www.ibm.com/software/data/**soliddb**

IBM solidDB **In-Memory Database User Guide**

Version 6.0 | April 2009

solidDB™

solidDB In-Memory Database User Guide

Copyright © Solid Information Technology Ltd. 2007, 2009
Document number: IMDB60
Product version: 06.00.1059
Date: 2009-04-22

Table of Contents

List of Tables

Chapter 1. Welcome

IBM solidDB (solidDB) In-memory Engine allows you to choose the optimal balance of maximum performance and the ability to handle large volumes of data by providing a unique dual-engine Database Management System (DBMS) architecture. Inside the database server, there are two engines: an in-memory engine for fastest possible access to performance-critical data and a "traditional" on-disk engine for efficiently handling virtually any volume of data.

solidDB In-memory Engine is built on solidDB Disk-based Engine and FlowEngine capabilities, which means that solidDB In-memory Engine inherits all functionality of these products. solidDB In-memory Engine can be used in embedded systems, requiring virtually no administration or maintenance. You can make solidDB In-memory Engine suitable for highly available systems by purchasing the CarrierGrade High Availability option. You may also purchase the SmartFlow option, which enables multiple solidDB In-memory Engine and solidDB Disk-based Engine servers to share and synchronize data with each other.

solidDB provides other options, as well. These are described in *solidDB Administration Guide*.

1.1 About This Guide

solidDB In-Memory Database User Guide introduces you to the features that allow you to optimize your database server's performance by using in-memory database technology.

1.1.1 Organization

Chapter 2, *Basic Features*, describes the basics of in-memory tables vs. disk-based tables. It also contains some information about compatibility and migration with respect to other solidDB servers.

Chapter 3, *Temporary Tables and Transient Tables*, explains two specialized types of in-memory tables: Temporary Tables and Transient Tables.

Chapter 4, *Optimizing and Tuning the Server*, gives you some tips on getting the best performance from solidDB In-memory Engine.

Appendix A, *Calculating Maximum BLOB Size*, shows how to calculate the maximum size of a character or binary column value that will fit in your in-memory tables.

Appendix B, *Calculating Storage Requirements*, explains how to calculate the approximate maximum amount of storage space (memory or disk) required for in-memory tables and disk-based tables.

Appendix C, *Configuration Parameters*, gives a brief introduction of how to modify the configuration parameters related to in-memory database.

Glossary provides definitions of solidDB terminology.

1.1.2 Audience

This guide assumes the reader has general knowledge of relational database management systems and familiarity with SQL. This guide also assumes that the reader has basic familiarity with the solidDB product family. We recommend that you read *solidDB Administration Guide* prior to reading this *solidDB In-Memory Database User Guide*. If you are not already familiar with relational databases, we recommend that you read the *solidDB Getting Started Guide* and solidDB SQL Guide first.

1.2 Conventions

1.2.1 About solidDB

solidDB provides advanced database solutions for mission-critical applications.

This documentation assumes that all options of solidDB are licensed for use. In some cases, however, a customer may choose not to license certain options. These include in-memory engine, disk-based engine, CarrierGrade Option (also known as "HotStandby" in previous releases), and SmartFlow Option. Please refer to your organization's contract with solidDB, or contact your solidDB account representative.

1.2.2 Typographic Conventions

This manual uses the following typographic conventions:

Table 1.1. Typographic Conventions

Format	**Used for**
Database table	This font is used for all ordinary text.
NOT NULL	Uppercase letters on this font indicate SQL keywords and macro names.
`solid.ini`	These fonts indicate file names and path expressions.

`SET SYNC MASTER YES;`
`COMMIT WORK;`
	This font is used for program code and program output. Example SQL statements also use this font.

run.sh	This font is used for sample command lines.
`TRIG_COUNT()`	This font is used for function names.

Format	**Used for**
`java.sql.Connection`	This font is used for interface names.
`LockHashSize`	This font is used for parameter names, function arguments, and Windows registry entries.
`argument`	Words emphasised like this indicate information that the user or the application must provide.
solidDB Administration Guide	This style is used for references to other documents, or chapters in the same document. New terms and emphasised issues are also written like this.
File path presentation	File paths are presented in the Unix format. The slash (/) character represents the installation root directory.
Operating systems	If documentation contains differences between operating systems, the Unix format is mentioned first. The Microsoft Windows format is mentioned in parentheses after the Unix format. Other operating systems are separately mentioned.

1.2.3 Syntax Notation

This manual uses the following syntax notation conventions:

Table 1.2. Syntax Notation Conventions

Format	**Used for**
`INSERT INTO` *`table_name`*	Syntax descriptions are on this font. Replaceable sections are on *`this`* font.
`solid.ini`	This font indicates file names and path expressions.
[ ]	Square brackets indicate optional items; if in bold text, brackets must be included in the syntax.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{ }	Curly brackets delimit a set of mutually exclusive choices in a syntax line; if in bold text, braces must be included in the syntax.

Format	**Used for**
...	An ellipsis indicates that arguments can be repeated several times.
. . .	A column of three dots indicates continuation of previous lines of code.

1.3 solidDB Documentation

Below is a complete list of documents available for solidDB. solidDB documentation is distributed in PDF format.

Electronic Documentation

- *Release Notes*. This file contains installation instructions and the most up-to-date information about the specific product version. This file (`releasenotes.txt`) is copied onto your system when you install the software.

- *solidDB Getting Started Guide*. This manual gives you an introduction to the solidDB.

- *solidDB SQL Guide*. This manual describes the SQL commands that solidDB supports. This manual also describes some of the system tables, system views, system stored procedures, etc. that the engine makes available to you. This manual contains some basic tutorial material on SQL for those readers who are not already familiar with SQL. Note that some specialized material is covered in other manuals. For example, solidDB "administrative commands" related to the High Availability (HotStandby) Option are described in the *solidDB High Availability User Guide*, not the *solidDB SQL Guide*.

- *solidDB Administration Guide*. This guide describes administrative procedures for solidDB servers. This manual includes configuration information. Note that some administrative commands use an SQL-like syntax and are documented in the *solidDB SQL Guide*.

- *solidDB Programmer Guide*. This guide explains in detail how to use features such as solidDB Stored Procedure Language, triggers, events, and sequences. It also describes the interfaces (APIs and drivers) available for accessing solidDB and how to use them with a solidDB database.

- *solidDB In-Memory Database User Guide*. This manual describes how to use the in-memory database of solidDB In-memory Engine.

- *solidDB SmartFlow Data Replication Guide*. This guide describes how to use the solidDB SmartFlow technology to synchronize data across multiple database servers.
- *solidDB AcceleratorLib User Guide*. Linking the client application directly to the server improves performance by eliminating network communication overhead. This guide describes how to use the AcceleratorLib library, a database engine library that can be linked directly to the client application.

  This manual also explains how to use two proprietary Application Programming Interfaces (APIs). The first API is the solidDB SA interface, a low-level C-language interface that allows you to perform simple single-table operations (such as inserting a row in a table) quickly. The second API is SSC API, which allows your C-language program can control the behavior of the embedded (linked) database server

  This manual also explains how to set up a solidDB to run without a disk drive.
- *solidDB High Availability User Guide*. solidDB CarrierGrade Option (formerly called the HotStandby Option) allows your system to maintain an identical copy of the database in a backup server or "secondary server". This secondary database server can continue working if the primary database server fails.
- *Getting Started With solidDB For VxWorks*. This guide describes how to take into use solidDB on the VxWorks environment. It also provides guidelines for application development and performance tuning. This manual is included only in packages for VxWorks.

Chapter 2. Basic Features

solidDB In-memory Engine combines the high performance of in-memory tables along with the nearly unlimited capacity of disk-based tables. This is something that other solutions on the market are not capable of. 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 were slow because the data structures that are optimal for disk-based tables are far from being optimal for in-memory tables.

solidDB's solution is to provide a single database server that contains two optimized servers inside it: one server is optimized for disk-based access and the other is optimized for in-memory access. Both servers co-exist inside the same process, and a single SQL statement may access data from both engines.

2.1 In-Memory Tables

2.1.1 In-Memory vs. Disk-Based Tables

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

In most respects, in-memory tables are similar to disk-based tables. Most importantly, both table types provide full persistence of data unless specified differently. You may perform the same types of queries on each of them. You may combine disk-based and in-memory tables in the same SQL query. You may combine in-memory and disk-based tables in the same transaction. You may query both types of tables when using the SA API (part of the AcceleratorLib). In addition, in-memory tables may be used with indexes, triggers, stored procedures, etc. In-memory tables also allow constraints, including primary key and foreign key constraints, although there are some limitations on foreign key constraints with non-persistent tables. (This topic is discussed in more detail later.)

With solidDB, the user may decide which tables will be in-memory tables and which tables will be disk-based tables. You may put heavily used tables in main memory so that they can be accessed more quickly. If you have enough memory, you may put all of your tables in main memory.

2.1.2 Types of In-Memory Tables

solidDB provides two different categories of in-memory tables: persistent and non-persistent. These are described below.

Persistent In-Memory Tables

As their name suggests, persistent in-memory tables persist indefinitely. Although client queries access the copy of the data in memory, the server stores the in-memory tables on disk when it shuts down, and therefore the data is available each time that the server starts. In-memory tables also use transaction logging, so that if the server is shut down unexpectedly (due to a power failure, for example), the server has a record of the transactions that have occurred and can update the tables to ensure that they have all the data from all the committed transactions. In-memory tables, like disk-based tables, have their data copied to the hard disk drive during checkpoints (for a description of checkpoints, see *solidDB Administration Guide*).

In-memory tables may also be used with solidDB's HotStandby feature; data in in-memory tables is copied to the Secondary server from where it is available if the Primary server fails.

