

HP 9000 Computer Systems
ALLBASE/SQL
Performance and Monitoring Guidelines



HP Part No. 36217-90185
Printed in U.S.A. April 1994

First Edition
E0494

Copyright © 1987, 1988, 1989, 1991, 1992, 1993, 1994 by Hewlett-Packard Company.

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DoD U.S. Government Departments and agencies are as set forth in FAR 52.227-19 (c) (1,2).

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Restricted Rights Legend

Printing History

This is the first edition of the *ALLBASE/SQL Performance and Monitoring Guidelines*. It is compatible with release G.0 of ALLBASE/SQL. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions. Prior to release G.0, chapters one through five were published as the *ALLBASE/SQL Performance Guidelines*. Because four new chapters were added to document SQLMON, this manual is now called the *ALLBASE/SQL Performance and Monitoring Guidelines*.

ALLBASE/SQL Documents

The following table lists all manuals in the ALLBASE/SQL document set:

Title	Part Number
<i>ALLBASE/ISQL Reference Manual</i>	36217-90188
<i>ALLBASE/NET User's Guide</i>	36217-90093
<i>ALLBASE/SQL Advanced Application Programming Guide</i>	36217-90186
<i>ALLBASE/SQL C Application Programming Guide</i>	36217-90014
<i>ALLBASE/SQL COBOL Application Programming Guide</i>	36217-90058
<i>ALLBASE/SQL Database Administration Guide</i>	36217-90005
<i>ALLBASE/SQL FORTRAN Application Programming Guide</i>	36217-90013
<i>ALLBASE/SQL Message Manual</i>	36217-90009
<i>ALLBASE/SQL Pascal Application Programming Guide</i>	36217-90007
<i>ALLBASE/SQL Performance and Monitoring Guidelines</i>	36217-90185
<i>ALLBASE/SQL Reference Manual</i>	36217-90001
<i>HP ALLBASE/QUERY User's Guide</i>	92534-64001
<i>HP PC API User's Guide for ALLBASE/SQL</i>	36217-90187
<i>Up and Running with ALLBASE/SQL</i>	36389-90011

Preface

This book presents a series of guidelines for use in monitoring and tuning the performance of your ALLBASE/SQL G.0 system. These guidelines are applicable for the HP-UX 9.0 release. The chapters in this book are listed below:

- Chapter 1, “Basic Concepts in ALLBASE/SQL Performance” describes some very basic concepts to help you understand how to tune performance.
- Chapter 2, “Guidelines on Logical and Physical Design” describes how your database design and file storage affect performance.
- Chapter 3, “Guidelines on Query Design” shows how to write queries for best performance.
- Chapter 4, “Guidelines on Transaction Design” describes techniques for obtaining the greatest concurrency.
- Chapter 5, “Guidelines on System Administration” describes how to improve performance in the areas of database, network, and operating system administration.
- Chapter 6, “Getting Started With SQLMON” describes the basic operations of SQLMON, the performance monitoring tool.
- Chapter 7, “Troubleshooting with SQLMON” provides examples on how to use SQLMON to troubleshoot ALLBASE/SQL performance situations.
- Chapter 8, “SQLMON Screen Reference” documents all of the SQLMON screens in alphabetical order.
- Chapter 9, “SQLMON Command Reference” describes each of the SQLMON commands, with syntax and examples.
- Appendix A, “Design for a High-Performance Interactive Table Editor,” sketches an approach to table editing that avoids concurrency problems and promotes high performance through the use of BULK operations.

You can find additional basic information relating to database performance in the following chapters in the ALLBASE/SQL document set:

- “Logical Design” and “Physical Design” in the *ALLBASE/SQL Database Administration Guide*.
- “Concurrency Control” in the *ALLBASE/SQL Reference Manual*.
- “Transaction Management with Multiple DBEnvironment Connections” in the *ALLBASE/SQL Advanced Application Programming Guide*.
- “Processing with Cursors” in the ALLBASE/SQL application programming guide for your programming language.

What's New in This Release

The following table highlights the new or changed functionality in this release, and shows you where each feature is documented.

New Features in ALLBASE/SQL Release G.0

Feature (Category)	Description	Documented in . . .
Stored procedures (Usability)	Provides additional stored procedure functionality for application programs. Allows declaration of a procedure cursor and fetching of multiple rows within a procedure to applications. New statement: ADVANCE. Changed syntax: CLOSE, CREATE PROCEDURE, DECLARE CURSOR, DESCRIBE, EXECUTE, EXECUTE PROCEDURE, FETCH, OPEN.	<i>ALLBASE/SQL Reference Manual</i> , "SQL Statements" and "Using Procedures" in "Constraints, Procedures and Rules;" <i>ALLBASE/SQL Advanced Application Programming Guide</i> , "Using Procedures in Application Programs."
Case insensitivity (Usability)	Adds an optional attribute to the character and varchar type column attributes of tables. Allows search and compare of these columns in a case insensitive manner. Four new SQLCore data types are added. Changed syntax: ALTER TABLE, CREATE TABLE.	<i>ALLBASE/SQL Reference Manual</i> , "Comparison Predicate" in "Search Conditions," CREATE TABLE in "SQL Statements."
Support for 1023 columns (Usability)	Increases the maximum number of columns per table or view to 1023. Increases maximum sort columns and parameters in a procedure to 1023.	<i>ALLBASE/SQL Reference Manual</i> , CREATE TABLE and CREATE VIEW in "SQL Statements;" <i>ALLBASE/SQL Database Administration Guide</i> , "ALLBASE/SQL Limits" appendix.
ISQL HELP improvements (Usability)	Gives help for entire command instead of only the verb.	<i>ALLBASE/ISQL Reference Manual</i> , HELP in "ISQL Commands."
EXTRACT command (Usability)	Extracts modules from the database and stores them in a module file. Allows for creation of a module file at any time based on the current DBEnvironment without preprocessing. New command: EXTRACT. Changed syntax: INSTALL.	<i>ALLBASE/ISQL Reference Manual</i> , "Using Modules" in "Using ISQL for Database Tasks," EXTRACT, INSTALL in "ISQL Commands."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
New SQLGEN GENERATE parameters (Usability)	Generates SQL statements necessary to recreate modified access plans for module sections. New syntax for GENERATE: DEFAULTSPACE, MODOPTINFO, PARTITION, PROCOPTINFO, SPACEAUTH.	<i>ALLBASE/SQL Database Administration Guide</i> , "SQLGEN Commands" appendix.
Row level locking (Usability)	Permits multiple transactions to read and update a table concurrently because locking is done at row level. Since the transaction will obtain more locks, the benefits must be weighed against the costs. (Previously documented in an addendum after F.0 release.)	<i>ALLBASE/SQL Reference Manual</i> , "Concurrency Control through Locks and Isolation Levels;" <i>ALLBASE/SQL Database Administration Guide</i> , "Effects of Page and Row Level Locking" in "Physical Design."
Increased number of users (Usability)	Removes the limitation of 240 users supported by pseudotables. (Maximum is system session limits: 2000 on HP-UX; 1700 on MPE/iX.)	<i>ALLBASE/SQL Database Administration Guide</i> , "ALLBASE/SQL Limits" appendix.
POSIX support (Usability)	Improves application portability across MPE/iX and HP-UX. Enhances the ALLBASE/SQL preprocessors to run under POSIX (Portable Operating System Interface) on MPE/iX.	<i>ALLBASE/SQL Advanced Application Programming Guide</i> , "POSIX Preprocessor Invocation" in "Using the Preprocessor."
Application thread support (Performance, Usability)	Provides the use of threads in an application. Allows ALLBASE/SQL to be used in an application threaded environment on MPE/iX. Application threads are light weight processes that share some resources and last for the duration of a transaction. Threaded applications reduce the overhead of context switching and improve the performance of OpenTP applications.	<i>ALLBASE/SQL Advanced Application Programming Guide</i> , "Using the Preprocessor."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
High Availability	Provides a collection of features to keep systems available nonstop including: Partial STORE and RESTORE, Partial rollforward recovery, DBEFiles in different groups (MPE/iX), detaching and attaching database objects, CHECKPOINT host variable, changing log files, console messages logged to a file, generating fewer log records by using TRUNCATE TABLE to delete rows, and new system catalog information. See the following features for new and changed syntax.	<i>ALLBASE/SQL Reference Manual</i> , “SQL Statements;” <i>ALLBASE/SQL Database Administration Guide</i> , “Maintaining a Nonstop Production System” in “Maintenance” chapter and “SQLUtil” appendix.
Partial rollforward recovery (High Availability)	Supports partial rollforward recovery through PARTIAL option on SETUPRECOVERY. Used to recover specific DBEFiles while allowing access to other DBEFiles.	<i>ALLBASE/SQL Database Administration Guide</i> , “Backup and Recovery” chapter and SETUPRECOVERY PARTIAL in “SQLUtil” appendix.
Partial STORE and RESTORE (High Availability)	Gives more flexibility in backup and recovery strategies by allowing partial store and restore of DBEFiles, DBEFileSets or combinations of both. See “New and changed SQLUtil commands for increased availability” later in this table.	<i>ALLBASE/SQL Database Administration Guide</i> , “Backup and Recovery” chapter and “SQLUtil” appendix.
DBEFile group change on MPE/iX (High Availability)	Manages DBEFiles so they can be placed in a particular group or on a particular volume (MPE/iX). Use either CREATE DBEFILE or MOVEFILE.	<i>ALLBASE/SQL Reference Manual</i> , CREATE DBEFile in “SQL Statements;” <i>ALLBASE/SQL Database Administration Guide</i> , “Maintaining a Nonstop Production System” in “Maintenance” chapter and MOVEFILE in “SQLUtil” appendix.
Detaching and attaching database objects (High Availability)	Detaches or attaches a DBEFile or DBEFileSet from the DBEnvironment. This is useful for data that is accessed infrequently such as tables containing historical data only. New SQLUtil commands: DETACHFILE, ATTACHFILE.	<i>ALLBASE/SQL Database Administration Guide</i> , “Maintaining a Nonstop Production System” in “Maintenance” chapter and DETACHFILE, ATTACHFILE in “SQLUtil” appendix.

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
New and changed SQLUtil commands for increased availability (High Availability)	Adds support for high availability and System Management Intrinsic. Intended for non-stop, continuously available operations. New SQLUtil commands: ATTACHFILE, CHANGELOG, DETACHFILE, RESTORE PARTIAL, STORE PARTIAL, STOREINFO, STOREONLINE PARTIAL, WRAPDBE. Modified SQLUtil commands: MOVEFILE, RESTORE, RESTORELOG, SHOWDBE, SETUPRECOVERY, STORE, STORELOG, STOREONLINE.	<i>ALLBASE/SQL Database Administration Guide</i> , “SQLUtil” appendix.
List files on backup device (High Availability)	Lists physical names of files stored on backup device with new SQLUtil command: STOREINFO.	<i>ALLBASE/SQL Database Administration Guide</i> , “Backup and Recovery” chapter and STOREINFO in “SQLUtil” appendix.
Log file improvements (High Availability)	Allows changing log files, switching of console messages to a file, and gives advance warning for log full. Increased maximum size of a single DBE log file to 4 gigabytes. A DBEnvironment can have up to 34 log files configured. Changed syntax: CHECKPOINT. New SQLUtil command: CHANGELOG.	<i>ALLBASE/SQL Reference Manual</i> , CHECKPOINT in “SQL Statements;” <i>ALLBASE/SQL Database Administration Guide</i> , “Maintaining a Nonstop Production System” in “Maintenance” chapter, CHANGELOG in “SQLUtil” appendix, and “ALLBASE/SQL Limits” appendix.
New SET SESSION and SET TRANSACTION statements (Standards, Performance)	Provides additional flexibility and improved performance. Allows setting and changing transaction and session attributes.	<i>ALLBASE/SQL Reference Manual</i> , SET SESSION and SET TRANSACTION in “SQL Statements.”
FIPS flagger (Standards)	Meets Federal Information Processing Standard (FIPS) 127.1 flagger support. Flags non-standard statement or extension. Invoked with a flagger option in the preprocessor command line or the SET FLAGGER command in ISQL. Updatability rules are different when flagger is invoked. New syntax: DECLARE CURSOR, WHENEVER. Changes to C and COBOL host variable declaration.	<i>ALLBASE/SQL Reference Manual</i> , DECLARE CURSOR in “SQL Commands” and “Standards Flagging Support” appendix; <i>ALLBASE/SQL Advanced Application Programming Guide</i> , “Flagging Non-Standard SQL with the FIPS Flagger;” <i>ALLBASE/ISQL Reference Manual</i> , SET in “ISQL Commands.”

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
Optimizer enhancement (Performance)	Uses a more efficient algorithm that significantly reduces the time to generate the access plan.	<i>ALLBASE/SQL Performance and Monitoring Guidelines</i> , "Optimization" in "Basic Concepts in ALLBASE/SQL Performance."
Access plan modification (Performance)	Allows modification of access plans for stored section to optimize performance. View the plan with SYSTEM.SETOPTINFO. New statement: SETOPT.	<i>ALLBASE/SQL Reference Manual</i> , SETOPT in "SQL Statements;" <i>ALLBASE/SQL Database Administration Guide</i> , SYSTEM.SETOPTINFO in "System Catalog."
Syntax added to disable access plan optimization (Performance, Usability)	Specifies that the optimization information in the module file is not to be used. Changed syntax: EXTRACT, INSTALL, VALIDATE.	<i>ALLBASE/SQL Reference Manual</i> , VALIDATE in "SQL Statements;" <i>ALLBASE/ISQL Reference Manual</i> , EXTRACT, INSTALL in "ISQL Commands."
Application Development Concurrency (Performance, Usability)	Provides enhancements to improve preprocessing performance when simultaneously accessed by multiple users. Page or row level locking on any system base table and processing without storing sections. See the related features in this table. New SQL parameter: SET DEFAULT DBEFileSet. SQL changed syntax: ALTER TABLE, GRANT, REVOKE, UPDATE STATISTICS. ISQL changed syntax: INSTALL. Changed SYSTEM and CATALOG view. New STOREDSECT tables. Special owners HPRDBSS and STOREDSECT. Changed syntax for Full Preprocessing Mode.	<i>ALLBASE/SQL Reference Manual</i> , "Names" and "SQL Statements;" <i>ALLBASE/SQL Advanced Application Programming Guide</i> , "Using the Preprocessor;" <i>ALLBASE/ISQL Reference Manual</i> , "ISQL Commands;" <i>ALLBASE/SQL Database Administration Guide</i> , "Database Creation and Security" and "System Catalog."
System Catalog tables (Performance)	Provides greater concurrency by allowing users to specify table, page, or row level locking of any system table owned by STOREDSECT through the ALTER TABLE statement.	<i>ALLBASE/SQL Reference Manual</i> , "Names;" <i>ALLBASE/SQL Database Administration Guide</i> , "System Catalog."
Preprocessors (Performance)	Allows optional specification of a DBEFileSet for storage of sections. Allows preprocessing without storing sections in DBEnvironment.	<i>ALLBASE/SQL Advanced Application Programming Guide</i> , "Using the Preprocessor."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
I/O performance improvement (Performance)	Optimizes I/O for initial load, index build, serial scans, internal data restructuring, file activity, pseudo mapped files and temporary files. See the following features for new and changed syntax.	<i>ALLBASE/SQL Reference Manual</i> , "SQL Statements."
TRUNCATE TABLE statement (Performance)	Deletes all rows in a specified table leaving its structure intact. Indexes, views, default values, constraints, rules defined on the table, and all authorizations are retained. TRUNCATE TABLE is faster than the DELETE statement and generates fewer logs. New statement: TRUNCATE TABLE.	<i>ALLBASE/SQL Reference Manual</i> , TRUNCATE TABLE in "SQL Statements."
New scans (Performance)	Reads tables with a new parallel sequential scan. The tables are partitioned and files are read in a round robin fashion to allow OS prefetch to be more effective. Allows the I/O for a serial scan to spread across multiple disc drives.	<i>ALLBASE/SQL Performance and Monitoring Guidelines</i> , "Using Parallel Serial Scans" in "Guidelines on Query Design."
Load performance improvement (Performance)	Improves performance with new SET and SET SESSION attributes, a new binary search algorithm, and deferred allocation of HASH pages. New attributes for SET SESSION statement: FILL, PARALLEL FILL.	<i>ALLBASE/SQL Reference Manual</i> , SET SESSION in "SQL Statements."
ISQL enhanced to improve the performance of LOADs (Performance)	Uses new parameters of the ISQL SET command to set load buffer size and message reporting. Improves load performance. Choose a procedure, command file, or new ISQL command to set constraints deferred, lock table exclusively, and set row level DML atomicity. Changed syntax: SET (see the following feature).	<i>ALLBASE/ISQL Reference Manual</i> , SET in "ISQL Commands."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
Modified SET options (Performance)	Provides better performance for LOADs and UNLOADs. Specify buffer size, status reporting for LOAD/UNLOAD or exclusive lock for data table. AUTOSAVE row limit increased to 2147483647. New and changed SET options: LOAD_BUFFER, LOAD_ECHO, AUTOLOCK, AUTOSAVE.	<i>ALLBASE/ISQL Reference Manual</i> , SET in “ISQL Commands;” <i>ALLBASE/SQL Performance and Monitoring Guidelines</i> , “Initial Table Loads” in “Guidelines on Logical and Physical Design.”
SQLMON (Tools)	Monitors the activity of ALLBASE/SQL DBEnvironment. Provides information on file capacity, locking, I/O, logging, tables, and indexes. Summarizes activity for entire DBEnvironment or focuses on individual sessions, programs, or database components. Provides read-only information.	<i>ALLBASE/SQL Performance and Monitoring Guidelines</i> , chapters 6-9.
Audit (Tools)	Provides a series of features to set up an audit DBEnvironment which generates audit log records that you can analyze with the new SQLAudit utility for security or administration. Includes the ability to set up partitions. See <i>ALLBASE/SQL Database Administration Guide</i> for SQLAudit commands. Modified statements: ALTER TABLE, CREATE TABLE, START DBE NEW, START DBE NEWLOG. New statements: CREATE PARTITION, DROP PARTITION, DISABLE AUDIT LOGGING, ENABLE AUDIT LOGGING, LOG COMMENT.	<i>ALLBASE/SQL Reference Manual</i> , “SQL Statements;” <i>ALLBASE/SQL Database Administration Guide</i> , “DBEnvironment Configuration and Security” chapter and “SQLAudit” appendix.
Wrapper DBEnvironments (Tools)	Creates a DBEnvironment to wrap around the log files orphaned after a hard crash of DBEnvironment. New SQLUtil command: WRAPDBE.	<i>ALLBASE/SQL Reference Manual</i> , “Wrapper DBEnvironments” in “Using ALLBASE/SQL;” <i>ALLBASE/SQL Database Administration Guide</i> , WRAPDBE in “SQLUtil.”
HP PC API is now bundled with ALLBASE/SQL.	PC API is an application programming interface that allows tools written with either the GUPTA or the ODBC interface to access ALLBASE/SQL and IMAGE/SQL from a PC.	<i>HP PC API User's Guide for ALLBASE/SQL</i> .

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
Increased memory for MPE/iX (HP-UX shared memory allocation is unchanged) (Performance)	Increases memory up to 50,000 data buffer pages and 2,000 run time control block pages. Increases the limits significantly allowing allocation of enough data buffer pages to keep the entire DBEnvironment in memory if desired for performance.	<i>ALLBASE/SQL Reference Manual</i> , STARTDBE, STARTDBE NEW, and START DBE NEWLOG in “SQL Statements;” <i>ALLBASE/SQL Database Administration Guide</i> , “ALLBASE/SQL Limits” appendix.
ALLBASE/NET enhancements (Connectivity, Performance)	Improves performance of ALLBASE/NET, allows more client connections on server system, and reduces number of programs on MPE/iX.	<i>ALLBASE/NET User's Guide</i> , “Setting up ALLBASE/NET.”
ALLBASE/NET commands and options for MPE/iX (Connectivity, Usability)	Adds option ARPA. Adds option NUMSERVERS to check status of listeners and number of network connections. Changed syntax: ANSTART, ANSTAT, ANSTOP. Changed NETUtil commands: ADD ALIAS, CHANGE ALIAS.	<i>ALLBASE/NET User's Guide</i> , “Setting up ALLBASE/NET” and “NETUtil Reference.”
ALLBASE/NET and NetWare (Connectivity)	ALLBASE/NET listener for NetWare now works with the 3.11 version of Novell's NetWare for UNIX (HP NetWare/iX).	<i>ALLBASE/NET User's Guide</i> , “Setting up ALLBASE/NET.”
Changed restrictions for executing NETUtil commands for MPE/iX (Connectivity, Usability)	Adds SM or AM (in the specified account) to MANAGER.SYS for adding, changing, or deleting users for MPE/iX.	<i>ALLBASE/NET User's Guide</i> , “Setting up ALLBASE/NET.”
ARPA is only TCP/IP interface for data communication through ALLBASE/NET beginning with HP-UX 10.0 (Connectivity)	Remote database access applications that specify NS will not work if the client and/or server machine is an HP 9000 Series 700/800 running HP-UX 10.0 or greater. Server Node Name entry must be changed from NS node name to ARPA host name. For the NETUsers file, the “Client Node Name” must be changed from the NS node name to the ARPA host name. New NETUtil commands: MIGRATE USER, MIGRATE ALIAS.	<i>ALLBASE/NET User's Guide</i> , “Setting up ALLBASE/NET” and “NETUtil Reference.”

Conventions

UPPERCASE In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

COMMAND

can be entered as any of the following:

command Command COMMAND

It cannot, however, be entered as:

comm com_mand comamnd

italics In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

COMMAND *filename*

punctuation In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*filename*):(*filename*)

underlining Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

Do you want to continue? >> yes

{ } In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either **ON** or **OFF**:

**COMMAND { ON }
 { OFF }**

[] In a syntax statement, brackets enclose optional elements. In the following example, **OPTION** can be omitted:

COMMAND *filename* [OPTION]

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select **OPTION** or *parameter* or neither. The elements cannot be repeated.

**COMMAND *filename* [OPTION
 parameter]**

Conventions (continued)

[...] In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

[, *parameter*] [...]

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

[*parameter*] [, ...]





| ... | In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **A**, **AB**, **BA**, or **B**. The elements cannot be repeated.




$\left\{ \begin{array}{l} \mathbf{A} \\ \mathbf{B} \end{array} \right\} | \dots |$

... In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

Δ In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

(*parameter*)Δ(*parameter*)

 The symbol  indicates a key on the keyboard. For example,  represents the carriage return key or  represents the shift key.

 *C*  followed by an uppercase character indicates a control character. For example, **Y** means that you press the control key and the Y key simultaneously.

Contents

1. Basic Concepts in ALLBASE/SQL Performance	
DBEFile Organization	1-1
Page Organization	1-1
Page Table Pages	1-2
Rows of Data on Pages	1-2
Structure of a Page	1-2
Storage of Table Data on DBEFile Pages	1-3
Slot Table	1-4
Indirect Rows	1-5
Hash Storage	1-5
Page Compression	1-6
Storage of Index Data on DBEFile Pages	1-6
How Indexes are Used	1-7
How PCRs are Stored	1-9
Page Splitting	1-9
Data Buffering	1-10
System Catalog	1-14
Directory Caching	1-15
Log File Organization	1-15
Log Buffering	1-16
No-Log Pages	1-16
Locking and Latching	1-17
Locks	1-17
Latches	1-17
Pins	1-18
Sequence of Events in Locking Data	1-18
Sorting	1-18
Optimization	1-19
How Optimization is Done	1-19
Table Size	1-19
Selectivity	1-20
Index Size	1-20
Cluster Count	1-20
Using GENPLAN	1-20
Using SETOPT	1-20
Sections and Validation	1-21
Section Caching	1-21
Validation	1-21

2. Guidelines on Logical and Physical Design	
Logical Data Design	2-1
Normalization Issues	2-1
Denormalizing Tables that are Consistently Joined	2-2
Horizontal Partitioning	2-3
Vertical Partitioning	2-3
Including Calculated Data in Tables	2-4
B-Tree Index Design	2-4
Choosing Keys	2-4
Building Indexes on Large Tables	2-5
Maintaining Indexes	2-6
Clustering Indexes	2-6
Using Clustering Indexes	2-7
Monitoring the Cluster Count	2-7
Reclustering a Table	2-8
Using Hash Structures	2-8
Choosing Appropriate Index Types	2-9
Updating Statistics	2-10
Authorization Design	2-10
Using a Duplicate Database	2-10
Physical Data Design	2-11
Creating DBEFileSets	2-11
Avoiding the SYSTEM DBEFileSet for User Data	2-11
Placing Large Tables in Separate DBEFileSets	2-12
Gathering Small Tables into DBEFileSets	2-12
Creating DBEFiles	2-13
Avoiding Extra DBEFile Space	2-13
Creating Tables	2-13
Avoiding NULL and Variable Length Data	2-14
Using INTEGER Rather than SMALLINT Data	2-14
Initial Table Loads	2-15
Unloading Data	2-16
Unloading and Reloading to Remove Indirect Rows	2-16
Unloading and Reloading to Remove Overflow Pages	2-16
Tips on Deletions from Tables	2-17
3. Guidelines on Query Design	
Avoiding Serial Scans Through Query Design	3-1
Arithmetic Expressions	3-1
Columns from One Table on Both Sides of the Relational Operator	3-2
Data Conversions	3-2
Predicates with INTEGER = DECIMAL(n,0) Factors	3-4
Using Subqueries	3-4
When Not to Use DISTINCT in Subqueries	3-6
Using UNION	3-6
Avoiding Conversions	3-6
Defining Indexes for UNION Queries	3-7
Using MIN/MAX Functions in Predicates	3-7
Using OR Predicates	3-8
How OR Predicates are Optimized	3-8
Choosing an Index for OR Factors	3-8

Using Predicates with LIKE	3-9
Using Predicates with BETWEEN	3-10
Using Fetch Unique Scans	3-10
Updating Key Columns	3-10
Avoiding User Propagation of Filters	3-11
Using TID Scans	3-11
Using Parallel Serial Scans	3-12
Using the BULK Option	3-12
Analyzing Queries with GENPLAN	3-12
Modifying the Access Optimization Plan with SETOPT	3-13
4. Guidelines on Transaction Design	
General Tips on Managing Transactions	4-1
Using Short Transactions and Savepoints	4-2
Controlling Locking	4-2
Using CS, RC, and RU Isolation Levels	4-3
Using Row Level Locking	4-4
Benefits of Row Level Locking	4-4
Shared Memory Considerations	4-4
Page Locking on PUBLICROW Tables	4-5
Using KEEP CURSOR	4-6
Removing Non-Database Processing from Transactions	4-6
Using Procedures and Rules	4-7
Tuning Performance of Dynamic Statements	4-7
Using Dynamic Parameters	4-7
Using Semi-Permanent Sections	4-8
5. Guidelines on System Administration	
DBA Guidelines	5-1
Validating Your Applications Before Run Time	5-1
Developing Application Programs	5-2
Balancing System Load	5-2
Placing Concurrently Used Objects on Different Drives	5-2
Calculating Shared Memory Allocation	5-3
Choosing a Number of Data Buffer Pages	5-3
Keeping a Small Group of Pages in Memory	5-4
Basic Example	5-5
First Threshold for Performance Gain	5-6
Second Threshold for Performance Gain	5-8
Cautions	5-9
An Empirical Approach	5-9
Choosing the Size of the Runtime Control Block	5-9
Choosing a Number of Log Buffer Pages	5-10
Choosing the Number and Size of Log Files	5-10
Nonarchive Log Guidelines	5-10
Archive Log Guidelines	5-11
Sorting Operations	5-11
Creating Temporary Spaces	5-11
Tips for Using Temporary Spaces	5-12
Disk Space for Sorting	5-12
Controlling the Use of Temporary Space	5-12

Memory Utilization in Sorting	5-13
Performance Hints for Large Sorts	5-13
Join Methods	5-13
Temporary Space in the SYSTEM DBEFileSet	5-13
Section Caching and Directory Caching	5-13
Setting Limits for Section Caching	5-14
Using Multiconnect Functionality	5-14
Using Timeouts to Tune Performance	5-14
Network Guidelines	5-15
HP-UX System Guidelines	5-15
Using HP-UX Raw Files for DBEFiles and Logs	5-16

6. Getting Started With SQLMON

Introduction	6-1
Starting SQLMON	6-1
Leaving SQLMON	6-2
Specifying the DBEnvironment	6-2
Invoking SQLMON Screens	6-3
Leaving an SQLMON Screen	6-7
Navigating SQLMON Subsystems	6-7
Setting SQLMON Variables	6-9
Accessing Online Help	6-10
Invoking the Help Facility	6-10
Leaving the Help Facility	6-10
Issuing Help Commands	6-10
Creating Batch Reports	6-12
Overhead Generated by SQLMON	6-12
Monitoring Tasks	6-13

7. Troubleshooting with SQLMON

Overview Subsystem	7-1
Transaction Limit Reached	7-1
Lock Contention	7-2
Memory Limit Reached	7-2
High Data Buffer Miss Rate	7-3
Log Full Condition	7-3
IO Subsystem	7-4
Insufficient Data Buffer Space	7-4
Insufficient Log Buffer Space	7-5
Load Subsystem	7-7
Transaction Delays	7-7
Rollbacks	7-7
Lock Contention	7-8
Lock Subsystem	7-9
Lock Waits	7-9
Overview Session Screen	7-9
Lock Session Screen	7-9
Lock Impede Screen	7-10
Deadlocks	7-11
Step 1 Open Four Windows	7-11
Step 2 Set Up the Freeze	7-11

Step 3 Create a Deadlock	7-12
Step 4 Examine the Locks with SQLMON	7-13
Step 5 Release the Frozen Session	7-14
Lock Allocation Failures	7-14
Step 1 Open Three Windows	7-14
Step 2 Set Up the Freeze	7-15
Step 3 Generate the Error	7-15
Step 4 Investigate the Session with SQLMON	7-15
Step 5 Release the Frozen Session	7-16
Freezing DBEnvironment Sessions	7-16
Releasing DBEnvironment Sessions	7-17
SampleIO Subsystem	7-18
Using the SET SAMPLING Command	7-18
Using the SET DISPLAYSAMPLES Command	7-20
A Sample Batch Job	7-20
Understanding the Internals of Sampling	7-22
Static Subsystem	7-23
Full DBEFileSets	7-23
Poorly Clustered Indexes	7-23
Indirect Rows	7-24
Hash Overflow Pages	7-24

8. SQLMON Screen Reference

IO Screen	8-2
IO Data Program Screen	8-4
IO Data Session Screen	8-6
IO Log Program Screen	8-8
IO Log Session Screen	8-10
Load Screen	8-12
Load Program Screen	8-14
Load Session Screen	8-16
Lock Screen	8-18
Lock Impede Screen	8-20
Lock Memory Screen	8-23
Lock Object Screen	8-25
Lock Session Screen	8-28
Lock TabSummary Screen	8-31
Overview Screen	8-34
Overview Program Screen	8-36
Overview Session Screen	8-38
SampleIO Screen	8-40
SampleIO Indexes Screen	8-42
SampleIO Objects Screen	8-44
SampleIO TabIndex Screen	8-46
SampleIO Tables Screen	8-48
Static Screen	8-50
Static Cluster Screen	8-51
Static DBEFile Screen	8-53
Static Hash Screen	8-55
Static Indirect Screen	8-57
Static Size Screen	8-59

9. SQLMON Command Reference

EXIT	9-2
HELP	9-3
QUIT	9-5
SET	9-6
SET CYCLE	9-7
SET DBECONNECT	9-8
SET DBEFILESET	9-9
SET DBEINITPROG	9-10
SET DBENVIRONMENT	9-11
SET DISPLAYSAMPLES	9-12
SET ECHO	9-14
SET LOCKFILTER	9-15
SET LOCKOBJECT	9-18
SET LOCKTABFILTER	9-20
SET MENU	9-21
SET OUTPUT	9-22
SET REFRESH	9-24
SET SAMPLING	9-25
SET SORTIODATA	9-26
SET SORTIOLOG	9-27
SET SORTLOAD	9-28
SET SORTLOCK	9-29
SET SORTSAMPLEIO	9-30
SET TOP	9-31
SET USERTIMEOUT	9-32
!	9-33

A. Design for a High-Performance Interactive Table Editor

Example Table	A-1
User Interface	A-1
Internal Algorithms	A-2
SELECT	A-2
DELETE	A-2
UPDATE	A-2

Index

Figures

6-1. SQLMON Road Map	6-8
7-1. Deadlock Example	7-12

Tables

6-1. SQLMON Screens	6-4
6-2. Abbreviated Screen Commands	6-6
6-3. SQLMON Help Commands	6-11
6-4. Monitoring Disk Usage	6-14
6-5. Monitoring Memory Usage	6-14
6-6. Monitoring Tables	6-15
6-7. Monitoring Hash Structures	6-15
6-8. Monitoring Indexes and Referential Constraints	6-16
6-9. Monitoring Transactions	6-17
6-10. Monitoring Sessions	6-18
6-11. Monitoring I/O for Data	6-19
6-12. Monitoring I/O for Logging	6-19
6-13. Additional Monitoring for Logging	6-19
6-14. Monitoring Locking	6-20

Basic Concepts in ALLBASE/SQL Performance

Before presenting specific tips and tricks for tuning your DBEnvironments, this chapter presents some information about how data is passed about in ALLBASE/SQL among files, buffers, applications, and other elements in a typical large-scale production environment. The following topics are included:

- DBEFile organization.
- Data Buffering.
- System Catalog.
- Log File Organization.
- Log Buffering.
- Locking and Latching.
- Sorting.
- Optimization.
- Sections and Validation.

Before reading on, you may wish to review the material found in the “Physical Design” chapter of the *ALLBASE/SQL Database Administration Guide* and the “Concurrency Control” chapter of the *ALLBASE/SQL Reference Manual*.

DBEFile Organization

Many performance issues depend on understanding the structure of ALLBASE/SQL DBEFiles. DBEFiles are disk files that store data and indexes for both user and system tables. The following paragraphs outline the structure of DBEFiles and the format of data they contain.

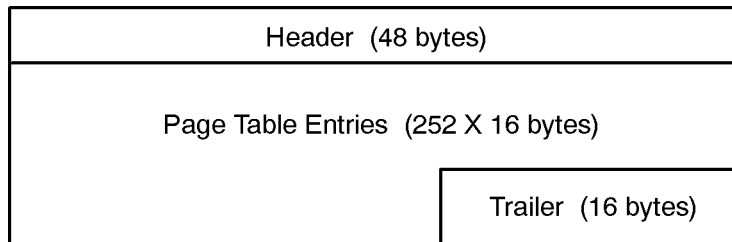
Page Organization

Data is read from and written to DBEFiles in 4096-byte blocks known as **pages**. Each page read from or written to disk constitutes one I/O operation. The number of I/Os for a given query is an important measure of performance. When you create a DBEFile for the DBEnvironment, you specify its size as a number of pages, and the size is recorded in the system catalog.

Page Table Pages

The first page in a DBEFile is known as a **page table page**, so called because it contains a table of the contents of the following 252 pages in the file. Before each group of 252 data or index pages, a new page table page is included. Page table pages are composed of entries that indicate whether the following pages are allocated, full, free, or in some other state. These pages are accessed during serial scans, and they are modified when data is added to or dropped from a table.

