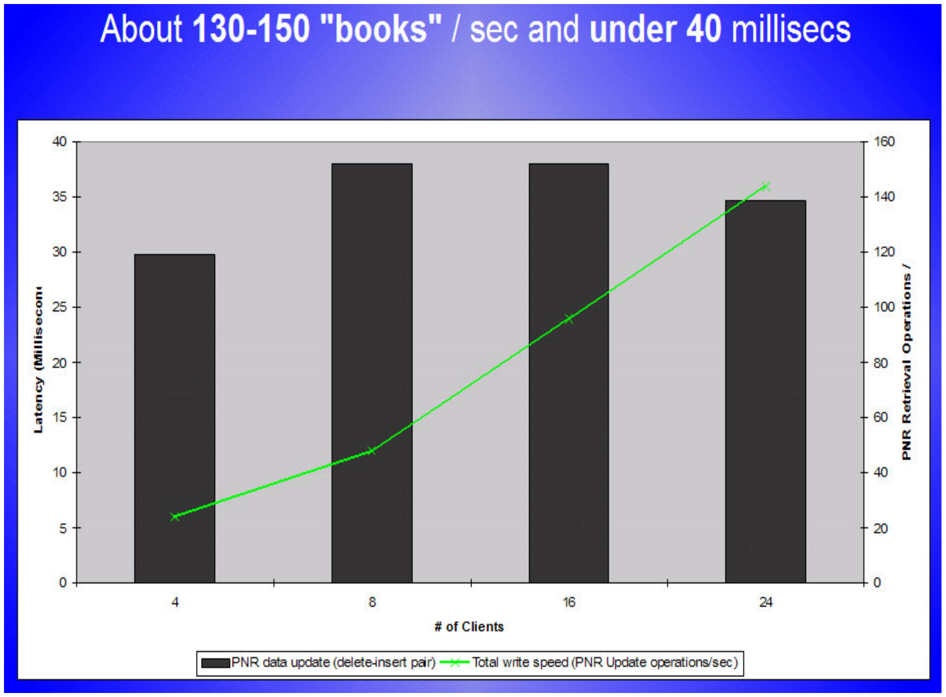


Figure 7-16 PNR data update throughputs and response times

Hotel Reservation Workload

The Hotel Reservation Workload demonstrates the benefit of solidDB Universal Cache in a hotel reservation scenario. The workload driver is a simple C application that executes hotel room availability queries against the in-memory cache. Any bookings are recorded directly against the back-end database, and changes to cached tables are propagated into the cache.

The workload use a read-only cache that is well suited for applications where workload is read dominated; with hotel reservations, the "look-to-book" ratio is often as high as several hundreds to one.

The workload uses a database schema in which the solidDB front-end cache contains only two tables:

- Room information: availability, type, location
- Features: bed type, view, other amenities

Figure 7-17 shows results of a case where the workload has been executed against the following three setups:

- Back end using ODBC connection
- solidDB Universal Cache with solidDB V6.3 using ODBC connection
- solidDB Universal Cache with solidDB V6.5 using SMA connection

As Figure 7-17 shows, the transactional throughput grows significantly after the solidDB Universal Cache is implemented in front of the disk based database. The improvement is further increased when SMA is used to access solidDB data.