There are a few differences between in-memory and on-disk tables that should be taken into consideration when designing an application that uses in-memory tables. Most importantly, in-memory tables use always pessimistic row-level concurrency control (locking) whereas disk-based tables use optimistic (versioning) concurrency control by default. Therefore, in in-memory tables read operations block write operations for the duration of the read transaction. Moreover, deadlocks are possible with in-memory tables whereas they cannot occur on versioning disk-based tables. On the other hand, concurrency conflicts may occur when optimistic concurrency control is in use.

It is important to keep these considerations in mind when designing the error handling of your application. Depending on the type of table used, the error handling needs to take different error codes into account.

Another significant difference is the checkpointing algorithm used. The checkpointing of in-memory tables is entirely different from the algorithm used on disk-based tables. The major benefit is that checkpointing in-memory tables doesn't block the transactions' access to the tables in any way during the checkpoint. Thus, the predictability of response times is better with in-memory tables than with disk-based tables.

The third major difference between in-memory and disk tables is that with in-memory tables, the secondary indices are never written to the disk. Instead, they are maintained in-memory only and re-built when the server is started. Hence, their impact on write performance is significantly smaller than with disk-based tables. Moreover, all indices of in-memory tables are equally fast whereas on disk-based tables, the primary key is significantly faster than the other indices.

For all other practical purposes, in-memory tables are indistinguishable from disk-based tables, except that in-memory tables are generally significantly faster.

Non-Persistent In-Memory Tables

Non-persistent in-memory tables are not written to disk when the server shuts down. Therefore, any time that the server shuts down, whether normally or abnormally, the data in non-persistent tables is lost. Their data is not logged or checkpointed. That makes them irrecoverable but remarkably faster than persistent tables.

Non-persistent tables are useful primarily as "scratchpad" tables. For example, you might copy data from a persistent table and then run a series of analyses and then discard the non-persistent copy. Or you might do a long series of transformations and then copy the final results back to persistent tables.

There are two different types of non-persistent in-memory tables: Transient Tables and Temporary Tables. The differences between the two are discussed in detail in Chapter 3, *Temporary Tables and Transient Tables*. We will give only a brief overview here.

Transient Tables

Transient Tables last until the database server shuts down. Multiple users may use the same Transient Table, and each user will see all other users' data.

Transient Tables have some limitations compared to persistent in-memory tables. For example, there are some limitations on using foreign keys (referential constraints) with Transient Tables. Also, Transient Tables are not copied to a HotStandby Secondary.

Temporary Tables

The data in Temporary Tables is visible only to the connection that inserted the data, and the data is retained only for the duration of the connection. Temporary Tables are like private scratchpads that no one else can see.

In addition to avoidance of logging they also do not use any type of concurrency control mechanism (such as record locking), which makes them even faster than transient tables.

2.1.3 How In-Memory Tables Improve Performance

If data is stored in a disk-based table, then the data must be read into memory before it can be used, and must be written back to the disk after it has been used. In-memory tables provide higher performance because all the data resides always in the main memory and hence the server may use more efficient techniques to provide the maximum performance for accessing and manipulating data.

Of course, almost any database server will perform faster if it has more memory and can store a larger percentage of its data in the cache memory of the server. However, solidDB In-memory Engine's high-performance in-memory technology does much more than merely copy data into memory. solidDB In-memory Engine also uses index structures that are optimized to work with data that is stored entirely in memory. solidDB In-memory Engine also takes into account issues that arise with in-memory tables, such as memory "fragmentation" when tables grow or shrink.

2.1.4 How to Decide Which Tables to Designate as In-Memory Tables

Ideally, your computer will have enough memory to store all of your tables in memory and thus provide the best possible performance for database transactions. However, in the real world, most users will have to choose a subset of tables to store in memory, while the remaining tables will be disk-based.

Not surprisingly, if you can't fit all the tables in memory, then you want to try to put the most-frequently-used data in memory. It's easy to see that small, frequently-used tables should go into memory, and large, rarely-used tables can be left on disk. But how about the other possible combinations, such as large tables that are heavily used, or small tables that aren't heavily used?

The best way to think about this is to think about the "density" of access to a table. The higher the number of accesses per megabyte per second, the better off you are putting that table in memory. For a more detailed algorithm, see Section 4.1, “Algorithm for Choosing Which Tables to Store in Memory”.

Once you've decided to store a table in memory, you must choose whether to store the data in a persistent table, a Transient Table, or a Temporary Table. The basic rules are as shown below. Note that these rules are guidelines, not rigid rules. You should read the detailed descriptions of Transient Tables and Temporary Tables before making a final decision.

Note that when we use the term "server session", we mean a single "run" of the server, from the time that it starts until the time that it is either deliberately shut down or it goes down for an unexpected reason (such as a power failure). A "connection" lasts from the time that a single user connects to the server until the time that user disconnects the same connection. (A user may establish multiple connections, but each of these is independent.)

You can decide on the most appropriate type of table by asking yourself the questions below until you reach the first question for which you answer "Yes".

1. Do you need the data to be available again the next time that the server starts? If so, then use a persistent table.
2. Do you need the data to be copied to the Secondary HotStandby server? If so, then use a persistent table.
3. Do you need the data only during the current server session, but the data must be available to multiple users (or multiple connections from the same user)? If so, then use a Transient Table.
4. If none of the above rules applied, then use a Temporary Table.

Again, you should read the details about Temporary and Transient Tables before making a final decision. Temporary and Transient Tables have a few restrictions, such as limits on foreign keys (referential constraints) that may affect your decision.

2.1.5 How to Specify that a Table Is Stored in Memory

There are two ways to explicitly specify whether tables are to be located in-memory or on disk. The system creates disk-based tables by default.

1. Use the STORE clause of the CREATE TABLE or ALTER TABLE command. E.g.

   ```
   CREATE TABLE employees (name CHAR(20)) STORE MEMORY;
   CREATE TABLE ... STORE DISK;
   ALTER TABLE network_addresses SET STORE MEMORY;
   ```

   (For more information about the syntax of the CREATE TABLE and ALTER TABLE statement, see solidDB SQL Guide.)

2. Specify the default in the `solid.ini` file. For convenience, if most or all of the new tables that you create should be of the same type (e.g. in-memory), then you may tell the server to automatically use this storage type unless otherwise specified. To specify the default type for new tables, set the following parameter in the `solid.ini` configuration file:

   ```
   [General]
   DefaultStoreIsMemory=yes
   ```

   If this parameter is set to 'yes', new tables will be created as in-memory tables unless specified otherwise in the CREATE TABLE statement. If this parameter is set to 'no', then new tables will be created as disk-based tables unless specified otherwise in the CREATE TABLE statement. As with other parameters in the `solid.ini` file, if this parameter is changed, the new value will not take effect until the next time that the server is started. See *solidDB Administration Guide* for more information about the *`Default-StoreIsMemory`* parameter.

Note

These instructions apply to persistent tables only. Tables that are declared to be Temporary Tables or Transient Tables are automatically stored in memory, even if you don't use the STORE MEMORY clause.

2.1.6 Memory Consumption

Understanding memory consumption is important, because if the database server uses up all of the available virtual memory in the system, you will be unable to add or update data. If the server uses up all of the physical memory and starts to use virtual memory, the server will continue to operate, but performance will be greatly reduced.

The In-memory Database main memory usage differs from the standard solidDB. The In-memory Database resides in its own memory pool. For more information on solidDB memory consumption, refer to *solidDB Administration Guide*.

solidDB In-memory Engine provides commands and configuration parameters to help you monitor and control memory consumption of the server. These commands and parameters focus on the server's In-Memory Database feature, not the server as a whole.

Monitoring Memory Consumption

The command

ADMIN COMMAND 'info imdbsize';

returns the current amount of memory that the in-memory data uses. The value returned is a VARCHAR, and it indicates the number of kilobytes used by the server. Note that this returns the amount of virtual memory used, not the amount of physical memory used.

In time, the imdbsize can grow, because returning data back to operating system can only be done in allocation units which need to be completely unused before they can be returned back to the operating system.

Transient memory allocations (such as SQL execution graphs) are excluded from the **ADMIN COMMAND 'info imdbsize';** report.

There are also several performance counters available, which include the run-time information related to the in-memory database server. Entering the command

ADMIN COMMAND 'pmon mme';

produces the following list of current values of counters.

```
RC TEXT
-- ----
 0 Performance statistics:
 0 Time (sec)                            30    21    Total
 0 MME current number of locks       :     0     0        0
 0 MME maximum number of locks       :     0     0        0
 0 MME current number of lock chains :     0     0        0
 0 MME maximum number of lock chains :     0     0        0
 0 MME longest lock chain path       :     0     0        0
 0 MME memory used by tuples         :     0     0        0
 0 MME memory used by indexes        :     0     0        0
 0 MME memory used by page structures:     0     0        0
10 rows fetched.
```

The command

ADMIN COMMAND 'mem';

only reports the allocated dynamic memory heap size. In heap-based memory allocation, memory is allocated from a large pool of unused memory area called the heap. The size of the heap memory allocation can be determined at run-time. Transient memory allocations (such as SQL execution graphs) are included in the **ADMIN COMMAND 'mem';** report.

Controlling Memory Consumption

There are two configuration parameters, *`ImdbMemoryLimit`* and *`ImdbLowPercentage`* to control memory consumption. Both of these parameters are in the *`[MME]`* section of the `solid.ini` file.

ImdbMemoryLimit

The *`ImdbMemoryLimit`* parameter specifies the maximum amount of virtual memory that can be allocated to in-memory tables (including Temporary Tables, Transient Tables, and "normal" in-memory tables) and the indexes on those in-memory tables.