Page table page entries are 16 bytes long; each page table page contains one entry for each of the following 252 pages. The page table page also contains a 48-byte header and a 16-byte trailer at the end for a total of 4096 bytes, as the following diagram shows:



Rows of Data on Pages

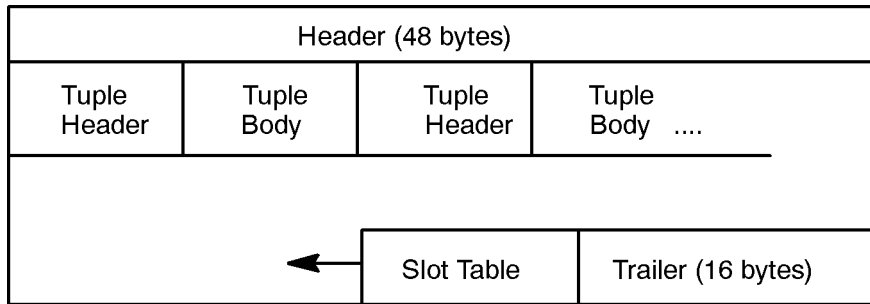
Data from tables and indexes is stored on DBEFile pages as rows or **tuples**. A given page can store data for only one table or index, though the DBEFile as a whole can contain pages for many different objects. A DBEFile of type TABLE can only contain data for tables, and a DBEFile of type INDEX can only contain index data. A DBEFile of type MIXED may include pages for both tables and indexes. A page table page entry indicates which object has data stored on that particular page.

Structure of a Page

Understanding how data is arranged on these pages can assist you in deciding how to reduce the number of I/Os required for particular database operations. Since table data and index data are stored somewhat differently, the two types of storage are presented in separate sections that follow.

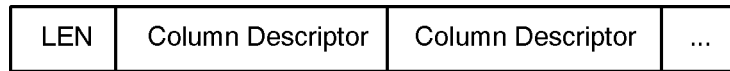
Storage of Table Data on DBEFile Pages

Tuples from user tables are stored on DBEFile pages as in the following diagram.



The tuples in a TABLE DBEFile are rows from user tables. The tuple header contains descriptive information about the columns in a tuple, and the tuple body contains the actual data.

The format of a tuple header is as follows:



The tuple header contains the following fields:

- Length of the header (2 bytes)
- A 2-byte column descriptor for each column in the tuple. Each column descriptor has bits that indicate the following:
 - Whether or not the column contains a null value.
 - Data type of the column.
 - Byte offset to the end of the column within the tuple.

The total length of a tuple header is given by the formula:

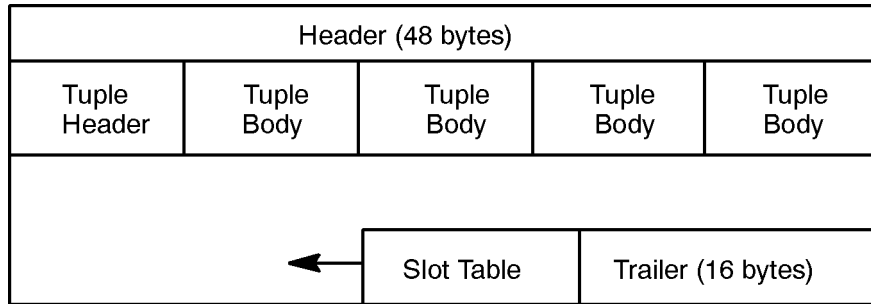
$$Length = 2 * (Ncols + 1)$$

where Ncols is the number of columns in the tuple.

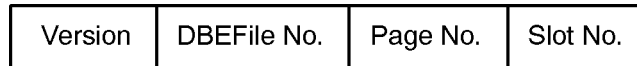
At the beginning of the data area on a page, following the page header and before the first tuple on the page, there is a tuple header which contains information about the first tuple. This header is also known as the **shared header** because it can be shared with other tuples on the page. For all tuples other than the first one, a header is only stored with the tuple body if it is different from the shared header.

To illustrate the use of the shared header, suppose that a page holds 100 rows of 4 columns each. In this case, header data requires 10 bytes per tuple. If each header were different from the shared header, the total space used on the page by all tuple headers would be 1000 bytes. But if all 100 rows used the same header, then header data would only occupy a total of 10 bytes, leaving more room for tuples.

A page with a shared tuple header looks like the following:



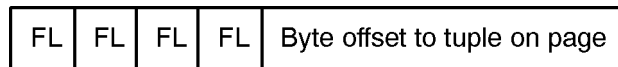
A flag is set in the **slot table** to indicate whether or not the shared tuple header is used by a particular row. The slot table is further described below. Each tuple can be uniquely identified by a **tuple ID (TID)**, which has the following format:



For individual rows, the TID is usually expressed as three numbers separated by colons (example: 10:5:8), where the first number is the DBEFile number; the second is the page number; and the third is the slot number. Version numbers are not used for identifying rows of data, but they are used in identifying other objects in the DBEnvironment, such as tables. The DBEFile number and page number indicate the file page in which the tuple is found. The slot number, further described below, indicates an offset on the page where the row is located.

Slot Table

At the end of the page is a **slot table**, which contains an entry for each tuple on the page. When ALLBASE/SQL uses a TID, the slot number found in the last field in the TID is an index into this table. Each slot table entry has the following format:



Each entry is 16 bits long; the first four bits are flags (FL), and the remaining 12 bits are the byte offset to the tuple within the page, counting from the beginning of the page. There is a maximum of 256 entries in a slot table, which means that a maximum of 256 rows can be stored on a page.

Indirect Rows

An **indirect row** is created when a row's length increases during an update, and the page that currently holds the row does not have enough space to store the new information. In cases like this, the updated row is stored on another page, and the original page is updated with the TID of the new row on the new page. An indirect row can only be accessed by first fetching one page to find the address of the row and then fetching a second page to obtain the row itself.

An indirect row may be created in any of the following circumstances:

- A variable length data field is updated with a longer value than the one previously stored.
- A NULL column is updated to contain a non-null value.
- An ALTER TABLE statement adds a column to a table and supplies a default value that must be added to each row.

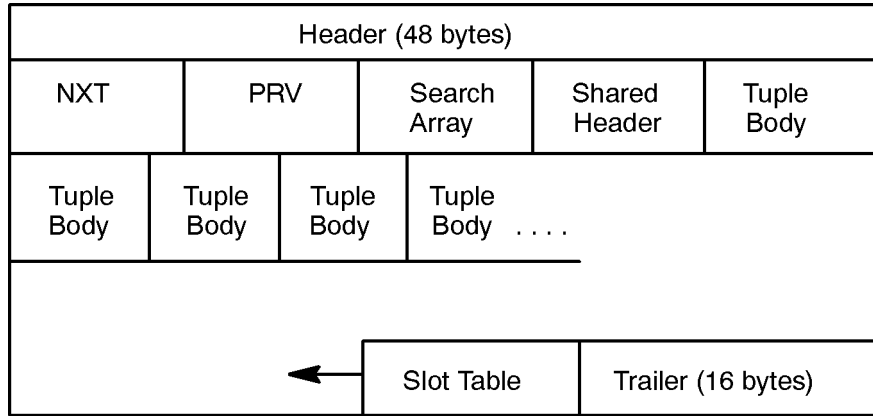
To determine the percentage of indirect rows, run SQLMON and examine the TABLE INDIRECT ROW field of the Static Indirect screen.

Hash Storage

Hash structures in ALLBASE/SQL are tables that you define to be hashed according to a specific key at the time you create them. In addition to the key, you specify a number of primary pages, which become the hashed locations of the data in the DBEFile or DBEFiles holding the table. Rows are inserted based on the value in the hash key; a row is said to hash to a specific page, which means that the primary page location is calculated from the value of the key, which must be unique. A search array is maintained on each page. This structure contains slot numbers in key order. ALLBASE/SQL does a binary search using this array to arrive at a specific row quickly. When a row is inserted, the array is updated.

If there is not enough room for an inserted row on a hash page, the row is placed on an **overflow page**. Overflow pages are linked to the primary page and to one another using the NXT and PRV pointers on the page. A large number of overflow pages means slower access to data. To avoid this, choose a good hash key with a uniform distribution of values within the table. Evenly distributed key values result in hashing to an evenly distributed set of pages. A key with a skewed distribution of values would result in hashing to a skewed set of pages. If the hash key does not have a uniform distribution, then the time required to access some rows will be much slower than the time required to access other rows.

Except for the search array and the NXT and PRV pointers, the format of a hash data page is similar to that of an ordinary data DBEFile page, as shown in the following diagram:



To get information about the hash structures of a DBEnvironment, run SQLMON and go to the Static Hash screen.

Page Compression

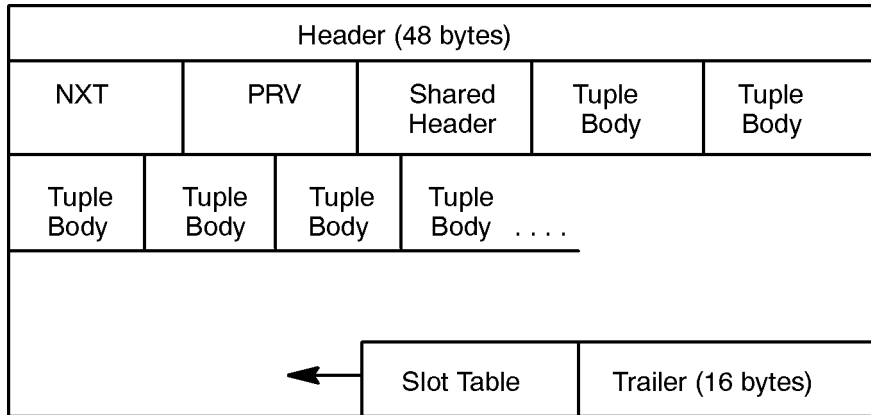
When a row of data is inserted on a DBEFile page, it enters the region known as the **free area** on the page. The free area is all the space that is marked as available for data. When a row is deleted, its space does not immediately return to the free area. Thus, additional inserts following a deletion will first fill up all the space in the free area on the page. If the free area becomes full at insert or update time, the space left over from deletions is returned to the page's free area. This process is known as **page compression**.

Storage of Index Data on DBEFile Pages

Index entries are stored just as data entries are, with the following exceptions:

- Index pages are either leaf pages or non-leaf pages.
- Leaf pages actually point to rows on DBEFile pages. Each leaf page tuple contains a key value and the TID of a data row containing that key.
- Non-leaf pages contain tuples with key values and pointers to other non-leaf pages or to leaf pages.

A leaf index page for a common type of index looks like the following:



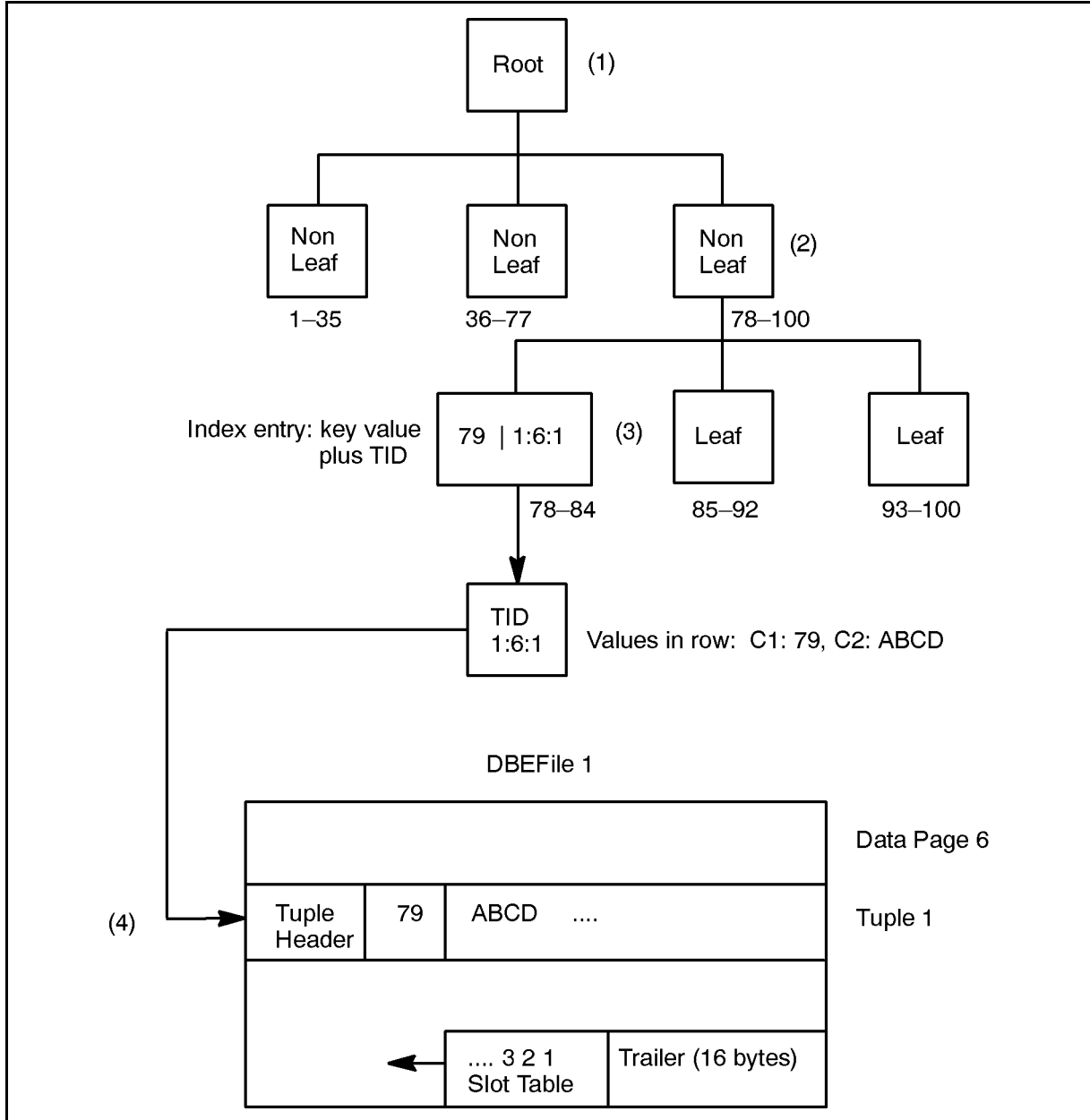
The NXT and PRV fields point to the next and previous leaf pages in the index. These pointers make an index scan in key order extremely fast. The shared header describes the tuple's length and characteristics. The tuple body on an index leaf page has two parts: a key and a data TID. The key is the actual index key value as it appears in the table, and the data TID is the address of the row pointed to by this index entry.

To monitor the indexes of a DBEnvironment, run SQLMON and go to the SampleIO Indexes and Static Cluster screens.

How Indexes are Used

The basic index structure in ALLBASE/SQL is a B-tree or balanced tree. The B-tree consists of a root page and a number of non-leaf and leaf pages. Typically, the root page points to a non-leaf page that contains entries for particular ranges of key values, and each non-leaf page points to still other pages containing entries for progressively narrower ranges of values. The lowest level in the tree is called the leaf page, which contains pointers to (TIDs of) specific rows in DBEFiles.

The following diagram shows how a B-tree provides access to data. After deciding to use a particular index, ALLBASE/SQL accesses the root page (1). Then it reads non-leaf pages (2) until it obtains a leaf page (3) which contains the TID of a qualifying row. Finally, it accesses the data page containing the row (4). Assume that the values of C1 in the following are percentages between 1 and 100:

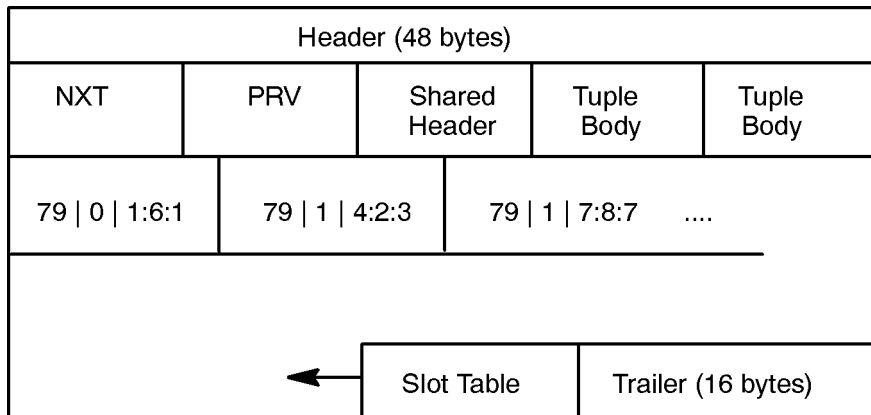


How PCRs are Stored

A **PCR** (parent-child relationship) is a special kind of B-tree that supports a referential relationship between two tables—the parent table and the child table. This kind of index has entries that point to the rows in the referring table and different entries that point to the rows in the table referred to. The table referred to must also have a unique index defined on it.

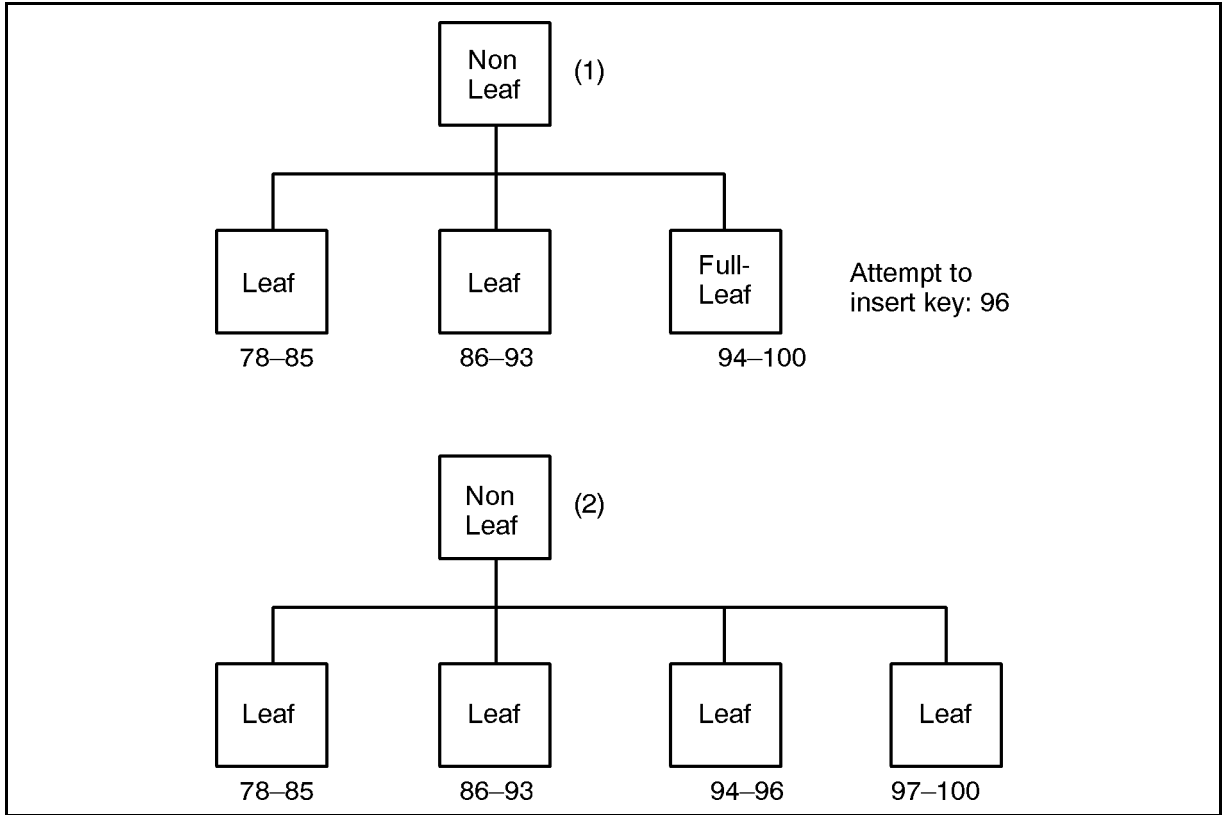
In the leaf pages of a PCR, an entry for the parent table (that is, the table referred to) precedes the entries for child tables (tables referring to the parent). Keys are distinguished by adding a 0 to the end of a parent key, and a 1 to the end of a child key.

The leaf index page in a PCR would have tuples like the following:



Page Splitting

As rows are inserted into a table, index entries are also inserted into the B-tree. As the pages of a B-tree fill up, new branches of the tree are created through a process known as **page splitting**. In page splitting, two new index pages are allocated, and the index entries on the old page are copied to the new pages—half the entries to each new page; then the new entry is inserted, and the old page is freed for reuse. The process is shown for a typical case in the following figure. (1) shows a portion of an index before splitting; (2) shows the index after splitting.



When the key value being inserted is greater than the largest value already in the index or smaller than the lowest value, page splitting is one-way. This means that only one new page is allocated, and half the entries from the old page are moved to it, after which the new key value is added to the new page. In the previous example, an attempt to insert a value of 110 on a full leaf page where the highest value is 100 would result in one-way page splitting.

Data Buffering

ALLBASE/SQL uses a system of buffers to provide access to data and index pages by concurrent transactions. Three sets of buffers actually are used:

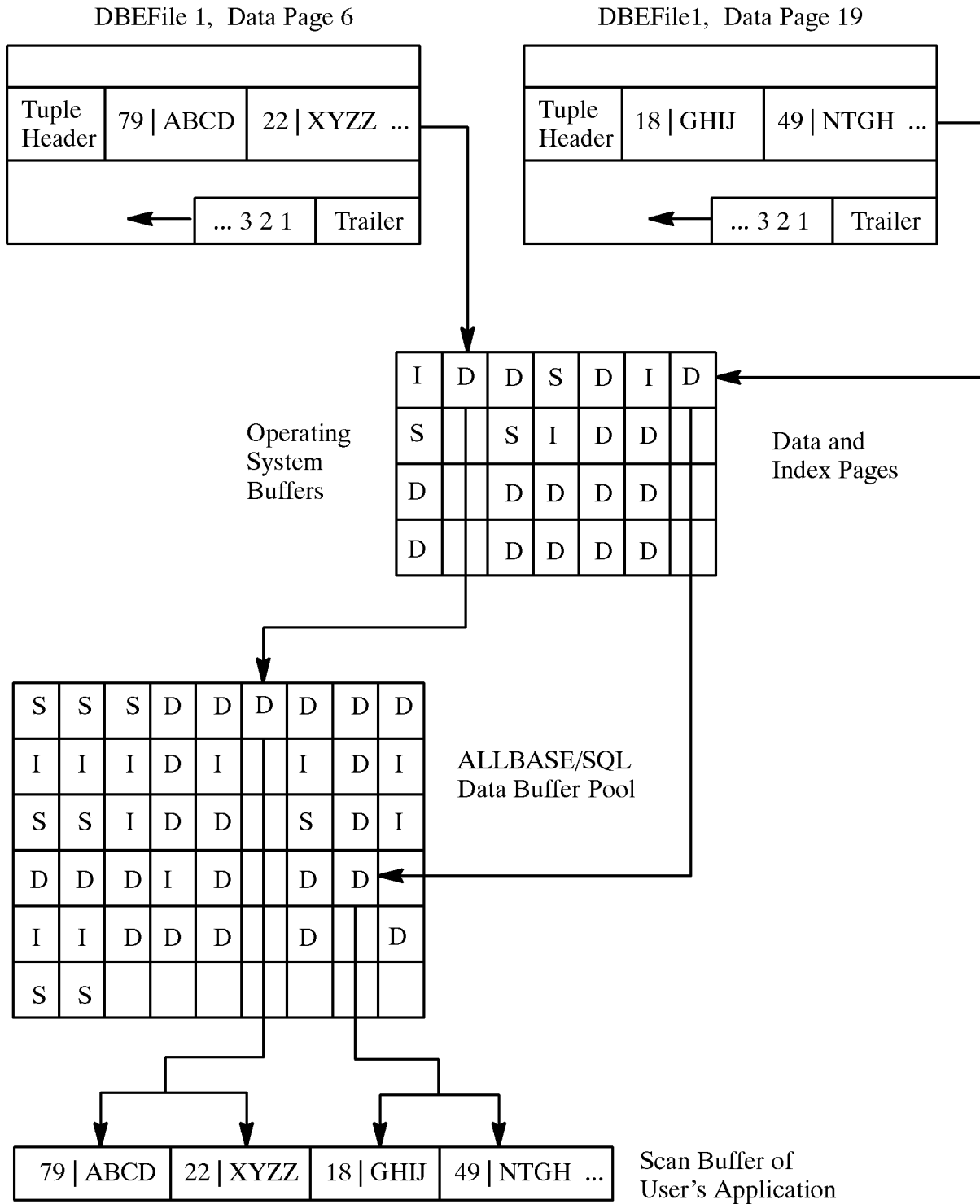
- Operating system buffers.
- ALLBASE/SQL data buffer pool, shared by all processes accessing the same DBEnvironment.
- ALLBASE/SQL scan buffer (or tuple buffer)—one per user process.

In what follows, the emphasis is on the latter two items, which are components of ALLBASE/SQL. Because the ALLBASE/SQL data buffer pool resides in shared memory, many users can access the same pages in memory. For example, if many users need to access information from the same system catalog pages, these pages do not need to be read into the buffer every time a transaction needs them. Provided there is enough buffer space, pages may remain in ALLBASE/SQL shared memory for long periods. A page that has not been

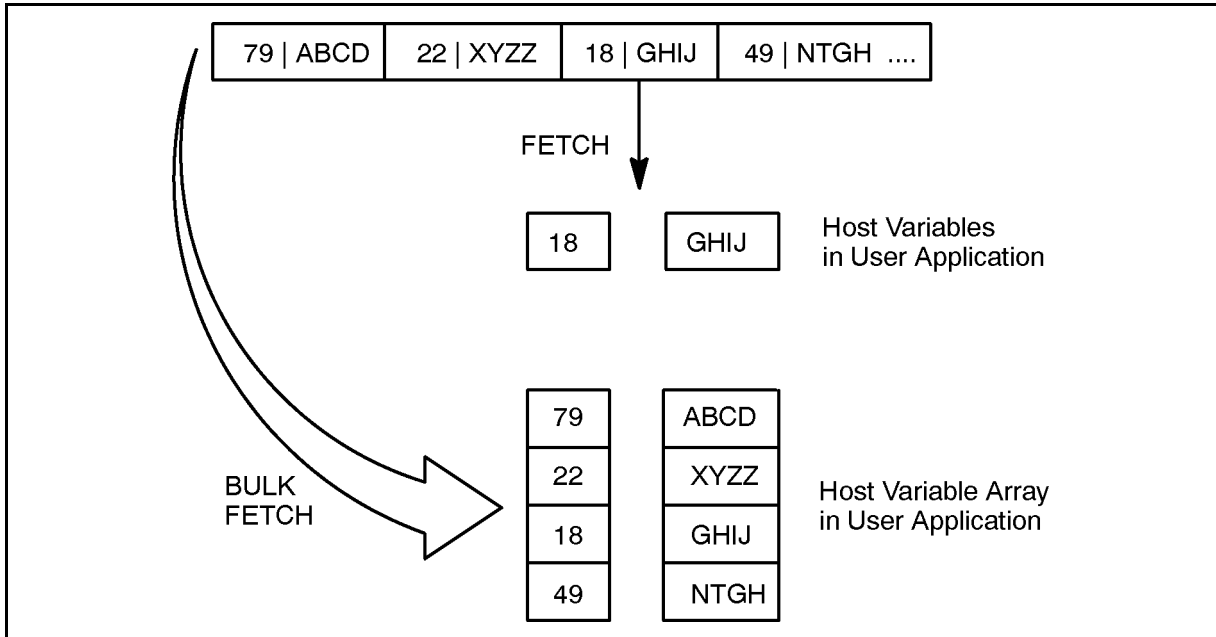
modified at all or a page that has been modified and written to disk is considered **clean**, while a page that has been modified but not written out to disk is considered **dirty**.

If there is not enough empty space in the buffer, dirty pages are **swapped out**, that is, written back to disk, and clean pages are overwritten, on the basis of a least recently used (LRU) algorithm.

As the following figure shows, pages flow through the operating system's buffers into the shared ALLBASE/SQL data buffer. Then individual tuples from data pages are read into the tuple buffer associated with an application program. In the figure, D indicates user data pages, I indicates user index pages, S indicates system catalog pages. The illustration shows primarily the movement of data pages, but index pages move in and out of the data buffer in the same way.



From the scan buffer, the application fetches data into host variables, as in the following:



Individual rows are fetched (or groups of rows are BULK fetched) into host variables or arrays declared within the application program.

It is important to understand that each layer of buffering requires additional copying of data from one place to another. More significantly for performance, the movement of data from DBEFiles into the operating system's buffer pool and back requires I/O. The movement of pages between the operating system buffers and the ALLBASE/SQL data buffer also may require additional I/O.

When your applications read large numbers of data pages, they may displace other pages which are still in the buffer, though they are not being used. Dirty pages in the buffer are swapped to disk, and new pages are read in.

To maximize performance, you should define a large enough data buffer for your specific queries, and you should attempt to eliminate as much swapping to disk as possible. This topic is discussed fully in the "System Administration" chapter.

For more information about buffering, refer to the chapter on "Concurrency Control through Locks and Isolation Levels" in the *ALLBASE/SQL Reference Manual*. To monitor data buffer I/O, invoke the IO screen in SQLMON.

System Catalog

The **system catalog** in ALLBASE/SQL is a database of runtime code and system information used by SQLCore to carry out internal operations. Like other databases, the system catalog is a set of tables. The base tables underlying SYSTEM views are owned by special user HPRDBSS, and they are located in the SYSTEM DBEFileSet.

The runtime code in the system catalog consists of stored sections for application programs, procedures, and views, together with validity information and authorization data. At run time, code is fetched from the system catalog and stored in the user's memory heap. You can examine the informational part of the system catalog by doing queries on a set of views owned by SYSTEM or CATALOG.

The system catalog is accessed in two ways: by user queries and by internal access. When you perform a query on the system catalog, locks are obtained and released just as in any other query in your transactions, and subject to the same isolation levels. However, when ALLBASE/SQL accesses the system catalog internally on your behalf, it uses the Repeatable Read (RR) isolation level. For example, if you issue the query

```
SELECT * FROM PurchDB.Parts
```

at the RU isolation level, no locks are obtained by your transaction on the Parts table. Internally, ALLBASE/SQL acquires share locks at the RR isolation level on several system tables as it performs the query. Even though you may have selected RU, ALLBASE/SQL still reads the system catalog on your behalf at the RR level.

Note

The locking of system catalog resources is different for dynamic statements than it is for statements in preprocessed applications. Consider the following query:

```
SELECT * FROM SYSTEM.TABLE
```

If you issue this query in a preprocessed application at the RU isolation level, your transaction does not obtain any locks on the SYSTEM.TABLE view or the base table HPRDBSS.TABLE at run time, provided the section that incorporates the query is valid. In a dynamic statement (including a query issued within ISQL), ALLBASE/SQL has to read HPRDBSS.TABLE to obtain information about SYSTEM.TABLE, so it therefore applies share locks on your transaction's behalf.

For more information about the locks that are applied on system catalog resources, refer to the appendix "Locks Held on the System Catalog by SQL Statements," in the *ALLBASE/SQL Database Administration Guide*.

What effect does internal locking of the system catalog have on performance? If a transaction is doing data definition, it obtains exclusive locks on system tables. This prevents other system access from taking place until the data definition transaction is finished. You can prevent data definition from taking place and thereby prevent lock waits and deadlocks on the system catalog by disabling data definition. You do this by using the SQLUtil ALTDDBE command to set the *DDL Enabled* flag to NO in the DBECon file.

Note Setting the *DDL Enabled* flag to NO does not disable section validation, which obtains exclusive locks on the system catalog.

Directory Caching

When DDL is disabled (DDL Enabled set to NO), certain system catalog information is cached in shared memory where it is available for quick access.

Log File Organization

The ALLBASE/SQL **log** is organized as a series of 512-byte pages. Logs are written and read in page-size blocks. Within ALLBASE/SQL, information that is logged is written out as a series of log records, which contain enough information to reconstruct the state of the database that existed before a user's transaction started. In general, only data change (write) operations are logged, since read operations do not affect the structure of the database. Changes to data, index, and system catalog pages all are logged. A log record may span many log pages.

A log may be used in archive or nonarchive mode. In nonarchive mode, the log maintains a record of all data change operations by transactions that were uncommitted at the time of the last checkpoint; and it maintains a record of all data change operations since the last checkpoint. In archive mode, the log maintains a record of *all* data change operations, whether from committed or uncommitted transactions. Both nonarchive and archive logs allow transactions to be rolled back. An archive log also allows you to reapply transactions after restoring a DBEnvironment following a hard crash.

Log records from all concurrent transactions are stored in sequential log records in the log buffer. When you issue a COMMIT WORK, when the log buffer becomes full, or when a CHECKPOINT statement is issued, the contents of the log buffer are written to a disk file. (Changed data buffer pages are also written to disk at checkpoint time.) In the case of nonarchive logging, a checkpoint may make log file space available for use by new log records.

The log itself consists of one or more files of different sizes. Initially, a log is configured in the START DBE NEW or START DBE NEWLOG statement as a single or dual file. Subsequently, you can add additional files (two at a time in dual logging) with the SQLUtil ADDLOG command. Adding log files allows log switching. When one log fills up, ALLBASE/SQL can immediately switch to the next file with no pause except to close the first file and open the second. If no file is available for log switching, and if no space can be reclaimed with a checkpoint, a LOG FULL condition will occur when the end of the log file is reached. LOG FULL causes your transaction to be rolled back.

A directory of log files is kept in the DBECon file; you can view this directory by using the SQLUtil SHOWLOG command. To examine the current capacity of the log files, invoke the Overview screen in SQLMON.

Log Buffering

For operations involving a change to data, ALLBASE/SQL writes log records to a log file, so that these operations can be rolled back if necessary or reapplied in the event of a system failure. In logging, the transaction enters log records into the log buffer, which is periodically flushed (written to disk). The log buffer pool consists of a number of 512-byte pages configured with the START DBE statement or with the SQLUtil ALTDBE command.

Regardless of log type, the log buffer pool is used as the collection point for log records. The maximum log buffer size is 1024 pages, but 120 pages is generally recommended. For detailed information about creating and manipulating log files, refer to the “Backup and Recovery” chapter in the *ALLBASE/SQL Database Administration Guide*. To monitor logging I/O, or to check the maximum size of the log buffer, run SQLMON and invoke the IO screen.

No-Log Pages

Some operations that change data in a DBEnvironment allocate new pages for insertion in a DBEFile. These new pages are often called **no-log pages**. The use of no-log pages provides a performance benefit in non-archive mode. In non-archive mode, no-log pages are not logged; only a log record is created for the allocation of the page. If the operation must be rolled back, then the page can simply be de-allocated. If a savepoint is defined following the allocation of a no-log page, the page is marked for logging, since rollback of a no-log page is an all-or-nothing operation; that is, the page could be deallocated, but not partially unfilled. Since the content of a no-log page is not logged, the page is forced to disk at COMMIT WORK time.

No-log pages include the following:

- Pages allocated at INSERT time in non-archive logging. When you are inserting rows, your transaction obtains an exclusive lock on the page being allocated, so other transactions cannot write on that page. Thus it is possible to log only the allocation of the page, not the data being inserted by your transaction.
- Pages allocated at CREATE INDEX time during non-archive logging.
- Pages allocated in a B-tree split during non-archive logging, if your transaction has obtained an exclusive lock on the table. If you must insert a large number of rows into a table that already has an index defined on it, B-tree index pages will split as they begin to fill up. If your transaction has locked the table in exclusive mode, it is only necessary to log the allocation of new pages in non-archive mode. In the event of a rollback, those pages can simply be deallocated.