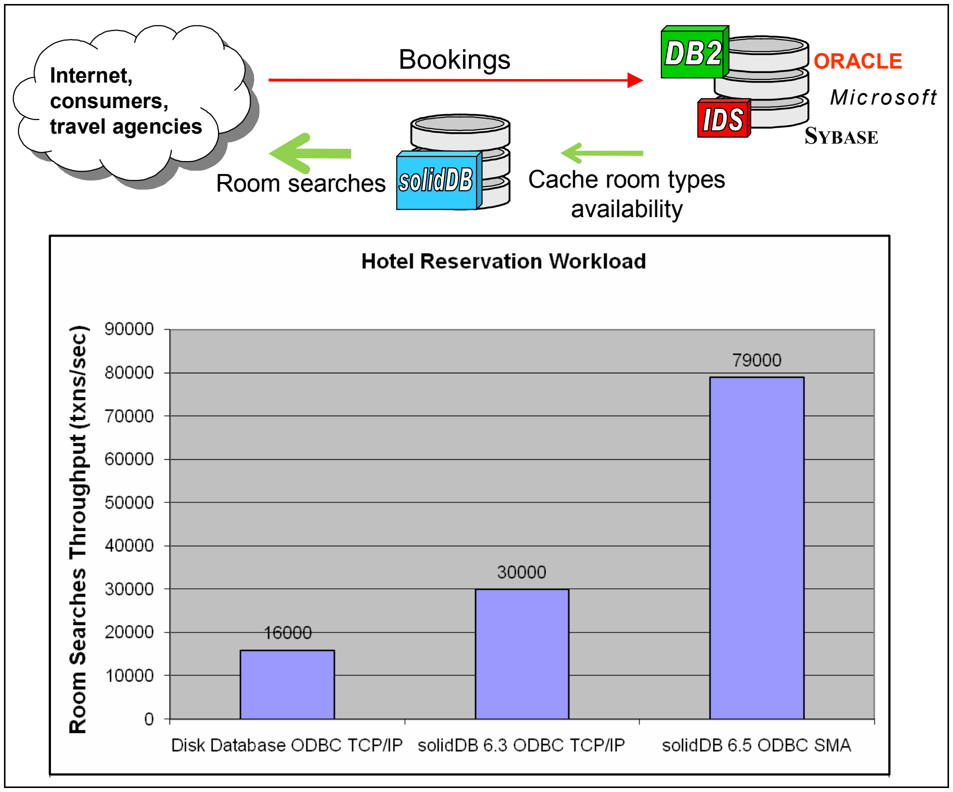


Figure 7-17 Results of the Hotel Reservation Workload executed against solidDB Universal Cache

7.4.7 Media

This section describes how the media delivery industry can benefit from the solidDB products.

Media delivery and systems have undergone a vast number of changes over the past decades, from early days of radio and television, to the modern era of on-demand delivered content. Waiting for scheduled programming is no longer satisfactory for most consumers; viewers are getting used to, and beginning to expect, the ability to watch any show they want at any time. Any solution to this problem eventually requires that the content can be recorded and replayed.

This idea is not a particularly novel one either; video cassette recorders (VCRs) have existed since the 1980s. However, the advent of digital television allows more modern devices to replace the analog recording onto a magnetic tape with storing of digital data onto a hard-drive. Many digital video recorder (DVR) solutions are available on the market, but in most cases the physical device resides in the consumer’s home. This technological choice results in a number of limiting factors. For instance, multiple devices or some type of digital network connectivity are needed to serve the programming to multiple monitors in different rooms or even different locations, like the vacation house. Also, all data is stored on the recording device, so there is no possible recovery from device failure, and expanding the storage requires changes to the physical recorder.

Remote DVRs are offering to solve all these problems for end customers, bringing additional opportunities to the content providers. This approach involves storing all recordings on a centralized system consisting of a large number of high end storage units and a secure metadata repository. High performance storage arrays take care of the data volumes needed to support thousands of concurrent video recordings by shredding the videos across many disks; built-in redundancy of the storage system offers protection from data loss.

A metadata repository contains all information necessary to reconstruct the recording and serve it to the customer at a later time. There are clear options for providers to optimize the business by facilitating the sharing of recorded bits among multiple customers who have recorded a particular show, thus reducing overall system cost, or by inserting targeted and personalized advertisements into the programming at replay time, thus increasing marketing revenue.