The default value for *`ImdbMemoryLimit`* is 0, which means "no limit". We strongly recommend that you not use the default value. You should set the parameter to a value that will ensure that the in-memory data will fit entirely within physical memory. Naturally, you must take into account the following factors:

- the amount of physical memory in the computer
- the amount of memory used by the operating system
- the amount of memory used by solidDB (the program itself)
- the amount of memory set aside for the solidDB server's cache (the *`CacheSize`* `solid.ini` configuration parameter)

- the amount of memory required by the connections, transactions and statements running concurrently in the server. The more concurrent connections and active statements there are in the server, the more working memory the server requires. Typically, you should allocate at least 0.5MB of memory for each client connection in the server.
- the memory used by other processes (programs and data) that are running in the computer

When the limit is reached, i.e. when the in-memory tables are using up 100% of the memory specified by *`ImdbMemoryLimit`*, the server will prohibit UPDATE operations on in-memory tables. (Before the limit is reached, the server will prohibit creation of new in-memory tables and INSERT operations on those tables. See the description of the ImdbLowPercentage parameter for more details.)

Example:

```
[MME]
ImdbMemoryLimit=1000MB
```

ImdbLowPercentage

The ImdbLowPercentage variable sets a "low water mark", expressed as a percentage of the memory (i.e. as a percentage of the *`ImdbMemoryLimit`*). When the server has consumed the specified percentage of the memory, the server will start to limit activities in order to prevent memory consumption from continuing to grow. For example, if *`ImdbMemoryLimit`* is 1000 megabytes, and if ImdbLowPercentage is 90%, then if the memory allocated to the in-memory tables exceeds 900 megabytes, then the server will start limiting activities. Specifically, the server will:

- Prohibit further creation of in-memory tables (including Temporary Tables and Transient Tables) and indexes on in-memory tables.
- Prohibit INSERTs into in-memory tables.

When the upper limit itself (i.e. *`ImdbMemoryLimit`*) is reached, the server will also prohibit UPDATE operations on records in in-memory tables.

Valid values for *`ImdbLowPercentage`* range from 60-99 (percent).

What to Do If You Reach the ImdbMemoryLimit

If you get an error message indicating that this limit has been reached, you will need to take strong action immediately. You must address both the immediate problems and the long term problem. The immediate problems are to prevent users from experiencing serious errors, and to free up some memory before shutting down the server so that you are not in the same situation (out of memory) when you restart the server. The long term problem is to make sure that you do not get back into this situation in the future as tables expand.

To address the immediate problem, you should typically do the following:

1. Notify users that they should disconnect from the server. This will accomplish two things: it will minimize the number of people who will be impacted if the situation deteriorates; and, if any of the users who disconnect were using Temporary Tables, then disconnecting will free up memory. You may wish to have a policy or error-checking code to ensure that users and/or programs will attempt to disconnect gracefully if they see this error.

2. If there were not enough Temporary Tables to make much difference in memory consumption, then drop some Transient Table indexes or Transient Tables themselves if any exist.

If there were not enough Temporary Tables and Transient Tables to make much difference in memory consumption, then you will have to take more drastic action.

1. Drop one or more indexes on in-memory tables..

2. Shut down the server.

3. If there was absolutely nothing in memory that you could discard (e.g. you had only "normal" in-memory tables, none of which had indexes, and all of which had valuable data), then increase the *`ImdbMemoryLimit`* slightly before re-starting the server. This may force the server to start paging virtual memory which will greatly reduce performance, but it will allow you to continue using the server and address the long-term problems. If you previously set the *`ImdbMemoryLimit`* a little bit lower than the maximum, you will be able to raise it slightly now without forcing the system to start paging virtual memory.

4. Re-start the server.

5. Minimize the number of people using the system until you have had time to address the long-term problem. Ensure that users do not create Temporary Tables or Transient Tables until the long-term problem has been addressed.

If you have solved the immediate problem and have ensured that the server has at least a little bit of free memory, then you are ready to address the long-term problem.

To address the long-term problem, you will need to reduce the amount of data stored in in-memory tables. The ways to do this are to reduce the number or size of in-memory tables (including Temporary Tables and Transient Tables), or reduce the number of indexes on in-memory tables.

If the problem was caused solely by heavy usage of Temporary Tables (or Transient Tables), then you may merely need to ensure that not too many sessions create too many large Temporary Tables (or Transient Tables) at the same time.

If the problem was caused by using too much memory for "normal" in-memory tables, and if you cannot increase the amount of memory available to the server, then you will have to move one or more tables out of main memory and onto the disk. Fortunately, this is not very difficult. To move a table from memory to disk, do the following:

1. Create an empty disk-based table with the same structure (but a different name) as one of the tables in memory.
2. Copy the information from the in-memory table to the disk-based table.[1]
3. Drop the in-memory table.
4. Rename the disk-based table to have the original name of the now dropped in-memory table.

It is a good precaution to deliberately set the *`ImdbMemoryLimit`* to a slightly lower value than the maximum you really have available. This way, if you run out of memory and have no unnecessary in-memory tables or indexes that you can get rid of, you can increase the *`ImdbMemoryLimit`* slightly, then re-start the server with enough free memory that you can address the long-term need.

Not all situations require you to reduce the number of in-memory tables. In some cases, the most practical solution may be to simply install more memory in the computer.

Also, keep in mind that it is better to prevent the problem than to solve it. We very strongly recommend that you set the *`ImdbMemoryLowPercentage`* parameter to an appropriate value so that you get a reliable warning before you use up all the memory available to your in-memory tables.

2.1.7 Calculating Disk Space Requirements

When calculating the amount of disk space required to store your database, you must take into account the amount of space required to store the in-memory tables as well as the disk-based tables. The reason for this

[1]If you try to copy a big table's records to another table using a single SQL statement (INSERT INTO ...VALUES SELECT FROM), please keep in mind that the entire operation occurs in one transaction. Such an operation is efficient only if the entire amount of data fits in the cache memory of the server. If transaction size outgrows the cache size, the performance degrades significantly. Therefore, it is strongly recommended that copying data of a large table to another table is done in smaller transactions (e.g. few thousands of rows per transaction) using a simple stored procedure or application.

Note also that the intermediate table does not need indices. The indices should be recreated in the new "actual table" after the data has been successfully copied.

is that when the server shuts down, it stores the in-memory data to the disk drive. (When the server re-starts, it reads this information back into memory.)

Note also that because the server must write in-memory data to the disk drive when the server shuts down, a server with in-memory tables usually requires more time to shut down than a server that has only disk-based tables. Similarly, because the server must reload data from the disk drive into memory when the server starts, the server usually requires extra time to start up if it has in-memory tables.

2.1.8 Standards Compliance

The in-memory tables feature is not part of the ANSI standard for SQL-99.

2.1.9 Limitations of In-Memory Tables

Physical Memory and Virtual Memory

The most obvious limitation is that the total size of the in-memory database tables cannot exceed the amount of virtual memory available.

Important

Since virtual memory is swapped to disk frequently, using virtual memory negates part of the advantage of in-memory tables. We strongly recommend that you limit your in-memory tables to less than the size of the available physical memory, not the size of the available virtual memory.

When calculating the amount of space required for tables, don't forget to take into account "BLOb" data. Generally, BLOb data should be kept on disk-based tables as the maximum size of a BLOb column is significantly reduced on main-memory tables.

Keep in mind that the amount of space required to store a table includes the space not only for the data that is in the table, but also for any indexes on that table, including any indexes created in support of primary key and foreign key constraints. Also, tables occupy significantly more space in memory than on disk.

If the server runs out of virtual memory when it tries to allocate memory (e.g. to expand a table during an INSERT or ALTER TABLE operation), you will get an error message.

Changing a Table from In-Memory to Disk-Based or Vice-Versa

It is possible to alter the type of a table from in-memory table to disk-based table or vice versa, if the table is empty. To do this, use the

```
ALTER TABLE table_name SET STORE MEMORY | DISK
```

command. If the table contains data, you need to create a new table (with different name) to which you copy the data. After copying the data to the new table, you can drop the old table and rename the new table with

```
ALTER TABLE current_table_name SET TABLE NAME new_table_name
```

command.

Transaction Isolation

Serializable Isolation Level Is Not Supported

In-memory tables may not be used in transactions where the transaction isolation level is SERIALIZABLE. The levels of transaction isolation that are supported for in-memory tables are REPEATABLE READ and READ COMMITTED.

On HotStandby Secondary, Transaction Isolation Is Always Read Committed

If you are using HotStandby (solidDB CarrierGrade) and you connected to the HotStandby Secondary server, then when you read data from in-memory tables, the transaction isolation level is automatically set to READ COMMITTED, even if you specified REPEATABLE READ. (In-memory tables do not support SERIALIZ-ABLE on either the Primary or the Secondary.)

2.2 Other In-memory Engine Enhancements

In addition to in-memory tables, solidDB In-memory Engine has two more characteristics that enhance performance. First, read operations do not wait for disk access, even when the system is engaged in activities such as checkpointing and transaction logging. Second, with relaxed transaction logging, write operations will not wait for disk access. (See *solidDB Administration Guide* for a discussion of relaxed vs. strict transaction logging.)

2.3 Using solidDB AcceleratorLib and HotStandby (CarrierGrade Option) with solidDB In-memory Engine

2.3.1 AcceleratorLib

AcceleratorLib provides the server in the form of a linkable library. A user may link her application directly to this library and access it via function calls without going through a network communications protocol.

The AcceleratorLib is compatible with solidDB In-memory Engine 's in-memory table feature. You may create in-memory tables in an Accelerator server.

For more information about AcceleratorLib, see solidDB AcceleratorLib User Guide.

2.3.2 HotStandby (CarrierGrade Option)

solidDB High Availability option provides "hot standby" capability. This means that your database server is paired with a second server, and the data on the two is automatically kept synchronized. If one server fails, the other can still be used.

solidDB In-memory Engine 's in-memory feature is compatible with the solidDB CarrierGrade option.

Persistent in-memory tables (i.e. in-memory tables that are not specifically designated TEMPORARY or TRANSIENT) will be replicated from the HotStandby Primary server to the Secondary server.

Note that Temporary Tables and Transient Tables are NOT replicated to the Secondary.

2.4 Incompatibilities with Previous solidDB Products

solidDB In-memory Engine is almost 100% compatible with previous versions of solidDB products, such as solidDB FlowEngine 3.7 and solidDB Database Engine 4.1. However, there are a few exceptions. See the `releasenotes.txt` file for information about compatibility issues.