This behavior of no-log pages makes it possible to save considerable logging activity in non-archive mode for LOAD and INSERT operations that lock the table in exclusive mode. However, there may be an increase in I/O activity as pages are forced to disk at COMMIT WORK time.

No-log pages are not created in archive mode. In archive mode, all changes to the DBEnvironment are logged.

Locking and Latching

Concurrent access to database objects, buffers, and other shared elements in a DBEnvironment is regulated by means of three kinds of controls within ALLBASE/SQL:

- Locks.
- Latches.
- Pins.

Locks regulate access to tables, pages, and rows of data when different users contend for them. Latches regulate system access to in-memory resources such as buffers, ensuring that one buffer operation is complete before another is allowed to proceed. Pins are used to keep resources in memory instead of allowing ALLBASE/SQL to swap them out by the LRU (least recently used) algorithm.

Detailed information about concurrency control, including locking, appears in the chapter “Concurrency Control through Locks and Isolation Levels” in the *ALLBASE/SQL Reference Manual*. To monitor locking activity, run SQLMON and go to the Load and Lock screens.

The following paragraphs highlight some specific performance-related issues.

Locks

A lock is a logical object that is created in memory whenever a user or the ALLBASE/SQL system accesses data at the RC, CS, or RR isolation levels. Physically, a lock is represented in the ALLBASE/SQL runtime control block as a 92-byte lock control block. Individual transactions issue lock requests for the specific objects that are required for data access. ALLBASE/SQL then either grants the request or places the transaction on a wait queue, requiring the transaction to wait until a lock is released.

ALLBASE/SQL maintains a specific area in shared memory for deadlock detection. As a lock request is registered, it is compared with existing locks and with other requests in the wait queue, and if a deadlock is discovered, the transaction with the lowest priority (highest priority number as assigned in the BEGIN WORK statement) is rolled back.

Each table, page, or row that is locked receives its own lock control block. For example, if you are using a large PUBLICROW table and locking individual rows, the transaction may require the allocation of many separate 92-byte control blocks. Space for control blocks is configured for the DBEnvironment as a whole when you specify a number of runtime control block pages. ALLBASE/SQL also uses these pages for other types of control blocks, but lock management is the single largest user of the runtime control block.

Latches

A latch regulates access to a data structure in memory. Latches are generally held for a very short period of time. An example is a memory-to-memory copy of a data page. While the copy takes place, it is important that no other transaction is allowed to alter the content of the data structure. The most commonly latched objects in ALLBASE/SQL are buffer pages, though other data structures are also latched. ALLBASE/SQL latches both data and log pages. For the log buffer, latching is the only mechanism used to regulate concurrent access. Log records are not locked.

Like locks, latches are also represented in shared memory as control blocks, but these are not allocated following a user's request; instead, they are allocated when the DBEnvironment

starts up. Latches are more efficient than locks, because deadlocks are not automatically detected in latching. Instead, deadlocks are avoided. Unlike locking, latching does not require the overhead necessary for deadlock detection.

Pins

A pin is similar to a latch, but its purpose is more specialized. A pin freezes a data structure (such as a page) in ALLBASE/SQL shared memory so that it cannot be overwritten while it is being read or written by a transaction. Multiple transactions can pin the same data structure. So long as one pin is still in place, the structure cannot be overwritten. After a read or write is complete, the data structure is unpinned.

Sequence of Events in Locking Data

Once a lock request is granted, a data page is pinned to a buffer frame in ALLBASE/SQL memory, then the page is latched while data is accessed by a transaction. When the access is complete, the latch is released, and the pin is released. The lock may be released or held, depending on the kind of lock and the isolation level of the transaction. In the case of updates and repeatable reads, locks are held until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

Sorting

Allbase/SQL uses sorting in several ways:

- When creating an index on a table that contains data.
- When processing a query that contains an ORDER BY, DISTINCT, UNION, or GROUP BY clause.
- When processing a join query with the sort/merge join algorithm.

Sorting operations require both disk space and CPU time, so they often raise performance issues. Space for sorting may be required in temporary files which can be in the current group or directory or in TempSpaces you define for the purpose. You can avoid the use of temporary files at run time by creating an index on the sort key. In many cases, the index can be used to carry out the sort.

Space is also needed for the temporary storage of sort output. This space is allocated in the SYSTEM DBEFileSet, which must have enough TABLE or MIXED DBEFile space to accommodate each query result.

Performance issues in sorting are discussed in more detail in the section “Sorting Operations” in the “Guidelines on System Administration” chapter.

Optimization

Optimization is the process by which ALLBASE/SQL finds the most appropriate access path to data for the execution of a particular SQL statement. An access path is one of the following ways of getting at data:

- Sequential scan—reading the entire table.
- Index scan—using a B-tree or PCR to access particular rows or the entire table.
- Hash scan—accessing a hash page directly, then finding the row on that page or on an overflow page.
- TID scan—accessing a single row directly using its page address or TID.

Usually the most appropriate path is the one requiring the fewest I/O operations needed to read requested data.

Optimization also includes decisions about join order, join method (nested loop or sort/merge), and sort operations (indexed or not).

How Optimization is Done

Optimization is primarily done on the basis of elements in the WHERE clause of an SQL statement. The WHERE clause may contain one or many components, known as **factors**, which limit the amount of data that will be manipulated by the statement. Information about these factors can potentially be used to improve the performance of a query. The following example has two EQUAL factors:

```
SELECT * FROM T1 WHERE C1 = 'Jones' AND C2 = 123534
```

Assume that C1 is defined as CHAR(40) NOT NULL and C2 is defined as INTEGER NOT NULL.

To retrieve the data, the query processor must scan the table to retrieve the rows in which C1 is *Jones* and C2 is 123534. There are several ways of doing this:

- Scan the entire table, choosing only the rows that meet both criteria.
- Scan the table for rows that satisfy the criterion for C1 and then eliminate the rows that do *not* satisfy the criterion for C2.
- Scan the table for rows that satisfy the criterion for C2 and then eliminate the rows that do *not* satisfy the criterion for C1.

Assuming there are B-tree indexes on both columns, what would the best approach be? In order to answer this, we need to consider several variables:

- Size of the table.
- Selectivity of each index.
- Index cluster count of each index.
- Sizes of the two indexes.

Table Size

If the table occupies only 3 pages in the DBEFile, the most that would be required to access the data without using an index is 3 I/Os (plus some I/Os to access system information; we ignore this hereafter). If the index occupies only 2 pages, the same query might need to access as few as 5 pages, but it could conceivably access many more than this, if the cluster count is high. There is clearly no advantage in using an index in this case.

As the size of the table increases, the use of an index may become more advantageous. Assume that the table has 10,000 pages, that data is stored in sorted order and that key values are found on 100 data pages. In this case, 10,000 I/Os would be needed to do the query with a serial scan. If a B-tree index on the table has 3 levels and if key values are found on two index leaves and 100 data pages, then only 104 I/Os would be needed for the same query. The index scan clearly makes sense now.

Selectivity

How can the optimizer tell whether to use a B-tree index on C1 or on C2, assuming that both exist? Index predicate selectivity and cluster count are the most important factors; the size of the index and the characteristics of the key also determine the choice. As an example of selectivity, suppose the index on C1 resulted in the selection of 200 rows, but the index on C2 resulted in 2000. Other things being equal, the optimizer would select C1 as the index with the higher selectivity, that is, the index that retrieves the smallest number of rows for its predicate.

Index Size

An index on C1 would be considerably larger than one on C2. Since key values of C1 are 40 bytes, and key values in C2 are 4 bytes, the B-tree on C1 would be ten times the size of the B-tree on C2, and the number of I/Os needed to retrieve the same number of key values using C1 would be ten times the number needed for the index on C2. In this case, if selectivity is the same, the optimizer would choose C2.

Cluster Count

The **cluster count** of an index indicates the number of times ALLBASE/SQL has to access a different data page to retrieve the next row during an index scan. Assume the table is sorted in the order of keys in C1. In this case, the cluster count for a B-tree on C1 would be equal to the number of pages on which rows are found. Let's say this is 100 pages for C1. Assume the same selectivity for C2, but random distribution of key values over the pages of the DBEFile in which data is stored. Each row retrieved with C2 would require a new I/O, and the cluster count could be closer to the number of rows in the table. In the present case, C1 is chosen. Cluster count is described in more detail in the "Guidelines on Logical and Physical Design" chapter. To monitor the cluster count, run SQLMON and go to the Static Cluster screen.

The optimizer makes all its choices in a similar fashion, weighing the relative cost of each scan plan and eventually choosing the most appropriate one.

Using GENPLAN

You can observe the result of the optimizer's work by using the SQL GENPLAN statement followed by a query on the system catalog pseudotable SYSTEM.PLAN. See the "Analyzing Queries with GENPLAN" section in the "Guidelines on Query Design" chapter for more information.

Using SETOPT

To override the access plan chosen by the optimizer, use the SETOPT statement. For more information, See the SETOPT section in the *ALLBASE/SQL Reference Manual*.

Sections and Validation

A **section** is a stored representation of an access path chosen by the optimizer. Sections are created by SQLCore when you do the following:

- Preprocess an application using the ALLBASE/SQL preprocessors.
- Issue a statement in ISQL.
- Use dynamic operations in an application. Such operations are built using the PREPARE and EXECUTE statements.
- Issue a CREATE PROCEDURE statement.

Sections are one of three types, depending on how you create them:

- Permanent sections, stored in the system catalog.
- Semi-permanent sections, deleted from memory after your session ends.
- Temporary sections, deleted from memory after your transaction ends.

Permanent sections are stored for non-dynamic statements you include in preprocessed applications. These sections are located in a system table called HPRDBSS.STOREDSECT, and their names are listed in the SYSTEM.SECTION view. You cannot access HPRDBSS.STOREDSECT, since it is an ALLBASE/SQL internal table.

Section Caching

Reading sections into memory from the system catalog at run time can cause a lot of I/O activity. Section caching allows sections to remain in memory between transactions. By default, up to 12 sections can be cached. Refer to the “System Administration” chapter for information about changing these defaults.

Validation

A section becomes invalid when something it depends on is no longer as it was when the section was originally marked valid. For example, a section containing a query on a particular table will be marked invalid if an index on that table is dropped, or when the UPDATE STATISTICS statement is used on the table.

A status of VALID means the section contains an appropriate access path for getting to data; a status of INVALID means the access path may no longer be appropriate.

When a section becomes invalid, it must be revalidated before it can be used again. SQLCore attempts to validate application program sections at run time. If a necessary section cannot be revalidated, an error results. During revalidation, exclusive locks are obtained on the system catalog base tables HPRDBSS.SECTION, HPRDBSS.STOREDSECT, and HPRDBSS.DEPENDENCY.

You can use the VALIDATE statement to revalidate sections singly or in groups at any time, such as after issuing the UPDATE STATISTICS statement. For best performance, issue all the UPDATE STATISTICS statements first, then issue all the VALIDATE statements. This ensures that you only invalidate and revalidate a section once, even if it is dependent on several tables.

Guidelines on Logical and Physical Design

This chapter shows how to adjust the logical and physical design of your databases for best performance. Additional general information on this subject appears in the “Logical Design” and “Physical Design” chapters of the *ALLBASE/SQL Database Administration Guide*. Refer also to the “Guidelines on System Administration” chapter below.

Logical Data Design

Logical data design concerns normalization of data, followed by the creation of tables, views, indexes, and authorization schemes. When designing logical objects in ALLBASE/SQL, *know your queries beforehand*. The following paragraphs present ways of adjusting your design to arrive at the best performance for the functionality you need to implement. The following topics are discussed:

- Normalization Issues.
- Including Calculated Data in Tables.
- B-Tree Index Design.
- Clustering Indexes.
- Choosing Appropriate Index Types.
- Authorization Design.
- Using a Duplicate Database.

Normalization Issues

Normalization results in tables with a relatively small number of columns, and a minimum amount of redundancy within a given table. In general, this yields an efficient design, but there are some drawbacks. The following table shows some of the pros and cons of normalization.

Pros and Cons of Normalization

Normalization Pros	Normalization Cons
Less or no data redundancy	Relatively larger number of table accesses (for joins)
Relatively more efficient use of storage space	Additional indexes (PCRs) required to maintain referential integrity.
Reduced data maintenance, since you don't need to update as many occurrences of a value	Increased overhead for processing more join queries.

Here are some tips:

- In general, design your tables to be in third normal form so that updates can be carried out against the smallest number of tuples.
- Avoid overnormalization of tables in logical design. If two reasonably static tables are joined frequently, consider making them into one table.

For example, consider the following two tables in third normal form:

employee	department

department	location

If you have a continual need to find the location of the employee, you might consider combining the two tables. In this case, you will incur extra maintenance if the location of the department changes, since it now appears in every tuple of Table C:

employee	department	location

When should you *not* combine tables, but stick with third normal form? Retain the third normal form for tables in which you must maintain several rows of data which are fully dependent on one particular key:

custno	address	state	zip	phone

custno	orderno	orderdate

Since Table E may contain many orders for each customer number, the amount of redundancy would be unacceptable if you were to include Table D's columns in each row of Table E.

Combining tables can also increase the concurrency between batch processes and on-line activities, since there is less contention for I/O resources within the system.

Denormalizing Tables that are Consistently Joined

Look for tables that are consistently being joined, and consider combining tables or duplicating some columns to prevent the need for the join. Considerations:

- Not all joins are expensive. If join columns are supported with unique indexes, the cost of doing a join is not really an issue.
- Only consider denormalizing when there is a performance problem that can be attributed to the join.
- Assess the impact of denormalizing on the performance of other processes that use the tables. Consider how often each process occurs, and tune for the most used processes.

2-2 Guidelines on Logical and Physical Design

- Duplicate data wastes storage space and can cause update problems.
- Tables that are combined in denormalization should have a 1:1 relationship with each other. The combining should not result in an excessive row length or an excessive number of indexes.
- Tables should have the fewest indexes possible.

Horizontal Partitioning

Large tables that contain historical data may be good candidates for division into two tables with the same column definition. To identify potential candidates for horizontal partitioning, review all your queries, looking for table scans with a low number of rows returned. Review other processes which would be impacted by dividing the table in two. Note especially the impact on processes which need *all* rows. Tune for the most critical or most frequently used processes.

As an example, consider a table containing daily stock prices. If the current year's prices are the only ones that are accessed frequently in looking up prices, then earlier data can be moved to a different table. Suppose the table is created as follows:

```
CREATE PUBLIC TABLE SECURITIES
  (NAME CHAR(10),
   DATE DATE,
   PRICE DECIMAL (6,2))
```

You can define a different table with the same column definition for the historical data:

```
CREATE PUBLIC TABLE PRE1992
  (NAME CHAR(10),
   DATE DATE,
   PRICE DECIMAL (6,2))
```

Then use a Type 2 INSERT to transfer the rows:

```
INSERT INTO PRE1992 SELECT *
  FROM SECURITIES
  WHERE DATE < '1992-01-01'
```

Finally, delete the 1992 data from the Securities table:

```
DELETE FROM SECURITIES
  WHERE DATE < '1992-01-01'
```

Vertical Partitioning

If some columns are infrequently used or if they require special security, consider moving them to their own table. Be sure to include the primary key values with every partition.

Vertical partitioning will make the rows smaller and more efficient to process (since there will be more rows per page). A disadvantage of having more rows per page is the potential of reducing concurrency because of page level locking. The use of row level locking may help offset this disadvantage.

Including Calculated Data in Tables

You can include calculated data in tables instead of performing calculations when data is retrieved. In this case, the overhead of the calculation occurs at INSERT or UPDATE time rather than at query time. Stored procedures are a useful way of recalculating whenever data values change. Some calculations can be done using ALLBASE/SQL functions such as SUM, MIN, AVG, etc.

You can create a view to return the results of a calculation, but this may degrade performance in some cases. Specifically, do not create views that contain aggregate functions or GROUP BY clauses. For better performance, define the view without aggregates or GROUP BY clauses and then apply the aggregation and grouping to the SELECT done on the view.

B-Tree Index Design

The use of indexes affects performance directly, since an index scan is generally faster than a serial scan under the following conditions:

- When only a small fraction of the rows in a table will be retrieved.
- When a large number of rows must be sorted.

Indexes are fast for retrievals, but slow for inserts, deletes, and some updates. You should consider the following questions when designing indexes:

- How often will an index be used in data retrieval? in joins?
- How often will an index be used in place of sorting? DBCore sorts data on columns associated with a GROUP BY, ORDER BY, DISTINCT, or UNION.
- How often will there be inserts, updates, or deletes to indexed columns?

If you are using unique or primary key constraints, ALLBASE/SQL automatically creates unique indexes which can be used for data retrieval. If you are using referential constraints, ALLBASE/SQL creates a PCR (parent-child relationship), which is an index on the two tables in the referential relationship. These may be less efficient than an index you create yourself.

Choosing Keys

- Create indexes on columns that frequently appear in an ORDER BY or a GROUP BY clause, or in the select list of a SELECT DISTINCT statement.
- For frequently performed joins, try to ensure that at least one side of each predicate in the WHERE clause has an index. For example, if you are joining the SupplyPrice table with the OrderItems table in the sample DBEnvironment (PartsDBE), you can improve performance by creating indexes on the common column, VendPartNumber, in both tables.
- Create indexes on columns frequently involved in a LIKE predicate. In such cases, be sure the literal string in the LIKE predicate does not begin with a wildcard character.
- Do not create indexes on columns containing YES/NO or other columns that can have only a few values (for example, day of the week).
- Always identify the primary and foreign keys in each entity before creating tables. This immediately identifies which indexes to create and which columns to specify as PRIMARY KEY or FOREIGN KEY columns.

2-4 Guidelines on Logical and Physical Design

- INTEGER and CHAR columns are more suitable as index keys than DECIMAL or DATE. INTEGER keys are best for extremely large tables (gigarows). Order of efficiency:
 - INTEGER
 - FLOAT
 - CHAR
 - DECIMAL
 - DATE
- Index keys greater than 20 bytes are not recommended.
- Consider creating an index of concatenated keys for WHERE clauses that contain AND's along with an EQUAL predicate (=).
- Create separate indexes on the columns on each side of an OR in a WHERE clause when the columns are from a single table.
- If you use compound indexes, be sure to use at least the first column of the compound index in your WHERE clauses.

Building Indexes on Large Tables

Indexes for large tables require special consideration:

- When creating an index for a very large table, be sure there is enough disk space for the index DBEFiles and for temporary sort space used while building the index. Make sure the directory has plenty of space, and make sure the index DBEFiles are large enough. See “Using TempSpaces” in the chapter “Guidelines on System Administration” for information on creating TempSpaces.
- When defining multiple indexes on very large tables (more than 3 million rows), use the following procedure in the specified order:
 1. Create DBEFileSet.
 2. Create DBEFiles for data (type = TABLE).
 3. Create table.
 4. Load data into table.
 5. Create one DBEFile (type = INDEX) large enough to hold the first index plus sufficient additional space for growth.
 6. Create the first index.
 7. Create one DBEFile to hold the next index.
 8. Create the next index.
 9. Repeat the previous two steps until all indexes are created.

Since the algorithm used when creating an index uses the last DBEFile defined, each index will now reside in its own DBEFile. (Caution: do not create a TABLE type DBEFile in the middle of the process of defining indexes, since this will cause the search for a DBEFile to start at the beginning instead of using the last DBEFile defined.)

If you use this process, page splits on indexes will point forward in the same DBEFile, resulting in better performance for inserts (though not for retrievals) on very large tables.

- For special applications, indexes can be added or dropped at any time. In these cases, consider how much data is involved, and estimate the time and resources needed to drop and recreate.

- Creating a B-tree index after loading data into the table causes minimal logging. In this case, key data is sorted and then loaded into index pages without page splitting. The result is faster index creation and a more compact index than when creating the index before loading data.
- Tables and indexes for which the most I/O is performed should be placed on the fastest available disk. If a choice must be made, place the index (not the table) on the fastest disk. Use the SQLUtil MOVEFILE command to place DBEFiles on different devices.

Maintaining Indexes

Periodic review of indexes ensures the greatest efficiency. Use the following recommendations:

- Review all indexes for usage, considering all queries (including ad hoc queries). Drop all indexes that are not used.
- Review available data about process usage, looking for table scans that return only minimum amounts of data. Consider creating an index.
- B-tree indexes grow in depth as rows are inserted; that is, the number of intermediate levels between root page and leaf page increases. However, as rows are deleted, the depth of the B-tree does not decrease. If many additions and deletions have been done, consider dropping and recreating the indexes.
- If update time is more critical than retrieval time, consider dropping indexes.
- Be sure to keep the number of indexes small on update-intensive tables.
- Use SQLMON to monitor index usage. The SampleIO Indexes screen indicates which indexes are used the most. The Static Size screen displays the size of each index, and the Static Cluster screen displays an index's cluster count.

Clustering Indexes

Applications which frequently access rows in index order will have better performance if the rows are physically stored on disk in index order. Performance is better because I/O is minimized. For example, a query that needs to read all rows that have a certain value will be faster if all those rows reside on one page, instead of on many pages.

To automatically maintain table data in index order, ALLBASE/SQL allows B-tree indexes and referential constraints to be defined as **clustering** when they are created. When a clustering index has been defined on a table, rows added to the table are placed physically near other rows with similar key values *whenever there is space available on the page*. If space cannot be found on the desired page, then the next row is inserted onto the first page in the DBEFileSet that has space.

A clustering index does a good job of maintaining index order when the number of insertions is approximately equal to, or less than, the number of deletions on the table. Under these conditions, space for a newly inserted row will usually be found on a page having rows with similar keys. But when the number of insertions is greater than the number of deletions, space will usually not be found on the desired page for the row. In this case, time and I/O will be wasted in calculating, locating, and examining the desired page even though the row must be inserted in some other page in the DBEFileSet. Thus, you pay the price at INSERT time for the attempt to optimize the placement of the row using a clustering index.

If space does not usually exist on table pages, then a clustering index should not be defined on the table, even if your applications frequently access the data in index order. You should use a non-clustering index instead, and manually maintain the table data in index order. You do this by monitoring the cluster count of the index, and performing UNLOAD operations followed by sorting and reloading. This process is explained more full later in this chapter, in the sections “Monitoring the Cluster Count” and “Reclustering a Table.”

Using Clustering Indexes

Consider using a clustering index when accessing a subset of rows with the same key value or when retrieving a sequential range (including an ORDER BY, GROUP BY, DISTINCT, or UNION). Note the following:

- Clustering indexes are more expensive than non-clustering indexes for inserts and index column updates.
- You can create empty spaces on data pages by adding “dummy” rows during the initial table load and later deleting them. This process is known as “sparsing” the table.
- You may have only one clustering index per table.

If a table rarely has rows deleted from it, and if it frequently has rows added to it, then you should not define a clustering index. Extra time (and I/O) will be wasted when rows are inserted in looking for free space for the row.

If you decide to use clustering, sort your data before loading the table. Create the clustering index after sorting data and loading the table.

Monitoring the Cluster Count

To measure how well clustered a B-tree index is (whether it is defined as clustering or non-clustering), ALLBASE/SQL uses the cluster count value, which is stored in the CCOUNT field of SYSTEM.INDEX or SYSTEM.CONSTRAINTINDEX when you do an UPDATE STATISTICS. The CCOUNT is the number of data page accesses that were made during the scan. The CCOUNT is increased by one each time the next row is not stored on the same page as the last row seen. The same page can be accessed multiple times if the data is not physically stored in index order. The best case is when the CCOUNT value is equal to the number of pages in the table. If the CCOUNT is higher than this value, then more I/O than is minimally necessary might be required for an index scan over the entire table. The worst case is when the CCOUNT value equals the number of rows in the table.

To monitor the cluster count with SQLMON, invoke the Static Cluster screen and examine the CCOUNT, TOT ROWS, and UNLOAD/LOAD SUGGESTD fields.

You can also query the system catalog to examine the cluster count for all B-tree indexes:

```
SELECT T1.INDEXNAME, T1.OWNER, T1.TABLENAME,  
T1.CCOUNT, T2.NROWS, T2.NPAGES  
FROM SYSTEM.INDEX T1, SYSTEM.TABLE T2  
WHERE T1.OWNER = T2.OWNER  
AND T1.TABLENAME = T2.NAME;
```

Use the following query to examine the cluster count for all unique or referential constraints:

```

SELECT T1.CONSTRAINTNAME, T1.OWNER, T1.TABLENAME,
T1.CCOUNT, T2.NROWS, T2.NPAGES
FROM SYSTEM.CONSTRAINTINDEX T1, SYSTEM.TABLE T2
WHERE T1.OWNER = T2.OWNER
AND T1.TABLENAME = T2.NAME;

```

You should use this command *after* doing an UPDATE STATISTICS on the tables of interest to make sure the cluster count is up-to-date. For indexes used in sorting or for retrieving large numbers of rows, the cluster count is especially important. A low cluster count may indicate good performance. Note that a clustering index does not necessarily result in a good cluster count.

Reclustering a Table

If the UNLOAD/LOAD SUGGESTED value of the SQLMON Static Cluster screen is high, or if the CCOUNT in SYSTEM.INDEX or SYSTEM.CONSTRAINTINDEX grows to twice the number of pages in a table, you should recluster the table.

You may be able to improve performance by reclustering the table using the following procedure:

1. Unload the data from the table using the INTERNAL option. Use a SELECT with an ORDER BY on the index key so that data is unloaded in sorted order.
2. Drop any referential constraints that exist on other tables referring to the unloaded table.
3. Drop the table. This will also drop all B-tree indexes that have been defined on the table.
4. Issue a COMMIT WORK statement. This makes the space occupied by the table and its related indexes and constraints available for reuse.
5. Issue the CREATE TABLE statement to recreate the table. Do *not* specify any constraints.
6. Load the data back into the table using the INTERNAL option.
7. Use the CREATE INDEX statement to recreate the clustering index, or use the ALTER TABLE statement to ADD a clustering constraint. Note that only one clustering index or constraint can be defined on a given table.
8. Use the ALTER TABLE statement to add any other constraints that should be defined for the table.
9. Recreate any other B-tree indexes or referential constraints that may have been dropped.
10. Do an UPDATE STATISTICS to update the system catalog information about the table and its indexes and constraints.
11. Issue a COMMIT WORK statement.

Using Hash Structures

For some tables, hashing offers better performance than B-tree indexes. Hashing works best with a fairly uniform distribution of key values, spreading the corresponding rows evenly over the number of pages available. A key with a skewed distribution will attempt to place all rows on a correspondingly skewed set of pages. The worst key results in hash values that cluster tightly in a narrow range of primary pages, leaving others empty.

You must define the hash at the same time you create the table. *The hash key must be unique.* Considerations:

- Hash retrieval is faster than B-tree retrieval unless there is a bad overflow situation. The larger the number of overflow pages, the slower the access to the average tuple.
- You may also define B-tree indexes on a hash table.

2-8 Guidelines on Logical and Physical Design

- An insert to a hash structure must perform a separate calculation to arrive at the correct page for each row of data. An ordinary non-indexed insert must find an available page in the DBEFileSet, and finding this page may require several I/Os; but once the location for new inserts is determined, ordinary inserts are less expensive than hash inserts. B-tree inserts are more expensive than either hash or non-indexed inserts.
- You cannot update the hash key column directly; instead, you must delete and re-insert the row.
- You cannot drop a hash index without dropping the table itself.
- Smaller rows mean more rows per page, which means less chance for overflow when several rows hash to the same page.
- The DBEFile or DBEFiles for the hash must be empty at the time you create the table, and there must be at least enough DBEFile space to hold all primary pages. You should add an appropriate number of pages (say, 30%) for growth and overflow. To determine how much empty space a DBEFile has, run SQLMON and go to the Static DBEFile screen.
- Use a prime number for the number of primary pages. An odd number is almost as good.
- Single-column integer values provide better hashing than multi-column or character indexes.
- You can use hashing to spread rows evenly among the pages of a small table for the greatest page-level concurrency.
- Do *not* load a hash table in key-sorted order. Doing so may result in a performance penalty.
- If the number of overflow pages grows too large, you can unload and reload the table. See “Unloading and Reloading to Remove Overflow Pages” later in this chapter.

Choosing Appropriate Index Types

It is important to consider whether a B-tree index is an appropriate choice for a given table. The larger the table and the lower the level of update activity, the greater the benefit of a B-tree index. A B-tree never hurts a read-only application. Indeed, B-tree indexes are usually defined to improve the read access requirements of an application. B-tree indexes can, however, degrade the performance of an application that updates the table on which the B-tree is defined because whenever a row is inserted or deleted, the B-tree index must also be updated. In addition, if the application updates a key column in any row, then the B-tree index must also be updated.

The needs of the applications that access a table must carefully be considered before a B-tree index is defined on the table. If the mix of transactions involving the table is primarily read-intensive, then a B-tree index can be defined to improve performance. If the mix is primarily update-intensive, then consider the following:

- If rows are frequently inserted and/or deleted, then hashing should probably be used instead of a B-tree index. Hashed tables do not require a second update, as B-tree indexes do. However, hashed tables may require more manual maintenance for optimal performance. Such maintenance may involve occasional unloading and reloading of data.
- If updates are made to the key columns in the row, then performance degradation cannot be avoided: a B-tree index will automatically be updated; and an update on the hash key requires that the old row be deleted and the new row inserted in two separate operations. You might consider changing the hash key or index key to a less volatile set of columns if this is possible.

Updating Statistics

In order for the optimizer to make the best choice among available indexes, use the UPDATE STATISTICS statement on tables after creating indexes and after doing many inserts or deletes. After updating statistics, use the VALIDATE statement to revalidate any stored sections that were invalidated by the UPDATE STATISTICS statement. While revalidation is taking place, performance may be impaired due to obtaining exclusive locks on the system catalog. To avoid concurrency problems, use the VALIDATE statement after using the UPDATE STATISTICS statement. Preferably, you should use both statements during periods of low activity.

Note

If you do not update statistics at appropriate times, the optimizer cannot choose the appropriate access path to data. In order to avoid performance degradation, be sure your statistics are up-to-date. You can query SYSTEM.TABLE to determine the time the last UPDATE STATISTICS was done for a given table. The following example shows the latest US time for PurchDB.Parts:

```
SELECT NAME, OWNER, USTIME
FROM SYSTEM.TABLE
WHERE NAME = 'PARTS'
AND OWNER = 'PURCHDB'
```

Authorization Design

The creation of authorization schemes can also affect performance, since ALLBASE/SQL has to look up authorizations before carrying out specific commands. The more complex the authorization scheme, the more system table pages must be read by the query processor. The only authorization verified at run time for non-dynamic queries is RUN authority. Therefore, it helps to keep a simple hierarchy for RUN authorization. In general, user group hierarchies should be kept simple (one or two levels) unless it is absolutely necessary to use more. If the user running an application is the owner of the module or has DBA authority, the authorization check is faster.

The overhead of authorization checking is higher for dynamic queries. This is because the *first* execution of a dynamic query in a transaction requires the query to be re-authorized. It is therefore all the more important to keep the authorization hierarchy simple.

In designing authorization schemes, you need to weigh the needs of security and ease of maintenance against your requirements for performance.

Using a Duplicate Database

To avoid lock contention, some database designers create a main database for updating and a duplicate or subset database for reporting. Data can be transferred periodically from the main database to the duplicate in batch mode.

Physical Data Design

Physical data design means the arrangement of data inside tables, which reside in DBEFiles, and the arrangement of DBEFiles in DBEFileSets. This section includes the following topics:

- Creating DBEFileSets.
- Creating DBEFiles.
- Creating Tables.
- Initial Table Loads.
- Unloading Data.
- Unloading and Reloading to Avoid Indirect Rows.
- Unloading and Reloading to Remove Overflow Pages.
- Tips on Deletions from Tables.

Additional information relating to physical design is found in the section “Load Balancing” in the chapter “Guidelines on System Administration.” You may also find it useful to review the “Physical Design” chapter in the *ALLBASE/SQL Database Administration Guide*.

Creating DBEFileSets

DBEFileSets are logical groupings of DBEFiles. DBEFileSets are also the logical home of tables, which are always created in a particular DBEFileSet. The following guidelines offer suggestions on how to distribute tables among DBEFileSets.

Avoiding the SYSTEM DBEFileSet for User Data

Although you can create user-defined tables in the SYSTEM DBEFileSet, it is recommended that you create new DBEFileSets to keep user data separate from system catalog data to facilitate maintenance and performance tuning. Here are the reasons why:

- System catalog data is accessed frequently. If user data is kept in the SYSTEM DBEFileSet, then the pages used for system data become intermingled with the pages used for user data. If ALLBASE/SQL needs to access a system page that physically resides near the end of the DBEFileSet, then it must first look through all the page table pages that come before it. The average time to access a particular system page increases because of the extra data stored within the DBEFileSet.
- Sometimes ALLBASE/SQL uses pages in the SYSTEM DBEFileSet for temporary space. If this space is not available, processing will terminate in an unsuccessful state. Mixing user and system data makes it more likely that temporary space will not be available when needed. See the section “Temporary Space in SYSTEM” in the chapter “Guidelines on System Administration.”
- Grouping tables in separate DBEFileSets allows you to identify the space requirements for particular tables. If you store all tables in the SYSTEM DBEFileSet and the DBEFileSet runs out of space, it will be difficult to tell how fast each table is growing. It will also be harder to determine which table’s update or insert operation caused the system to run out of space.

In particular, you should avoid creating hash structures in the SYSTEM DBEFileSet.

Placing Large Tables in Separate DBEFileSets

You should place tables larger than 1000 pages in their own DBEFileSets. When more than one table is stored in the DBEFileSet, the pages of DBEFiles in the DBEFileSet become a mixture of pages from the different tables. In doing a sequential scan, if ALLBASE/SQL needs to access a data page that physically resides near the end of the DBEFileSet, it must first look through all the page table pages that come before it. The average time to access a data page thus increases because of the extra data stored within the DBEFileSet. The time increases severely when several large tables are stored in the DBEFileSet. To determine the size of a DBEFileSet, run SQLMON and go to the Static DBEFile screen.

Maintenance functions such as UPDATE STATISTICS also will take longer if many large tables are stored in a single DBEFileSet. Therefore, create a separate DBEFileSet for each table or group of tables that you want to maintain separately.

Gathering Small Tables into DBEFileSets

For smaller tables, there may be a performance advantage in grouping them together into DBEFiles of 1000 pages or less. When each small table occupies its own DBEFileSet, you can expect some of the following performance problems:

- I/O might increase, for several reasons:
 - Each page table page has little information on it. More pages would have to be read in from disk because the data physically resides on more separate pages.
 - Because each page table page has little information on it, it will not stay in the data buffer pool. Pages with more information have more locality of reference and are more likely to remain in the buffer pool (thus reducing real I/O).

If the data buffer pool were large enough, the actual increased I/O might be negligible.

- Each DBEFileSet requires at least one DBEFile, which is an actual operating system file. Opening a file to read from it is a slow operation. Putting each table into its own DBEFileSet increases the number of files, and therefore increases the average time needed to access the data in these tables.
- Operating systems have limits on the number of files that can be open simultaneously. To avoid exceeding the limit, successive close and open file operations may be required, which will degrade performance. ALLBASE/SQL imposes its own limit of 50 files that may be open simultaneously.
- The operating system must search for the correct file descriptor, which may take longer with more files open.
- Disk space usage would increase, since many virtually empty page table pages would exist.

When small tables are placed in a single DBEFileSet, disk space is used efficiently and I/O is minimized. For example, if you have fifty tables that average 5 pages, they will all fit comfortably into a single DBEFile and use only one page table page.