Remote DVR systems can thus be said to require the following properties from the metadata repository; and solidDB in-memory database is a perfect fit for such technical requirements and business needs:

- No data loss
- Quick transaction response times
- Real-time database properties, such as quickly reacting to administrative requests
- Ability to predictably handle large peak loads
- Ability to sustain high throughput

Because metadata is needed to store, recover, and reconstruct the recordings, any loss of metadata amounts to the loss of the customer recording. Furthermore, during the time the system needs to detect and recover a database failure none of the recordings are accessible, effectively taking the DVR solution out of service. Using solidDB HotStandby (HSB) provides high-availability options that guarantee no data loss in case of a single database failure, while maintaining excellent performance characteristics, fast failure detection and database failover, and fast recovery back to the fully active HSB system.

As demonstrated in the previous sections, solidDB in-memory technology provides low and predictable transactional response times and sustains high throughputs. This is particularly important because the remote DVR systems need to manage extremely high peak loads - imagine millions of users wanting to record the U.S. Superbowl football game within a few minutes before the game starts. Because DVR metadata repositories are not likely to require complicated database schemas or transactional workloads, they fit in the solidDB “sweet spot” areas discussed in 7.1, “solidDB and Universal Cache sweet spots” on page 220.

Conclusion

Over the last few years the world has experienced a true data volume explosion. Studies indicate that the amount of data routinely being processed to yield actionable information is growing faster than Moore's law[1]. Additionally, there are many new classes of structured and unstructured data which we may be able to convert into knowledge and base decisions: for example, every single click on the Internet, or every time a light switch is flipped, or every time a stock price falls after a large volume trade, or every time a car enters a section of a highway, Also, think about that growth in data volumes and new classes of information in terms of a large and growing worldwide environment.

In this worldwide environment, we have to understand that the way information is being extracted and derived must change to be able to keep up with the overwhelmingly increasing demands. Simply following existing paradigms and expecting hardware advances, such as those bound by Moore's law to double the capacity roughly every two years, to provide the necessary bandwidth will not suffice.

Adopting and using fast and efficient in-memory database technology is a part of the answer. It provides the necessary paradigm shift toward answering a set of questions more effectively. When used well, it does more with less, and this is the unbeatable opportunity that should not be missed in the present-day competitive business environment.

[1] http://www.intel.com/technology/mooreslaw/

8.1 Where are you putting your data

As the amount of collected data grows over the next decade[2] (Figure 8-1), new approaches to processing, analyzing and information management will become a necessity.

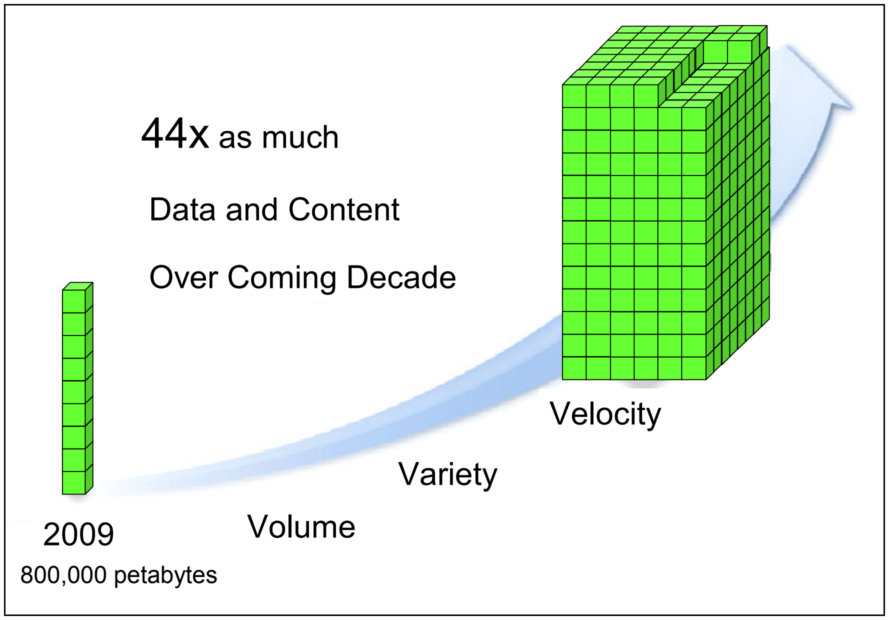


Figure 8-1 Expected data growth over the next decade

Examples of highly demanding processing workloads can be found in any number of industries such as financial services, communications, and web, to name a few. Consider the following requirements:

- Brokerage application
  - Receive market feeds
  - Evaluate equity positions
  - Check for fraud
  - Evaluate tens of thousands of rules for thousands of trades per second and millions of trades per day
- Telecommunications online charging
  - Authenticate and authorize
  - Initiate service

[2] IDC, John Gantz and David Reinsel, *The Digital Universe Decade – Are You Ready?*, May 2010; http://idcdocserv.com/925

  - Manage credit balance
  - Manage volume discounts
  - Hundreds of thousands of concurrent requests
  - Needing microsecond database response times
- Web 2.0
  - Authenticate users and manage personal profiles
  - Generate page contents with targeted advertising
  - Facebook has millions of concurrent sessions; billions of page views daily
  - Wikipedia has thousands of page views per second
  - Needing tens of thousands of database requests per second

One obvious approach to managing huge data sets is classifying them into multiple layers of *hotness*, or importance. This concept is not novel, because all respectable database systems already make such distinctions by pulling active data into memory buffer pools, or by archiving historical or rarely used data on tape or inexpensive disks.

However, the number of storage tiers is increasing, and forthcoming technical advances add even more complexity to the picture. Data access time increases for each consecutive tier, in some cases by multiple orders of magnitude. However, the cost per byte stored and physical size limits are also reduced significantly.

Starting from the fastest and most expensive, the following data storage mechanisms are currently available:

- CPU cache
- Volatile DRAM (Dynamic Random Access Memory) main memory
- Non-volatile DRAM (likely battery-backed) main memory[3]
- Non-volatile PRAM (Phase-Change RAM) main memory[3]
- Non-volatile Flash based main memory[3]
- SSD (Solid State Disk) I/O devices
- HDD (Hard Disk Drive) I/O devices
- Magnetic tape I/O devices

The future of effective information management will require intelligent and business-driven choices regarding what data is to be kept within each of the storage tiers. An equally important question will be “What products yield optimal performance characteristics for any given storage type?”

[3] Not yet available in the market, however the amount of available research material indicates strongly that such technologies will be in the market within this decade.

8.2 Considerations

Historically disk-based databases have been the easy answer because data had to be persisted within one of the I/O device types, for which disk-based databases are optimized. Main memory was effectively used only as a volatile “staging and manipulation area” while transporting data between I/O based storage and the CPU.

With anticipated advances in directly addressable main memory technologies (including non-volatility, larger available sizes, lower costs) the importance of new database systems optimized for main memory access will greatly increase. Though adoption of these new technologies presently looks like a choice, gated mostly by the cost of introducing a new software solution into the existing system, it may not be long before doing so becomes a necessity.

Bringing data closer to the application allows us to use the fastest and most efficient data access paradigms, yielding more results faster. This unique value proposition is realized by the IBM solidDB product family.

Glossary

1-safe algorithm. A method of transaction processing in HotStandby setups. In 1-safe systems, transactions are committed on the primary server and then propagated to the secondary server after If the primary server fails before it sends the transactions to the secondary server, the transactions will not be visible on the secondary server. Also see 2-safe algorithm.

2-safe algorithm. A method of transaction processing in HotStandby setups. In 2-safe systems, transactions are not considered committed on the primary server until the transaction is confirmed committed on the secondary server. All updates to the data are applied to both copies synchronously. If the secondary server fails, the primary server stops accepting transactions. Also see 1-safe algorithm.

Access mode. The access mode of a solidDB parameter defines whether the parameter can be changed dynamically through an ADMIN COMMAND, and when the change takes effect. The possible access modes are RO, RW, RW/Startup, RW/Create.

application programming interface (API). An interface provided by a software product that enables programs to request services.

binary large object (BLOB). A block of bytes of data (for example, the body of a message) that has no discernible meaning, but is treated as one entity that cannot be interpreted.