2.5 Migrating from solidDB FlowEngine

With the exceptions listed in the `releasenotes.txt` file, this product is upwards compatible with respect to solidDB FlowEngine.

This product will read a FlowEngine 3.7 or solidDB Database Engine 4.1 database file and the `solid.ini` configuration file.

If you migrate from FlowEngine to solidDB In-memory Engine, then the existing tables will be disk-based tables.

There are no changes to the communication protocols between clients and the server. Current applications and ODBC/JDBC drivers will work with solidDB In-memory Engine just as they did with earlier products. However, we recommend that you upgrade drivers when you update servers.

Chapter 3. Temporary Tables and Transient Tables

Temporary and Transient tables offer higher performance than standard in-memory tables. The data in Temporary and Transient tables is not permanent, however, and therefore you may need to copy data from the Temporary or Transient table to another table if you want to store it permanently.

By default, when you create an in-memory table, that table is "persistent". The table is written to disk when the server shuts down, and is read back from disk when the server starts up again. However, solidDB In-memory Engine offers two types of in-memory tables that are not persistent: Temporary Tables, and Transient Tables. Both of these types of tables are intended for use with "temporary" data.

Temporary Tables and Transient Tables provide higher performance for the following reasons:

- Data in Temporary Tables and Transient Tables is stored solely in memory; it is never written to disk. (If you shut down and re-start the server, or if the server terminates abnormally, the data is lost. In the case of Temporary Tables, the data is discarded at the end of the user session -- it does not even remain until the server is shut down.)

- Temporary and Transient Tables do not log transaction data to disk. (The data is not recoverable after an abnormal server termination.)

- When the server does its periodic "checkpoint" operations, which write database data to the disk drive, the data in Temporary Tables and Transient Tables is not written to the disk. (For a more detailed explanation of checkpoints, see *solidDB Administration Guide*.)

- Temporary Tables and Transient Tables not only use solidDB's high-performance technology for in-memory tables, but also use a more efficient data storage structure than regular in-memory tables use.

- Temporary Tables have a further performance advantage. Sessions (connections) do not see each other's records in a Temporary Table, and therefore Temporary Tables do not need sophisticated concurrency control (e.g. there is no need to check for locking conflicts on records within the table).

Temporary Tables and Transient Tables are especially useful as "scratchpads". For example, you can copy data from a persistent table, do a series of intensive operations on the data while it's in the Temporary Table, and then store the results back in a persistent table. This allows you to maximize performance, yet still keep part or all of the data when you are done. If, for some reason, your work is interrupted, the original data is still safe in the persistent table, and you can re-start the processing.

The main difference between Temporary Tables and Transient Tables is that the data of a Temporary table is visible to a single connection whereas data of a Transient table is visible to all users.

Below is more specific information about each of these two types of tables. Temporary Tables and Transient Tables have a lot in common, and therefore you will find some repetition in the following two sections. We describe each of these two types of table separately and completely, then we highlight the differences between them. We discuss the differences further in Section 4.2, “Performance Tuning Information for Temporary and Transient Tables”.

3.1 Temporary Tables

The data in Temporary Tables has very limited visibility and very limited duration.

3.1.1 Limited Visibility

The data has limited visibility because only the session (connection) that inserted the data can see it. If your session creates a temporary table and inserts data into it, then even if you grant privileges on that table, no other user session will be able to see your data. Note that multiple sessions may use the same table simultaneously, but each session will see only its own data. (Note that since each session will see only its own data, you do not need to coordinate with other sessions to make sure that you insert unique values into the table, even if the table has a unique constraint. For example, if you create a Temporary Table that has a unique constraint on the ID column, you and another session might both insert records that have the ID set to the value 1.) Since each session sees only its own data, operations such as UPDATE and DELETE affect only the session's own data.

3.1.2 Limited Duration

The data has limited duration because as soon as you exit your current session (i.e. as soon as you disconnect from the server), the data is discarded. If you connect again, you will not see your data.

It is important to understand that the word "Temporary" in the name "Temporary Tables" refers to the data, not the table itself. The server actually stores the definition of the Temporary Table (but not the data) in the server's system tables and keeps that definition even after you disconnect. Thus, if you reconnect to the server later, you will find that the table still exists, but is empty. Thus, once you create the table, you do not need to create it again in future sessions. In fact, if you or another user try to create a Temporary Table with the same name as an existing Temporary Table, you will get an error message. This behavior may be unexpected if you think that a "Temporary Table" means that the table (not just the data) will disappear as soon as you disconnect.

Naturally, since the tables persist (even though the data does not), you should use the DROP TABLE command to drop the table definition after you no longer need it.

Because the table persists, if you export a database schema definition, the output will include the commands to re-create the Temporary Tables.

Because a session's temporary tables are cleared when the user disconnects, the server's CPU usage may seem high for some time after a session with a lot of temporary table data disconnects.

3.1.3 Additional Limitations

Below are some additional limitations with Temporary Tables.

- data of Temporary Tables is not replicated to the Secondary server when you use the HotStandby (CarrierGrade) feature. Note that the Temporary Table definitions themselves are replicated to the HotStandby Secondary server. Thus, if you have to fail over to your Secondary, you do not need to re-create any Temporary Tables that you've already created. You will, however, have to re-create any data in them.
- Temporary Tables cannot be used as "master" tables in a SmartFlow system. (They may be used as replica tables in a SmartFlow system, however.)
- Temporary Tables have restrictions on how they can be used with referential constraints. A Temporary Table may reference another Temporary Table, but may not reference any other type of table (i.e. Transient or persistent). No other type of table may reference a Temporary Table. See table Referential Constraints in this chapter.

With the exceptions of the limitations listed in this section, Temporary Tables behave like normal (persistent) in-memory tables. For example,

- Temporary Tables may have indexes on them.
- Temporary Tables may be used in Views.
- Temporary Tables may have triggers on them.
- Temporary Tables may contain BLOb columns (but the length of those columns is limited to a couple of kilobytes).
- Temporary tables reside in a specific catalog and schema.
- Privileges apply to Temporary Tables; in other words, the creator of the Temporary Table may grant and revoke privileges on the table. The DBA may also grant and revoke privileges on the table. Remember, however, that when a session puts data into a Temporary Table, the data cannot be seen by any other session, even if that session is by a DBA or by a user that has been granted SELECT privilege on the Temporary Table. Therefore, granting privileges on a table merely grants the other user the right to use

your table, not your data. Note that the default privileges on Temporary Tables are the same as the default privileges on persistent tables.

To create a Temporary Table, use the syntax shown below, where "<...>" denotes syntax that is the same as for any other type of table.

```
CREATE [GLOBAL] TEMPORARY TABLE <...>;
```

(For the complete syntax of the CREATE TABLE command, see *solidDB SQL Guide*.)

A Temporary Table is always an in-memory table. If you use the STORE DISK clause, the server will give you an error. If you use STORE MEMORY, or if you omit the STORE clause altogether, the server will create the Temporary Table as an in-memory table.

solidDB's implementation of Temporary Tables fully complies with the ANSI SQL:1999 standard for "Global Temporary Tables". All Temporary Tables of solidDB are global tables regardless of whether the keyword GLOBAL is specified or not. solidDB does not support "Local Temporary Tables", as defined by ANSI.

3.2 Transient Tables

The data in Transient Tables has limited duration. The server keeps the data only until the server shuts down.

Data in Transient Tables has the same "scope" or visibility as data in persistent tables. The data that you insert into a Transient Table can be seen by other users' sessions if those users have appropriate privileges.

Transient Tables have some limitations:

- Transient Table data is not replicated to the Secondary server when you use the HotStandby (CarrierGrade) feature. Note that the Transient Tables themselves (but not their data) are replicated to the HotStandby Secondary server. Thus, if you have to fail over to your Secondary, you do not need to re-create any Transient Tables that you've already created. You will, however, have to re-create any data in them.

- Transient Tables have restrictions on how they can be used with referential constraints. Transient Tables may reference other Transient Tables and may reference persistent tables. They may not reference Temporary Tables. Neither Temporary Tables nor persistent tables may reference a Transient Table. See table Referential Constraints in this chapter.

- Transient Tables cannot be used as "master" tables in a SmartFlow system. (They may be used as replica tables in a SmartFlow system, however.)

With the exception of the limitations listed in this section, Transient Tables behave like "normal" (persistent) in-memory tables. For example:

- Transient Tables may be used in Views.
- Transient Tables may have indexes on them.
- Transient Tables may have triggers on them.
- Transient Tables may contain BLOb columns (but the length of those columns is limited to a couple of kilobytes as is the case with all in-memory tables).
- Privileges apply to Transient Tables.
- Transient tables reside in a specific catalog and schema.

If you export a database with a Transient Table, the data in the Transient Tables will be exported (as will the structure of the tables).

To create a Transient Table, use the syntax shown below, where "<...>" denotes syntax that is the same as for any other type of table.

```
CREATE TRANSIENT TABLE <...>;
```

(For the complete syntax of the CREATE TABLE command, see *solidDB SQL Guide*.)

Transient Tables are always in-memory tables. If you use the STORE DISK clause, the server will give you an error. If you use STORE MEMORY, or if you omit the STORE clause altogether, the server will create the Transient Table as an in-memory table.

Note that the server actually stores the definition of the Transient Table (but not the data) in the server's system tables and keeps that definition even after the server is shut down. If you restart the server later, you will find that the table is still there, but the data is not. Thus, once you create the table, you do not need to create it again. In fact, if you or another user try to create a Transient Table with the same name as an existing Transient Table, you will get an error message, even if the server has been shut down and restarted since the time that the table with that name was originally created. This behavior may be unexpected if you think that a "Transient Table" will disappear as soon as you shut down the server.

Naturally, since a Transient Table persists (even though the data does not), you can use the DROP TABLE command to drop the table after you no longer need it.

You can import data into Transient Tables using the solload utility.