Creating DBEFiles

DBEFiles are the physical repositories of ALLBASE/SQL data on the operating system. Since you have great flexibility in defining DBEFile types, sizes, and assignment to DBEFileSets, your choices often affect the overall performance of your system. Here are some suggestions for creating DBEFiles:

- Create DBEFiles in sizes that are multiples of 253 pages to minimize the space used by page table pages. Each 253-page segment contains one page table page plus 252 pages of data following the page table page.
- Create separate TABLE and INDEX DBEFiles, especially for large tables, so that they can be placed on separate disks to improve I/O performance.
- Create a MIXED DBEFile in its own DBEFileSet for a small table (less than 1000 pages) with a single index. (In this case, data and index rows will be on alternating pages, which will result in faster processing and better space utilization).
- Use dynamic DBEFile expansion to avoid running out of DBEFile space at run time. In the CREATE DBEFILE statement, specify the initial size, the maximum size, and the increment size.
- Use raw files to shorten the I/O path length and eliminate operating system buffering of database pages. This feature is appropriate only for large DBEFiles containing large tables accessed randomly. Standard I/O is more appropriate for small DBEFiles or DBEFiles that are frequently accessed sequentially.

For information on raw files, refer to the appendix “Using HP-UX Raw Files for DBEFiles and Logs” in the *ALLBASE/SQL Database Administration Guide*.

Avoiding Extra DBEFile Space

When sequential scans are made frequently for queries involving tables in a particular DBEFileSet, extra space should be minimized. When sequential scans are made, every page table page in the DBEFileSet is read to determine if pages might qualify for the query. When a large amount of extra space exists, then extra I/O is required to fetch the page table pages for empty DBEFile space. One way to minimize the extra space in your DBEFiles is by specifying dynamic space expansion in the CREATE DBEFILE statement.

Creating Tables

Here are some general suggestions on table creation:

- Create PUBLIC and PUBLICROW tables for maximum concurrency. PUBLICREAD can improve performance by reducing internal page locking. Use PRIVATE for special purposes. Note that PRIVATE is the default; usually, you want to specify something else.
- Choose data types which suit the programming language so as to avoid data conversions.
- Native language (NLS) users should be aware that indexes and predicates that use USASCII (n-computer or C) columns perform faster than indexes and predicates that use native language columns. If there is a choice, use USASCII for index columns in your tables, create indexes on USASCII columns, and use USASCII columns in your predicates.
- Integer values can be 5 - 15% more efficient than packed decimal values.

- When adding a column with ALTER TABLE and adding a default value, every row in the table is immediately updated to add the new value. This causes all rows to become indirect rows. It may be better to unload data, drop and recreate the table with the new column and default, then reload the data.
- If a column is to be used consistently in the WHERE clause, it should have an index.
- Wherever possible, load data for non-hash tables in sorted order. This is essential for tables with clustering indexes. If a multi-column index exists, the order of its columns should be the same as the order of the columns in the load file. Hash tables should *not* be loaded in sorted order.
- Consider combining tables that are constantly joined together. It is a good idea to revisit normalization issues when actually creating tables.
- For large tables, place data and indexes on separate disks using the SQLUtil MOVEFILE command. In general, spread the load of database and log files over your disks, and keep data and log files on separate devices. Refer to the section “Load Balancing” in the “Guidelines on System Administration” chapter for more information.

Avoiding NULL and Variable Length Data

- Avoid NULLs and variable length data types (VARCHAR and VARBINARY) for the following reasons:
 - Nulls always require additional CPU to check for null values. Using NOT NULL can save as much as 5% in CPU because of the overhead of checking for nulls.
 - Use of NULL and VARCHAR or VARBINARY may cause wasted space due to the inability to use a shared tuple header. (A separate header must be stored for each tuple that has a different size.) However, the use of nulls or variable length columns may actually save space if there are only a few such rows per page. If the difference between the maximum size of the row and average size of data stored in the row is greater than the size of the header, then use of NULL or variable size columns may be more efficient.
 - Updating a NULL, VARCHAR, or VARBINARY column may cause the data to move to a different page, leaving an indirect tuple behind and increasing the number of I/Os required to access the data tuple.
 - You can create a table with default values instead of nulls so that when columns are updated they will not have to move to another page because of tuple size increase.
- Use an appropriate (not an unreasonably long) length for variable length columns. The maximum length can affect the performance of BULK SELECT and cursor operations.
- Indexes whose keys will be updated should never include NULL columns as keys, and rarely should they include VARCHAR or VARBINARY columns. Variable length keys can cause long index chains and rearranging when index values are updated.

Using INTEGER Rather than SMALLINT Data

Since SQLCore only performs 4-byte arithmetic, all SMALLINT values are automatically converted to INTEGERS before processing by SQLCore. These conversions do not affect the optimizer’s choice of a scan plan, but they may be costly in CPU for very large databases. To avoid the conversion, consider defining your SMALLINT columns as INTEGER columns. The drawback, of course, is that INTEGERS require twice as much storage space as SMALLINTs.

Initial Table Loads

For optimal performance, use the following ISQL commands before starting the load operation:

- **SET LOAD_BUFFER**

Use this statement to enlarge the load buffer beyond the default size of 16,384 bytes.

- **SET AUTOLOCK ON**

Avoid lock contention by locking the table in exclusive mode when the load is performed. For PUBLIC and PUBLICROW tables, locking the table in exclusive mode avoids the overhead of obtaining locks as pages of data are added to the table. If you are loading a table that has an index defined on it, locking the table in exclusive mode also provides optimal logging.

- **SET AUTOCOMMIT ON and SET AUTOSAVE**

Together these commands cause the number of rows specified with AUTOSAVE to be automatically committed when the load buffer is full. Should the load operation subsequently fail, you can insert the remaining rows with the LOAD PARTIAL command.

- **SET SESSION DML ATOMICITY AT ROW LEVEL**

Setting the DML atomicity to row level guarantees that savepoints will not be generated during a load, reducing logging overhead when running in non-archive mode.

- **SET SESSION UNIQUE, REFERENTIAL, CHECK CONSTRAINTS DEFERRED**

By deferring constraints, you avoid problems caused by the order in which dependent values are inserted into a table when foreign or primary key constraints exist. Constraint checking is deferred until the end of the load operation.

You may wish to load a table first, then add constraints to it using the ALTER TABLE statement. When the constraint is added, the data in the table is checked for consistency, and if inconsistent data is found, an error is generated. In this case, it is your responsibility to remove the inconsistent tuples before attempting to add the constraint.

Deferring constraints and setting atomicity to the row level together have the effect of reducing logging overhead greatly in non-archive mode because of the use of no-log pages. However, logging is still performed for the allocation of new pages, and the non-archive log file must be large enough to include a record for each page.

Additional tips for improving load performance:

- For subsequent unload and load operations, use the INTERNAL option of the ISQL UNLOAD and LOAD commands. LOAD INTERNAL is 3-5 times faster than LOAD EXTERNAL.
- Using archive logging during loading requires additional log file space, but it actually requires fewer I/Os than nonarchive logging. However, the saving of I/Os must be weighed against the time required to back up log files so they can be reused.
- Set the number of log buffers to at least 120 while loading.
- Set data buffer pages to 200 to reduce the size of the linked list and the expense of maintaining it.
- Add indexes after data is loaded. For clustering indexes, sort the data before the load.

- If you wish to use rules tied to INSERT operations, create the rules after loading the table. Otherwise, the operation of the rule will degrade the performance of the load. Note, however, that the operation of the rule is not retroactive, so you must ensure the consistency of your data at the time the rule is created.

Unloading Data

UNLOAD operations are subject to the following performance considerations:

- To recluster data to improve the cluster count, unload the table with an ORDER BY clause, delete all rows from the table, then reload the data in the new sorted order. This process is described more thoroughly earlier in this chapter under “Reclustering a Table.” Also, see the next item.
- For large tables, you can UNLOAD INTERNAL to tape. The tape must be labeled. It is important to use a blocking factor that is as close to 16K bytes as possible, since ISQL uses 16K byte chunks. If your block size is less than 16K, the unload and load will take longer and use more tape.
- For UNLOAD operations on large amounts of data that you intend to sort, consider unloading the data first and then sorting with an external utility rather than using the ORDER BY as part of the UNLOAD. In this case, you need to use UNLOAD EXTERNAL.

Unloading and Reloading to Remove Indirect Rows

The existence of indirect rows increases the amount of I/O that must be performed to obtain data. Indirect rows also consume more disk space, since they frequently are not able to use the shared tuple header. Therefore, indirect rows should be avoided. Use the following procedure to remove indirect rows from a table:

1. Unload the data from the table using the INTERNAL option.
2. Drop any referential constraints on other tables that refer to the unloaded table.
3. Drop the table. This will also drop any B-tree indexes that have been defined on the table.
4. Issue the CREATE TABLE statement to recreate the table. Do not include constraints.
5. Issue the COMMIT WORK statement to reclaim space.
6. Load the data back into the table using the INTERNAL option.
7. Use the CREATE INDEX statement to recreate any B-tree indexes that may have been dropped, and use the ALTER TABLE statement to add back constraints (including referential constraints) that may have been dropped.
8. Do an UPDATE STATISTICS for the table to update system catalog information about the table and its indexes and constraints.
9. Issue the COMMIT WORK statement.

Unloading and Reloading to Remove Overflow Pages

Since many key values can hash to the same page address, it is possible for a page in a hash structure to become full. When this happens, a new row must be inserted on an overflow page. Overflow pages increase the amount of I/O that must be performed to obtain table data. The larger the number of overflow pages, the slower the average access to any tuple.

Observe the AVGLEN column in SYSTEM.HASH or the AVGOVERFLOW field of the SQLMON Static Hash screen to see the average number of page accesses required to retrieve a particular row. As the number of overflow pages increases, so will this number. Increasing

the capacity of the table should reduce AVGLEN, and thus improve performance. Use the following procedure:

1. Unload the data from the table using the INTERNAL option.
2. Drop any referential constraints on other tables that refer to the unloaded table.
3. Drop the table. This will also drop any B-tree indexes that have been defined on the table.
4. Issue a COMMIT WORK statement. This makes the space occupied by the table and its related indexes and constraints available for reuse.
5. If necessary, create additional TABLE or MIXED DBEFiles and add them to the DBEFileSet that will contain the redefined table.
6. Issue the CREATE TABLE statement, specifying a larger number of primary pages than when you previously created the table. The CREATE TABLE statement should not include any constraints.
7. Load the data back into the table using the INTERNAL option.
8. Use the CREATE INDEX statement to recreate any B-tree indexes that may have been dropped, and use the ALTER TABLE statement to add back constraints (including referential constraints) that may have been dropped.
9. Issue an UPDATE STATISTICS statement for the table to update system catalog information about the table and its indexes and constraints.
10. Issue the COMMIT WORK statement.

Tips on Deletions from Tables

- When deleting all rows from a table, it is more efficient to use the TRUNCATE TABLE statement instead of the DELETE statement. On a large table, the DELETE statement may cause the log to run out of space, whereas the TRUNCATE TABLE statement incurs minimal logging.
- The performance of the TRUNCATE TABLE statement degrades when the table is hashed. Dropping the table and recreating it may be faster than using the TRUNCATE TABLE statement.
- If the table specified by the TRUNCATE TABLE statement is included in a referential constraint, it may be more efficient to drop the constraint, issue the TRUNCATE TABLE statement, and readd the constraint.

Guidelines on Query Design

Careful query design is of primary importance for performance. In most queries, the most efficient retrieval of data occurs when an index is used. By default, the ALLBASE/SQL query optimizer decides whether to use an index or not, and if so, it decides which one to use. However, you can override the query optimizer's choice with the SETOPT statement. Also, through careful query design, you can ensure that the optimizer is able to choose an available index. Topics included in this chapter are as follows:

- Avoiding Serial Scans Through Query Design.
- Using Subqueries.
- Using UNION.
- Using MIN/MAX Functions in Predicates.
- Using OR Predicates.
- Using Predicates with LIKE.
- Using Predicates with BETWEEN.
- Using Fetch Unique Scans.
- Updating Key Columns.
- Avoiding User Propagation of Filters.
- Using TID Scans.
- Using Parallel Serial Scans.
- Using the BULK Option.
- Analyzing Queries with GENPLAN.
- Modifying the Access Optimization Plan with SETOPT

Avoiding Serial Scans Through Query Design

To design efficient ALLBASE/SQL queries, you should keep in mind the conditions which force the optimizer to perform serial scans rather than faster index scans. Since certain classes of predicates cannot be evaluated using an index scan, you should avoid using such predicates in queries on large tables.

Arithmetic Expressions

An index scan cannot be used to evaluate a predicate that contains an arithmetic expression. For example, ALLBASE/SQL performs a serial scan for the following query:

```

SELECT  W
FROM    T1
WHERE   X = Y + Z - W

```

The following approach avoids the problem by assigning the result of the computation to a host variable which is then used in the predicate:

```

HostVar = Y + X - W

SELECT W
FROM T1
WHERE X = :HostVar

```

Columns from One Table on Both Sides of the Relational Operator

An index scan cannot be used to evaluate a predicate with columns from the same table on either side of a relational operator. For example, ALLBASE/SQL performs a serial scan for the following query:

```

SELECT  W
FROM    T1
WHERE   T1.X = T1.Y

```

Data Conversions

Conversion takes place in the predicates of queries. For example, in the predicate WHERE $X = Y$, a type conversion takes place whenever X and Y are not of the same data type. In general, you should avoid data type conversion, since it may mean that the optimizer will choose *not* to use an index. Avoiding conversions is partly a matter of table design—in defining columns with compatible data types—and partly a matter of query design. Therefore, *it is important to understand the queries that will be used with the database before creating the tables.*

For best results, columns constantly compared in the WHERE clause should be of the same data type and size; thus DECIMAL types should have the same precision and scale.

You can ensure that the optimizer is able to choose an available index by making sure that data conversion in your predicates is of an acceptable type, that is, that data types are **compatible**. In some cases, conversions are required which may result in the loss of significant information. In these cases, an index is not used. The following conversions *do not* result in the loss of significant information, because the data elements in them are compatible:

1. CHARACTER to CHARACTER or VARCHAR
2. VARCHAR to CHARACTER or VARCHAR
3. INTEGER to DECIMAL(p,s) where (p-s) >= 10
4. INTEGER to FLOAT
5. SMALLINT to INTEGER
6. SMALLINT to DECIMAL(p,s) where (p-s) >= 5
7. SMALLINT to FLOAT
8. DECIMAL to FLOAT
9. DECIMAL(p1,s1) to DECIMAL(p2,s2) where s2 >= s1 and (p2-s2) >= (p1-s1)
10. DATE, TIME, DATETIME, or INTERVAL to CHARACTER or VARCHAR
11. CHARACTER or VARCHAR to DATE, TIME, DATETIME, or INTERVAL

3-2 Guidelines on Query Design

In comparisons that result in conversions from INTEGER or SMALLINT constants or host variables to DECIMAL, an index may be used if the number of places to the left of the decimal point in the DECIMAL type (i.e., p-s) can accommodate the largest value yielded by the INTEGER (or SMALLINT). For example, a SMALLINT value is compatible with a DECIMAL (10,2), but an INTEGER value is not compatible with a DECIMAL (10,2). Comparisons that result in conversions between an INTEGER or SMALLINT *column* and a DECIMAL expression can only use an index if the following are true:

- The DECIMAL scale is 0 and
- The DECIMAL precision is
 - less than 10 for an INTEGER or
 - less than 5 for a SMALLINT

In the case of DECIMAL fields, the scale of both DECIMAL elements plays a role in determining whether or not an index can be used. For example, in converting between two DECIMAL elements, ALLBASE/SQL does use an index if the scale of the comparison value is greater than the scale of the column data type.

In the following query, an index can be used because both sides of the conversion are decimal, and the right side (comparison value) is convertible to the left (column data type) without loss of significant information:

SELECT W	decimal (7,2)
FROM T1	-----
WHERE X = 25.0	x y z w

Table T1

In this example, the left side (p2,s2) is DECIMAL (7,2) and the right side (p1,s1) is (3,1). We see that s2 is greater than s1 and p2 - s2 (5) is greater than p1 - s1 (2).

To retrieve data from an established database, you might need to compare columns in the WHERE clause that contain different data types. But there are several ways to help the optimizer by doing your own data conversion. For example, you can express an INTEGER constant as a DECIMAL when it is to be compared with a DECIMAL. In the following, assume that X is decimal:

Instead of	SELECT W FROM T1	use	SELECT W FROM T1
	WHERE X = 25		WHERE X = 25.0

Predicates with INTEGER = DECIMAL(n,0) Factors

Factors of the form Integer = Decimal (n,0) can be optimized by ALLBASE/SQL provided n is less than or equal to 10. Factors of the form Smallint = Decimal (n,0) can be optimized provided n is less than or equal to 5. In the following, an index may be chosen, since the decimal scale is zero and the precision is within the appropriate bounds. Assume that X is an integer:

```
SELECT W FROM T1
WHERE X = 45.
```

Using Subqueries

You can nest queries within the predicates of other queries. This makes it easier to express complex queries, and it makes it possible to formulate queries in several different ways. Thus you can create different queries which return the same rows but which do so with vastly different performance. In the following discussion, queries 1, 2, and 3 are all equivalent.

Most subqueries can also be expressed in an equivalent join form. For example, assuming that the PartNumber column in PurchDB.Parts is unique, the query:

```
(1) SELECT * FROM PurchDB.SupplyPrice
     WHERE PartNumber IN (SELECT PartNumber from PurchDB.Parts
                          WHERE PartName = 'Cache Memory Unit')
```

can also be written:

```
(2) SELECT PurchDB.SupplyPrice.*
     FROM PurchDB.SupplyPrice, PurchDB.Parts
     WHERE PurchDB.SupplyPrice.PartNumber = PurchDB.Parts.PartNumber
     AND PurchDB.Parts.PartName = 'Cache Memory Unit'
```

Both queries return the same information. In general, while the subquery is more easily understood and easier to formulate, the join actually improves performance because it gives the optimizer more efficient choices of how to execute the query. There are two types of subqueries: correlated and non-correlated. A correlated subquery is one in which the subquery makes reference to one or more columns of an outer query. For example,

```
(3) SELECT * FROM PurchDB.SupplyPrice
     WHERE EXISTS (SELECT PartNumber
                   FROM PurchDB.Parts
                   WHERE PartName = 'Cache Memory Unit'
                   AND PartNumber = PurchDB.SupplyPrice.PartNumber)
```

This is a correlated subquery, since the subquery makes reference to PartNumber from the outer query. In this case, the subquery must be executed for each row returned from the outer query. For the above query, since the SupplyPrice table has 69 rows in it, there must be one scan on the outer query and 69 scans on the inner query for a total of 70 scans. The equivalent join shown in query (2) can be executed using a sort/merge technique, which takes two sorts and two scans.

It should be noted that the predicate WHERE PartNumber IN of query (1) is internally the same as the predicate WHERE EXISTS in query (3). All quantified predicates involving

3-4 Guidelines on Query Design

subqueries get transformed into EXISTS predicates internally. Therefore the nested query using the IN predicate also takes 70 scans.

In general, non-correlated subqueries are faster than joins, and correlated subqueries are slower than joins. You should know whether the subquery will return a single row or not so as to take advantage of the speed of non-correlated subqueries. Since we know that each part in the Parts table is unique, there is no reason why we could not express query (1) as a non-correlated subquery by simply replacing the IN with an equal sign (=). Then the subquery does not depend on the outer query and the subquery can be executed only once. In this case, there is one scan for the outer query and one scan for the subquery for a total of two scans. This is even faster than the sort/merge join since there is no sorting involved.

One exception to the rule that correlated subqueries are slower than an equivalent join is when the equivalent join involves an aggregate. For example, consider the following query, which is transformed into a correlated subquery internally by the query processor:

```
SELECT VendorNumber, PartNumber, DiscountQty
FROM PurchDB.SupplyPrice
WHERE DiscountQty < ALL (SELECT DiscountQty
                        FROM PurchDB.SupplyPrice
                        WHERE VendorNumber = 9010)
```

becomes

```
SELECT VendorNumber, PartNumber, DiscountQty
FROM PurchDB.SupplyPrice sp1
WHERE NOT EXISTS (SELECT DiscountQty
                 FROM PurchDB.SupplyPrice sp2
                 WHERE VendorNumber =9010
                 AND sp2.DiscountQty <= sp1.DiscountQty)
```

The equivalent join would require the use of an aggregate, and would therefore be slower:

```
SELECT sp1.VendorNumber, sp1.PartNumber, sp1.DiscountQty
FROM PurchDB.SupplyPrice sp1, PurchDB.SupplyPrice sp2
WHERE sp2.VendorNumber = 9010
GROUP BY sp1.VendorNumber, sp1.PartNumber, sp1.DiscountQty
HAVING sp1.DiscountQty < MIN(sp2.DiscountQty)
```

In this case, because of the complexity of the query, the subquery is a better choice than the join. The best solution, however, is an equivalent non-correlated subquery:

```
SELECT VendorNumber, PartNumber,
       DiscountQty from PurchDB.SupplyPrice
WHERE DiscountQty < (SELECT MIN(DiscountQty)
                   FROM PurchDB.SupplyPrice
                   WHERE VendorNumber = 9010)
```

When Not to Use DISTINCT in Subqueries

In general, you should avoid using the DISTINCT keyword in subqueries. DISTINCT does not change the query result and, in fact, hinders performance.

The following two queries return the same result:

```
(4) SELECT * FROM T1 WHERE C1 IN (SELECT DISTINCT C2 FROM T2);
```

```
(5) SELECT * FROM T1 WHERE C1 IN (SELECT C2 FROM T2);
```

ALLBASE/SQL transforms IN predicates and also other quantified predicates where the subquery can return multiple values to make them similar to an EXISTS predicate. Therefore, queries (4) and (5) would be transformed to the two following queries:

```
(6) SELECT * FROM T1 WHERE EXISTS (SELECT DISTINCT C2 FROM T2
    WHERE T1.C1 = T2.C2);
```

```
(7) SELECT * FROM T1 WHERE EXISTS (SELECT C2 FROM T2
    WHERE T1.C1 = T2.C2);
```

These transformed queries are correlated in nature, meaning that the subquery result depends on values of the outer query. This means that the subquery needs to be reevaluated for each row of the outer query. Therefore, the performance of the subquery is critical.

When you use the DISTINCT keyword, the subquery must do a complete scan of the table, sort the values, eliminate duplicates, and then return TRUE if any rows are returned. Without the DISTINCT keyword, the subquery can scan until it finds the first qualifying row, return the row, return TRUE, and then terminate. As you can see, subqueries without the DISTINCT keyword are faster.

Using UNION

When using UNION in your queries, make sure you avoid data conversions, and be sure to define indexes on appropriate columns.

Avoiding Conversions

SQLCore checks the data types of each select list in a UNION query, and determines the result data type. For all select lists that have data types different than the result data type, a conversion is carried out if the types are compatible. Refer to the description of the SELECT statement in the “SQL Statements” chapter of the *ALLBASE/SQL Reference Manual* for a table showing the kinds of conversions.

The following example requires conversions:

```
CREATE TABLE T1 (Item CHAR(40), Price SMALLINT)
CREATE TABLE T2 (Item CHAR(40), Price INTEGER)
CREATE TABLE T3 (Item CHAR(40), Price DECIMAL(10,2))
CREATE TABLE T4 (Item CHAR(40), Price FLOAT)
```

```
SELECT Item, Price
FROM T1
UNION
SELECT Item, Price
from T2
UNION
SELECT Item, Price
FROM T3
UNION
SELECT Item, Price
FROM T4
```

Since the result data type in this UNION is FLOAT, sources 1, 2, and 3 require conversion to FLOAT, which is the largest common denominator type. Now if all columns were of the same type—for example, FLOAT—no conversions would be required and the performance of such a query would be faster than the conversion example.

Defining Indexes for UNION Queries

Each source SELECT in a UNION query is optimized individually, and SQLCore tries to pick the best access method for each source. Therefore, you should create indexes on all sources, if possible, to maximize performance. The result of the UNION is not optimized.

If you know that there are no duplicate rows generated by the query, or if you do not need to exclude duplicate rows from the result, the UNION ALL form is faster than UNION, because it does not sort the query result.

Using MIN/MAX Functions in Predicates

An index scan may be used for a query containing a single MIN() or MAX() function in the select list if an index exists on the column the function is applied to, and if the function is applied to the first column in the index. The index scan is considered for queries with or without predicates, including MIN() or MAX() on a join column in a nested loop join, provided the MIN() or MAX() is applied to a column in the outermost table.

An index scan plan for MIN() or MAX() cannot be considered for queries in which there is an expression, an ORDER BY or GROUP BY clause, or where tables are joined using the sort/merge join method.

Using OR Predicates

An index scan may be used for a query that has an OR predicate.

How OR Predicates are Optimized

Most predicates involving OR factors are transformed to conjunctive normal form to make the choice of an index scan during optimization more likely. In addition, the optimization of OR predicates involves internally ANDing additional factors to the predicate you supply in order to eliminate duplicates. **Conjunctive normal form** expresses a predicate as the conjunction of factors rather than the disjunction of factors. For example, if a predicate has the following elements:

```
(c1=10 AND c2=20) OR (c1=10 AND c3=30)
```

there are two factors in disjunctive normal form. ALLBASE/SQL transforms the predicate as follows:

```
(c1=10 OR c1=10) AND (c2=20 OR c1=10) AND  
(c1=10 OR c3=30) AND (c2=20 OR c3=30)
```

Now there are 4 factors in conjunctive normal form.

The transformation of a predicate into conjunctive normal form increases the number of factors in the predicate. This can result in exceeding the maximum number of factors in a predicate (currently 256). You can avoid this problem by writing your predicates in conjunctive normal form yourself, as in the following:

```
(c1=10) AND (c2=20 OR c3=30)
```

The only exception to this rule is a predicate containing OR and BETWEEN, as in the following:

```
(c1 BETWEEN 10 AND 20) OR (c2 BETWEEN 30 AND 40)
```

Each BETWEEN predicate is actually a conjunction of two range predicates:

```
(c1 >= 10 AND c1 <= 20) OR (c2 >= 30 AND c2 <= 40)
```

Such a predicate is not in conjunctive normal form. However, in this case, you should *not* rewrite the predicate in conjunctive normal form, nor does ALLBASE/SQL transform the predicate. Both range predicates from a BETWEEN predicate can be used in a single index scan. Therefore, ALLBASE/SQL can make best use of this predicate when it is in the original form.

Choosing an Index for OR Factors

For index scan plans to be chosen for an OR factor (for example, c1=10 OR c2=20), the following conditions must be met:

- All columns involved in the OR factor must come from a single table.
- Each column involved in the OR factor must have an index defined on it.
- The OR factor must require no data type conversions that result in the loss of significant information.

Based on these conditions, here are some suggestions for query and index design:

3-8 Guidelines on Query Design

- If a query predicate includes *only* OR factors (all columns in one OR factor from a single table), it is a good idea to define a multicolumn index on all columns involved.
- If a query predicate includes both OR factors and simple EQUAL factors (for example, (c1=10 OR c2=20) AND c3=30), the EQUAL factor (c3=30) may yield a cheaper plan than the OR factor. In such cases, it is a good idea to ensure that there is an index on the columns involved in the EQUAL factor.
- To ensure that no unacceptable data type conversions are required, columns and expressions being compared in the factors should be of compatible data types. For example, if c1 is a column of type integer and c2 is a column of type decimal (2,1), the following predicate would require unacceptable data type conversions, and therefore index scan plans may not be chosen:

```
(c1=1.0 OR c2=2.00)
```

The following predicate would not require any unacceptable data type conversions, and therefore index scan plans might be chosen:

```
(c1=1 OR c2=2.0)
```

Using Predicates with LIKE

An index scan may be used for a query with a LIKE predicate. However, there are situations in which a table scan is a better choice for good performance. For example, for LIKE pattern values that start with a wildcard, ALLBASE/SQL needs to scan the whole relation. In this case, a table scan is usually the optimal scan to use. Here are some suggestions for the use of LIKE in predicates:

- For a multicolumn index to be chosen by the optimizer, the LIKE predicate must be on the first column in the index if only one column is used. For example, a predicate that includes LIKE C1 might benefit from an index defined on C1,C2 but it would not benefit from an index defined on C2,C1.
- Avoid using a LIKE pattern (value or host variable) that starts with a wildcard, since the entire table must be scanned in this case.
- For a LIKE predicate on a column that contains clustered data, the performance is improved dramatically when an index is used. Since the data is clustered together, no extra I/O is needed to search for the next tuple.
- Avoid using NOT LIKE, since an index plan is not generated for a NOT LIKE predicate.
- For LIKE predicates, an index can only be used if all previous columns in the index have values supplied in the predicate. For example, if a multicolumn index exists on (C1, C2, C3) in a table, the following predicate may result in the choice of an index scan:

```
WHERE C1 = 12 AND C2 = :HostVar AND C3 LIKE 'J%'
```

However, assuming the same three-column index, the following would *not* result in the choice of an index scan:

```
WHERE C1 = 12 AND C3 LIKE 'J%'
```

Predicates containing LIKE are only optimized if the language of the column is n-computer.

Using Predicates with BETWEEN

For the BETWEEN clause, the optimizer makes the decision depending on the following (among other things):

- The range of the BETWEEN.
- The range of the HIGH and LOW values of the index column(s).

For example, if the range in the BETWEEN predicate is very wide relative to the HIGH and LOW values stored in the index, the optimizer expects a large number of rows to be returned and is therefore more likely to choose a serial scan. (Note that the optimizer assumes a uniform distribution of values.) In the case of BETWEEN predicates with host variables, the actual range of values cannot be known until run time, so the optimizer uses default selectivity values.

Using Fetch Unique Scans

Fetch Unique is a special faster kind of index scan method which is automatically used when a predicate meets the following criteria:

- There is exactly one EQUAL factor for each key column.
- All factors must be exactly matching with a unique index.
- No non-unique index columns occur in the WHERE clause.
- The factors in the WHERE clause are ANDed together.
- There are no indicator variables in the WHERE clause.
- Data retrieval is done with a singleton SELECT, not a cursor. (A singleton operation is one in which only a single row qualifies.)

Providing a non-cursor SQL SELECT, UPDATE, or DELETE meets these criteria, it uses a Fetch Unique scan.

Updating Key Columns

Ordinarily, you can update columns with the benefit of an indexed scan. When you want to update the key column or columns on which an index is based, the index scan is only be used in some circumstances. In general, the optimizer chooses an indexed update if there is an EQUAL predicate, as in the following examples:

```
UPDATE T1 SET c1 = c1 + 10 WHERE c1 = 20
UPDATE T1 SET c1 = 20 WHERE c1 = 10
```

However, the optimizer chooses a table scan in updating the key column if the EQUAL predicate is combined with an OR factor:

```
UPDATE T1 SET c1 = c1 + 10 WHERE c1 = 10 OR c1 = 20
```

The optimizer also chooses a table scan in updating the key column if there is anything other than EQUAL (for example, GREATER THAN) in the predicate involving the index column:

```
UPDATE T1 SET C1 = C1 + 10 WHERE C1 >= 20
```

Avoiding User Propagation of Filters

A **filter** is an element in a predicate that reduces the size of the query result by eliminating a category of result rows. Consider the following join query fragment:

```
... WHERE table1.col1 = table2.col2
      AND table1.col1 <= 200
```

Note that, logically, table2.col2 has to be <= 200 as well; that is, it is subject to the same filter as table1.col1. ALLBASE/SQL propagates such filters from one joined table to the other joined tables without your explicitly doing so. Thus the above query is translated internally into:

```
... WHERE table1.col1 = table2.col2
      AND table1.col1 <= 200
      AND table2.col2 <= 200
```

This internal translation is done for all equal-joins on multiple tables. Both range predicates (<, <=, >, >=) and the equals predicate (=) are propagated. The result is a performance improvement on equal-joined queries when there are one or more range or equal predicate filters on one or more of the tables.

Since the optimizer does not check for duplicates in filters, you should *not* explicitly propagate filters in writing the queries. If you do, SQLCore processes the same filter twice. This causes performance to deteriorate without adding anything significant.

Using TID Scans

When the tuple ID (TID) is known, you can access a particular row without using an index. Use the TID() function in a SELECT statement to provide TIDs for later use in manipulating an individual row.

The TID scan provides the fastest possible data access when you need to access a single row in a singleton SELECT, or in a non-cursor UPDATE or DELETE statement. Cursor operations do not benefit from the use of the TID() function.

Using Parallel Serial Scans

When you need to sequentially read a large table, you can improve performance by using a parallel serial scan. The ALLBASE/SQL optimizer uses a parallel serial scan when it is able to prefetch pages from multiple disk drives in parallel. A parallel serial scan is useful only for large tables that must be read sequentially. It is not beneficial for small tables or for tables that are accessed only by an index.

To take advantage of a parallel serial scan, the following conditions must be met:

- The DBEFileset containing the table must contain multiple DBEFiles.
- The DBEFiles containing the table must be placed on separate disk drives. You can specify the disk drive with the `DEVICE` clause of the `CREATE DBEFILE` statement. If the DBEFile has already been created, you can use the `SQLUtil MOVEFILE` command.
- The DBEFiles cannot be raw because the file system's prefetching must not be bypassed.
- The DBEFiles must not contain pages from other tables or index pages from the same table. Put other tables in different DBEFilesets. Use separate `INDEX` and `TABLE` DBEfiles for the index and data pages of the table.

Using the BULK Option

Using the BULK feature in embedded SQL with the `INSERT`, `SELECT` and `FETCH` statements can reduce the number of SQLCore calls. Even though it uses a large internal buffer to process queries, ALLBASE/SQL makes a separate call to SQLCore for each row in a simple `SELECT` or `FETCH`. If the BULK option is used, ALLBASE/SQL needs a smaller number of calls to SQLCore, thus reducing overhead. Since BULK operations can be 2 to 10 times faster than their non-BULK counterparts, you should use them wherever possible. Note that you cannot employ `UPDATE WHERE CURRENT` or `DELETE WHERE CURRENT` statements when using BULK operations.

When using the BULK option, declare an array as close as possible to 12K bytes or a multiple of 12K bytes, since this is the size of SQLCore's tuple buffer.

Analyzing Queries with GENPLAN

The GENPLAN statement can be useful in determining the way to write a `SELECT`, `UPDATE`, or `DELETE` statement for maximum performance. GENPLAN lets you see the optimizer's access plan for one ALLBASE/SQL statement at a time. Issue the GENPLAN statement in ISQL, then simply do a `SELECT` on the temporary table `SYSTEM.PLAN` within the same transaction. A knowledgeable user may be able to use this information to formulate queries differently, make other design changes to improve performance, or use the `SETOPT` statement to override the optimizer's access plan.

Whenever users have a performance concern about a query, they can use the GENPLAN statement to generate the access plan for the query and display the plan information by doing a `SELECT` on the temporary table `SYSTEM.PLAN`. By using GENPLAN and `SELECT` statements, users can easily investigate the performance question.

Refer to the *ALLBASE/SQL Advanced Application Programming Guide* chapter, “Analyzing Queries with GENPLAN,” and the *ALLBASE/SQL Reference Manual* for more information about the GENPLAN statement.

Modifying the Access Optimization Plan with SETOPT

By default, the ALLBASE/SQL optimizer chooses the access plan for queries. You can specify the access plan with the SETOPT statement. For example, you can specify the use of either serial, index, or hash access for a particular statement. See the “SETOPT” section in the *ALLBASE/SQL Reference Manual* for more information.