Bonsai Tree. A small active index (data storage tree) that stores new data (deletes, inserts, updates) in central memory efficiently, while maintaining multiversion information.

cache database. The solidDB database in a Universal Cache setup. Also called *cache* or *front-end*.

concurrency control. A method for preventing two different users from trying to update the same data in a database at the same time.

Data Definition Language (DDL). An SQL statement that creates or modifies the structure of a table or database, for example, CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE DATABASE.

Data Manipulation Language (DML). An INSERT, UPDATE, DELETE, or SELECT SQL statement.

data store (InfoSphere CDC). A management entity that represents the InfoSphere CDC instance in Management Console.

deploy. The process of making operational the configuration and topology of the solidDB Universal Cache.

disk-based table (D-table). A table that has its contents stored primarily on disk so that the server copies only small amounts of data at a time into memory. Also see in-memory table.

distributed application A set of application programs that collectively constitute a single application.

durability level. A feature of transactionality that controls how solidDB handles transaction logging. solidDB supports three durability levels: strict, relaxed, and adaptive.

Dynamic SQL. SQL that is interpreted during execution of the statement.

Instance (InfoSphere CDC). A runtime instance of the InfoSphere CDC replication engine for a given DBMS.

Java Database Connectivity (JDBC). An API that has the same characteristics as ODBC but is specifically designed for use by Java database applications.

Java developer kit. A software package used to write, compile, debug, and run Java applets and applications.

Java Message Service. An application programming interface that provides Java language functions for handling messages.

Java runtime environment. A subset of the Java developer kit that allows you to run Java applets and applications.

In-memory table (M-table). A table whose contents are entirely stored in memory so that the data can be accessed as quickly as possible. Also see disk-based table.

main-memory engine (MME). The solidDB component that takes care of operations concerning in-memory tables.

meta data. Typically called data (or information) about data. It describes or defines data elements.

multi-threading. A capability that enables multiple concurrent operations to use the same process.

Open Database Connectivity (ODBC). A standard API for accessing data in both relational and non-relational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface. ODBC is based on the call level interface (CLI) specification of the X/Open SQL Access Group.

optimization. The capability to enable a process to execute and perform in such a way as to maximize performance, minimize resource utilization, and minimize the process execution response time delivered to the user.

partition. Part of a database that consists of its own data, indexes, configuration files, and transaction logs.

Primary Key. A field in a table that is uniquely different for each record in the table.

process. An instance of a program running in a computer.

replication (InfoSphere CDC). InfoSphere CDC replication is based on an asynchronous, push-based model. Unidirectional subscriptions can be created for real-time propagation of data changes from the source side to the target side. Bidirectional capability is achieved by setting up two subscriptions with mirrored source and target definitions.

replication (HotStandby). In HotStandby (HSB) setups, data changes in the primary are propagated to the secondary using a push-based replication protocol. The protocol can be set to synchronous (2-safe) or asynchronous (1-safe).

replication (advanced replication). In advanced replication setups, an asynchronous pull-based replication method enables occasional distribution and synchronization of data across multiple database servers.

read-only (RO). Parameter access mode where the value cannot be changed; the current value is always identical to the startup value.

read-write (RW). Parameter access mode where the value can be changed through an ADMIN COMMAND and the change takes effect immediately.

RW/Startup. Parameter access mode where the value can be changed through an ADMIN COMMAND and the change takes effect the next time that the server starts.

RW/Create. Parameter access mode where the value can be changed through an ADMIN COMMAND and the change applies when a new database is created.

server. A computer program that provides services to other computer programs (and their users) in the same or other computers. However, the computer that a server program runs in is also frequently referred to as a server.

shared nothing. A data management architecture where nothing is shared between processes. Each process has its own processor, memory, and disk space.

SQL pass-through. The act of passing SQL statements to the back end, instead of executing statements in the front-end.

static SQL. SQL that has been compiled prior to execution. Typically provides best performance.

subscription (InfoSphere CDC). A connection that is required to replicate data between a source data store and a target data store.