Transient Tables are not defined by the ANSI standard for SQL. Transient Tables are a solidDB extension to the SQL standard.

3.3 Differences Between Temporary Tables and Transient Tables

The main differences between Transient Tables and Temporary Tables are:

- Transient Tables allow all sessions (connections) in the system to see the same data. Temporary Tables allow only the user who created a piece of data to see that data.

  Because users may access the same data, Transient Tables do use concurrency control. Currently, Transient tables support only pessimistic concurrency control ("locking").

  Because Temporary Tables do not use concurrency control, Temporary Tables are faster than Transient Tables.

- The data in Transient Tables lasts until the server is shut down, while data in Temporary Tables lasts only until the user logs out of the session. This means that if one session inserts data into a Transient Table, then other sessions may see that data even after the creator of the data disconnects.

- Data in Transient Tables is exportable using solexp tool. Data in Temporary Tables is not.

- The referential integrity rules for the two table types are different.

  The table below shows which table types are allowed to refer to other types. For example, if a transient table is allowed to have a foreign key that references a persistent table, then you will see "YES" in the cell at the intersection of the row "Transient Child" and the column "Persistent Parent". If the foreign key constraint is not allowed, you will see a dash ("-").

Table 3.1. Referential Constraints

REFERENCED TABLE **REFERENCING TABLE**	**Persistent Disk-based Table**	**Persistent In-Memory Table**	**Transient Table**	**Temporary Table**
Persistent Disk-based Table	YES	YES	-	-
Persistent	YES	YES	-	-

REFERENCED TABLE **REFERENCING TABLE**	**Persistent Disk-based Table**	**Persistent In-Memory Table**	**Transient Table**	**Temporary Table**
In-Memory Table				
Transient Table	YES	YES	YES	-
Temporary Table	-	-	-	Yes

Every type of table may reference itself. In addition, Transient Tables may reference Persistent Tables (but not vice-versa). All other combinations are invalid.

Chapter 4. Optimizing and Tuning the Server

This chapter gives more details about how to optimize and tune the server by using the features available in solidDB In-memory Engine.

4.1 Algorithm for Choosing Which Tables to Store in Memory

If you have enough memory to put some, but not all, tables in memory, here is a strategy that will guide you in choosing which tables to put in memory.

The principle is to take into account the "density" of access. Obviously, the higher the frequency of access, the higher the access "density". Similarly, the larger the table, the lower the access "density" for a given number of accesses per second.

The access density is measured in units of accesses per megabyte per second, which we'll write as rows/MB/s. (For simplicity, we assume one access per row.) For example, if you have a 1 megabyte table, and you access 300 rows in a 10-second period, then the density is

30 rows/MB/s = 300 rows / 1 MB / 10 seconds

As a second example, suppose that you have a 500KB table and you access 300 rows per second. The access density is

600 rows/MB/s = 300 rows / 0.5 MB / second

Clearly, the second table has a higher access density than the first one, and if you can only fit one of these tables into memory than you should put the second one into memory.

This formula is somewhat simplified.

1. You may want to take into account the number of bytes accessed each time. This is typically the average row size, although it may be different if you are using binary large objects, or if the server can find all the information that it needs by reading just an index rather than the entire table.

   Note that because the server normally reads data from the disk in multiples of a "block" (where a block is typically 8KB), the number of bytes per access or the number of bytes per row gives you only slightly more precise figures than the formula without these. Whether you read a 10-byte row or a 2000 byte row, the server does approximately the same amount of work.

2. Remember that when taking into account the size of the table, you must also take into account the size of any indexes on that table. Each time that you add an index, you add more data that is stored about that table. Furthermore, when you add a foreign key constraint to a table, the server will create an appropriate index (if there isn't already one) to speed up certain types of lookup operations on that table. When you calculate the size of your table in memory, you must take into account the table, all its indexes, and all its BLOBs.

Once you have calculated the access density of all your tables, you rank order those tables from highest to lowest. Then, starting with the table that has the highest density, you work your way down the list, designating tables as in-memory tables until you use up all of the available physical memory.

Unfortunately, the process is not this simple because this description assumes that you have perfect information and that you can change a table from disk-based to in-memory (or vice-versa) at any time. In fact, you may not know the total amount of free memory in your computer. You might accidentally designate more in-memory tables than the computer has room in physical memory for. The result may be that tables are swapped to disk. This may substantially reduce performance. Also, you may not really know how frequently each table is accessed until that table has a substantial amount of data in it. Yet the current version of this solidDB server requires that you designate a table as in-memory or disk-based at the time that you create the table, before you have put any data into it. Thus your calculations are going to have to be based on estimates of the amount of usage each table gets, estimates of the size of each table, and estimates of the amount of free memory. It also assumes that the average access density does not change over time.

This approach also assumes that you aren't planning to add still more tables in the future, and it assumes that your tables do not grow in size. In a typical situation, you should not use up all the memory that you have - you should leave enough space to take into account that your tables are likely to grow in size, and you should leave a little bit of a margin for error so that you don't run out of memory.

Nonetheless, this algorithm gives you a basic guide to selecting which tables to put in memory.

Important

Since virtual memory may be swapped to disk frequently, using virtual memory negates the advantage of in-memory tables. Always make sure that the entire DBMS process fits into the physical memory of the computer.

4.2 Performance Tuning Information for Temporary and Transient Tables

Caution

When deciding whether to use Temporary Tables or Transient Tables, keep in mind that data in Temporary and Transient tables is temporary data, not persistent data. The data is not stored to disk. If you are using HotStandby (CarrierGrade), the data is not copied to the HotStandby Secondary server. The data is also not written to the transaction log and therefore cannot be recovered if the server terminates abnormally. Do not use Temporary Tables or Transient Tables if you cannot afford to lose the data in them and re-start your work.

If you are trying to decide between Temporary Tables and Transient Tables, the main things to remember are:

- Data in Temporary Tables does not last past the end of the session; data in Transient Tables lasts until the server is shut down.
- Data in Temporary Tables is not visible to other sessions/connections.
- Temporary Tables are faster than Transient Tables because they do very little checking for concurrency conflicts.

If you don't want to "share" your data and you don't need your data after you disconnect from the current session, then choose Temporary Tables because they have less overhead and higher performance.

4.3 Indexes

If a table is stored in memory, then all indexes on that table are also stored in memory. Not surprisingly, this speeds up performance, but also consumes memory space.

In general, in-memory indexes can be extremely fast, and we recommend that you use them to ensure fast access to the data of the tables.

If you do not have enough memory to store all your tables and indexes in memory, then in some cases adding a particular index may not be the best choice because even though it will speed up some queries, it will slow up other queries by using memory that otherwise could be used to put other tables in memory.

Chapter 5. Configuring In-Memory Database

This chapter describes how to configure solidDB In-memory Engine to meet your environment, performance and operational needs. This description is a subset of the information contained in Chapter 3, "Configuring solidDB," of *solidDB Administration Guide*.

See Appendix C, *Configuration Parameters* for an overview for the full list of available parameters relevant to solidDB In-memory Engine.

5.1 Configuration Files and Parameter Settings

solidDB gets most of its configuration information from the `solid.ini` file. To be more specific, there are two different `solid.ini` configuration files, one on the server and one on the client. Neither configuration file is obligatory. If there is no configuration file, the factory values are used. The `solid.ini` configuration files contain configuration parameters for the client and for the server, respectively. The client-side configuration file is used if the ODBC driver is used and the file must be located in the working directory of the application.

Note

In solidDB documentation, references to `solid.ini` file are usually for the server-side `solid.ini` file.

When solidDB starts, it attempts to open `solid.ini` first from the directory set by the *`SOLIDDIR`* environment variable. If the file is not found from the path specified by this variable or if the variable is not set, the server or client attempts to open the file from the current working directory. (The current working directory is normally the same as the directory from which you started the solidDB server, or a client application. You may specify a different working directory by using the "-c" server command-line option. For more information about command-line options, see *Appendix B, solidDB Command Line Options* in *solidDB Administration Guide*.

The configuration files contain settings for the solidDB parameters. If a value for a specific parameter is not set in the `solid.ini` file, solidDB will use a factory value for the parameter. The factory values may depend on the operating system you are using.

Generally, factory values offer good performance and operability, but in some cases modifying some parameter values can improve performance.

You can modify the configuration by setting parameter name/value pairs in the `solid.ini` file. For example, to specify the network address of the server, you use the parameter name *`Listen`* and an appropriate value, for example,

```
Listen=tcp 192.168.255.1 1315
```

This specifies that when the server listens for client requests, it should listen using the TCP/IP protocol, the network address 192.168.255.1, and the port number 1315.

Parameters are grouped according to section categories in the configuration file. See *Appendix A, Server-Side Configuration Parameters* and *Appendix B, Client-Side Configuration Parameters* in *solidDB Administration Guide*. for an overview of the section categories and all available parameters

Each section category starts with a section name inside square braces, for example:

```
[com]
```

The *`[com]`* section lists communication information. Note that section names are case insensitive. The section names "*`[COM]`*", "*`[Com]`*", and "*`[com]`*" are equivalent.

Below is a sample section from a server-side `solid.ini` configuration file:

```
[IndexFile]
FileSpec_1=C:\soldb\solid1.db 1000M
CacheSize=64M
```

5.2 Managing Server-Side Parameters

You can view and modify solidDB parameters and their values in the following ways:

- Entering the commands:

  ADMIN COMMAND 'parameter'

  and

  ADMIN COMMAND 'describe parameter'

  in SolidConsole or solidDB SQL Editor (teletype).

- Using the Administration→Configuration page in SolidConsole.

  The SolidConsole Configuration page lets you display a parameters listing in a tree node format and change configuration settings through a dialog box. For details, refer to SolidConsole Online Help available by selecting Help on the menu bar.

- Directly, by editing the `solid.ini` file in the solidDB directory.

The sections below contain instructions for managing parameters with ADMIN COMMAND and `solid.ini`.