Guidelines on Transaction Design

Transaction design is important for performance because it determines the type and duration of locks held by your applications. By selecting appropriate lock types, and by avoiding unnecessary locking, you can improve performance. Topics in this chapter include the following:

- General Tips on Managing Transactions.
- Controlling Locking.
- Using Isolation Levels.
- Using Row Level Locking.
- Using KEEP CURSOR.
- Removing Non-Database Processing.
- Using Procedures and Rules.
- Tuning Performance of Dynamic Statements.

General Tips on Managing Transactions

- Non-cursor SELECT involves less overhead than FETCH with a cursor. Use SELECT or BULK SELECT if you have a choice.
- Use the KEEP CURSOR option with the OPEN statement to permit frequent release of locks during long cursor operations. A combination of Cursor Stability and KEEP CURSOR can improve concurrency by letting you scan and update a large table without holding locks for the duration of the entire scan. (Do not forget to use the COMMIT WORK statement immediately following an OPEN for a kept cursor.)
- Use DML only mode (i.e., DDL Enabled set to NO through SQLUtil) to improve concurrency on access to system catalog tables.
- Try to perform data definition in single-user mode as much as possible. Data definition involves such activities as CREATE, DROP, ALTER, UPDATE STATISTICS, GRANT, and REVOKE. It is advisable to perform these operations outside of a production application if at all possible, since these operations place exclusive locks on the system catalog tables. These locks, like all other exclusive locks, are held for the duration of the transaction, and will reduce concurrency as other users attempt to access the system catalog tables. For more information on system catalog locks, refer to the appendix, “Locks Held on the System Catalog,” in the *ALLBASE/SQL Database Administration Guide*.

Using Short Transactions and Savepoints

At the end of a transaction, the COMMIT WORK statement makes changes permanent to disk, which causes I/O. Short transactions free locks and buffers more frequently, which improves concurrency, but they also increase log I/O (the increase is slight, since the use of group commits in logging dilutes the effect of increased log I/O in this case). Longer transactions minimize I/O, but they hold locks longer and thus reduce concurrency. In general, you should keep your transactions as short as possible to improve the performance of the DBEnvironment.

A **savepoint**, created in a transaction with a SAVEPOINT statement, marks a place you can roll back to in the transaction, releasing locks that were obtained since you issued the statement. Savepoints can be used to reduce the number of transactions that must be resubmitted because part of the transaction was unsuccessful.

Note Whether or not you use savepoints, the entire transaction is rolled back in the event of a deadlock.

Controlling Locking

ALLBASE/SQL supports a variety of lock granularities, lock types, and isolation levels to enable a transaction to lock only what is necessary to keep other transactions from interfering with its work. For a complete general discussion of locking and concurrency issues, refer to the chapter “Concurrency Control through Locks and Isolation Levels” in the *ALLBASE/SQL Reference Manual*. This section concentrates primarily on locks and performance. To monitor lock activity, run SQLMON and access the screens in the Load and Lock subsystems.

Locking degrades performance in two ways:

- A transaction must wait if the object it needs is already locked in an incompatible mode by some other transaction.
- Deadlocks sometimes occur.

An application that is well tuned for performance has a low rate of deadlock and a high rate of concurrency. In reality, however, a tradeoff is usually necessary. A low deadlock rate is often achieved by limiting the number of users attempting to obtain a lock by locking at a coarse level of granularity (for example, the table level rather than the page level). This strategy tends to increase the wait time for the lock (thereby reducing concurrency). Conversely, a short wait time for locks is usually achieved by locking at a finer level of granularity (for example, the page level rather than the table level). This strategy tends to increase the number of deadlocks.

When any ALLBASE/SQL statement is executed, page locks are acquired on one or more system tables (that is, tables owned by the special user HPRDBSS). Page locks and row locks are also acquired on certain ALLBASE/SQL internal tables (that is, tables owned by DBCore). You cannot directly change the locking behavior of these tables.

When ALLBASE/SQL statements that reference a user table are executed, row, page, and table locks of different kinds may be obtained on the table. You can help control what kind of locks are obtained and how long they are held by one of the following strategies:

4-2 Guidelines on Transaction Design

- You can modify the implicit locking structure of the table by changing the table type with the ALTER TABLE statement.
- You can use the LOCK TABLE statement to override the implicit lock mode for a given transaction.
- Instead of RR, you can use the CS (Cursor Stability), RC (Read Committed), and RU (Read Uncommitted) isolation levels to reduce the duration of certain locks in a transaction.

These strategies can help promote improved concurrency and reduced deadlocks.

Using CS, RC, and RU Isolation Levels

You can improve concurrency by using the Cursor Stability (CS) or Read Committed (RC) isolation levels with the BEGIN WORK statement. The effect of these options is to release shared locks before the transaction ends, whereas Repeatable Read (RR) holds them until the end of the current transaction. Read Uncommitted (RU) promotes still greater concurrency by not obtaining any locks on user tables for read operations. The greatest benefit is obtained with RU if all your applications are using it. This allows a minimum of locking in the system, and a minimum of waiting for other locks to be released.

Use appropriate isolation levels for the kind of read/write operations you are performing:

- Use Cursor Stability for long serial reads with occasional updates. Cursor Stability will increase concurrency during serial reads. It also improves the throughput for a single writer waiting on multiple readers. When using Cursor Stability in a transaction, row and page level READ locks are released behind you as you move through a table.
- Use Read Committed for read-only operations in which it is important to access committed data. Read Committed releases locks as soon as data is read.
- Use Read Uncommitted for read-only operations in which it is not important that all the data you read has been committed. Read Uncommitted does not obtain locks, so it permits you to read dirty pages, that is, pages that may be in the process of being updated by some other transaction. RU operations are known as **dirty reads**.

Users of CS, RC, and RU should be aware of the following:

- Regardless of isolation level, write operations (INSERT, UPDATE, DELETE) obtain exclusive locks, which are not released until the end of the transaction.
- Despite the choice of a different isolation level, system catalog pages are still locked at the RR level. Therefore, deadlocks and lock waits involving system catalog pages are possible if your applications use DDL (data definition language).

All isolation levels work with sorted query results and with Type 2 INSERT statements. For more information, see the chapter “Concurrency Control Through Locks and Isolation Levels” in the *ALLBASE/SQL Reference Manual*. Also see the description of the BEGIN WORK statement in the “SQL Statements” chapter of the *ALLBASE/SQL Reference Manual*.

Using Row Level Locking

Row level locking provides the finest level of lock granularity, where only the row that is read or updated is locked. By locking the row alone, ALLBASE/SQL allows other concurrent transactions to access other rows on the same page. This is in contrast with page level locking, where an entire page containing the row is locked, with the result that concurrent transactions accessing the same page must wait until the lock is released.

You enable row level locking for a specific table by defining the table to be of type PUBLICROW in the CREATE TABLE statement. For PUBLICROW tables, ALLBASE/SQL uses row rather than page level locking. A table may also be changed to PUBLICROW by using the ALTER TABLE statement, as in the following example:

```
ALTER TABLE PurchDB.Parts SET TYPE = PUBLICROW
```

Benefits of Row Level Locking

In general, row level locking can be used to reduce or eliminate the frequency and length of lock waits on **hot spots**. Hot spots are data storage areas, such as pages, that are accessed frequently by concurrent transactions. By locking at a finer level of granularity, the overall throughput of the system can be increased by reducing lock waits.

Small tables are good candidates for row level locking. This is especially true if the row size is small, and many rows fit on the same page, and if the table is frequently accessed by concurrent transactions.

Large tables with hot spots may also be good candidates for row level locking. An example is a history table in which small portions of the table, such as the most recent history, are read and updated frequently by concurrent transactions, even though most of rest of the table remains untouched. In such cases, use row level locking with caution, especially if the table may be used by some transactions in serial scans.

Hot spots that develop at either end of an index are *not* alleviated by row level locking. This situation can arise if there is a series of inserts or deletes of keys at either end by concurrent transactions. For example, problems can occur on inserting successive rows containing CURRENT_DATETIME into a table with an index on the DATETIME column. Hot spots arise because key inserts and deletes lock the data pointed to by the neighbor index entry to enforce repeatable read and constraints.

Shared Memory Considerations

A locked table, page, or row is represented in shared memory by means of a **lock object**. Lock objects are stored in lock control blocks in shared memory. The greater the number of lock objects, the more control blocks are required in shared memory. If a page is locked, then all rows on the page are implicitly locked. In this case, only one lock object is needed to represent the lock in memory. However, if rows are locked in a page, then a separate lock object is needed for every row on the page.

Use row level locking carefully. In allocating a row level lock, ALLBASE/SQL will place an intention lock on the table, another intention lock on the page, and the requested lock on the row. Thus row locking uses more CPU than table or page level locking and is less efficient. Row locking also generally uses more runtime control block pages than page and table locking. For these reasons, you should avoid using row level locking if the entire table will be scanned using an index.

4-4 Guidelines on Transaction Design

As an example, consider a table having 100 pages containing 100 rows each. A scan of the whole table will acquire 100*100 row locks for row locking, in addition to 100 intention locks on the pages and an intention lock on the table, for a total of 10,101 lock objects. With page level locking, the total is only 101 lock objects.

A table that is a good candidate for row level locking is one in which the following are true:

- Rows are small. If each row takes up a page, then page level locking has about the same effect as row level locking, but page level locking is more efficient.
- There will be concurrent write operations or concurrent read and write operations. If there will only be concurrent read operations, then the table should be PUBLICREAD. If only one transaction will access the table at a time, then the table should be PRIVATE or PUBLICREAD.
- A relatively small number of rows will be locked. This is true in the following cases:
 - The number of rows in the table is small.
 - The number of rows in the table is large, but all transactions are trying to access the same small number of rows.
 - The overall table size is small compared to the number of concurrent transactions that are expected to access the table. High concurrency becomes more critical when more transactions are trying to access the same data.

If a large number of rows will be locked, then more shared memory and CPU will be required to manipulate the lock objects than with page level locking.

As an example, if 240 concurrent transactions randomly access a table which contains 20 pages and if each page contains 200 tuples, then the table is a good candidate for row level locking. If the table contained only one tuple per page, or if only read operations were expected, or if the table were 20,000 pages in size, then the table would not be a good candidate for row level locking.

Page Locking on PUBLICROW Tables

Sometimes during updates, ALLBASE/SQL needs to acquire locks on pages instead of rows even if the table is a PUBLICROW table. The following are examples:

- When a new page must be allocated before inserting a new row.
- When a page must be deallocated after deleting the last tuple on a page.
- When a page must be compressed to reclaim freed space.

Page locks are also acquired when the transaction is performing a serial scan on the table at the Repeatable Read (RR) isolation level.

Using KEEP CURSOR

After you specify `KEEP CURSOR` in an `OPEN` statement, a `COMMIT WORK` does not close the cursor, as it normally does. Instead, `COMMIT WORK` keeps the cursor open and begins a new transaction while maintaining the cursor position. This makes it possible to update tuples in a large active set, releasing locks as the cursor moves from page to page, instead of requiring you to reopen and manually reposition the cursor before the next `FETCH`.

Locks on the page of data corresponding to the current cursor position are either held (the default) or released, depending on whether you specify `WITH LOCKS` or `WITH NOLOCKS`. The `KEEP CURSOR` option retains the current isolation level (`RR`, `CS`, or `RC`) that you have specified in the `BEGIN WORK` statement. Moreover, the exact pattern of lock retention and release for cursors opened using `KEEP CURSOR WITH LOCKS` depends on the current isolation level. With the `RC` and `RU` isolation levels, no locks are maintained across transactions because locks are released at the end of the `FETCH`. Therefore, `KEEP CURSOR WITH LOCKS` has no effect on locks at the `RC` or `RU` isolation levels. Also, the `WITH LOCKS` option releases exclusive (`X`) locks on user tables at `COMMIT WORK` time in transactions using `RC` and `RU`. Remember that system catalog tables are locked at the `RR` isolation level; exclusive locks on these tables are retained until the `CLOSE` cursor statement that is immediately followed by a `COMMIT WORK`. For more information and examples, refer to the chapter “Processing with Cursors” in the appropriate *ALLBASE/SQL Application Programming Guide*.

Removing Non-Database Processing from Transactions

Transactions affect performance because they hold locks until they are terminated with a `COMMIT WORK` or `ROLLBACK WORK` statement. If the transaction contains processing that could actually be performed outside the transaction, this processing takes time, which means that locks may be held longer than needed, which reduces concurrency and degrades performance. Such extra processing should be removed from the transaction.

Whenever possible, avoid holding locks around terminal reads. As a general rule, all user input should be retrieved before the start of the transaction. This helps to keep the transaction as short in duration as possible, and it has the following advantages:

- Locks are not held around user prompts, thus avoiding application “hangs.”
- The result of the transaction can be viewed and used by other users sooner, thus improving throughput.

When user input must be accepted during a transaction, you can use `ALLBASE/SQL` features that help avoid excessive locking. In cursor operations, you can use `KEEP CURSOR WITH NOLOCKS` when reading data. For cursor or non-cursor operations, you can use the `RC` or `RU` isolation level. When using these options, you can use the `REFETCH` statement to acquire locks and revalidate data before updating tuples.

Using Procedures and Rules

Stored procedures, used independently or invoked by rules, can improve performance in the DBEnvironment. For applications that access a local DBEnvironment, combining multiple SQL statements and logic in a stored procedure can improve performance by reducing the overhead of multiple calls to SQLCore.

For applications that operate on a remote DBEnvironment or in a client/server environment, each call to SQLCore must travel across the network. In this environment, the use of stored procedures can dramatically reduce network traffic and thus improve performance.

It is generally better to place conditions in the *FiringCondition* of a rule rather than including them as conditional statements within the procedure invoked by the rule, to minimize the overhead of procedure execution. The specification of a *FiringCondition* minimizes overhead by avoiding unnecessary procedure calls. For each row affected by the statement, the *FiringCondition* is tested, and the procedure is invoked only when the *FiringCondition* is satisfied. Thus, the specification of a *FiringCondition* can improve the performance of data manipulation operations on the table on which the rule is defined.

Since a procedure executed by a rule may itself execute statements that fire rules, the developer should carefully analyze the relationships among rules and the procedures invoked by rules to ensure that the performance implications on data manipulation statements that fire a chain of rules are well understood.

Tuning Performance of Dynamic Statements

A given ALLBASE/SQL statement can be processed either statically or dynamically. At run time, a dynamic statement must be preprocessed before executing; therefore, a static section may offer better performance. However, dynamic processing is often desirable for reasons of portability or flexibility. (For a comparison of the two types of statements, see the “Comparing Static and Dynamic Statements” section of the *ALLBASE/SQL Advanced Application Programming Guide*.)

When your application must execute a dynamic statement more than once, you can achieve improved performance by using dynamic parameters or semi-permanent sections.

Using Dynamic Parameters

When your application uses dynamic processing, parameter substitution offers added flexibility and improved performance. Although you can use this technique in any dynamic processing application involving prepared sections, it could be most useful for applications where the same SQL statement type must be re-executed multiple times using a different set of actual parameter values each time.

A statement containing dynamic parameters must be dynamically preprocessed at run time by using the PREPARE statement. The dynamic section created by PREPARE can then be executed as many times as required with the option of assigning a different set of dynamic parameter values for each execution without the overhead of preprocessing each time input values change.

For example, the following UPDATE statement specifying two dynamic parameters could be put into either a string or a host variable (in this case a string) in your program, then prepared and executed:

```
PREPARE Cmd FROM
'UPDATE PurchDB.Parts SET SalesPrice = ? WHERE PartNumber = ?;'
```

Execute the dynamic command using host variables to provide dynamic parameter values:

```
EXECUTE Cmd USING :SalesPrice, :PartNumber
```

You could now loop back to provide different values for SalesPrice and PartNumber. Note that the dynamic command does not have to be prepared again.

When your application will be inserting multiple rows of data, you might be able to use a BULK INSERT statement containing dynamic parameters to provide efficient performance.

The chapter “Using Parameter Substitution in Dynamic Statements” in the *ALLBASE/SQL Advanced Application Programming Guide* contains further information including detailed code examples.

Using Semi-Permanent Sections

Semi-permanent sections improve performance when your application executes dynamic queries more than once. Unlike temporary sections, semi-permanent sections are retained in memory when the current transaction ends. Semi-permanent sections are deleted from memory only when the DBEnvironment session ends. Semi-permanent sections, like temporary sections, are not stored in the DBEnvironment. When using semi-permanent sections, set the Authorize Once per Session flag to ON with the SQLUtil ALTDBE command:

```
>> ALTDBE
DBEnvironment Name: PartsDBE
.
.
.
Authorize Once per Session (on/off) (opt):on
Alter DBEnvironment Startup Parameters (y/n)?y
>>
```

To make a section semi-permanent, include the REPEAT clause in the PREPARE statement. In the following example, the section containing the UPDATE statement is semi-permanent:

```
PREPARE REPEAT Cmd FROM
'UPDATE PurchDB.Parts SET SalesPrice = 100.00 WHERE PartNumber = ''1124-P-02'''
```

Guidelines on System Administration

System administration includes both DBA functions for ALLBASE/SQL and system administrator functions for HP-UX. Careful coordination between these roles is important in performance tuning. This chapter includes information on

- DBA guidelines
- Network guidelines
- HP-UX system guidelines

DBA Guidelines

Several aspects of database administration affect performance. To improve performance in these areas, you need DBA authority.

Validating Your Applications Before Run Time

When you run an application, ALLBASE/SQL first checks each section for validity before executing it. Then, ALLBASE/SQL revalidates invalid sections (if possible) before executing them.

You may notice a slight delay as the sections are being revalidated. Revalidating sections can reduce concurrency, because it involves updating system catalog tables. For each update, ALLBASE/SQL must obtain an exclusive page lock on the system catalog tables. The locks are retained until the transaction is committed.

To improve performance, you can revalidate your applications before run time by either:

- Preprocessing the program again.
- Using the `VALIDATE` statement on the affected sections. For the complete syntax of the `VALIDATE` statement, refer to the *ALLBASE/SQL Reference Manual*.

You should also be careful when you use the `UPDATE STATISTICS` statement, because it invalidates sections. For best performance, after the initial statistics for a table have been established, use `UPDATE STATISTICS` only during periods of low DBEnvironment activity. Then, after you use `UPDATE STATISTICS`, use `VALIDATE` to revalidate modules and procedures.

You may want to use `VALIDATE` after executing an `UPDATE STATISTICS` statement or any DDL statement, since these statements invalidate stored sections. If you issue such statements during a period of low activity for the DBEnvironment (at night, for example), the DBEnvironment experiences minimal performance degradation.

Developing Application Programs

The following tips relate to preprocessing of application programs:

- Use separate development and production DBEnvironments, because during development, frequent re-preprocessing of applications locks system catalog pages for extended periods. You can later use the ISQL INSTALL command to move finished modules into your production DBEnvironments. The INSTALL command locks the same system catalog pages as the preprocessor, but it only does so a single time.
- Avoid using DDL when running in multiuser mode, since DDL places exclusive locks on system catalog resources and can reduce throughput significantly. (DDL includes the UPDATE STATISTICS and CREATE INDEX statements, among other statements.) You can also restrict DDL to single-user applications that run during off-peak hours. When you are not using DDL, you can set the DDL Enabled flag to NO to achieve significantly higher throughput in a multiuser environment.
- Use section caching. Note that the first connect to a DBEnvironment is slower when DDL is disabled, since extra caching is being done. See the section “Section Caching and Directory Caching” in this chapter for more information.
- Use the TERMINATE USER statement to abort a user program or ISQL session, if necessary. DBCore does a set critical while accessing the DBEnvironment, and it only allows a program to abort at certain strategic points. If DBCore is filling the tuple buffer, it completes this operation before allowing a break. TERMINATE USER allows the program to abort immediately.

Balancing System Load

Balancing system load is an effort to spread out I/O in your system among the available disk devices for the most efficient operation. Large systems can benefit significantly from an analysis of system load, followed by reallocating the DBEFiles for specific DBEFileSets or within a DBEFileSet. In general, the DBEFiles on which the most I/O is performed should be stored on the fastest devices available.

For best performance, separate data DBEFiles, index DBEFiles, and log files across the available disks on your system. Specifically, make sure to:

- Store log files on separate devices from data DBEFiles to improve performance and ensure against a possible head crash. For the same reason, store dual log files on different devices.
- Move DBEFiles to different devices with the SQLUtil MOVEFILE command (for DBEFiles) and the SQLUtil MOVELOG command (for log files).
- Separate index DBEFiles on large tables from their data DBEFiles. You can use the SQLUtil MOVEFILE command to move the index DBEFile. For small tables, this technique probably yields a less noticeable performance gain.

Placing Concurrently Used Objects on Different Drives

If two database objects are used concurrently, you should store them on different disk drives to minimize disk head contention.

This includes any two tables that show I/O at the same time, any two B-tree indexes that show I/O at the same time, or a table that shows I/O at the same time as the B-tree index defined on it. To monitor I/O, run SQLMON and access the SampleIO subsystem. When you

access objects simultaneously, the disk heads may move back and forth between the objects. This movement slows access to data. To avoid this problem, use the SQLUtil MOVEFILE command to place the competing objects on different drives.

Remember that the mapping of database objects to DBEFiles is not one-to-one, and that MOVEFILE moves DBEFiles, not DBEFileSets. In order to achieve the goal of placing different objects on different drives, you must ensure that they are in different DBEFiles. Therefore, it is useful to create separate INDEX and TABLE DBEFiles, and to define a different DBEFileSet for each table or index that may need to be separate from other objects.

Calculating Shared Memory Allocation

As you adjust the various configurable parameters in ALLBASE/SQL, be sure you do not exceed the amount of shared memory available on your system. You can use the following formula to derive an approximate figure (in kilobytes) for the total shared memory used:

$$SM = (4.2*DBP) + (0.5*LBP) + (0.16*NTXN) + (4.1*CBP)$$

where

SM = Total Shared Memory in kilobytes
DBP = Data Buffer Pages [15 - the limit of shared memory]
LBP = LogBufferPages [24-1024; default 24]
NTXN = Number of Concurrent Transactions [No limit]
CBP = Control Block Pages [17-800; default 37]

The result of this equation must be within the amount of shared memory you have available. This equation also gives an idea of the relative effect of changing different parameters on the total shared memory size.

For the default parameters, you can use

$$SM = 4.2*DBP + 164$$

And for the maximum parameters, you can use

$$SM = 4.2*DBP + 3830$$

These equations will help you arrive at a first approximation of the amount of shared memory that has been used.

Choosing a Number of Data Buffer Pages

Data buffers are used to cache data pages touched by ALLBASE/SQL. Up to the limit of buffer space, a data page is retained in memory until the space it occupies is needed for another page that is on disk. You can set the number of data buffer pages your DBEnvironment uses. The minimum number of data buffer pages is 15, and the default number is 100. The maximum is determined by the amount of shared memory available. How many data buffer pages should you allocate? There is no explicit equation to find the ideal buffer pool size. The following general suggestions may help, though this is not an exhaustive list:

- Total shared memory should not exceed free real memory available on the system. Available real memory can be determined only at run time, since it depends on the total number of processes on the system and on the type of application programs running on the system. The DBA may be able to make estimates. Excessive shared memory causes page faults.

- You never get a performance benefit by defining more page space than stays in real memory. If the data buffers force paging of virtual space, having too many buffers degrades performance. Shared memory only holds buffers temporarily, since real buffering occurs in the operating system. Try to use a large buffer pool within the limits of the first two suggestions. However, a larger pool usually slows down checkpoints if the fraction of dirty pages remains the same. There is a trade-off between checkpoint time and response time.
- When possible, the buffer pool should accommodate all pages that are accessed frequently. If a small table or index is frequently accessed, then the buffer pool should accommodate all its pages, if possible.
- If users are accessing a table without much locality of reference (that is, with almost random access) and if the total size of the table is much larger than any practical buffer pool size, then increasing the size of the buffer pool is not helpful. For example, if a table is 100 MBytes large, then a 2000-page buffer pool does not work much better than a 1000-page pool.
- When there is a high hit ratio, increasing the number of buffer pages can help performance. For example, if the table has a 2000-page index that is used frequently, then a 2000-page buffer pool performs better than a 1000-page pool.

When should you increase the number of buffer pages? If the set of applications does *not* have locality of data and index pages, it may not matter how many data buffers there are. Once you have enough data buffer pages, performance is not very sensitive to adding more.

Keeping a Small Group of Pages in Memory

How do you keep a small, frequently used group of pages in memory consistently? The trick is to cause other pages to be swapped out first. ALLBASE/SQL employs a **least recently used** (LRU) algorithm to discard older pages and make room for new pages. To keep a special group of pages in memory, you must have other pages in memory as well that can age past the pages in the special group. If you use a large enough number of pages outside the special group, each page will age to the point of being least recently used before any of the special group is swapped out.

Use the following suggestions to determine how many data buffer pages is enough:

- Examine the performance-crucial transactions. (Usually there are only a few that are sensitive or dominate performance). Look at the tables that are used in these transactions.
 - INSERT actions tend to cluster at the end of a table, so if there is insert activity, add a buffer page for each active user (that is, the instantaneously active count).
 - If a table or index is small enough to fit in real memory and is heavily accessed, add the number of pages in the table or index.
 - If an index is large, but the number of high level index pages is small and the table is heavily accessed, add a number of pages corresponding to the number of index levels above the leaf level.
 - If a table is too big to fit in real memory and the access is random, add a buffer page for aging for each transaction before the small table is accessed. This allows for LRU.
 - Estimate the table access pattern in your transactions. If small tables are accessed in 50% of them or more, you should gain by having a large enough buffer pool to hold the small tables.

- The right number of buffers is approximately the number needed to hold the small tables and the aged pages of the large tables.

Basic Example

For this example, assume that a sample application

- updates a small table
- updates a large table
- inserts to a third table

Assume that the tables for the sample occupy the following space in DBEFiles:

- The small table and its index use 50 pages: one root index page, 10 leaf index pages, and 39 data pages.
- The large table uses 50,000 pages: one root index page, 100 second-level index pages, 11,899 leaf index pages, and 38,000 data pages.

Also assume that:

- Each insert into the third table requires 40 bytes, and the pages of the third table fill up about every 100 transactions.
- There are about 10 active transactions in the database at any instant.

At a given instant, the buffer pool looks like this:

Source	Operation	Pages Already In Data Buffer	Pages Requiring Physical I/O
Small table	Update	11 (index root and leaf) 10 (data)	40
Large table	Update	1 (index root) 10 (second-level index pages active per transaction) 10 (leaf pages active per transaction) 10 (data pages active per transaction)	100 (total index second-level pages) 12,000 (total index leaf pages) 50,000 (total data pages)
Third table	Insert	10 (data pages active per transaction)	
Total in cache		62	

Based on the numbers of pages given above, and assuming the default number of data buffer pages (100), the application will require an average of 7 I/Os per transaction, as shown in the following table:

Source	Operation	Number of I/Os
Small table	Read data	1
	Write data	1
Large table	Read second-level index	1
	Read leaf index	1
	Read data	1
	Write data	1
Third table	Write data	.01
Commit	Write log	1
Total		7.01

There is a 25% chance that a random access to the small table will want a page that is already in the buffer pool. There is a 10% chance that the large index second-level page is already in the buffer pool. These reduce the I/O count to about 6.4 instead of 7, as shown in the following table:

Source	Operation	Number of I/Os
Small table	Read data	.75
	Write data	.75
Large table	Read second-level index	.9
	Read leaf index	1
	Read data	1
	Write data	1
Third table	Write data	.01
Commit	Write log	1
Total		6.41

First Threshold for Performance Gain

System performance improves if you add more data buffers. The first threshold of performance gain is when there are enough data buffers to prevent the pages of the small table from being swapped out. This will happen only if the small table's pages never become least recently used.

Suppose that we want 62 pages to stay in memory from the previous calculation, and that we will insert about 10 pages. In each transaction, accessing the large table will require touching a root page, two more index pages, and a data page. If the two nonroot index pages and the data page are read each time, we touch three pages that are new.

We need enough pages in memory so that pages from the large table are less recently used than pages from the small table. A minimal estimate would be 120 pages; a more comfortable estimate would allow 160 pages. Using the 120-page minimum, the buffer pool would look like this:

Source	Operation	Pages Already In Data Buffer	Pages Requiring Physical I/O
Small table	Update	11 (index root and leaf) 40 (all pages in buffer)	
Large table	Update	1 (index root) 40 (second-level index pages—active and “aging”) 40 (leaf pages—active and “aging”) 40 (data pages—active and “aging”)	100 (total index second-level pages) 12,000 (total index leaf pages) 50,000 (total data pages)
Third table	Insert	10 (data pages active per transaction)	
Total in cache		182	

Here is the resultant pattern of I/Os:

Source	Operation	Number of I/Os
Small table	Read data	0
	Write data	0 (deferred until checkpoint)
Large table	Read second-level index	.9
	Read leaf index	1
	Read data	1
	Write data	1
Third table	Write data	.01
Commit	Write log	1
Total		4.91

Second Threshold for Performance Gain

The next performance gain comes from fitting all the pages from the second level of the large table's index into the data buffer cache. There are 100 pages in the second-level index, so the buffer pool in this scenario looks like the following:

Source	Operation	Pages Already In Data Buffer	Pages Requiring Physical I/O
Small table	Update	11 (index root and leaf) 40	
Large table	Update	1 (index root) 100 (second-level index pages) 100 (leaf pages—active and “aging”) 100 (data pages—active and “aging”)	12,000 (total index leaf pages) 50,000 (total data pages)
Third table	Insert	10 (data pages active per transaction)	
Total in cache		362	

Here is the resultant pattern of I/Os:

Source	Operation	Number of I/Os
Small table	Read data	0
	Write data	0 (deferred until checkpoint)
Large table	Read second-level index	0
	Read leaf index	1
	Read data	1
	Write data	1
Third table	Write data	.01
Commit	Write log	1
Total		4.01

This analysis predicts one plateau of performance at about 180-200 pages in the buffer pool, a second plateau at about 360-400 pages. The next plateau would require something like 25,000 pages, which is impractical for most installations.

In practice, there is little penalty for extra pages, so use a generous allocation (that is, at least 200 or 400 pages), if you can afford to.

Note The previous calculations assume only the pages involved in one transaction type involving a small and a large table. If you have other transactions contending for the database, be sure to allow adequate buffer space for them too.

Cautions

- The size of the page pool combined with other demands on memory should not exceed available real memory.
- Checkpoints may take longer if the buffer pool is larger. A checkpoint writes to disk all dirty pages in the buffer pool. If there is a consistent fraction of dirty pages, the checkpoint takes longer with a large buffer pool. However, if the large buffer pool is filled with read-only data, the checkpoint may not take much longer. A rough estimate can be obtained by looking at the buffer pool size derived in the analysis above.

An Empirical Approach

Using SQLMON, you can observe the data buffer miss rate and adjust the number of data buffer pages accordingly. Start by allocating the largest number of data buffer pages you are willing to reserve for ALLBASE/SQL. This number should be less than the following:

$$\begin{aligned} & (\text{Total Size Virtual Memory}) - (\text{Number of Log Buffer Pages}) * 0.5K \\ & \quad - (\text{Number of Run-Time Control Block Pages}) * 4K \\ & \quad - 45K \end{aligned}$$

As you observe the data buffer miss rate on SQLMON's IO screen, gradually reduce the number of data buffer pages until the miss rate increases sharply. Allocate just enough data buffer pages to avoid the sharp increase of the data buffer miss rate. Since the load on your system probably varies, you should monitor the data buffer miss rate at different times throughout the day.

Choosing the Size of the Runtime Control Block

The runtime control block is an area of shared memory containing global, runtime information for the DBEnvironment. ALLBASE/SQL allocates control blocks from the runtime control block. Lock management is the single greatest user of control blocks. Each table, page, or row lock acquired needs one control block. As the granularity of the lock decreases, the number of locks required is likely to increase.

For example, locking an entire table with row level locking requires more locks than locking the table with the LOCK TABLE statement. The following table lists the maximum number of locks that can be associated with table, page, and row-level locking:

Granularity	Maximum Number of Locks
Table	1
Page	(number of pages in table) + 1
Row	(number of pages in table) + (number of rows in table) + 1

An application that manages locks well is less likely to deplete the amount of shared memory available. If the runtime control block is too small, ALLBASE/SQL is unable to allocate the necessary control blocks, an error is returned, and the transaction is rolled back. If the runtime control block is too large, other processes on the system may not have adequate access to memory.

To allocate pages for the runtime control block, you can use the START DBE statement or the ALTDBE command. To monitor the usage of the runtime control block, start SQLMON and go to the Overview screen, as described in the chapter “Getting Started with SQLMON.”

Choosing a Number of Log Buffer Pages

The number of log buffers is independent of the number of data buffers. The log buffer pages are only 512 bytes (in contrast with the data buffer pages, which are 4096 bytes). If the log buffer pool becomes full before any transaction completes, then at least two I/Os are needed to complete the transaction.

You should provide enough log buffers to allow all active transactions to have log space. Additional buffers do not help performance, but adding pages probably doesn't hurt either. To monitor the I/O incurred by logging, run SQLMON and access the IO screen.

For example, suppose that a user is making changes to a sample database and that

- Before and after images of the changed data cause about 600 bytes to be logged.
- The insert is 100 bytes.
- We allow 100 bytes for log overhead.

The figures listed above total 800 bytes. You can then round up to 1024 bytes, or 2 512-byte pages. This implies that 20 log buffer pages should be sufficient for 10 transactions. If there are occasional bursts of more than the average, 40 or 50 pages might be a good idea.

Choosing the Number and Size of Log Files

ALLBASE/SQL gives you great flexibility in choosing the number and size of log files. The following suggestions may be useful in making choices for your system.

Nonarchive Log Guidelines

The size of nonarchive log files determines the frequency of automatic system checkpoints, and it therefore determines rollback recovery time. The total size of the log (all files taken together) determines the largest transaction that can be accommodated in the DBEnvironment.

Some guidelines for determining the size of the nonarchive log are listed below:

- Total log space should be large enough to accommodate changes from the largest transaction. Otherwise, large transactions may abort. To determine the amount of log file space available, run SQLMON and go to the Overview screen.
- ALLBASE/SQL performs an automatic checkpoint each time a log file becomes full. By choosing a specific file size, you can force a checkpoint to take place at specific intervals, thereby controlling the amount of time required for rollback recovery.
- In general, a large log file is good for nonarchive logging. However, a checkpoint can cause significant loading to be put on the system when the log file is large. When a checkpoint occurs, all data pages relating to the transactions currently held in the log file must be forced to disk if they have not already been swapped out. If you use very large log files, the checkpoint process can flood the I/O subsystem and cause interference with other higher priority processing. If such flooding occurs, use a smaller log file or use the CHECKPOINT statement to force a checkpoint at regular intervals.
- The transaction that runs out of log space triggers a checkpoint, and this causes all transactions to wait until the buffers are flushed.

5-10 Guidelines on System Administration

- When you use a large log buffer size, you avoid frequent flushing of the buffer to disk.
- Large nonarchive log files can reduce the number of system checkpoints. To display the number of system checkpoints, run SQLMON and examine the IO screen. If you observe a large number of checkpoints and log buffer writes on the IO screen, you should increase the size of the nonarchive log files.
- Checkpoints should be postponed as far as possible, as long as rollback recovery time is within limits. The reason is that checkpoints flush dirty pages, and this prevents the saving of writes in those cases where the same page gets updated frequently. Also, checkpoints increase disk contention.

Archive Log Guidelines

- The transaction that runs out of log space triggers a checkpoint, and this causes the transaction to wait until the buffers are flushed. Other transactions may continue if logging is in archive mode.
- In archive mode, use at least two log files, so that one can be backed up while the other is being used.
- In archive mode, use large log files to minimize the frequency of log backups and checkpoints.