Abbreviations and acronyms

ACID	atomicity, consistency, isolation, durability
ACS	access control system
ADK	Archive Development Kit
AIX	Advanced Interactive eXecutive from IBM
API	application programming interface
ASCII	American Standard Code for Information Interchange
BE	back end data server
BLOB	binary large object
CDC	change data capture
CLI	call level interface
CLOB	character large object
CPU	central processing unit
DBA	database administrator
DBMS	database management system
DDL	Data Definition Language
DES	Data Encryption Standard
DLL	dynamically linked library
DML	Data Manipulation Language
DSN	Data Source Name
D-table	Disk-based Table
EJB	Enterprise Java Beans
ERE	External Reference Entity
FE	front end data server
FP	Fix Pack
FTP	File Transfer Protocol
Gb	gigabit
GB	gigabyte
GUI	graphical user interface
HAC	solidDB High Availability Controller
HADR	High Availability Disaster Recovery - DB2
HAM	solidDB High Availability Manager
HSB	solidDB HotStandby
I/O	input/output
IBM	International Business Machines Corporation
ID	Identifier
IDS	Informix Dynamic Server
ISV	Independent Software Vendor
IT	Information Technology
ITSO	International Technical Support Organization
J2EE	Java 2 Platform Enterprise Edition
JAR	Java Archive
JDBC	Java DataBase Connectivity
JDK	Java developer kit
JE	Java Edition
JMS	Java Message Service
JRE	Java runtime environment
JTA	Java Transaction API
JVM	Java virtual machine
KB	kilobyte (1024 bytes)
LDAP	Lightweight Directory Access Protocol
LLA	linked library access
Mb	megabit
MB	megabyte
MBCS	multibyte character set
MME	main-memory engine

M-table	in-memory table
ODBC	Open DataBase Connectivity
OLE	Object Linking and Embedding
ORDBMS	object relational database management system
OS	operating system
pmon	performance counter
RDBMS	relational database management system
RR	repeatable read
SA API	solidDB application programming interface
SBCS	single byte character set
SDK	software developers kit
SMA	shared memory access
solcon	solidDB remote control utility program
soldd	solidDB Data Dictionary utility program
solexp	solidDB Export utility program
solload(o)	solidDB Speed Loader utility program
solsql	solidDB SQL Editor utility program
SQL	Structured Query Language
SSC API	solidDB server control API
TATP	Telecom Application Transaction Processing
TC	transparent connectivity
TF	transparent failover
TSN	transaction start number
URL	Uniform Resource Locator
VLDB	very large database
VTrie	variable length trie
WAL	write-ahead logging
XA	X/Open XA
XML	eXtensible Markup Language

Index

IBM

Redbooks

IBM solidDB: Delivering Data with Extreme Speed

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

IBM solidDB Delivering Data with Extreme Speed

Provides low latency, high throughput, and extreme availability

Offers fully featured relational in-memory database software

Has universal cache with shared memory access

The world seems to be getting smaller and business moving much faster. To be successful in this type of environment you need instantaneous access to any information, immediate responses to queries, and constant availability, on a worldwide basis, and in a world where the volume of data is growing exponentially. You need the best resources you can get, and ones that can satisfy those needs. IBM can help.

A primary component that can affect performance is access to disk-based data. And, as data volumes grow, so does the performance impact. To improve performance, it is time to look for technology enhancements that can mitigate that impact.

IBM solidDB is powerful relational, in-memory caching software that can accelerate traditional disk-based relational database servers by caching performance-critical data into one or more solidDB in-memory database instances. This capability can enable significant performance improvements. It brings data closer to the application so you can use a faster and more efficient data access paradigm. The result? Faster delivery of information for your queries to enable faster analysis and decision-making that can give you a significant business advantage.

Have questions? Many of the answers you need are contained in this IBM Redbooks publication.

SG24-7887-00 ISBN 0738435457