Note

For details on viewing and setting server communication protocol parameters only, read Chapter 6, "Managing Network Connections" in *solidDB Administration Guide*.

5.2.1 Viewing and Setting Parameters with ADMIN COMMAND

With ADMIN COMMAND, you can change the parameters remotely through a solidDB server without re-starting it. All parameters are accessible even if they are not present in the `solid.ini` configuration file. If the parameter is not present, the factory value is used.

Viewing Parameters

A summary view of many parameters of one parameters may be obtained with the command

```
ADMIN COMMAND 'parameter [-r] [section_name[.parameter_name]]';
```

where:

- -r option specifies that only the current value is required
- *`section_name`* is the category name where the parameter is located in `solid.ini`

To view all parameters, enter the following command in SolidConsole or solidDB SQL Editor (teletype):

ADMIN COMMAND 'parameter';

A list of all parameters with *current*, *default*, and *factory values* is returned. You can restrict the viewed parameters to a specific section by adding a section name, e.g.:

```
ADMIN COMMAND 'parameter logging';
```

You can view the values related a single parameter by giving a full parameter name, like in:

```
admin command 'parameter logging.durabilitylevel';
    RC TEXT
    -- ----
    0 Logging DurabilityLevel 3 2 2
1 rows fetched.
```

The three values shown are (in this order):

- *current value*
- *startup value* that was used when the server was started up
- *factory value* preset in the product

If desired, you can also qualify this command with a -r option to display only the current values. For example:

ADMIN COMMAND 'parameter -r';

Viewing the Description of a Specific Parameter

You can also view a more detailed description of a specific parameter, which includes valid parameter types and access modes. This is useful information, especially because parameters may need to be handled dynamically; parameter support may vary between products, platforms, or releases.

To view a parameter's description, enter the following command using SolidConsole or solidDB SQL Editor (teletype):

```
ADMIN COMMAND 'describe parameter [section_name[.parameter_name]] ';
```

A result set for a single parameter looks like this:

```
admin command 'describe parameter logging.durabilitylevel';
    RC TEXT
    -- ----
    0 DurabilityLevel
    0 Default transaction durability level
    0 LONG
    0 RW
    0 2
    0 3
    0 2
7 rows fetched.
```

The rows of the resultset are:

- *Parameter name* is the name of the parameter, for example *`CacheSize`*.
- *Description* of the parameter
- *Data type*
- *Access mode* that may be one of the following:
  - RO: read-only, the value cannot be changed dynamically
  - RW: read/write, the value may be changed dynamically and the change takes effect immediately
  - RW/STARTUP: the value may be changed dynamically but the change takes effect upon next server startup.
  - RW/CREATE: the value may be changed dynamically but the change takes effect when a new database is created
- *Startup value* displays the parameter's startup value
- *Current value* displays the parameter's current value
- *Factory value* displays the value preset in the product.

Setting a Parameter Value

To set a value for a specific parameter, enter the following command using solidDB SQL Editor (teletype) or SolidConsole:

```
ADMIN COMMAND 'parameter section_name.parameter_name=value [temporary]';
```

where:

`value` is a valid parameter value.

Note

If no value is specified, this sets the parameter with a factory (or unset) value. Furthermore, if you assign a parameter value with an asterisk (*), the parameter will be set to its factory value.

When temporary is set, the changed value is not stored in the `solid.ini` file.

Note that, optionally, you can provide blanks around the equal sign.

Example:

```
--set communication trace on
ADMIN COMMAND 'parameter com.trace = yes';
```

Note

Parameter management operations are not part of a transaction and cannot be rolled back.

The commands return the new value as the resultset. If the parameter's access mode is RO (read-only) or the value entered is invalid, the ADMIN COMMAND statement returns an error.

Persistence of Parameter Modifications

All the changes made to parameters having the access mode RW* are stored in the `solid.ini` file at the next checkpoint. This does not apply to values set with the *`temporary`* option.

It is also possible to request an immediate storing of changed values, with the command:

```
ADMIN COMMAND 'save parameters [ini_file_name]';
```

When *`ini_file_name`* is not specified, the current `solid.ini` file is re-written. Otherwise, a full configuration file is written to a new location. This is a convenient way to save configuration file checkpoints for later use.

5.2.2 Viewing and Setting Parameters in `solid.ini`

1. Open the `solid.ini` file located in the working directory of your solidDB process.

2. View the value of the parameter.

The parameters displayed are the parameters currently active in the server. If you have not set a parameter value, the factory value is used at start-up. The factory value may depend on the operating system that solidDB runs on.

3. If necessary, add the section, the parameter, and the parameter's value.

4. Save the changes.

You must restart the server to activate the changes.

5.2.3 Constant Parameter Values

The parameter access mode for the *`Blocksize`* parameter in the *`IndexFile`* section of the configuration file is RO. The parameter is set when the database is created and cannot be modified afterwards.

If you want to use a different constant value, you have to create a new database. Before creating a new database, set the new parameter constant value by editing the `solid.ini` file in the solidDB directory.

The following example sets a new block size for the index file by adding the following lines to the `solid.ini` file :

```
[IndexFile]
Blocksize = 4096
```

After editing and saving the `solid.ini` file, move or delete the old database and log files, and start solidDB.

Note

The log block size can be changed between startups of the server.

Appendix A. Calculating Maximum BLOB Size

A.1 Purpose

One important difference between in-memory tables and disk-based tables is that column values in in-memory tables must fit into a single "page" (the page size is specified in the `solid.ini` configuration file, and its maximum is 32kB). Therefore, in-memory tables cannot store character or binary files larger than the page size. Smaller binary files, however, are supported.

This appendix shows how to calculate the maximum size of a character or binary column value that will fit in your in-memory tables.

A.2 Background

Many applications today use data that cannot be easily stored in the standard data types INT, CHAR, etc. Instead, a long character or binary format may be better suitable. In these cases, the data may be stored as *CLOBs* and *BLOBs*, *Character* and *Binary Large OBjects*, respectively. A CLOB includes interpretable characters whose number may be up to 2 billion. A BLOB data type can hold virtually any data that can be stored as a series of binary numbers (8-bit bytes). Typically, BLOBs are used to store large, variable-length data that cannot be easily interpreted as numbers or characters. For example, BLOBs may hold digitized sound (e.g. the music on a Compact Disc), multi-media files, or time-series data read from sensors.

In solidDB BLOBs are widely supported and there are several different data types to choose from: BINARY, VARBINARY and LONG VARBINARY, of which the latest is mapped to standard data type BLOB.

CLOB is implemented with six data types, CHAR, WCHAR, VARCHAR, WVARCHAR, LONG VARCHAR and LONG WVARCHAR. The two latest data types are mapped to standard data types CLOB and NCLOB. For detailed information about CLOB and BLOB data types see chapters "Character Data Types" and "Binary Data Types" in Appendix A in *solidDB SQL Guide*.

For disk-based tables, solidDB's implementation of BLOB storage balances speed of access with the need to be able to store large amounts of data. Regardless of the data type (VARCHAR, VARBINARY, etc.), short values are generally stored in the table, while longer values have part or all of their data stored in a separate area in the database storage tree. This is entirely transparent to the user; the user simply decides on the data type, and the solidDB database engine takes care of the rest. Your data will always be accessed the same way, and will appear to be stored in the table, regardless of the actual physical location of the data. In disk-based tables, the maximum length of a VARCHAR or VARBINARY field is 2 gigabytes.

For in-memory tables, BLOB data is stored entirely in the table itself, and the maximum length of a BLOB is limited by the "block size" (no row of an in-memory table may exceed the length of a page or "block"). In this appendix, we give you some information to help you estimate the largest size VARCHAR or VARBINARY data that you can store in an in-memory table.

A.3 Calculating

Please note that the algorithm for calculating the space available for BLOBs is approximate. Make a photocopy of the table below, then fill it in with the values appropriate for your table. Follow the steps to calculate the remaining space available for BLOB data.

Table A.1. Calculating the Space Available for BLOB Data

	VALUE	WHAT TO ENTER IN VALUE	WHAT THE VALUE MEANS
1		In the space to the left, enter either your block size or 32767 (whichever is smaller). The block size will be either the value that you set in the `[IndexFile] BlockSize` `solid.ini` configuration file, or the default documented in *solidDB Administration Guide*.	The block size (page size) is the number of bytes in a "block", analogous to a disk block. Since each row must fit within a block, this represents the maximum size of a row.
2	17	Use the hard-coded value shown to the left.	This is the number of bytes of overhead per page.
3	10	Use the hard-coded value shown to the left.	This is the number of bytes of overhead per row. We'll assume that you have only 1 row per page if you have large BLOBs.
4		If you have declared an explicit primary key for your table, enter the value 10. Otherwise, enter 20.	This represents bytes used for columns that the server automatically adds to each table.
5		Enter the number of columns in your table, multipled by 2.	This is the number of bytes of overhead for the columns.
6		Enter the sum of the sizes of the fixed-size columns of data in your table. (See table #2 below for the size of each fixed-size data type.)	This represents space taken up by fixed-size columns.
7		Enter the number of blob columns.	This is the number of bytes used to terminate BLOB values (1 byte per value).
8		Sum the values in rows 2 through 7.	This is the total space used by everything except the BLOB values.
9		Subtract row 8 from row 1.	This is the approximate number of bytes available for BLOB data. If you have a single BLOB column

	VALUE	WHAT TO ENTER IN VALUE	WHAT THE VALUE MEANS
			in your table, then this is the approximate maximum size of that BLOB value.

Note

NOTE: The maximum block size is 64K; however, the maximum row size (and thus the maximum blob size) is only 32K (actually 32K-1, or 32767). If your block size is 64K or 32K, please enter 32767 instead of the block size in row 1 of the table.

The table below indicates the number of bytes required to store a value of each fixed-size data type. For example, it takes 8 bytes to store a value of type SQL FLOAT.