Sorting Operations

The sort method in ALLBASE/SQL is memory-intensive, using the tournament sort algorithm for sorting and merging. It sorts in two phases:

- *Run generation:* It reads the input rows, sorts them in the tournament tree, and generates runs. (A run is a list of sorted tuples.) The number of runs generated depends on the input data and on the sort memory size. For large sort operations, the sorted runs (intermediate results) may be written to scratch files created dynamically in temporary spaces you define and purged after sorting is completed. If all input tuples fit in sort memory, then the merge phase is completely avoided, and no scratch files are used.
- *Merge phase:* In the second phase, the sorted runs are merged. The merge width depends on the size of the sort memory. If the number of generated runs is less than the merge width, only one merge is done. Otherwise, several merge passes may be necessary.

Creating Temporary Spaces

Use the CREATE TEMPSPACE statement to create a directory that will contain temporary space for scratch files. ALLBASE/SQL needs scratch files to store the intermediate results of sorting and creating indexes. You should create at least one temporary space in the DBEnvironment you are working in. If you do not create a temporary space, ALLBASE/SQL opens scratch files in the /tmp directory.

The CREATE TEMPSPACE statement itself does not use any disk space, and ALLBASE/SQL does not create any scratch files when you issue the CREATE TEMPSPACE statement. Instead, they are dynamically created and purged during sort operations. When you define a temporary space, the amount of available disk space may change dynamically as ALLBASE/SQL creates and purges scratch files.

By default, the number of pages used for each temporary space file is 256. The total amount of space used is all that is available in the directory /tmp.

Tips for Using Temporary Spaces

- If you do large sorts or create large indexes, use CREATE TEMPSPACE to define temporary spaces. Be sure to specify a *MaxTempFileSize* large enough for your needs.
- Create one temporary space for each disk partition that is available for scratch files.
- ALLBASE/SQL opens a new scratch file after reaching the limit specified in the CREATE TEMPSPACE statement. If the sort is large and the default is small, ALLBASE/SQL may have to open many small scratch files. To avoid this, specify a large *MaxTempFileSize* value. For small sorts, ALLBASE/SQL does not create large scratch files even if *MaxTempFileSize* is large; hence, it is safe to specify *MaxTempFileSize* generously.
- If you define more than one temporary space in the DBEnvironment, ALLBASE/SQL opens scratch files by rotation in each temporary space.
- When doing large sorts, choose a *MaxTempFileSize* value that is large enough to prevent too many scratch files from being created. The number of files a process can open, including scratch files and other files, is limited by a value set in the kernel file. You can configure the limit, and its default value is 60. If ALLBASE/SQL reaches the limit, it returns DBERR 3061 and an HP-UX file system error. If you see DBERR 3061, drop your temporary spaces and recreate them with larger *MaxTempFileSize* values.

Disk Space for Sorting

All intermediate sort results are written to scratch files, not to any DBEFileSet in the DBEnvironment. Sorting may require 100% to 200% of the input data size in temporary disk files for intermediate results. If a join and an ORDER BY are both involved, the amount could be as much as 300%. However, scratch files are purged as soon as the sort operation is finished.

Final sort output for queries with GROUP BY, ORDER BY or DISTINCT is written to the SYSTEM DBEFileSet, requiring space worth 100% of the sorted output. For CREATE INDEX, the final output is directly written to the DBEFileSet containing the index, requiring space worth the size of the index.

The disk space required for sort operations is summarized below:

- CREATE INDEX: 100-200% of the size of the input key data in temporary files and 100% in the INDEX DBEFileSet.
- DML statements using a sort: 100-300% of the size of the input data in temporary files and 100% in the SYSTEM DBEFileSet.

Controlling the Use of Temporary Space

The order in which you carry out database operations may determine the amount of temporary space required. For example, if you create a large table, load it, then create an index, you will probably use scratch file space for sorting. However, if you create the table, then create the index, then load the data, you will not use scratch files. In the second case, the load is 3 to 10 times slower because ALLBASE/SQL creates index entries during loading. However, if you load unsorted data, the size of the index is about 50% larger.

Memory Utilization in Sorting

Sorting is memory-intensive, and the amount of memory required is estimated from the number of input tuples. The number of input tuples is in turn estimated from table statistics. Therefore, it is important to have up-to-date statistics when you execute `ORDER BY`, `GROUP BY`, or `CREATE INDEX` statements, especially on large tables.

Performance Hints for Large Sorts

- Issue the `UPDATE STATISTICS` statement after loading a large table and before issuing the `CREATE INDEX` statement. After you issue `CREATE INDEX`, issue another `UPDATE STATISTICS` statement to get the correct statistics for the index.
- Make sure that you have allocated enough temporary space to hold the scratch files.

Join Methods

ALLBASE/SQL uses two join methods: nested loop join and sort/merge join. Nested loop joins are usually much faster than sort/merge joins. By default, the optimizer chooses a join method depending on the query and the statistics in the system catalog for the tables involved. To override the join method the optimizer chooses, you can use the `SETOPT` statement.

The nested loop method scans the second table once for each qualifying row in the first table. If the scan on the second table is a serial scan (rather than an index scan), the nested loop join can require considerable I/O and CPU time when the table is larger than the buffer cache. However, a nested loop join may be the best approach for joins involving small tables.

A sort/merge join has two phases: the sort phase and the merge/join phase.

- *The sort phase:* If the scan on the table is an index scan on the joined column (that is, there is a join on `col1` and the index being picked is also on `col1` or on `col1, col2`), then the sort can be skipped, since the returned tuples from the scan are already sorted.
- *The merge/join phase:* Multiple tables are merged or joined without scanning a table repeatedly, as in a nested loop join. This saves a significant amount of I/O and CPU time, especially for large tables.

A sort/merge join can help with tables that have indexes that are larger than the buffer cache size.

Temporary Space in the SYSTEM DBEFileSet

Query results from queries using `ORDER BY` and `GROUP BY` clauses are stored in temporary pages in the SYSTEM DBEFileSet. Be sure to include enough TABLE or MIXED DBEFile space in SYSTEM to accommodate your largest sorted query. As your system catalog grows, you should monitor the DBEFile space in SYSTEM occasionally (after doing an `UPDATE STATISTICS` on at least one system view) to make sure enough temporary space is still available. As needed, add DBEFiles to SYSTEM to supply temporary pages.

Section Caching and Directory Caching

Section caching holds a number of sections in memory between transactions so that they do not need to be read again for the next execution. ALLBASE/SQL can keep up to 12 sections in memory. Section caching is more efficient when DDL Enabled is set to `NO`. **Directory caching** stores in memory the DBCore directory, which contains the locations of tables and

indexes. Directory caching is turned on when you set the DDL Enabled flag to NO by using ALTDBE in SQLUtil.

Section caching is subject to the following rules:

- When there are 12 or fewer sections in memory, the system does not try to delete any permanent sections at the end of a transaction.
- When there are more than 12 sections in memory, the system deletes only those sections that are not opened. Sections are considered opened if they are cursor sections and have been opened by the OPEN *CursorName* statement.
- The system does not delete dynamic sections, even if the cache limit is exceeded.
- The system can keep more than 12 (or 4) sections in memory, if all of them are opened cursors. The only limit is the amount of user heap space available.

This feature is helpful for an application that executes the same sections repeatedly, especially if it has a small number of sections.

Setting Limits for Section Caching

By default, ALLBASE/SQL allocates shared memory for up to 12 sections. You can increase this number by setting the environment variable *HPSQLsectcache* to any value from 4 to 128.

Using Multiconnect Functionality

It is possible to establish a maximum of 32 simultaneous database connections. When your application must access more than one DBEnvironment, there is no need to release one before connecting to another. Performance is greatly improved by using this method rather than connecting to and releasing each DBEnvironment individually.

For more detailed information, refer to the chapter “Transaction Management with Multiple DBEnvironment Connections” in the *ALLBASE/SQL Advanced Application Programming Guide* and the section “Using Multiple Connections and Transactions with Timeouts” in the *ALLBASE/SQL Reference Manual* chapter “Using ALLBASE/SQL.”

Using Timeouts to Tune Performance

When an application requests a database resource that is unavailable, the application is placed on a wait queue. If the application waits longer than the timeout value specified for its DBEnvironment connection, an error occurs, and the transaction is rolled back. Your strategy for specifying timeout values and handling timeout errors depends on the specific needs of your application and on your business procedures. By default, the timeout value is infinite.

For example, a transaction may require resources that are locked by a second transaction that requires a great deal of execution time. In this case, you could specify a reasonable amount of time for the first transaction to wait before a timeout occurs. Such a strategy might be essential in a distributed environment. For more detailed information, refer to the chapter “Transaction Management with Multiple DBEnvironment Connections” in the *ALLBASE/SQL Advanced Application Programming Guide* and the section “Setting Timeout Values” in the *ALLBASE/SQL Reference Manual* chapter “Using ALLBASE/SQL.”

Network Guidelines

If you are using ALLBASE/SQL in a network, you should structure your applications with remote access in mind.

- Remote database access is faster when you send BULK requests to the remote DBEnvironment. For example, if you use the simple FETCH statement on a cursor opened on a remote database, the network must process each successive FETCH as a separate request for data. But if you use BULK FETCH, you make only one request, for a larger number of rows. In your application, use a buffer size that is close to the size of the actual data retrieved.
- Use stored procedures to reduce network traffic between applications and the DBEnvironment. Refer to the section “Using Rules and Procedures” in the chapter “Guidelines for Transaction Design.”

HP-UX System Guidelines

This section presents some pointers about parameters in the HP-UX operating system that can affect the performance of ALLBASE/SQL.

- Use an HP-UX buffer cache that is at least as large as the default cache size (10% of physical memory). Use more than the default for large tables or indexed joins.
- Put swap space on a spindle different from DBEnvironment files and log files, especially for large multiuser operations. Increase the size of swap space if you are running more than 32 users.
- Use a file blocking factor of at least 4K. Serial scans may benefit from a blocking factor of 8K or larger.
- Use a larger interval for the sleep time for *hpdbdaemon* than the default of 30 seconds. The daemon is a process that wakes up every 30 seconds and connects to the DBEnvironment to determine whether there have been any abnormal terminations. If there have been abnormal terminations, the daemon cleans up by releasing locks and other system resources. If your system rarely experiences abnormal terminations, you can avoid the overhead of having daemons repeatedly connect to and release the DBEnvironment by increasing the sleep time. You should experiment with different intervals; a value of 60 seconds reduces the overhead by half. Remember, however, that increasing the sleep time also increases the time before cleanup takes place.

You can set the daemon’s sleep time with the C shell *setenv* command, as follows:

```
setenv hpdbdmons1 n
```

where the integer *n* is a number of seconds.

- Use the formula described under “Choosing the Number of Data Buffer Pages” to help determine the maximum number of data buffer pages to use on series 800 systems, where the maximum is limited by the size of swap space.
- You can use the HP-UX *nice* command to adjust the priority of executing processes. Refer to the manual page for *nice*.

Using HP-UX Raw Files for DBEFiles and Logs

Using raw files in HP-UX bypasses the operating system's normal buffering process. Considerable performance benefits can be derived from using raw files for logs and for DBEFiles when the primary data access method is random. However, there may be a performance penalty for sequential access to DBEFiles, since the file system's prefetching is bypassed.

In HP-UX, you can create raw DBEFiles and log files by specifying the pathname of a raw device in the CREATE DBEFILE statement or in the LOG DBEFILE clause of the START DBE NEW and START DBE NEWLOG statements.

For more information about raw files, refer to the appendix "Using HP-UX Raw Files for DBEFiles and Logs" in the *ALLBASE/SQL Database Administration Guide*.

Getting Started With SQLMON

This chapter describes the basic operations of SQLMON. You will learn how to start the program, exit the program, display screens, modify SQLMON variables, and access online help. The last section in this chapter, “Monitoring Tasks,” explains which screens and fields to check as you use SQLMON.

Introduction

SQLMON is an online diagnostic tool that monitors the activity of a DBEnvironment. SQLMON screens provide information on file capacity, locking, I/O, logging, tables, and indexes. They summarize activity for the entire DBEnvironment, or focus on individual sessions, programs, or database components. SQLMON is a read-only utility, and cannot modify any aspect of the DBEnvironment.

SQLMON can serve as

- a tool for learning about your DBEnvironment
- a development tool for application programmers
- a monitoring tool for tuning your DBEnvironment
- a troubleshooting tool for detecting performance problems

Starting SQLMON

To run SQLMON, log on as superuser or as DBEcreator, and issue the following command:

```
% sqlmon
```

After displaying a banner and several menus, SQLMON displays a prompt that shows that you are in the Overview subsystem:

```
SQLMONITOR OVERVIEW =>
```

SQLMON has six subsystems, named Overview, IO, Load, Lock, SampleIO, and Static. The SQLMON prompts identify the current subsystem. All of the prompts listed below are subsystem prompts:

```
SQLMONITOR OVERVIEW =>
```

```
SQLMONITOR IO =>
```

```
SQLMONITOR LOAD =>
```

```
SQLMONITOR LOCK =>
```

```
SQLMONITOR SAMPLEIO =>
```

```
SQLMONITOR STATIC =>
```

From a subsystem prompt, you can access SQLMON screens, issue SQLMON commands, or exit the program.

Leaving SQLMON

To exit SQLMON, enter either the EXIT or QUIT command at any subsystem prompt:

```
SQLMONITOR OVERVIEW => EXIT
```

```
SQLMONITOR OVERVIEW => QUIT
```

If you are actively displaying a screen, you must press Return to leave the screen before you enter EXIT or QUIT. For more information, refer to the section “Leaving SQLMON Screens.”

If you are using the help facility, you must enter //, EXIT, or QUIT to leave the help facility and then EXIT or QUIT again to leave SQLMON. For more information, refer to the section “Accessing Online Help” later in this chapter.

Specifying the DBEnvironment

Before you invoke an SQLMON screen, you must specify the DBEnvironment to be monitored. For example, to specify the PartsDBE DBEnvironment, you would enter:

```
SQLMONITOR OVERVIEW => SET DBENV PartsDBE
```

Invoking SQLMON Screens

To access an SQLMON screen, type the name of the screen at a subsystem prompt. For example, to invoke the Overview screen, issue the following command:

```
SQLMONITOR OVERVIEW => OVERVIEW
```

The commands are not case sensitive, and you can abbreviate them. For example, you can also access the Overview screen with this command:

```
SQLMONITOR OVERVIEW => o
```

The abbreviated commands are context sensitive and vary according to the current subsystem. For example, to access the Lock Memory screen from the Lock subsystem, you can use either of the following commands:

```
SQLMONITOR LOCK => m
```

```
SQLMONITOR LOCK => /loc m
```

But to access the Lock Memory screen from another subsystem, you must use the longer abbreviation:

```
SQLMONITOR IO => /loc m
```

Table 6-1 lists the SQLMON screens by subsystem.

Table 6-1. SQLMON Screens

Subsystem	Screen Name	Description
IO	IO	Performance information on the data and log buffer pools.
	IO Data Program	Data buffer pool information for each program being run by sessions attached to the DBEnvironment.
	IO Data Session	Data buffer pool information for each session attached to the DBEnvironment.
	IO Log Program	Log buffer pool information for each program being run by sessions attached to the DBEnvironment.
	IO Log Session	Log buffer pool information for each session attached to the DBEnvironment.
Load	Load	Information useful in measuring the throughput efficiency of the DBEnvironment.
	Load Program	Transaction throughput information for each program being run by sessions attached to the DBEnvironment.
	Load Session	Transaction throughput information for each session attached to the DBEnvironment.
Lock	Lock	Lock activity data for the entire DBEnvironment.
	Lock Impede	Locks granted to a particular session that are causing other sessions to wait.
	Lock Memory	Number of locks allocated at each granularity to each session.
	Lock Object	Sessions in the lock queue for particular locks.
	Lock Session	Lock activity data for a single session.
	Lock TabSummary	Number of locks allocated at each granularity level for each table.

Table 6-1. SQLMON Screens (continued)

Subsystem	Screen Name	Description
Overview	Overview	Important aspects of the DBEnvironment's performance, such as the data buffer pool miss rate and the amount of available runtime control block space.
	Overview Program	Session information for each program.
	Overview Session	Information about all sessions connected to the DBEnvironment.
SampleIO	SampleIO	DBEFile I/O information.
	SampleIO Objects	I/O information for database objects in the data buffer pool.
	SampleIO Tables	I/O information for tables in a DBEFileSet.
	SampleIO Indexes	Index and referential constraint
	SampleIO Tabindex	I/O information about a single table, and its indexes and referential constraints.
Static	Static	Information about indexes, referential constraints, and hashing for each table contained in a DBEFileSet.
	Static Cluster	Information about the clustering of indexes and referential constraints in a DBEFileSet.
	Static DBEFile	File capacity and fullness data for each DBEFile in a DBEFileSet.
	Static Hash	Overflow chain information for hashed tables.
	Static Indirect	Information about the indirect rows in each table of a DBEFileSet.
	Static Size	Information about the size of database objects in a DBEFileSet.

Table 6-2 summarizes the abbreviated commands used to invoke SQLMON screens. You can also issue the commands in the “Different Subsystem” column from the same subsystem.

Table 6-2. Abbreviated Screen Commands

Screen Name	Abbreviated Command	
	Different Subsystem	Same Subsystem
IO	/i	i
IO Data Program	/i d p	d p
IO Data Session	/i d s	d s
IO Log Program	/i l p	l p
IO Log Session	/i l s	l s
Load	/loa	l
Load Program	/loa p	p
Load Session	/loa s	s
Lock	/loc	l
Lock Impede	/loc i	i
Lock Memory	/loc m	m
Lock Object	/loc o	o
Lock Session	/loc s	s
Lock TabSummary	/loc t	t
Overview	/o	o
Overview Program	/o p	p
Overview Session	/o s	s
SampleIO	/sa	s
SampleIO Objects	/sa o	o
SampleIO Tables	/sa tabl	tabl
SampleIO Indexes	/sa i	i
SampleIO Tabindex	/sa tabi	tabi
Static	/st	st
Static Cluster	/st c	c
Static DBEFile	/st d	d
Static Hash	/st h	h
Static Indirect	/st i	i
Static Size	/st si	si

Leaving an SQLMON Screen

To exit an SQLMON screen, press Return. If the subsystem prompt does not appear immediately, the cycle option is probably set to a value other than OFF. Therefore, the screen is displayed for a certain number of refresh cycles, instead of disappearing immediately.

To reset the cycle option to OFF, enter

```
SQLMONITOR OVERVIEW => SET CYCLE OFF
```

Once you return to the subsystem prompt, you can access another screen, issue an SQLMON command, or exit the program.

Navigating SQLMON Subsystems

SQLMON has six subsystems, which are listed below:

- **Overview**, which shows general performance information
- **IO**, which provides information on data and log buffer activity
- **Load**, which displays transaction throughput data
- **Lock**, which shows locking activity
- **SampleIO**, which provides information on DBEFile, table, and index I/O
- **Static**, which describes indexes, tables, DBEFiles, and constraints

The name of a screen identifies the subsystem to which it belongs. Each subsystem has a primary screen, with the same name as the subsystem, and secondary screens, whose names begin with the subsystem name.

When you access a screen, you automatically move to the subsystem that screen belongs to.

Figure 6-1 shows how the SQLMON screens and subsystems are organized.

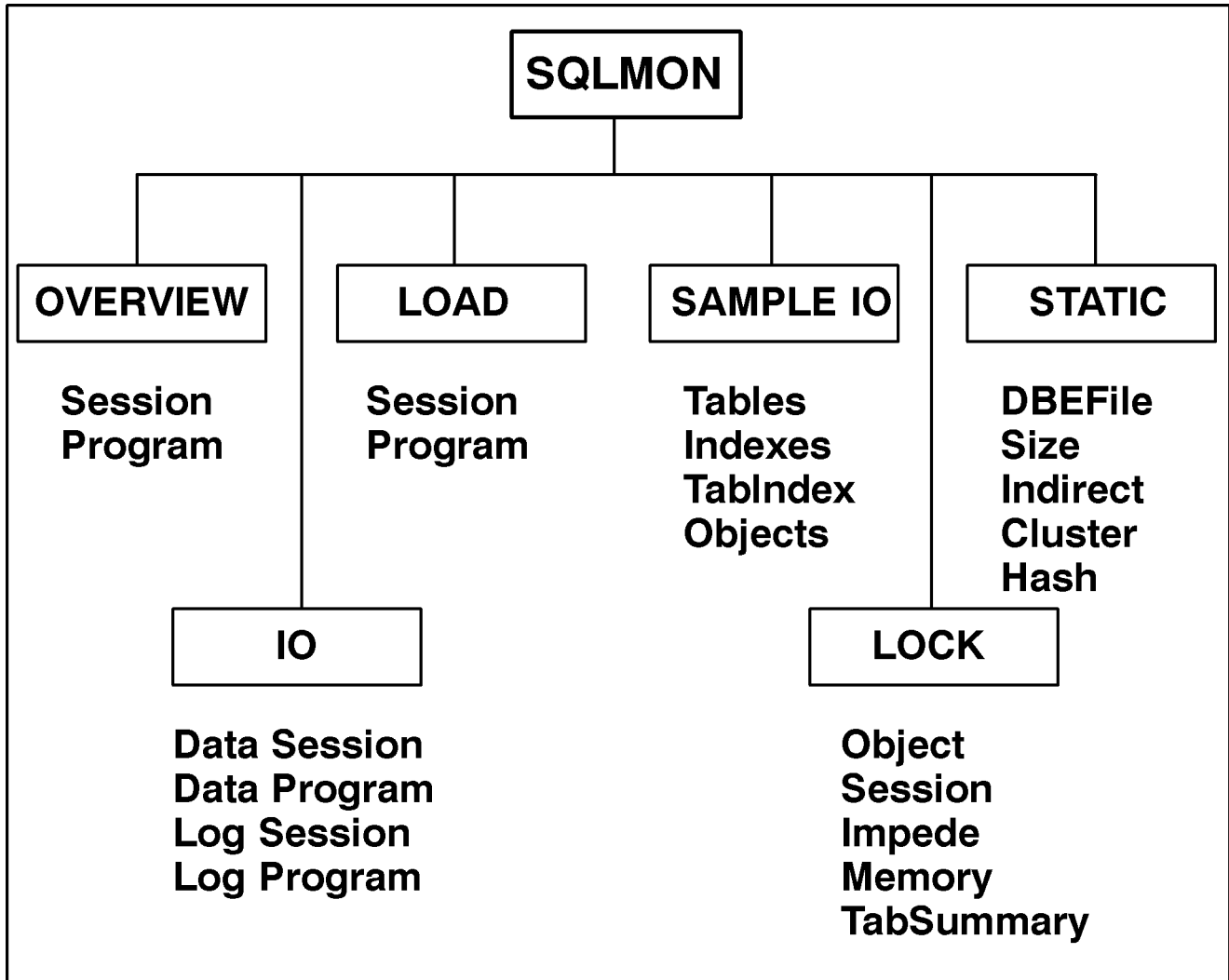


Figure 6-1. SQLMON Road Map

Setting SQLMON Variables

You can modify SQLMON environment variables using the SET commands. For example, the following command sorts rows on a screen according to the value in the screen's second column:

```
SQLMONITOR IO => SET SORTIODATA 2
```

If you omit the last parameter (the number after SORTIODATA), SQLMON issues a prompt:

```
SQLMONITOR IO => SET SORTIODATA
```

```
<0=OFF,1=BUFF ACCESS,2=DATA DISK RD,3=DATA DISK WR,4=MISS RATE>: 2
```

You can execute a SET command only from a subsystem prompt. If you issue a SET command at the help prompt, SQLMON displays help text for the command but does not execute the command.

To view the current variable settings, issue SET with no parameters:

```
SQLMONITOR OVERVIEW => SET
```

```
C[CYCLE]                OFF
DBEF[ILESET]            OFF
DBEN[VIRONMENT]        PartsDBE
DBEC[ONNECT]           ON
DBEI[NITPROG]          ON
E[CHO]                 OFF
LOCKF[ILTER]           SU/TPR/GWC/SXRsxr6v/1
LOCKO[BJECT]           ALL
LOCKT[ABFILTER]        OFF
M[ENU]                 ON
OUTP[UT]               OFF
R[EFRESH]              10
SA[MPLING]             ON
DI[SPLAYSAMPLES]       OFF
SORTIOD[ATA]           3
SORTIOL[OG]            OFF
SORTLOA[D]             3
SORTLOC[K]             5
SORTS[AMPLEIO]         3
T[OP]                  OFF
U[SERTIMEOUT]          5
```

Each variable displayed in the previous example has a corresponding SET command. For more information on the SET commands, see the chapter "SQLMON Command Reference."

Accessing Online Help

SQLMON provides extensive online help, with descriptions of screens, fields, subsystems, and commands, as well as performance tuning hints. The help facility is context sensitive; for example, if you request tuning information from within the Lock subsystem, you see only tuning hints related to locking issues.

Invoking the Help Facility

To invoke the SQLMON help facility, issue the following command from a subsystem prompt:

```
SQLMONITOR OVERVIEW => help
```

By default, SQLMON displays help information for the screen you last accessed. To override the default, you can specify the screen name on the command line. For example, to get help information on the IO screen from the Overview subsystem, enter

```
SQLMONITOR OVERVIEW => help io
```

Leaving the Help Facility

To leave the help facility, enter any of the following commands to return to the subsystem prompt:

```
SQLMONITOR HELP OVERVIEW => //
```

```
SQLMONITOR HELP OVERVIEW => EXIT
```

```
SQLMONITOR HELP OVERVIEW => QUIT
```

Once you return to the subsystem prompt, you can access SQLMON screens, issue SQLMON commands, or exit the program.

Issuing Help Commands

When you enter the help facility, it displays the help prompt for the current screen:

```
SQLMONITOR HELP OVERVIEW =>
```

Let's assume that you have just visited the Lock Memory screen. The easiest way to get help on the screen is to enter the command HELP from the prompt:

```
SQLMONITOR [ LOCK ] => help
```

You would then see information about using the Lock Memory screen, and then return to the help prompt, which looks like this:

```
SQLMONITOR [ HELP LOCK MEMORY ] =>
```

Notice that the name of the screen is displayed in the help prompt, not just the name of the Lock subsystem.

To see a description of each field that appears on the Lock Memory screen, enter

```
SQLMONITOR [ HELP LOCK MEMORY ] => help
```

You can then type additional commands to get more information:

6-10 Getting Started With SQLMON

```
SQLMONITOR [ HELP LOCK MEMORY ] => control
```

```
SQLMONITOR [ HELP LOCK MEMORY ] => tune
```

When you want to leave the help facility, type //, EXIT, or QUIT:

```
SQLMONITOR [ HELP LOCK MEMORY ] => //
```

```
SQLMONITOR [ LOCK ] =>
```

You can also add options to the HELP command when you enter the help facility. For example, after you visit the Static DBEFile screen, you can enter the following command to display a description of each field on the screen:

```
SQLMONITOR [ STATIC ] => help info
```

Then, to leave the help facility and return to the screens, enter

```
SQLMONITOR [ HELP STATIC DBEFILE ] => //
```

```
SQLMONITOR [ STATIC ] =>
```

Table 6-3 summarizes the parameters you can add to the HELP command.

Table 6-3. SQLMON Help Commands

Help Command	Description
<i>ScreenName</i>	Generates help text for the screen you specify.
CONTROL	Describes the SET commands that affect the current screen.
INFO	Describes each field belonging to the current screen.
SUBSYSTEM	Describes each of the screens in the subsystem.
TUNE	Generates a list of topics on database performance tuning.
TUNE <i>TuneNumber</i>	Provides detailed information on the tuning topic designated by <i>TuneNumber</i> .
MAIN	Displays general information on SQLMON.
<i>SetCommand</i>	Describes the <i>SetCommand</i> you specify.

The HELP command is documented fully in the chapter “SQLMON Command Reference.”

Creating Batch Reports

To generate a batch report containing SQLMON screen displays, create a file of SQLMON commands that will be used as input to SQLMON. The command file must include a SET CYCLE command to stop the display of the screens after some number of refresh intervals, and a SET OUTPUT command to specify the name of the file to which output will be sent.

For example, you can copy the Static screen to an output file named Report1. To do so, use a text editor to create a file named BatchIn, and add the following SQLMON commands to it:

```
SET ECHO 0N
SET DBENVIRONMENT PartsDBE
SET CYCLE 1
SET OUTPUT Report1
STATIC
EXIT
```

Before you run SQLMON, remove any earlier versions of Report1:

```
% rm Report1
```

Run SQLMON, using BatchIn as input and redirecting output to /dev/null to prevent the screens from appearing on standard output:

```
% sqlmon < BatchIn > /dev/null
```

When SQLMON finishes executing, you can print Report1 to create a hard copy of the Static screen:

```
% lp Report1
```

Overhead Generated by SQLMON

In most cases, SQLMON has little impact on system performance. However, be careful when you use the SampleIO subsystem, because it can generate enough overhead to impact the performance of other DBEnvironment sessions.

Be careful also when you use the Static subsystem. The Static subsystem differs from the other SQLMON subsystems, because it establishes a connection with the DBEnvironment. When you leave the Static subsystem for another subsystem, the connection closes.

SQLMON obtains information on the primary screen in the subsystem, the Static screen, when you issue a SET DBENVIRONMENT command. System performance should not be affected when you access the Static screen.

However, when you access other screens in the Static subsystem, SQLMON issues commands that are similar to the UPDATE STATISTICS statement. These commands perform serial scans on each DBEFileSet, which can be very time consuming (however, the commands do not update statistics in the system catalog or invalidate stored sections). SQLMON does not acquire locks on user tables during this processing.

To improve performance, you can issue the command

```
SET DBEFILESET !!DBEFileSetName!!
```


which improves performance, because

- SQLMON obtains information only about the objects in *DBEFileSetName*, thereby reducing the number of serial scans that must be performed
- Serial scans are performed only once on *DBEFileSetName*. Serial scans occur the first time you access a screen other than the primary screen in the subsystem, not when you issue the SET DBEFILESET command. When you access another screen in the subsystem, you view information obtained during the original scans.

If the DBEFILESET variable is OFF, SQLMON performs scans on each DBEFileSet each time that you access a screen in the Static subsystem.

Monitoring Tasks

This section summarizes the monitoring tasks you can perform with SQLMON, giving the appropriate SQLMON screens and fields for each task. The tasks fall into the following categories:

- disk usage
- memory usage
- tables
- hash structures
- indexes and referential constraints
- transactions
- sessions
- I/O
- logging
- locking

For example, if you are interested in checking the size of a DBEFile, look at the tasks listed in Table 6-4. To perform the task, go to the screen listed in the Screens column and the fields listed in the Fields column. If you need more information on a screen's fields, see the corresponding section in the "SQLMON Screen Reference" chapter.