Table A.2. Number of Bytes Required to Store Values

Data Type	Storage Size (in bytes)
TINYINT	1
SMALLINT	2
INT	4
BIGINT	8
DATE/TIME/TIMESTAMP	11
FLOAT / DOUBLE PRECISION	8
REAL	4
NUMERIC / DECIMAL	11
CHAR / VARCHAR / LONG VARCHAR	char_length(column_value) + 1
WCHAR / WVARCHAR / LONG WVARCHAR	char_length(column_value) * 2 + 1
BINARY / VARBINARY / LONG VARBINARY	octet_length(column_value) + 1

Appendix B. Calculating Storage Requirements

This appendix gives you information that will allow you to estimate how much memory or disk space is required to store a table and its indexes in memory or on disk.

Please note that the formulae given here are not precise for several reasons, including the following:

- solidDB compresses some data.
- Variable-length data (e.g. VARCHAR) requires different amounts of space, depending upon the actual lengths of the values stored.
- The in-memory data structures do not necessarily store the same number of "pointers" for every record.

In most of this discussion, we assume that the data is not compressed, and we assume the maximum number of pointers. Thus the results that you get by using these formulae will usually be somewhat conservative - i.e. the formulae will usually over-estimate the amount of space required.

In the formulae below, the notation

```
sum_of(x)
```

means to take the sum of the sizes of each "x". For example,

```
sum_of(col_size)
```

means to take the sum of the sizes of each of the columns in the table or index, and

```
sum_of(index_sizes)
```

means to take the sum of the sizes of all of the indexes on the table.

B.1 Disk-Based Tables

The general formula for the space required for a disk-based table is:

```
chkpt_factor x (table_size + sum_of(index_sizes))
where
        chkpt_factor is between 1.0 and 3.0 (explained below), and
        table_size =
            1.4 x rows x (sum_of(col_size + 1) + 12)
where
        rows is the number of rows; and
        sum_of(col_size + 1) is the sum of the sizes of the columns
            plus one byte per column.
The column sizes are shown in a table later.
```

For each disk-based index, the index_size is

```
1.4 x rows x (pkey_size + idx_size)
```

where pkey_size is the sum of the sizes of the columns in the primary key, and idx_size is the sum of the sizes of the columns in the index.

The chkpt_factor is needed to take into account that "checkpoint" operations may briefly require up to three times the size of the database. During a checkpoint operation, a copy of each of the changed pages in the database is copied from memory to the disk. If every page in the database has been updated, then it's possible to copy as many pages from memory as there already are on disk. Furthermore, the most recent successful checkpoint is not deleted until the current checkpoint is successfully completed. Therefore, during a checkpoint the disk may simultaneously have up to 3 copies of each page (1 copy for the page in the database, 1 copy in the most recent successful checkpoint, and 1 copy for the current checkpoint while it is executing). The checkpoint factor therefore can be between 1.0 and 3.0. Values approaching 3.0 are rare in most databases. A value of 1.5 is usually well sufficient even for small databases that have high levels of activity. Note that the less frequent the checkpoint, the larger the chkpt_factor may need to be.

Note

In a disk-based index, if you do not explicitly define a primary key, then the server uses a server-generated "row number" as the primary key. This forces the primary key index to store records in the same order that they were inserted.

B.2 In-Memory Tables

The general formula for an in-memory table is `table_size + sum_of(index_sizes)`

`table_size` =

```
1.3 x rows x (sum_of(col_sizes) + (3 x word_size) + (2 * num_cols) + 2)
```

where: rows is the number of rows;

`word_size` is the machine word size (e.g. 4 bytes for 32-bit OS and 8 bytes for 64-bit OS);

`num_cols` is the number of columns; and

sum_of(`col_sizes`) is the sum of the sizes of the columns.

For each in-memory index, the index size is

```
1.3 x rows x ((dist_factor x sum_of(col_sizes + 1)) + (8 x word_size) + 4)
```

where "dist_factor" is a value between 1.0 and 2.0 that depends upon the distribution of the key values. If key values are highly dissimilar, then use a value closer to 2.0. If key values are highly similar, then use a value closer to 1.0.

B.3 Table of Column Sizes

TINYINT: 2 bytes

SMALLINT: 2 bytes

INT: 4 bytes

BIGINT: 8 bytes

DATE/TIME/TIMESTAMP: 11 bytes

FLOAT / DOUBLE PRECISION: 8 bytes

REAL: 4 bytes

NUMERIC / DECIMAL: 12 bytes

CHAR / VARCHAR / LONG VARCHAR: char_length(column_value) + 5

WCHAR / WVARCHAR / LONG WVARCHAR: char_length(column_value) * 2 + 5

BINARY / VARBINARY / LONG VARBINARY: octet_length(column_value) + 5

B.4 Measuring Memory Consumption

After you have created your tables and indexes, you can measure the actual amount of memory consumed by using the command:

ADMIN COMMAND 'info imdbsize';

This command gives the total memory consumption of in-memory tables and indexes. The units are kilobytes.

B.5 Details

This chapter contains some detailed information about how the data is stored in the different storage trees. You may find this helpful, if you would like a better understanding of the basis for the preceding formulae.

B.5.1 Disk-Based Tables

On disk-based tables, data and indexes are stored in a B-tree. Each entry in the tree consumes space for the header and the data.

The space used by the actual data can be calculated using the table of column sizes shown earlier. The values in that table are the maximum lengths. Variable-length data (e.g. VARCHAR) or compressible data may require fewer bytes.

In addition, in disk-based tables, the server requires 1 additional byte per column; this byte is used as part of the length indicator (which also serves as a null indicator).

The header for each row uses 12 bytes:

Table B.1. Header Bytes

Number of Bytes	**Used for...**
3 bytes	Row header
3 bytes	Table id
6 bytes	Row version

If a disk-based table contains indexes (other than the primary key), the size of the entries in those indexes must be estimated separately using the same guideline. An index entry contains the following components:

- columns that are defined in the index
- columns of the primary key of the table
- a row header (12 bytes)

Additionally, there is usually some empty space (e.g. 20 - 40%) in the database pages. This is why the formulae include a multiplier of 1.4 for both tables and indexes.

For example: We have a disk-based table:

```
CREATE TABLE subscriber (
             id INTEGER NOT NULL PRIMARY KEY,
             name VARCHAR(50),
             salary FLOAT) ;
```

Additionally, we have created a secondary index:

```
CREATE INDEX subscriber_idx_name ON subscriber (name);
```

The index entry contains the NAME column; it also contains the primary key column, which in this case is ID. The space required by that index should be estimated separately. The total size of the disk-based table, assuming the "empty space factor" is 1.4, would be:

```
rows x 1.4                   // 1.4 = the empty space estimate.
   x ( (12 + 4 + (50+5) + 8 + 3)   // size of the table entry,
       + (12 + 4 + (50+5) + 2) )   // size of the secondary index entry
```

Representing this differently,

```
                   space required for          space required for
                   one row in table            one row in index
                           |                          |
                 -------------------------   ---------------------
                 |                       |   |                   |
                 |                       |   |                   |
  rows x 1.4 x ( (12 + 4 + (50+5) + 8 + 3) + (12 + 4 + (50+5) + 2) )
                   |   |    |  |    |   |      |   |    |  |    |
row header size <--    |    |  |    |   |      |   |    |  |    |
size of INT <----------     |  |    |   |      |   |    |  |    |
size of VARCHAR(50) <-------   |    |   |      |   |    |  |    |
VARCHAR overhead   <-----------     |   |      |   |    |  |    |
size of FLOAT  <--------------------    |      |   |    |  |    |
length indicators (1 byte per col) <----       |   |    |  |    |
row header size (in index) <-------------------    |    |  |    |
size of INT  <-------------------------------------     |  |    |
size of VARCHAR(50)  <-----------------------------------  |    |
VARCHAR overhead   <---------------------------------------     |
length indicator bytes (1 per column)   <-----------------------
```

B.5.2 In-Memory Tables

The space required by in-memory tables is estimated differently.

The size of each entry is the combined size of the data of the table plus three memory pointers (4 bytes each in 32-bit OS or 8 bytes each in 64-bit OS) of overhead per row. Additionally, there is overhead of two bytes per row and two bytes per each column of the row. Note that you do not need to add 1 byte per column to take into account the length indicator; that is included in the 2 bytes per row.

In addition, the main memory tables may have indexes, which are populated upon server startup. Each index entry contains the data of the columns defined in the index. Additionally, each index entry contains up to eight memory pointers. (Note that a copy of the primary key is NOT required for an in-memory index.)

Furthermore, there is some other overhead that depends on the actual data values of the index. This is a percentage of the data size of the index. An exact value cannot be given exactly because it depends on the key value distribution, but the multiplier ranges between 1.0 and 2.0.

Additionally, the index structure itself needs an average of 4 bytes per index entry (i.e. per row).

For the above example table and index, the memory consumption in a 32-bit operating system can be estimated to be:

```
rows * 1.3 x (
  ((3 x 4) + 2 + (4 + 2) + (50+5+2) + (8+2)) // Size of data in table
+ ((8 x 4) + 4 + 1.2 x 4))       // Size of the primary key index
+ ((8 x 4) + 4 + 1.2 x (50+5)))  // Size of the secondary index.
)
```

```
((3 x 4) + 2 + (4 + 2)+(50+5+2)+(8+2)) // Size of data in the table.
  |   |    |    |   |   |  | |   | |
  |   |    |    |   |   |  | |   |  -> 2 bytes per col
  |   |    |    |   |   |  | |    ---> 8 bytes for FLOAT (Salary)
  |   |    |    |   |   |  |  -------> 2 bytes per col
  |   |    |    |   |   |   ---------> 5 bytes overhead per VARCHAR
  |   |    |    |   |    ------------> 50 bytes for VARCHAR(50)
  |   |    |    |    ----------------> 2 bytes per col
  |   |    |     --------------------> 4 bytes for INT (ID)
  |   |     -------------------------> 2 bytes per row
  |    ------------------------------> pointer size (4 bytes on 32-bit OS)
   ----------------------------------> 3 pointers

((8 x 4) + 4 + 1.2 x 4))     // Size of the primary key index
  |   |    |    |    |
  |   |    |    |     ---------------> 4 bytes for INT
  |   |    |     --------------------> 1.2 index value distribution factor
  |   |     -------------------------> 4 bytes per index entry
  |    ------------------------------> pointer size (4 bytes on 32-bit OS)
   ----------------------------------> 8: up to 8 pointers


((8 x 4) + 4 + 1.2 x (50+5)))  // Size of the secondary index.
  |   |    |    |     |  |
  |   |    |    |     |   -----------> 4 bytes for VARCHAR overhead
  |   |    |    |      --------------> 50 bytes for VARCHAR(50)
  |   |    |     --------------------> 1.2 index value distribution factor
  |   |     -------------------------> 4 bytes per index entry
  |    ------------------------------> pointer size (4 bytes on 32-bit OS)
   ----------------------------------> 8 up to 8 pointers
```

In a 64-bit operating system, use a memory pointer size of 8 bytes rather than 4 bytes.