Table 6-4. Monitoring Disk Usage

Task	Screens	Fields
Determining log file capacity	Overview	LOG FULL % Used LgPgs Max LgPgs
Determining DBEFile capacity in a DBEFileSet	Static DBEFile	DBEFILE DBEFILE FULLNESS % USED PAGES MAX PAGES
Identifying DBEFile storage restrictions	Static DBEFile	DBEFILE TYP BD
Determining DBEFileSet capacity	Static DBEFile	DBEFILESET DBEFILESET FULLNESS % FSUSED PAGES FSMAX PAGES
Determining the size of database objects in a DBEFileSet	Static Size	DBEFILESET OWNER.TABLE TABLE PAGES INDEX PAGES TOTAL PAGES
Identifying detached DBEFiles	Static DBEFile	DBEFILESET FULLNESS DBEFILE FULLNESS

Table 6-5. Monitoring Memory Usage

Task	Screens	Fields
Identifying the size of the data buffer pool	IO	TOTAL DATA BUFFER PAGES
Identifying the size of the log buffer pool	IO	TOTAL LOG BUFFER PAGES
Determining the size of the runtime control block	Overview	RUNTIME CB % Used Pages Max Pages
Resolving memory problems	Lock Memory Lock TabSummary	all fields (hint: type HELP LOCK MEMORY TUNE 4 for more information)

Table 6-6. Monitoring Tables

Task	Screens	Fields
Identifying tables in a DBEFileSet	Static	DBEFILESET OWNER.TABLE
Identifying tables stored in TurboIMAGE data sets	Static	OWNER.TABLE IMAGE
Determining a table type	Static	OWNER.TABLE TYPE
Determining the number of indexes and referential constraints on a table	Static	OWNER.TABLE NUM INDEXES
Determining the size of a table	Static Size Static Cluster	OWNER.TABLE TABLE PAGES
Determining the number of rows in a table	Static Indirect Static Cluster	OWNER.TABLE TOTAL ROWS
Determining the percentage of indirect rows in a table	Static Indirect	OWNER.TABLE TABLE INDIRECT ROW %
Comparing the number of locks by table	Lock TabSummary	OWNER.TABLE G TOTAL LOCKS
Identifying the locks on a table	Lock	OWNER.TABLE G PAGE/ROW ID LOCK QUEUE (hint: use SET LOCKTABFILTER)
Monitoring table I/O	SampleIO Tables SampleIO TabIndex	OWNER.TABLE SWAPIN SWAPOUT TOTALIO
Identifying the tables currently residing in the data buffer pool	SampleIO Objects	OWNER.TABLE CURRENT PGS

Table 6-7. Monitoring Hash Structures

Task	Screens	Fields
Identifying hashed tables in a DBEFileSet	Static	DBEFILESET OWNER.TABLE HASH
Identifying hashed tables in a DBEFileSet	Static Hash	DBEFILESET OWNER.TABLE (hint: only hashed tables are displayed)
Monitoring primary pages	Static Hash	PRIMPAGES PRIMDATA PRIMOVERF
Monitoring overflow pages	Static Hash	OVERPAGES OVERFLOW CHAIN LNGTH MAXOVERFLOW AVGOVERFLOW
Identifying DBEFiles that are bound to hashed tables	Static DBEFile	DBEFILE BD

Table 6-8. Monitoring Indexes and Referential Constraints

Task	Screens	Fields
Identifying the indexes and referential constraints on a table or in a DBEFileSet	Static Size Static Cluster	DBEFILESET OWNER.TABLE
Determining the number of indexes and referential constraints on a table or in a DBEFileSet	Static	DBEFILESET OWNER.TABLE NUM INDEXES
Determining the size of an index or a referential constraint	Static Size	OWNER.TABLE INDEX PAGES
Identifying locks on a referential constraint	Lock	OWNER.TABLE/CONSTRAINT G PAGE/ROW ID LOCK QUEUE (use SET LOCKTABFILTER)
Monitoring index and referential constraint I/O	SampleIO Indexes SampleIO TabIndex	OWNER.TABLE[/INDEX,CONSTRAINT] SWAPIN SWAPOUT TOTALIO
Identifying the indexes and referential constraints currently residing in the data buffer pool	SampleIO Objects	OWNER.TABLE[/INDEX,CONSTRAINT] CURRENT PGS
Determining the efficiency of index scan over an index or referential constraint	Static Cluster	CCOUNT UNLOAD/LOAD SUGGESTED %

Table 6-9. Monitoring Transactions

Task	Screens	Fields
Determining the transaction identifier	Overview Program Overview Session Lock Session Lock Object Lock Impede Lock TabSummary	XID
Determining the isolation level	Overview Program Overview Session Lock Session Lock Object Lock Impede Lock TabSummary	ISO
Determining the transaction priority	Overview Program Overview Session Lock Session Lock Object Lock Impede Lock TabSummary	PRI
Determining the transaction label	Overview Program Overview Session Lock Session Lock Object Lock Impede Lock TabSummary	LABEL
Determining the maximum number of active transactions	Overview Load	MAX XACT
Determining the current number of active transactions	Overview Load	ACTIVE XACT
Determining the number of active transactions waiting for a lock	Overview Load	IMPEDE XACT
Determining the number of transactions waiting for a transaction slot	Load	THROTTLE WT
Monitoring throughput	Load Load Session Load Program	BEGIN WORK COMMIT WORK ROLLBK WORK DEADLOCK

Table 6-10. Monitoring Sessions

Task	Screens	Fields
Determining the number of sessions	Overview Overview Session Overview Program IO IO Data Session IO Data Program IO Log Session IO Log Program Load Load Session Load Program	SESSIONS
Determining the process identifier of a session	Overview Session	PID LOGIN NAME PROGRAM NAME
Determining the transaction information of a session	Overview Session Lock Session Lock Impede Lock TabSummary	XID ISO PRI LABEL
Identifying the program being run by a session	Overview Program Lock Session Lock Impede Lock TabSummary	PID LOGIN NAME PROGRAM NAME
Identifying the sessions running a particular program	Overview Program	PID LOGIN NAME PROGRAM NAME
Identifying all waiting sessions	Overview Session Overview Program	PID LOGIN NAME STATUS
Monitoring session lock activity	Lock Session	All fields
Monitoring session data buffer I/O	IO Data Session IO Data Program	All fields
Monitoring session log buffer I/O	IO Log Session IO Log Program	All fields
Comparing session throughput	Load Load Session Load Program	BEGIN WORK COMMIT WORK ROLLBK WORK DEADLOCKS

Table 6-11. Monitoring I/O for Data

Task	Screens	Fields
Identifying the size of the data buffer pool	IO	TOTAL DATA BUFFER PAGES
Monitoring data buffer I/O	IO IO Data Session IO Data Program	BUFF ACCESS DATA DISK RD DATA DISK WR MISS RATE
Monitoring DBEFile I/O	SampleIO	DBEFILE SWAPIN SWAPOUT TOTALIO
Monitoring table I/O	SampleIO Tables SampleIO TabIndex	OWNER.TABLE SWAPIN SWAPOUT TOTALIO
Monitoring index and referential constraint I/O	SampleIO Indexes SampleIO TabIndex	INDEX,CONSTRAINT SWAPIN SWAPOUT TOTALIO
Determining the number of data buffer pages occupied by an object	SampleIO Object	CURRENT PGS
Determining the percentage of indirect rows in a table	Static Indirect	OWNER.TABLE TABLE INDIRECT ROW %
Determining the efficiency of an index scan over an index or referential constraint	Static Cluster	CCOUNT UNLOAD/LOAD SUGGESTED %

Table 6-12. Monitoring I/O for Logging

Task	Screens	Fields
Identifying the size of the log buffer pool	IO	TOTAL LOG BUFFER PAGES
Monitoring log buffer I/O	IO IO Log Session IO Log Program	LOG BUFF WR LOG DISK RD LOG DISK WR
Monitoring checkpoints	IO	CHECKPOINTS

Table 6-13. Additional Monitoring for Logging

Task	Screens	Fields
Determining log file capacity	Overview	LOG FULL % Used LgPgs Max LgPgs
Identifying log mode	Overview IO	Archive Mode
Detecting logging errors	Overview	LOG ERRORS

Table 6-14. Monitoring Locking

Task	Screens	Fields
Determining the size of a runtime control block	Overview	RUNTIME CB % Used Pages Max Pages
Monitoring DBEnvironment lock activity	Load	LOCK REQTS LOCK WAITS LOCK WAIT %
Comparing the number of locks by table	Lock TabSummary	OWNER.TABLE G TOTAL LOCKS
Comparing the number of locks by session	Lock Memory	TABLE PAGE ROW TOTAL MAXTOTAL
Identifying locks on a table or referential constraint	Lock	OWNER.TABLE[/CONSTRAINT] G PAGE/ROW ID LOCK QUEUE (hint: use SET LOCKTABFILTER)
Determining the number of sessions that are accessing a particular lock	Lock	LOCK QUEUE
Determining the number of transactions that are waiting for locks	Overview Load	IMPEDE XACT
Determining the isolation level of a transaction	Overview Program Overview Session Lock Impede Lock Object Lock TabSummary Lock Session	XID ISO PRI LABEL
Identifying locks for which sessions are waiting	Lock	all fields (hint: use SET LOCKFILTER)
Identifying sessions that have obtained a particular lock	Lock Object	GWC MOD PID
Identifying sessions that are waiting to obtain (or to convert) a particular lock	Lock Object	GWC MOD NEW PID
Identifying lock activity for a particular session	Lock Session	all fields
Identifying locks obtained by a particular session that are causing other sessions to wait	Lock Impede	all fields
Detecting deadlocks	Load Load Session Load Program	DEADLOCKS
Resolving deadlocks	Lock Lock Object Lock Impede	all fields (hint: type HELP LOCK TUNE 13 for more information)

Troubleshooting with SQLMON

This chapter provides examples on how to troubleshoot performance problems with each of SQLMON's subsystems:

- Overview
- IO
- Load
- Lock
- SampleIO
- Static

Overview Subsystem

You can use the Overview subsystem to determine the overall cause of a performance problem, and then go to another subsystem for detailed information. For example, with the Overview screen you can detect

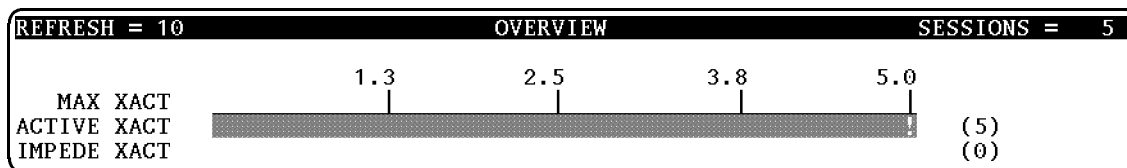
- transaction limit reached
- lock contention
- memory limit reached
- high data buffer miss rate
- log full condition

Transaction Limit Reached

If a session attempts to begin a transaction, but the number of active transactions is at the maximum, the session must wait until a transaction slot becomes available. How long the session waits depends on the timeout value in effect for the session. In the example that follows, the ISQL session times out because it cannot begin a new transaction:

```
isql=> SET USER TIMEOUT 3;
isql=> BEGIN WORK;
Timeout expired (3 seconds). (DBERR 2825)
```

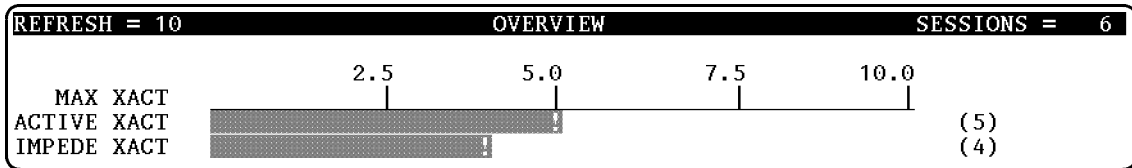
The following screen shows that the transaction limit, MAX XACT, and the number of active transactions, ACTIVE XACT, are both 5.



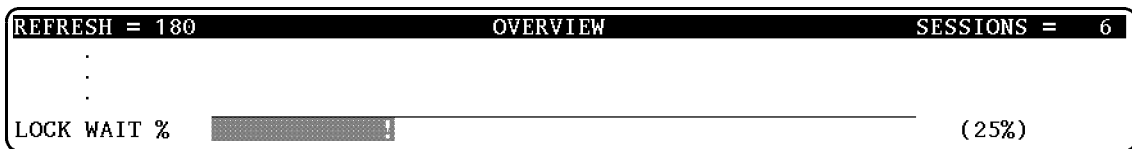
Therefore, you may need to raise the transaction limit. Use the Load subsystem for more information.

Lock Contention

The next screen shows that there are 5 active transactions (ACTIVE XACT) and 4 impeded transactions (IMPEDE XACT). This means that 4 out of 5 transactions are waiting to acquire a lock.



We also see that 25% of all lock requests are not granted immediately because other sessions hold incompatible locks.



This DBEnvironment definitely has a locking problem. You should use the Lock subsystem to get more information.

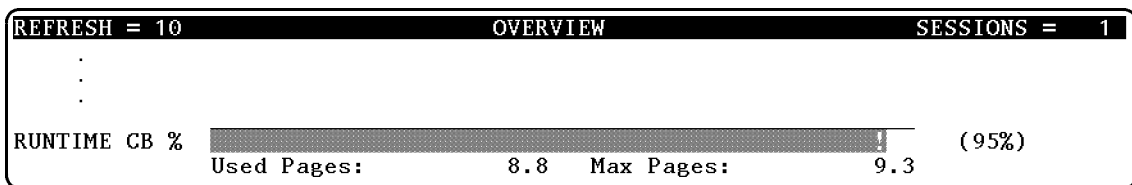
Memory Limit Reached

The runtime control block is an area of shared memory containing global runtime information for the DBEnvironment. All ALLBASE/SQL control blocks are allocated from the runtime control block, and the majority of control blocks are used for lock management. For more information, see the section “Shared Memory Considerations” in the chapter “Guidelines on Transaction Design.”

In the next screen, the runtime control block is 95% full.

You can either

- increase its size using the SQLUtil ALTDBE command. However, you must stop the DBEnvironment before you can issue this command.
- use the Lock Memory screen to identify the sessions that have the greatest number of locks.



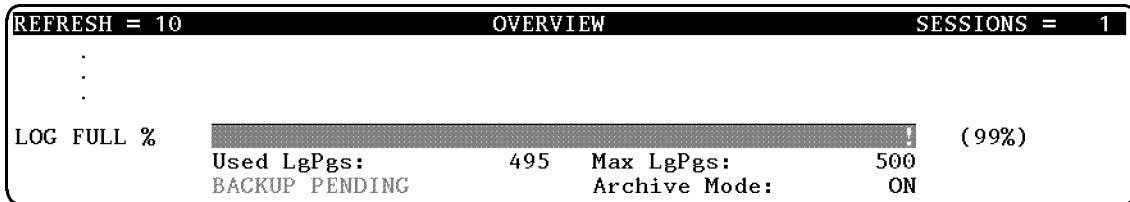
After you have identified the sessions that have the most locks, use the Lock TabSummary screen to identify the programs each session is running and the tables that have the greatest number of locks. You may wish to change some PUBLICROW tables to PUBLIC to reduce the memory overhead associated with them.

High Data Buffer Miss Rate

If the value of the DATA BUFFER MISS RATE field on the Overview screen is high, DBEnvironment performance degrades due to increased I/O. You can use the IO subsystem to identify the sessions and programs that are contributing to the data buffer miss rate.

Log Full Condition

You can monitor log file capacity by checking the LOG FULL field on the Overview screen. In the example that follows, the archive log is 99% full.



If you enter the following UPDATE statement to update the PurchDB.SupplyPrice table, it fails because the log is so full:

```
isql=> UPDATE PurchDB.SupplyPrice SET UnitPrice = UnitPrice * 1.2;
Log full. (DBERR 14046)
INSERT/UPDATE/DELETE statement had no effect due to execution errors.
(DBERR 2292)
Number of rows processed is 0
```

You can avoid this problem by using the Overview screen to monitor log file capacity and by adding log files before the logs are full. See the chapter “Backup and Recovery” in the *ALLBASE/SQL Database Administration Guide* for instructions.

IO Subsystem

Slow DBEnvironment performance is often caused by I/O activity. Use the IO subsystem to determine if the DBEnvironment has insufficient data buffer space or insufficient log buffer space.

Note You should use the Static subsystem to remove indirect rows, eliminate overflow chains from hashed tables, and recluster appropriate indexes before you use the IO subsystem to tune I/O.

Most of the information displayed on the IO screens is for logical I/O, not physical I/O. Logical I/O means that ALLBASE/SQL requests that the operating system read or write a page.

If ALLBASE/SQL requests to read a page, and if the page is in the operating system's buffer pool, no physical I/O occurs. If ALLBASE/SQL requests to write to a page, the operating system may record the write in its buffer pool to avoid physical I/O, unless ALLBASE/SQL forces it to write the page to disk. ALLBASE/SQL forces the operating system to write the page to disk when log disk writes occur. This ensures the integrity of the database in case of a system crash.

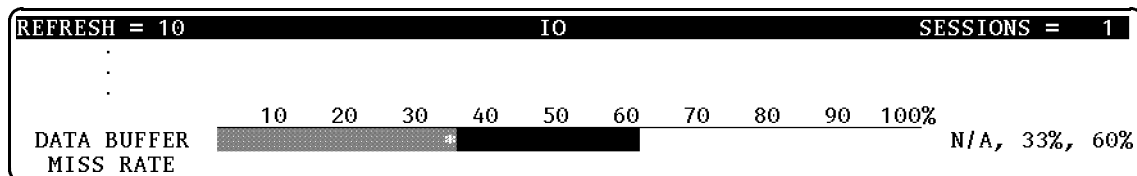
To minimize log disk writes, you can use the IO screens to tune the size of the ALLBASE/SQL buffer pools. Specifically, you would use the IO screen, the IO Log Program screen, or the IO Log Session screen and then adjust the the LOG DISK WR field, as described in the chapter "SQLMON Screen Reference."

To further improve I/O performance, you should also use performance tuning tools on the operating system.

Insufficient Data Buffer Space

A high data buffer miss rate leads to increased I/O and slower performance. If ALLBASE/SQL requests a page that is not in the data buffer pool, then the operating system must fetch the page, either from the operating system buffer pool or by a physical disk read.

If the data buffer miss rate of your DBEnvironment seems high, try increasing the number of data buffer pages. For more information, see "Choosing a Number of Data Buffer Pages" in the chapter "Guidelines on System Administration." On the IO screen that follows, the average data buffer miss rate is 33% and the maximum is 60%.



The BUFF ACCESS, DATA DISK RD, and DATA DISK WR fields of the IO screen provide more information on data buffer I/O activity. The following screen shows that on average BUFF ACCESS is 3, which means that 3 page requests are made to the data buffer pool every 10.0 seconds.

REFRESH = 10	IO										SESSIONS = 1		
per 10.0 sec	10	20	30	40	50	60	70	80	90	100	CUR,	AVG,	MAX
BUFF ACCESS											0,	3,	81
DATA DISK RD											0,	1,	26
DATA DISK WR											0,	0,	0

You can also see that on average, DATA DISK RD is 1, which means that 1 of the 3 pages does not reside in the pool. That page may need to be read from disk, resulting in physical I/O.

To determine which sessions are engaged in data buffer I/O, access the IO Data Session screen. In the example screen that follows, PID 27344 has issued the highest number of page requests (BUFF ACCESS), but PID 27332 has the highest data buffer miss rate.

REFRESH = 10	IO DATA SESSION								SESSIONS = 2	
SORTIODATA	(1)		(2)		(*3)		(4)			
PID	BUFF ACCESS		DATA DISK RD		DATA DISK WR		MISS RATE			
27344	192	69%	22	46%	4	100%	11%			
27316	0	0%	0	0%	0	0%	0%			
27332	87	31%	26	54%	0	0%	30%			

Insufficient Log Buffer Space

Log files are required to recover data in the event of a system crash, but the I/O incurred by logging can reduce the performance of your DBEnvironment. You can use the IO screen to monitor log buffer I/O. If the log buffer I/O seems excessive, try increasing the number of log buffer pages. For more information, see "Choosing a Number of Data Buffer Pages" in the chapter, "Guidelines on System Administration." Limiting the number of checkpoints issued can also reduce logging I/O.

To monitor the amount of log buffer I/O activity, check the fields shown in the following IO screen.

REFRESH = 10	IO										SESSIONS = 2		
per 0.5 sec	10	20	30	40	50	60	70	80	90	100	CUR,	AVG,	MAX
LOG BUFF WR											38,	2,	93
LOG DISK RD											10,	1,	32
LOG DISK WR											19,	1,	37

To determine which sessions are performing log buffer I/O, check the IO Log Session screen. In the following example, PID 6167 is responsible for all of the log buffer I/O activity.

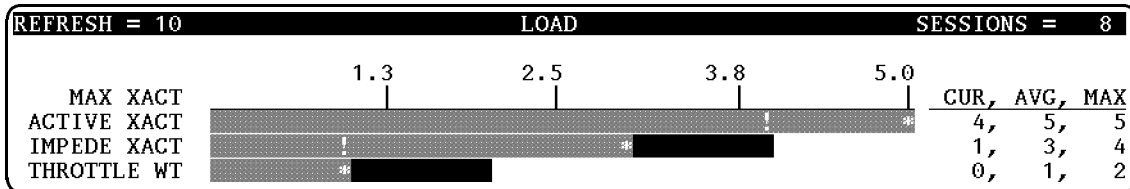
REFRESH = 10		IO LOG SESSION				SESSIONS = 3	
SORTIOLOG		(1)		(2)		(3)	
PID	LOG BUFF WR	LOG DISK RD	LOG DISK WR				
325	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%
5955	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%
6167	9332 100%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%

Load Subsystem

The Load subsystem is useful in troubleshooting throughput problems. This section describes how to handle transaction delays, rollbacks, and lock contention.

Transaction Delays

The ACTIVE XACT, IMPEDE XACT, and THROTTLE WT fields on the Load screen help identify the source of transaction delays. In the following screen, these fields indicate two problems.



The first problem is that the transaction limit has been reached. The transaction limit (MAX XACT) for the DBEnvironment is 5.0. The average number of active transactions (ACTIVE XACT) is also 5.0. When the transaction limit is reached, other sessions that attempt to begin a transaction are placed in the throttle wait queue. These sessions must wait until a transaction slot becomes free. Notice that on average one session is waiting in the throttle wait queue. You can increase the transaction limit by using

- The START DBE statement.
- The SQLUtil ALTDBE command. (However, remember that you must stop the DBEnvironment before you use this command.)

The second problem is that other sessions hold incompatible locks. The current value of IMPEDE XACT is 1, which means that one active transaction is waiting to acquire a lock. On average, IMPEDE XACT is 3 and ACTIVE XACT is 5, which means that 3 out of 5 transactions are waiting.

This DBEnvironment definitely has a locking problem. At this point, you should use the Lock subsystem to obtain more information.

Rollbacks

In a DBEnvironment with high throughput, transactions are completed quickly and successfully. The following screen shows that on average, 2 transactions are started and completed every 10.0 seconds.

REFRESH = 10		LOAD										SESSIONS = 4			
.															
.															
per 10.0 sec															
BEGIN WORK	[Progress bar]										14,	2,	80		
COMMIT WORK	[Progress bar]										10,	2,	80		
ROLLBK WORK	[Progress bar]										5,	0,	14		
DEADLOCKS	[Progress bar]										0,	0,	0		

On average, ROLLBK WORK is 0, which means that typically no transactions are rolled back. However, during the last refresh cycle of the LOAD screen, five transactions were rolled back.

To find out which sessions are rolling back transactions, access the Load Session screen. On the following screen, PID 26556 is responsible for most of the rolled back transactions.

REFRESH = 10		LOAD SESSION								SESSIONS = 4	
SORTLOAD	(1)	(2)		(*3)		(4)					
PID	BEGIN WORK	COMMIT WORK	ROLLBK WORK	DEADLOCKS							
26556	8	7%	1	1%	7	88%	0	0%			
26560	85	76%	84	81%	1	13%	0	0%			
26597	10	9%	10	10%	0	0%	0	0%			
27600	9	8%	9	9%	0	0%	0	0%			

You can see from the screen that the SET SORTLOAD 3 command has sorted the sessions in descending order according to ROLLBK WORK activity.

Lock Contention

The Load screen also provides an overall view of the amount of lock contention in the DBEnvironment. In the example that follows, 25% of the lock requests are forced to wait. For detailed information on locking, you should use the screens in the Lock subsystem.

REFRESH = 10		LOAD										SESSIONS = 8			
.															
.															
per 10.0 sec															
LOCK REQSTS	[Progress bar]										4,	6,	20		
LOCK WAITS	[Progress bar]										1,	0,	1		
LOCK WAIT %	[Progress bar]										25%,	0%,	25%		

Lock Subsystem

Use the Lock subsystem to troubleshoot performance bottlenecks caused by lock contention. This section describes how to handle lock waits, deadlocks, and lock allocation failures.

Lock Waits

If an end user complains of a hung session, perhaps his or her session is actually waiting for a lock. A session will wait for a lock if another session has already acquired an incompatible lock on the same object. For more information on lock waits, see the chapter “Concurrency Control through Locks and Isolation Levels” in the *ALLBASE/SQL Database Administration Guide*.

For example, we can use SQLMON to find out why the session of the user logged on as BNORTON appears to be hung. Since the user is connected to the PartsDBE DBEnvironment, issue the following command:

```
SQLMONITOR OVERVIEW => SET DBENV PartsDBE
```

To make the screens easier to read, set the lock filter to limit the lock information displayed to locks for which at least one session is waiting (W) and locks that are being converted to a stronger mode (C):

```
SQLMONITOR OVERVIEW => SET LOCKFILTER //WC//
```

Overview Session Screen

Invoke the Overview Session screen to obtain the PIDs of all database sessions connected to the DBEnvironment:

```
SQLMONITOR OVERVIEW => /o s
```

From the following screen, we can see that the BNORTON login has a PID of 26180 and a wait status.

REFRESH = 10		OVERVIEW SESSION					SESSIONS = 5
PID	LOGIN NAME	STATUS	XID	ISO	PRI	LABEL	
6167	DGREEN	Idle	47426	RR	127		
26180	BNORTON	Wait	47488	RR	127		
26923	SCHEN	Wait	47505	RR	127		
27333	RSMITH	Wait	47524	RR	127		
27783	JVALDEZ	Wait	47587	RR	127		

Lock Session Screen

To get locking information for PID 26180, access the Lock Session screen:

```
SQLMONITOR OVERVIEW => /loc s  
PID: 26180
```

The value of the GWC field in the following screen verifies that the database session that appears to be hung is actually waiting to convert a lock. The WAITS FOR column identifies PID 6167 as the session that is causing PID 26180 to wait.

REFRESH = 10	LOCK SESSION	LOCKFILTER = SU/TPR/WC/SXR _{sxr6v} /-
PID:26180	STATUS:Waiting for LOCK	XID:47488 ISO: RR
LOGIN NAME: BNORTON		LABEL: PRI:127
PROGRAM NAME: orderp		
G OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	GWC MOD NEW WAITS FOR:
T PURCHDB.VENDORS		G s x PID: 6167

Now you can invoke the Lock Session screen for PID 6167, which has acquired a lock on the PurchDB.Vendors table. To make the screen easier to read, limit the lock information displayed to the PurchDB.Vendors table. Change the lock filter so that all locks are displayed.

```
SQLMONITOR LOCK => SET LOCKTABFILTER PurchDB.Vendors
```

```
SQLMONITOR LOCK => SET LOCKFILTER ////
```

```
SQLMONITOR LOCK => /loc s 6167
```

REFRESH = 10	LOCK SESSION	LOCKFILTER = SU/TPR/GWC/SXR _{sxr6v} /-
PID: 6167	STATUS:Idle	XID:47426 ISO: RR
LOGIN NAME: DGREEN		LABEL: PRI:127
PROGRAM NAME: isql		
G OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	GWC MOD NEW WAITS FOR:
T PURCHDB.VENDORS		G S

Lock Impede Screen

To identify all sessions that are impeded by PID 6167, invoke the Lock Impede screen:

```
SQLMONITOR LOCK => i
PID: 6167
```

The following screen reveals that the lock held by PID 6167 is causing a number of other sessions, including PID 26180, to wait.

REFRESH = 10	LOCK IMPEDE
PID: 6167	STATUS:Idle XID:47426 ISO: RR
LOGIN NAME: DGREEN	LABEL: PRI:127
PROGRAM NAME: isql	
G OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID GWC MOD NEW WAITING:
T PURCHDB.VENDORS	G S x PID: 26180
	s PID: 26923
	s PID: 27333
	s PID: 27783

To solve this problem, you should investigate PID 6167. The Lock Session screen reveals that PID 6167 is running ISQL, and is run by the user logged on as DGREEN. If PID 6167 does not immediately end the transaction holding the incompatible lock, you should consider executing the ISQL statement TERMINATE USER. If the transaction has been marked with a label, you can clearly identify the part of the program the user is running. PID 6167 is executing a repeatable read (RR) transaction.

7-10 Troubleshooting with SQLMON

A better solution might be to change the transaction's isolation level. The `XID` field displays a system-generated integer that uniquely identifies the transaction, but it does not identify the SQL statements that have been executed. The `LABEL` field displays an 8-character string that the application program defines in the `BEGIN WORK` or `SET TRANSACTION` statements. If the application program uses the `LABEL` field carefully, it will be easy for you to debug all the transactions in your application programs.

Deadlocks

A deadlock occurs when two transactions are each waiting for a database object which the other has locked. `ALLBASE/SQL` automatically detects deadlocks and resolves them by rolling back one of the transactions. For a detailed discussion on deadlocks, see the chapter “Concurrency Control through Locks and Isolation Levels” in the *ALLBASE/SQL Database Administration Guide*.

To examine the locks that cause a deadlock, you must freeze one of the sessions before `ALLBASE/SQL` rolls it back and releases its locks. The steps in the following example show how to create a deadlock, freeze a session, and use `SQLMON` to examine the deadlocked session's locks. To perform these steps, you will need access to a Sample DBEnvironment and the ability to create at least four windows on your workstation or PC. For instructions on setting up the Sample DBEnvironment, see the chapter “Practice with `ALLBASE/SQL` Using PartsDBE” in *Up and Running with ALLBASE/SQL*.

Step 1 Open Four Windows

Use the windows as follows:

- Window 1 The first ISQL session that is deadlocked.
- Window 2 The second deadlocked ISQL session. Since this session will be rolled back by `ALLBASE/SQL`, it is the one you want to freeze.
- Window 3 An `SQLMON` session for examining the locks held by the deadlocked sessions.
- Window 4 The window used to create the file that releases the session in window 2.

Step 2 Set Up the Freeze

In window 2, set the `DBCORERR` environment variable to 1024 before running ISQL. If you are using the C shell, issue the following command:

```
% setenv DBCORERR 1024
```

Korn shell and Bourne shell users should use the following commands:

```
% DBCORERR=1024
% export DBCORERR
```

When troubleshooting a production system, set the `DBCORERR` environment variable for all sessions that might be involved in the deadlock, since it is difficult to predict which session will be rolled back. See the section “Freezing DBEnvironment Sessions” in this chapter for more information on setting the `DBCORERR` environment variable.

Step 3 Create a Deadlock

Run ISQL in windows 1 and 2, and issue the following statements:

```
Window 1    CONNECT TO PartsDBE
Window 2    CONNECT TO PartsDBE
Window 1    UPDATE PurchDB.Parts SET SalesPrice = 1.2 * SalesPrice
Window 2    SELECT * FROM PurchDB.SupplyPrice
Window 2    Exit from the ISQL browser but do not commit work.
Window 1    UPDATE PurchDB.SupplyPrice SET UnitPrice = 1.2 * UnitPrice
Window 2    SELECT * FROM PurchDB.Parts
```

When you complete step g, you see a message in window 2:

```
*****PID=4727*****RC=1024 : Wed Mar 31 14:08:07 1993
```

This message indicates that the DBEnvironment session having PID 4727 (that is, the session running in window 2) is frozen on DBCORE error 1024, which means that a deadlock has been detected. Under these circumstances, ALLBASE/SQL does not roll back the session until you release it. This will be described in “Step 5 Release the Frozen Session”.

The deadlock is illustrated in Figure 7-1.

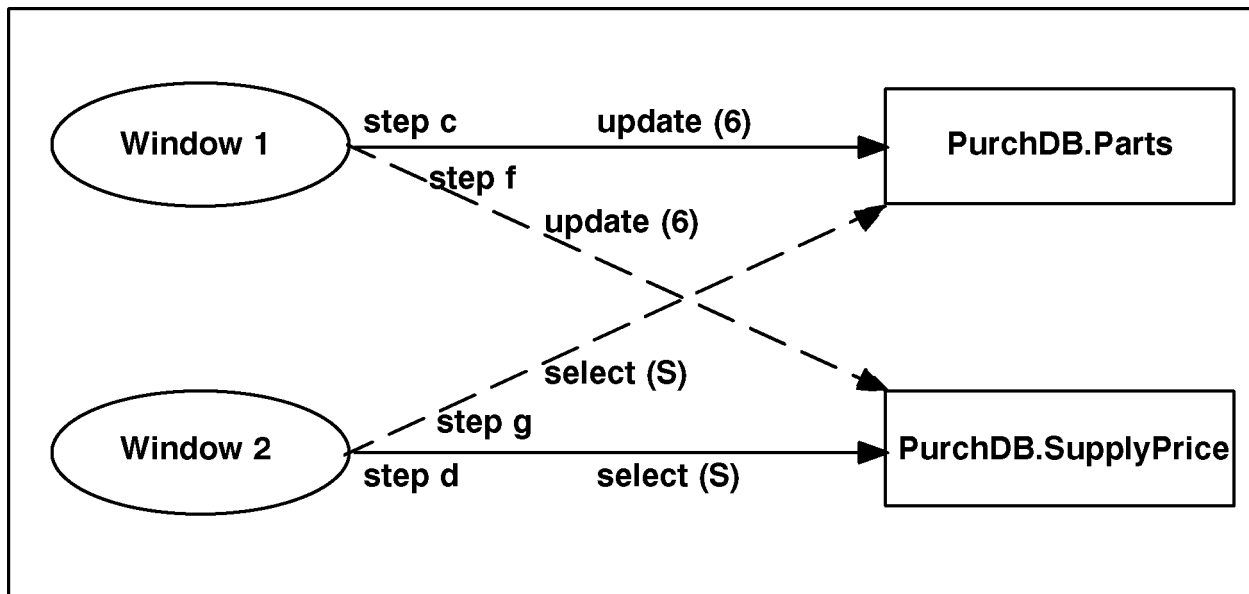


Figure 7-1. Deadlock Example

PID 4687 is running in window 1, and PID 4727 is running in window 2. The solid lines represent locks that have been granted, and the dashed lines represent sessions that have been waiting to acquire locks. The letter S designates a share lock, and the number 6 represents a share plus intent exclusive (SIX) lock. This lock information is displayed by the SQLMON screens accessed in the next step.

Step 4 Examine the Locks with SQLMON

In window 3, run SQLMON and set the DBEnvironment to PartsDBE. Issue the SET LOCKFILTER command to limit the lock information displayed to sessions that are waiting to acquire or convert locks.

```
% sqlmon
```

```
SQLMONITOR OVERVIEW => SET DBENV PartsDBE
```

```
SQLMONITOR OVERVIEW => SET LOCKFILTER //WC//
```

Invoke the Lock screen to see which tables have been locked:

```
SQLMONITOR OVERVIEW => /lock
```

REFRESH = 10		LOCK	LOCKFILTER = SU/TPR/WC/SXR _{sxr6v} /1	
G	OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	LOCK QUEUE	
T	PURCHDB.PARTS		6S	
T	PURCHDB.SUPPLYPRICE		S6	

In the Lock screen, you can see the PurchDB.Parts and PurchDB.SupplyPrice tables. There are two locks on each table: a share plus intent exclusive (6) lock, and a share (S) lock. The characters displayed in inverse video represent locks held by sessions that are waiting.

To identify the PID of each session in the lock queue of each of the objects shown above, access the Lock Object screen:

```
SQLMONITOR LOCK => o
```

In the following screen, PID 4687 has been granted (G) a share plus intent exclusive (6) lock on the PurchDB.Parts table. PID 4727 is waiting to acquire a share (S) lock on PurchDB.Parts. A share (S) lock has been granted to PID 4727 on the PurchDB.SupplyPrice table, and PID 4687 is waiting to acquire a share plus intent exclusive (6) lock on PurchDB.SupplyPrice.

REFRESH = 10			LOCK OBJECT		LOCKFILTER = SU/TPR/WC/SXR _{sxr6v} /1			
GRN	OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID						
GWC	MOD	NEW	PID	LOGIN NAME	XID	ISO	PRI	LABEL
				PROGRAM NAME				
T PURCHDB.PARTS								
G	6		4687	DGREEN	50209	RR	127	
				isql				
W		S	4727	BNORTON	50210	RR	127	
				isql				
T PURCHDB.SUPPLYPRICE								
G	S		4727	BNORTON	50210	RR	127	
				isql				
W		6	4687	DGREEN	50209	RR	127	
				isql				

Use the Lock Impede screen to verify that the sessions are deadlocked. To identify which sessions are impeded by PID 4687, invoke the Lock Impede screen as follows:

```
SQLMONITOR LOCK => i
PID: 4687
```

In the following screen, PID 4727 is listed in the WAITING column. PID 4727 is waiting for PID 4687 to release the lock it holds on PurchDB.Parts.

REFRESH = 10		LOCK IMPEDE				
PID: 4687		STATUS:Waiting for LOCK		XID:50209	ISO: RR	
LOGIN NAME: DGREEN		LABEL:		PRI:127		
PROGRAM NAME: isql						
G	OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	GWC	MOD	NEW	WAITING:
T	PURCHDB.PARTS		G	6	S	PID: 4727

If you specify PID 4727, you see the following screen.

REFRESH = 10		LOCK IMPEDE				
PID: 4727		STATUS:Running		XID:50210	ISO: RR	
LOGIN NAME: BNORTON		LABEL:		PRI:127		
PROGRAM NAME: isql						
G	OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	GWC	MOD	NEW	WAITING:
T	PURCHDB.SUPPLYPRICE		G	S	6	PID: 4687

Note that PID 4687 is waiting for PID 4727 to release the lock on the table PurchDB.Supplyprice. These two sessions are indeed deadlocked.

Step 5 Release the Frozen Session

In window 4, edit a file named /tmp/SQLunfrz.ALL and enter the deadlock error number, 1024, on the first line. When you save the file, the session is released, and a message such as the following is displayed in window 2:

```
***** UNFREEZING PID=4727*****RC=1024 : Wed Mar 31 14:54:38 1993
Deadlock detected. (DBERR 14024)
```

Lock Allocation Failures

If a session requests a lock when space is not available in the runtime control block, a lock allocation error (DBERR 4008) occurs. In the example that follows, the session that encounters DBERR 4008 is frozen, allowing you to examine the locks with SQLMON.

Step 1 Open Three Windows

To perform the steps in this example, your workstation or PC must have a windows environment. The windows will be used as follows:

- Window 1 The DBEnvironment session that encounters the lock allocation failure
- Window 2 An SQLMON session
- Window 3 The window used to create the file that releases the session in window 1

Step 2 Set Up the Freeze

In window 1, set the DBCORERR environment variable to 1035. If you are using the C shell, issue the following command:

```
% setenv DBCORERR 1035
```

Korn shell and Bourne shell users should use the following commands:

```
% DBCORERR=1035
% export DBCORERR
```

Step 3 Generate the Error

In window 1, run the application that encounters the lock allocation failure. When DBCORE detects the failure, it freezes the session and displays a message such as the following on standard output:

```
*****PID=17325*****RC=1035 : Wed Apr 7 11:09:57 1993
```

Step 4 Investigate the Session with SQLMON

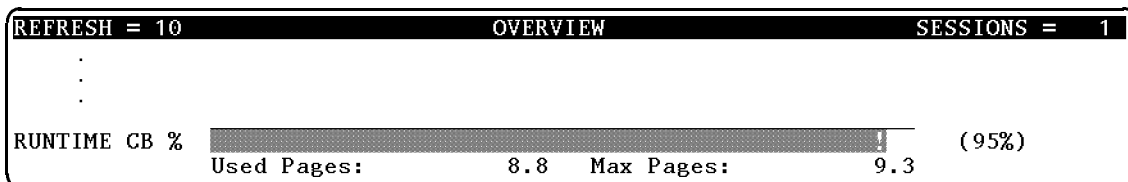
In window 2, run SQLMON, set the DBEnvironment option, and invoke the Overview screen:

```
% sqlmon
```

```
SQLMONITOR OVERVIEW => SET DBENV DBEnvironmentName
```

```
SQLMONITOR OVERVIEW => o
```

The Overview screen shown below indicates that the runtime control block space is nearly full and no more control blocks can be allocated.