The factor 1.2 in the above estimate is the "TRIE index value distribution factor" whose exact value depends on the actual values of the indexed column. Its value is typically between 1 and 2. With random value distribution, the value is closer to 2.0 and with sequential value distribution, it is closer to 1.0. The 4 bytes is the data overhead needed by an index entry on average.

The factor of 1.3 is to take into account the internal overhead of the memory allocator.

Note

Indexes of main memory tables are created dynamically each time that the server starts; they are never written to disk and therefore they don't occupy any disk space. However, the tables themselves are written to disk when the server shuts down (and during checkpoints), so the total amount of disk space that you have must be enough to store both the disk-based tables and the in-memory tables.

Appendix C. Configuration Parameters

Parameters are grouped according to section categories in the `solid.ini` configuration file. The parameters related to the solidDB In-memory Database are stored into the *[MME]* section of the configuration file. Additionally one parameter, *DefaultStoreIsMemory*, is stored into the *[General]* section.

You can change configuration parameters in either by manually editing the `solid.ini` configuration file or by entering the following command in solidDB SQL Editor (or in SolidConsole):

```
ADMIN COMMAND 'parameter section_name.param_name=value'
```

For example:

ADMIN COMMAND 'parameter mme.imdbmemorylimit=1gb';

Note

The server reads the configuration file only when it starts, and therefore changes to the configuration file do not take effect until the next time that the server starts.

The complete list of IMDB-related configuration parameters is presented below.

C.1 General Section

Table C.1. MME Parameters

[General]	**Description**	**Factory Value**	**Access Mode**
DefaultStore-IsMemory	If set to Yes, new tables are created as in-memory tables if they are created without an explicit STORE clause in the CREATE TABLE statement. If set to No, then by default new tables are stored on disk. You can override the default value by using the STORE clause in the CREATE TABLE statement. This parameter only applies to products that support solidDB In-memory Engine.	No	RW

[General]	Description	Factory Value	Access Mode
	Note that system tables are stored on disk, even if this parameter is set to Yes.		

C.2 MME Section

Note

The *DefaultStoreIsMemory* parameter (in the *[General]* section of the `solid.ini` file) is also related to solidDB In-memory Engine capability. For more information, see Section C.1, “General Section”.

Table C.2. MME parameters

[MME]	Description	Factory Value	Access Mode

ImdbMemoryLimit	This sets an upper limit on the amount of memory (virtual memory) that the server will allocate for in-memory tables and indexes on in-memory tables. Note that "in-memory tables" includes Temporary Tables and Transient Tables, as well as "normal" (persistent) in-memory tables.
	
	The limit may be specified in bytes, kilobytes (kb), megabytes (mb), or gigabytes (gb). For example:
	
	`ImdbMemoryLimit=1073741824`
	`ImdbMemoryLimit=1048576kb`
	`ImdbMemoryLimit=1024MB`
	`ImdbMemoryLimit=1GB`
	
	If you use the value 0, it means "no limit".
	
	As a general rule, for servers with 1GB or less of memory, the maximum amount that you should allocate to in-memory tables is usually 30% - 70% of the system's physical memory. The more memory the system has, the
		0
		
		Unit: 1 byte k=KB m=MB g=GB
			RW

[MME]	**Description**	**Factory Value**	**Access Mode**
	larger the percentage of it you may use for in-memory tables. For more details about controlling memory usage of in-memory tables, see *solidDB In-Memory Database User Guide*. Note: This parameter only applies only to solidDB In-memory Engine tables. It does not apply to other versions of solidDB or to disk-based tables. You can change this with the command: `ADMIN COMMAND 'parameter MME.ImdbMemoryLimit=n[kb|mb|gb]';` where 'n' is a positive integer. You may only increase, not decrease, this value while the server is running. The command takes effect immediately. The new value is written back to the `solid.ini` file at shutdown. **Caution** We strongly recommend that you ensure that your in-memory tables will fit within the available physical memory. If you exceed the amount of physical memory available, performance will decrease significantly. If you use up all of the available virtual memory, the server will abruptly limit inserts, updates, etc. and will return error codes.		
ImdbMemoryLowPercentage	Once you have set *ImdbMemoryLimit*, you may set this additional parameter to give you advance warning before you use up all of memory. This *ImdbMemoryLowPercentage* parameter allows you to indicate what percentage of memory you may use before the server	90	RW/Startup

[MME]	Description	Factory Value	Access Mode
	starts limiting your ability to insert rows into in-memory tables, etc. For example, if *ImdbMemoryLimit* is 1000MB and *ImdbMemoryLowPercentage* is 90 (percent), then the server will stop accepting inserts when you've used up 900 megabytes of memory for your in-memory tables. Valid values are between 60 and 99 (percent). For more details about controlling memory usage of in-memory tables, see *solidDB In-Memory Database User Guide*. **Note** This parameter only applies to solidDB In-memory Engine tables. It does not apply to other versions of solidDB or to disk-based tables.		
LockEscalationEnabled	Typically, when the server needs to use locks to prevent concurrency conflicts, the server locks individual rows. This means that each user affects only those other users who want to use the same row(s). However, the more rows are locked, the more time the server must spend checking for conflicting locks. In some cases, it is worthwhile to lock an entire table rather than a large number of the rows in that table. When *LockEscalationEnabled* is set to yes, the lock level is escalated from row-level to table-level after a specified number of rows (in the same table) have been locked within the current transaction. Lock escalation improves performance, but reduces concurrency, because it means that other users are temporarily unable to use the same table, even if they want to use different rows within that table. See the parameter *LockEscalationLimit*. The value may be "yes" or "no".	yes	RW/Startup

[MME]	**Description**	**Factory Value**	**Access Mode**
	Note This parameter applies to in-memory tables only.		
`LockEscalationLimit`	If *`LockEscalationEnabled`* is set to yes, then this parameter indicates how many rows must be locked (within a single table) before the server will escalate lock level from row-level to table-level. (See *`LockEscalationEnabled`* for more details.) The value may be any number from 1 to 2,147,483,647 (2^32-1). **Note** This parameter applies to in-memory tables only.	1000	RW/Startup
`LockHashSize`	The server uses a hash table (array) to store lock information. If the size of the array is remarkably underestimated the performance degrades. Too large hash table doesn't affect directly to the performance although it causes memory overhead. The *`LockHashSize`* determines the number of elements in hash table. This information is needed when the server is using pessimistic concurrency control (i.e. locking). The server uses separate arrays for in-memory tables and disk-based tables. This parameter applies to in-memory tables. In general, the more locks you need, the larger this array should be. However, it is difficult to calculate the number of locks that you need, so you will probably need to experiment to find the best value for your applications. The value that you enter is the number of hash table entries. Each table entry has a size of one pointer (4 bytes in 32-bit architectures). Thus, for example, if you choose a hash table size of 1,000,000, then the amount of memory required is 4,000,000 bytes (assuming 32-bit pointers).	1000000	RW/Startup

[MME]	**Description**	**Factory Value**	**Access Mode**
MaxCacheUsage	The value of *MaxCacheUsage* limits the amount of D-table cache used while checkpointing M-tables. The value is expected to be given in bytes. Regardless of the value of the *MaxCacheUsage* at most half of the D-table cache (*IndexFile.CacheSize*) is used for checkpointing M-tables. Value *MaxCacheUsage=0* sets the value unlimited, which means that the cache usage is *IndexFile.CacheSize/2*.	8MB	RW/Startup
ReleaseMemoryAtShutdown	When set to "yes", this tells the server that when it shuts down it should explicitly release memory used by in-memory tables, rather than relying on the operating system to clean up all memory associated with this process. Some operating systems (like VxWorks) may require you to set this to "yes" to ensure that all memory is released. The possible values are yes and no. The factory value is no because shutting down the server is faster that way.	No	RW/Startup

Glossary

This glossary gives you a description of terms used in this document. Note that *solidDB Administration Guide* contains a more extensive glossary.

D

D-table
: In some error messages, the term "D-table" is used as shorthand for "disk-based table".

I

Isolation Level
: See Transaction Isolation Level.

L

Log file (Transaction log)
: This file holds a log of committed operations executed by the database server. If a system crash occurs, the database server uses this log to recover all data inserted or modified after the latest checkpoint.

M

M-table
: In some error messages, the term "M-table" is used as shorthand for "in-memory table".

T

Transaction Isolation Level
: When multiple users are using a database at the same time, one user's changes should only be visible to other users in controlled ways. For example, you might choose the "COMMITTED READ" isolation level, which means that you do not want to see any other user's changes (e.g. new records) that have not yet been committed yet. Or you might choose an isolation level that guarantees that if you look at the same table repeatedly in the same transaction, then you will see the same records each time. The ANSI standard for SQL defines 4 different levels of isolation. These are discussed in *solidDB Administration Guide* and are defined in the ANSI standard for SQL.

Index