Access the Lock Memory screen as follows:

```
SQLMONITOR OVERVIEW => /loc m
```

In the screen that follows, PID 17325 has a total of 324 locks. Most of the locks are held on rows, because the table in this example was created with the PUBLICROW type. To solve this problem, you should either alter the table to PUBLIC to avoid numerous row level locks, or allocate additional runtime control block pages.

Leslie-Anne says that a new screen is needed

REFRESH = 10		LOCK MEMORY			SESSIONS = 1	
SORTLOCK	(1)	(2)	(3)	(4)	(*5)	
PID	TABLE	PAGE	TUPLE	TOTAL	MAXTOTAL	
17325	8	9	307	324	324	

Step 5 Release the Frozen Session

In window 3, edit a file named /tmp/SQLunfrz.ALL and enter the lock allocation failure error number, 1035, on the first line. When you save the file, the session is released, and a message such as the following appears in window 1:

```
***** UNFREEZING PID=17325*****RC=1035 : Wed Apr 7 11:29:26 1993
ALLBASE/SQL shared memory lock allocation failed in DBCore. (DBERR 4008)
```

Freezing DBEnvironment Sessions

ALLBASE/SQL allows you to freeze a session whenever it encounters a certain DBCORE error. While the session is frozen, you can use SQLMON to examine the session's locks.

To freeze a session, set the DBCORERR environment variable to one or more DBCORE error numbers. When the session encounters the errors you specify, it freezes. For example, to freeze a session that encounters either DBCORE error 1024 (deadlock) or 1035 (lock allocation failure), set DBCORERR from the C shell as follows:

```
% setenv DBCORERR "1024 1035"
```

Korn shell and Bourne shell users should use the following commands:

```
% DBCORERR="1024 1035"
% export DBCORERR
```

If you set DBCORERR to a negative number, a session freezes if it encounters an error whose number is greater than the absolute value of DBCORERR. For example, if DBCORERR is set to -4000, the session freezes if it encounters a DBCORE error greater than 4000.

When a session freezes, a message appears on standard output. The PID identifies the frozen session, and the RC field designates the DBCORE error number:

```
*****PID=18250*****RC=1024 : Mon Mar 22 11:14:53 1993
```

DBCORE is a set of internal ALLBASE/SQL routines. DBCORE errors have different error numbers than DBERR errors. For example, the DBCORE error number for detecting deadlocks is 1024, but the DBERR number is 14024. When setting the DBCORERR environment variable, be sure to use the DBCORE error number, not the DBERR error number.

Releasing DBEnvironment Sessions

To release a session, use a text editor to enter DBCORE error numbers in a file. When you save the file, the frozen session resumes execution. The DBCORE error numbers in the file must correspond to the value of the DBCORERR environment variable. To release a session on any DBCORE error number, enter an ASCII zero in the file.

The file can contain one or more lines, and each line can contain one or more numbers. You can name the file as follows:

- SQLunfrz Releases only those sessions running from the subdirectory containing this file.
- /tmp/SQLunfrz.*xxx* Releases only the session whose PID is specified in *xxx*. For example, if the PID is 8895, the name of the file should be /tmp/SQLunfrz.8895.
- /tmp/SQLunfrz.ALL Releases only the frozen sessions that qualify, according to the error numbers entered in the file.

For example, to release any session that froze because it encountered either DBCORE error 1024 or 1035, use an editor to create a file named /tmp/SQLunfrz.ALL, and add the following lines to it:

```
1024
1035
```

When ALLBASE/SQL releases a session, it displays a message on standard output like the one below:

```
***** UNFREEZING PID=18250*****RC=1024 : Mon Mar 22 11:16:59 1993
```

SampleIO Subsystem

The SampleIO subsystem is useful for balancing the I/O load of a DBEnvironment. The SampleIO screens display the amount of data buffer swapping activity for DBEFiles, tables, indexes, and referential constraints. For more information, refer to the section “Load Balancing” in the chapter “Guidelines on System Administration.”

You should use the IO subsystem to tune the size of the data buffer pool before you use the SampleIO subsystem to balance load.

Note Use the SampleIO subsystem sparingly, because it significantly increases CPU usage.

Using the SET SAMPLING Command

The SET SAMPLING command enables or disables sampling of the data buffer pool. Sampling only occurs if SAMPLING is ON and you access a SampleIO screen. The counters on the SampleIO screens are set to 0 when you issue SET DBENVIRONMENT, and are then incremented as sampling occurs. The counters only reflect activity that was “seen” during sampling. The counters are cumulative; they represent the total activity observed over all samples taken since you issued SET DBENVIRONMENT.

By default, SQLMON does not display SampleIO screens during sampling. Instead, it displays a scale that allows you to determine the amount of sampling that has occurred. For example, if you issue the SET REFRESH 10 and SET CYCLE 5 commands and then invoke a SampleIO screen, SQLMON displays the following:

```
SQLMONITOR SAMPLEIO => /sa

SAMPLING = ON
REFRESH = 10      (One set of samples will be taken every 10 seconds).
CYCLE = 5         (A total of 5 sets of samples will be taken, then you
                  will automatically return to the SQLMONITOR prompt).

1020304050
12345678901234567890123456789012345678901234567890
```

The command SET REFRESH 10 means that SQLMON will take one set of samples every 10 seconds, and SET CYCLE 5 means that SQLMON will take 5 sets of samples and then return to the prompt.

The scale includes the following elements:

1. The current value of the SAMPLING variable. When you see a refresh scale, SAMPLING is ON, because this scale is never printed if SAMPLING is OFF. To view the results of previous sampling, issue the command SET SAMPLING OFF and then invoke the SampleIO screen you want to see.
2. The current value of the REFRESH variable, which determines the number of seconds that SQLMON pauses between each refresh cycle.
3. The current value of the CYCLE variable, which determines the number of refresh cycles that occur before you return to the SQLMON prompt. If CYCLE is OFF, you must press Return to exit sampling and return to the prompt.

4. SQLMON displays a period each time a refresh cycle completes.

In the example above, SQLMON displays a period every 10 seconds, because REFRESH is set to 10. It displays 5 periods, because CYCLE is 5. SQLMON takes a total of 125 samples, which is 5 refresh cycles X 25 samples each.

After SQLMON has completed the sampling, issue the command SET SAMPLING OFF. If you are using SQLMON interactively, and if CYCLE has a value other than OFF, you should also issue SET CYCLE OFF. Then, you can invoke the SampleIO screens in which you are interested.

For example, the following screen lists the amount of I/O for each DBEFile.

REFRESH = 5		SAMPLEIO			
SORTSAMPLEIO	DBEFILESET	DBEFILE	(1) SWAPIN	(2) SWAPOUT	(*3) TOTAL IO
	SYSTEM	PARTSDBE0	1751	0	1751
	PURCHFS	PURCHDATAF1	192	31	223
	WAREHFS	WAREHDATAF1	142	0	142
	WAREHFS	WAREHINDEXF1	42	0	42

PartsDBE0, which exists in the SYSTEM DBEFileSet and contains the system catalog, has the most read I/O activity. The only DBEFiles with write I/O activity are PurchDataF1 and PartsDBE0.

DBEFiles provide storage for objects that exist within a DBEFileSet. I/O occurs for a DBEFile whenever I/O occurs for the tables, indexes, or referential constraints within the DBEFileSet. You can use the SampleIO screen to determine which DBEFiles (and therefore which DBEFileSets) have the most I/O.

In the SampleIO screen above, we see that objects in the PurchFS DBEFileSet are showing slightly more I/O activity than objects in the WarehFS DBEFileSet. You can use the SampleIO Tables, SampleIO Indexes, and SampleIO TabIndex screens to obtain I/O information about the individual objects in these DBEFileSets.

In the following screen, we see that PurchDB.SupplyPrice is the table with the most activity in the PurchFS DBEFileSet, and is responsible for the write I/O activity that we saw on the SampleIO screen.

REFRESH = 5		SAMPLEIO TABLES		
OWNER	TABLE	(1) SWAPIN	(2) SWAPOUT	(*3) TOTAL IO
DBEFILESET = PURCHFS				
PURCHDB	SUPPLYPRICE	95	31	126
DBEFILESET = WAREHFS				
PURCHDB	PARTS	91	0	91

PurchDB.Parts shows the most activity in the WarehFS DBEFileSet. The next example shows that an index on PurchDB.Parts is also undergoing I/O.

REFRESH = 5		SAMPLEIO INDEXES			
SORTSAMPLEIO	OWNER	TABLE/INDEX,CONST	(1) SWAP IN	(2) SWAPOUT	(*3) TOTAL IO
DBEFILESET = WAREHFS					
PURCHDB		PARTS			
		PARTNUMINDEX	42	0	42

Using the SET DISPLAYSAMPLES Command

If you want to display screens in the SampleIO subsystem during sampling, instead of seeing the refresh scale shown above, you can do so by issuing a SET DISPLAYSAMPLES ON command. When DISPLAYSAMPLES is ON, the screen is refreshed after each set of samples is obtained, that is, after each refresh cycle.

When DISPLAYSAMPLES is OFF, you see a refresh scale instead of the screen, and a period is displayed each time a refresh cycle completes. Because SQLMON does not display screen images, a smaller amount of output is created when sampling occurs, which might be especially desirable for batch jobs. In addition, SQLMON needs less CPU time, because some processing to sort and format the information displayed on the screen is avoided.

When SAMPLING is OFF, the SET DISPLAYSAMPLES command has no effect.

A Sample Batch Job

For example, you can use the following script within an SQLMON batch job to obtain SampleIO statistics for the day on a particular DBEnvironment.

```

/set dbenv MyDBE
/set menu off
#
#####
# SAMPLING is ON by default, so the following command is not really
# necessary.
#####
/set sampling on
#
#####
# Do not generate screens while sampling. This reduces the output
# generated by this job, and also reduces the amount of CPU SQLMON
# consumes when it gathers statistics. DISPLAYSAMPLES is OFF by default,
# so the following command is not really necessary.
#####
/set displaysamples off
#

```

```

#####
# Take samples every 10 minutes for 8 hours:
#####
# Take 1 set of 25 samples every 10 minutes (10 min x 60 sec/min = 600 sec)
/set refresh 600
#
# Take 48 sets of samples (8 hours = 480 min x (1 cyc/10 min) = 48 cyc)
/set cycle 48
#
#####
# Now perform the sampling. It does not matter which SampleIO screen is
# used. The following command causes the SampleIO screen to be
# visited 48 times (a 10-minute pause occurs between each visit).
#####
/sampleio
#
#####
# Now print a report. When SAMPLING is OFF, each screen is painted
# without taking additional samples of the data buffer pool. We simply
# view the statistics that were obtained as a result of all of the
# previous sampling.
#
# If you set the REFRESH variable to a large value for sampling, it's
# good practice to reset it to a lower (normal) value when turning
# sampling off. It's also good practice to reset the CYCLE variable.
#####
/set sampling off
/set refresh 10
/set cycle 1
#####
# The following commands cause tables and indexes from all DBEFileSets
# to be included in the report. They will be printed in descending order
# according to the total amount of I/O that was observed during sampling.
#####
/set dbfileset off
/set sortsamp 3
/set output myfile
/set
/sa
/sa tables
/sa indexes
/sa tabi owner.table1
/sa tabi owner.table2
/exit

```

Once you enter this script, you can use the HP-UX `at` command to execute the job at a particular time.

Understanding the Internals of Sampling

SampleI/O statistics are generated entirely by SQLMON. In other words, the data is not obtained by simply reading from some existing table where these statistics are maintained. The more often you perform sampling, the more complete the I/O statistics become. However, SQLMON uses CPU time whenever it takes samples. The larger the number of samples SQLMON takes, the larger the amount of CPU time it consumes. Furthermore, SQLMON needs more CPU time to examine large data buffer pools than small data buffer pools.

When SAMPLING is ON, SQLMON takes 25 “snapshots” of the data buffer pool during each refresh cycle. SQLMON pauses between successive snapshots. The length of the pause is the refresh rate divided by 25.

As SQLMON takes each snapshot, it keeps track of the pages that are currently in the data buffer pool and the pages that were in the data buffer pool during the last snapshot. The SWAPIN, SWAPOUT, and TOTALIO counters are then incremented to reflect the changes.

The SWAPIN value represents read I/O. The SWAPIN value will be incremented by 1 if

- A page is in the pool now, but it was not in the pool during the last snapshot.
- A page was in the pool last time, but it was dirty then and it is clean now. The page was swapped out to disk and then swapped back in again.
- A page was in the pool last time, but it was occupying a different buffer page location than it is now.

The SWAPOUT value represents write I/O. The SWAPOUT value will be incremented by 1 if

- A page is in the pool now, but it was not in the pool during the last snapshot. The page is also dirty (if it is not dirty, the SWAPOUT value is not modified).
- A page was in the pool last time, but it was clean then and is dirty now.

If the page was dirty during the last snapshot, and it is still dirty, the SWAPOUT column is not modified. When a dirty page stays in the buffer pool for a long time without being swapped out, ALLBASE/SQL is using the information well without paying a high I/O price.

The TOTAL value represents total I/O, and is obtained by adding the SWAPIN and SWAPOUT values.

Static Subsystem

The Static subsystem allows you to troubleshoot full DBEFileSets, poorly clustered indexes, indirect rows, and hash overflow pages.

Full DBEFileSets

A transaction fails if it attempts to insert a row into a table whose DBEFileSet is full. To prevent this, you should monitor DBEFileSet capacity on a regular basis. The Static DBEFile screen displays the capacity of each DBEFile and DBEFileSet in a DBEnvironment. In the following Static DBEFile screen, the InvoiceFS DBEFileSet is 92% full. It contains only 12 pages and should be expanded.

PartsDBE		STATIC DBEFILE					
DBEFILESET	DBEFILE TYP BD	DBEFILESET	FULLNESS	%	FSUSED PAGES	FSMAX PAGES	
		DBEFILE	FULLNESS	%	USED PAGES	MAX PAGES	
.							
.							
.							
INVOICEFS			!	92%	11	12	
	INVOICEDATAF1 MIX		!	92%	11	12	

You can plan for capacity by monitoring the growth of DBEFileSets and the database objects contained within them.

Poorly Clustered Indexes

A poorly clustered index forces ALLBASE/SQL to access more physical pages during index scans. Performance degrades because I/O increases. Applications that frequently access rows in index order perform better if the rows are physically stored together on disk in index order.


To check the clustering of an index, go to the Static Cluster screen. In the following example, the CCOUNT of the InvoiceIndex is approximately equal to the number of rows in the table, which indicates that the data is poorly clustered.

partsdb		STATIC CLUSTER				
DBEFILESET	OWNER, TABLE	UNLOAD/LOAD	SUGGESTD	%	TABLE PAGES	TOTAL ROWS
						CCOUNT
INVOICEFS						
PURCHDB.CUSTOMER					5	160
(INDEX)	CUSTINDEX		!	56%		92

If this index is used frequently to access rows, you should unload the PurchDB.Invoice table in sorted order and reload it with ISQL. For more information, see the section “Clustering Indexes” in the chapter “Guidelines on Logical and Physical Design.”

Indirect Rows

Avoid indirect rows, because they waste disk space and increase the amount of I/O needed to access data. Use the Static Indirect screen to detect the presence of indirect rows. On the screen that follows, the PurchDB.Invoice table has 12% indirect rows. For instructions on how to remove indirect rows, see “Unloading and Reloading to Remove Indirect Rows” in the chapter “Guidelines on Logical and Physical Design.”

partsdb		STATIC INDIRECT				
DBFILESET	OWNER, TABLE	TABLE	INDIRECT	ROW	%	TOTAL ROWS
	:					
	:					
	:					
INVOICEFS						
	STOREDSECT. INVOICEFS				0%	0
	PURCHDB. CUSTOMER				12%	160
	PURCHDB. INVOICE				0%	0

Hash Overflow Pages

As the number of hash overflow pages grows, the amount of I/O necessary to obtain table data increases. To monitor the overflow pages of a hash structure, access the Static Hash screen. In the next example, the PurchDB.Invoice table has seven overflow pages, and should be unloaded and reloaded to improve performance.

PartsDBE		STATIC HASH																							
DBFILESET	OWNER, TABLE	OVERFLOW CHAIN LNTH																							
	PRIMPAGES	PRIMDATA	PRIMOVERF	OVERPAGES	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	AVGOVERFLOW	
INVOICEFS																									
	PURCHDB. INVOICE																								
		11	1	1	7																				7
																									7

For instructions, see “Unloading and Reloading to Remove Overflow Pages” in the chapter “Guidelines on Logical and Physical Design.”

SQLMON Screen Reference

This chapter describes each of the SQLMON screens in alphabetical order.

You will find a table listing each of the SQLMON screens in the chapter “Getting Started with SQLMON.” For complete descriptions of the SQLMON commands, refer to the chapter “SQLMON Command Reference.”

IO Screen

This screen provides I/O information on the data and log buffer pools.

REFRESH = 20		IO										SESSIONS = 2		
per	1.0 sec	10	20	30	40	50	60	70	80	90	100	CUR,	AVG,	MAX
BUFF ACCESS		█										3,	0.4,	20
DATA DISK RD												0.1,	0,	0.4
DATA DISK WR		█										0,	0,	1
LOG BUFF WR		█										1,	0.1,	3
LOG DISK RD												0,	0,	0
LOG DISK WR		█										0,	0,	1
CHECKPOINTS												0,	0,	0
		10	20	30	40	50	60	70	80	90	100%			
DATA BUFFER MISS RATE		█										3%,	0%,	12%
		TOTAL DATA BUFFER PAGES:								100				
		TOTAL LOG BUFFER PAGES:								24				
		ARCHIVE MODE:								ON				
LEGEND:		█ CURRENT	█ AVG	█ BOTH CURRENT & AVG		█ MAX								

To invoke the IO screen, use the `i` command from the IO subsystem or the `/i` command from other subsystems.

When you issue the `SET DBENVIRONMENT` command, the counters on this screen are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
per sec	The scale used to identify the length of the inverse video bars and the data in the CUR, AVG, and MAX columns. In the example above, the MAX frequency of BUFF ACCESS is 20 per 1.0 second.
BUFF ACCESS	The number of page requests issued to the data buffer pool.
DATA DISK RD	The number of pages that are not in the data buffer pool at request time. The operating system must fetch these pages, either from the operating system buffer pool or by a physical disk read.
DATA DISK WR	The number of dirty pages from the data buffer pool that were logically written to disk.
LOG BUFF WR	The number of records written to the log buffer pool.
LOG DISK RD	The number of log pages fetched from disk. Unless you have used rollforward recovery, this value is zero. The operating system performs a physical read only if the log page is not found in the operating system buffer pool.

LOG DISK WR	The number of log pages written from the log buffer pool to disk. The operating system performs a physical write for each log buffer write.
CHECKPOINTS	The number of checkpoints taken. Checkpoints are performed automatically, when the nonarchive log becomes full, or when you issue a CHECKPOINT command.
DATA BUFFER MISS RATE	The percentage of pages that are not in the data buffer pool at request time. The percentage is calculated as follows: $\text{DATA BUFFER MISS RATE} = (\text{DATA DISK RD} / \text{BUFF ACCESS}) * 100$
TOTAL DATA BUFFER PAGES	The number of data buffer pages configured. To change the number of data buffer pages, issue the START DBE statement or the SQLUtil ALTDBE command.
TOTAL LOG BUFFER PAGES	The number of log buffer pages configured. To change the number of log buffer pages, issue the START DBE statement or the SQLUtil ALTDBE command.
ARCHIVE MODE	Either ON or OFF, to indicate archive or nonarchive logging.

Display Conventions

The characters displayed in the horizontal bars have special meanings, as explained in the following table.

Legend

Symbol	Corresponding Number	Description
!	CUR	Represents activity that has occurred during the most recent refresh interval.
*	AVG	Designates the average amount of activity that has occurred since SQLMON has been attached to the DBEnvironment.
\$	AVG and CUR	Indicates that the average value and the value of the most recent refresh interval are the same.
Full bright inverse video	MAX	Designates the maximum amount of activity that has occurred since SQLMON has been attached to the DBEnvironment.

Related SET Commands

Use the SET REFRESH command to modify the refresh rate of the screen.

IO Data Program Screen

This screen provides I/O data buffer pool information for each program being run by sessions attached to the DBEnvironment.

REFRESH = 10		IO DATA PROGRAM				SESSIONS = 2	
SORTIODATA		(1)	(2)	(*3)	(4)		
PID	BUFF ACCESS	DATA DISK RD		DATA DISK WR	MISS RATE		
PROGRAM NAME = isql							
AVERAGE		50%		50%		0%	2%
18371	698	76%	11	79%	0	0%	2%
18381	220	24%	3	21%	0	0%	1%

To invoke the IO Data Program screen, use the `d p` command from the IO subsystem or the `/i d p` command from other subsystems.

When you invoke this screen, its counters are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTIODATA	An indicator of how the programs are sorted. The programs are sorted in descending order by the value in the column marked with the asterisk. In the example above, the programs are sorted by the DATA DISK WR values. For more information, see the SET SORTIODATA command.
PID	The HP-UX process identification number of the DBEnvironment session.
BUFF ACCESS	The number of page requests the session has issued to the data buffer pool.
DATA DISK RD	The number of pages that are not in the data buffer pool at request time. The operating system must fetch these pages, either from the operating system buffer pool or by a physical disk read.
DATA DISK WR	The number of dirty pages from the data buffer pool that have been logically written to disk.

MISS RATE The percentage of pages that are not in the data buffer pool, calculated as follows:

$$\text{MISS RATE} = (\text{DATA DISK RD} / \text{BUFF ACCESS}) * 100$$

PROGRAM NAME The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment.

All of the sessions running a program are listed beneath the PROGRAM NAME.

AVERAGE For a given column, the average amount of work performed by the processes in the list.

Each program has a list of processes beneath it. Each process has a line of information, which shows percentages in each column. The line shows the exact percentages of work each process performs, in each column. The AVERAGE field, however, shows the average amount of work each process performs.

For example, in the above screen, process 18371, which is running the program ISQL, is responsible for 76% of the BUFF ACCESS activity in the DBEnvironment. Process 18381 is responsible for 24% of the BUFF ACCESS activity. On average, each process is responsible for 50% of the BUFF ACCESS activity.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 10
SET SORTIODATA	Sorts the programs in descending order according to the value of the specified column.	SET SORTIODATA 1
SET TOP	Limits the number of programs displayed.	SET TOP 5

IO Data Session Screen

This screen provides data buffer pool I/O information for each session attached to the DBEnvironment.

REFRESH = 10		IO DATA SESSION				SESSIONS = 2	
SORTIODATA	(1)	(2)		(*3)	(4)		
PID	BUFF ACCESS	DATA DISK RD	DATA DISK WR	MISS RATE			
18217	246 98%	1 100%	38 100%	0%			
18270	4 2%	0 0%	0 0%	0%			

To invoke the IO Data Session screen, use the `d s` command from the IO subsystem, or use the `/i d s` command from other subsystems.

When you invoke this screen, its counters are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTIODATA	An indicator of how the sessions are sorted. The sessions are sorted in descending order by the value in the column indicated by the asterisk. In the above example, sessions are sorted by their DATA DISK WR values. For more information, see the SET SORTIODATA command.
PID	The HP-UX process identification number of the DBEnvironment session.
BUFF ACCESS	The number of page requests the session has issued to the data buffer pool.
DATA DISK RD	The number of pages that are not in the data buffer pool at request time. The operating system must fetch these pages, either from the operating system buffer pool or by a physical disk read.
DATA DISK WR	The number of dirty pages from the data buffer pool that were logically written to disk.

MISS RATE The percentage of pages that are not in the data buffer pool at request time. The percentage is calculated as follows:

$$\text{MISS RATE} = (\text{DATA DISK RD} / \text{BUFF ACCESS}) * 100$$

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SORTIODATA	Sorts sessions in descending order according to the value of the specified column.	SET SORTIODATA 2
SET TOP	Limits the number of sessions displayed.	SET TOP 10

IO Log Program Screen

This screen provides log buffer pool I/O information for each program being run by sessions attached to the DBEnvironment.

REFRESH = 30		IO LOG PROGRAM				SESSIONS = 3	
SORTIOLOG		(1)		(2)		(3)	
PID	LOG BUFF WR	LOG DISK RD		LOG DISK WR			
PROGRAM NAME = isql							
AVERAGE		50%		0%		46%	
18371	42208	97%	0	0%	12	92%	
18381	1068	2%	0	0%	0	0%	
PROGRAM NAME = pasex71.r							
AVERAGE		1%		0%		8%	
18556	328	1%	0	0%	1	8%	

To invoke the IO Log Program Screen, use the `l p` command from the IO subsystem or the `/i l p` command from other subsystems.

When you invoke this screen, its counters are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTIOLOG	An indicator of how the programs are sorted. In the above example, the programs are sorted by program name, because the SORTIOLOG variable is set to off. Otherwise, the programs are sorted by the value in the column indicated by the asterisk. For more information, see the SET SORTIOLOG command.
PID	The HP-UX process identification number of the DBEnvironment session.
LOG BUFF WR	The number of records written by the session to the log buffer pool.
LOG DISK RD	The number of log pages fetched from disk. Unless the system has performed a rollforward recovery, this value is zero. The operating system performs a physical read only if the log page is not found in the operating system buffer pool.
LOG DISK WR	The number of log pages written from the log buffer pool to disk. To ensure data integrity in case of a system crash, ALLBASE/SQL instructs the operating system to perform a physical write for each log buffer write.

IO Log Program Screen

- PROGRAM NAME** The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment. All of the sessions running a program are listed beneath the PROGRAM NAME.
- AVERAGE** For a given column, the average amount of work performed by the processes in the list.
- Each program has a list of processes beneath it. Each process has a line of information, which shows percentages in each column. The line shows the exact percentages of work each process performs, in each column. The AVERAGE field, however, shows the average amount of work each process performs.
- (For example, in the screen above, process 18371 performs 97% of the LOG BUFF WR activity that occurs in the DBEnvironment.)

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SORTIOLOG	Sorts programs in descending order according to the value of the specified column.	SET SORTIOLOG 2
SET TOP	Limits the number of programs displayed.	SET TOP 10

IO Log Session Screen

This screen provides log buffer pool I/O information for each session attached to the DBEnvironment.

REFRESH = 10		IO LOG SESSION				SESSIONS = 2	
SORTIOLOG		(1)		(2)		(3)	
PID	LOG BUFF WR		LOG DISK RD		LOG DISK WR		
18371	42208	94%	0	0%	15	88%	
18381	2556	6%	0	0%	2	12%	

To invoke the IO Log Session screen, use the `l s` command from the IO subsystem or the `/i l s` command from other subsystems.

When you invoke this screen, its counters are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTIOLOG	An indicator of how the sessions are sorted. In the above example, the sessions are sorted by PID, because the SORTIOLOG variable is set to OFF. Otherwise, the sessions are sorted by the value in the column indicated by the asterisk. For more information, see the description of the SET SORTIOLOG command.
PID	The HP-UX process identification number of the DBEnvironment session.
LOG BUFF WR	The number of records written by the session to the log buffer pool.
LOG DISK RD	The number of log pages fetched from disk. Unless the system has performed a rollforward recovery, this value is zero. The operating system reads a log page from disk only if it cannot find a log page in the operating system buffer pool.
LOG DISK WR	The number of log pages written from the log buffer pool to disk. To ensure data integrity, ALLBASE/SQL instructs the operating system to write to disk each time it writes to the log buffer.

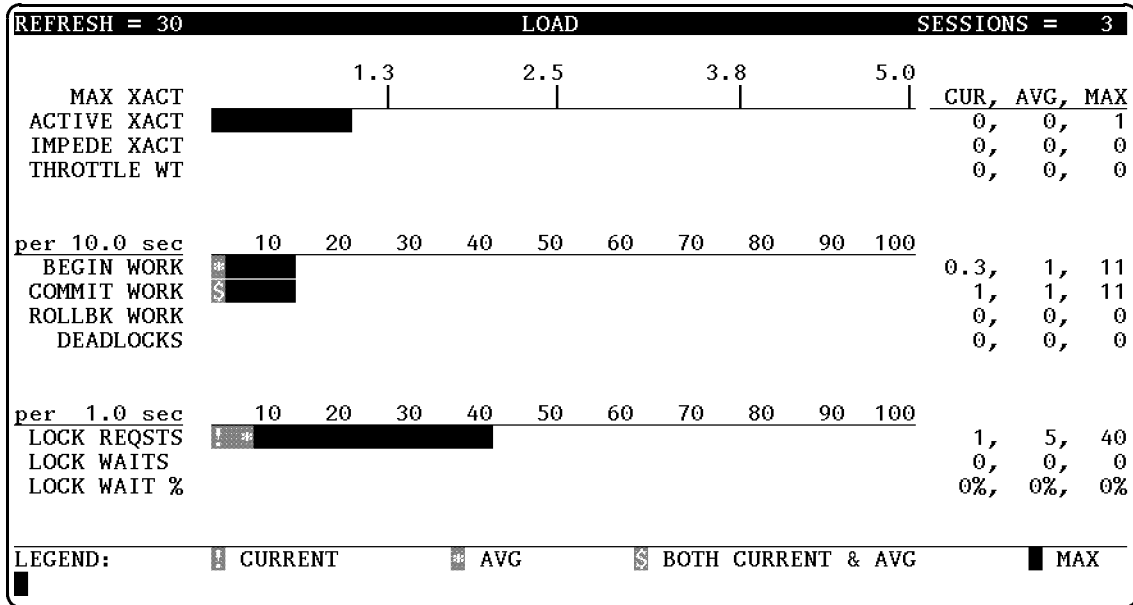
Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SORTIOLOG	Sorts sessions in descending order according to the value of the specified column.	SET SORTIOLOG 2
SET TOP	Limits the number of sessions displayed.	SET TOP 10

Load Screen

This screen provides information useful in measuring the transaction throughput efficiency of the DBEnvironment.



To invoke the Load screen, use the `l` command from the Load subsystem or the `/loa` command from other subsystems.

When you issue the `SET DBENVIRONMENT` command, the counters on this screen are set to zero. See the section “Display Conventions” on the next page for a description of the CUR, AVG, and MAX headings.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
MAX XACT	The maximum number of transactions. The maximum number that may be active at one time appears at the far right of the scale. The first three numbers on the scale are 1/4, 1/2, and 3/4 of the maximum.
ACTIVE XACT	The number of transactions that have started but have not terminated (that is, a <code>BEGIN WORK</code> statement has completed, but a <code>COMMIT WORK</code> statement has not yet begun).
IMPEDE XACT	The number of active transactions blocked because they are waiting for a lock owned by another session. Compare the <code>IMPEDE XACT</code> value with the <code>ACTIVE XACT</code> and <code>SESSIONS</code> values to determine the amount of lock contention.
THROTTLE WT	The number of sessions that are waiting for their <code>BEGIN WORK</code> statements to complete. These sessions are in the throttle wait queue.

per sec	The scale used to identify the length of the inverse video bars and the data in the CUR, AVG, and MAX columns. In the above example, the MAX frequency of LOCK REQSTS is 40 per 1.0 second.
BEGIN WORK	The number of BEGIN WORK commands being processed. This number includes both explicit BEGIN WORK commands that sessions issue and BEGIN WORK commands that ALLBASE/SQL generates.
COMMIT WORK	The number of COMMIT WORK commands being processed.
ROLLBK WORK	The number of ROLLBACK WORK commands being processed. This number includes both the ROLLBACK WORK commands the session issues and the ROLLBACK WORK commands ALLBASE/SQL issues against the session to resolve deadlocks.
DEADLOCKS	The number of deadlocks being encountered.
LOCK REQTS	The number of lock requests being processed.
LOCK WAITS	The number of lock requests postponed because another session held an incompatible lock.
LOCK WAIT %	The percentage of lock requests postponed because another session held an incompatible lock.

Display Conventions

Characters displayed in the horizontal bars have special meanings, as explained in the following table:

Legend

Symbol	Corresponding Number	Description
!	CUR	Represents activity that has occurred during the most recent refresh interval.
*	AVG	Designates the average amount of activity that has occurred since SQLMON has been attached to the DBEnvironment.
\$	AVG and CUR	Indicates that the average value is the same as the value of the most recent refresh interval.
Full bright inverse video	MAX	Designates the maximum amount of activity that has occurred since SQLMON has been attached to the DBEnvironment.

Related SET Commands

Use the SET REFRESH command to modify the refresh rate of the screen.

Load Program Screen

This screen provides transaction throughput information for each program being run by sessions attached to the DBEnvironment.

REFRESH = 30		LOAD PROGRAM				SESSIONS = 3				
SORTLOAD	(1)	(2)	(*3)	(4)						
PID	BEGIN WORK	COMMIT WORK	ROLLBK WORK	DEADLOCKS						
PROGRAM NAME = isql										
AVERAGE	49%	49%	0%	0%						
18371	32 94%	32 94%	0 0%	0 0%						
18381	1 3%	1 3%	0 0%	0 0%						
PROGRAM NAME = pasex71.r										
AVERAGE	3%	3%	0%	0%						
18556	1 3%	1 3%	0 0%	0 0%						

To invoke the Load Program screen, use the `p` command from the Load subsystem or the `/loa p` command from other subsystems.

When you invoke this screen, its counters are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTLOAD	An indicator of how the programs are sorted. The programs are sorted in descending order by the value in the column indicated by the asterisk. In the example above, the programs are sorted by ROLLBK WORK values. For more information, see the SET SORTLOAD command.
PID	The HP-UX process identification number of the DBEnvironment session.
BEGIN WORK	The number of BEGIN WORK commands the session has completed, including those that sessions issue explicitly and those that ALLBASE/SQL issues.
COMMIT WORK	The number of COMMIT WORK commands the session has completed.
ROLLBK WORK	The number of ROLLBACK WORK commands the session has completed. This number includes both the ROLLBACK WORK commands the session issued and the ROLLBACK WORK commands ALLBASE/SQL issued against the session to resolve deadlocks.
DEADLOCKS	The number of times the session was deadlocked and then rolled back.

Load Program Screen

PROGRAM NAME The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment. All of the sessions running a program are listed beneath PROGRAM NAME.

AVERAGE For a given column, the average amount of work performed by the processes in the list.

Each program has a list of processes beneath it. Each process has a line of information to the right of its ID. The line shows the exact percentages of work each process performs, in each column. The AVERAGE field, however, shows the average amount of work each process performs.

(For example, in the screen above, process 18371 performs 94% of the BEGIN WORK activity that occurs in the DBEnvironment.)

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SORTLOAD	Sorts programs in descending order according to the value of the specified column.	SET SORTLOAD 2
SET TOP	Limits the number of programs displayed.	SET TOP 5

Load Session Screen

This screen provides transaction throughput information for each session attached to the DBEnvironment.

REFRESH = 30		LOAD SESSION						SESSIONS = 3	
SORTLOAD		(1)		(2)		(*3)		(4)	
PID	BEGIN WORK	COMMIT WORK	ROLLBK WORK	DEADLOCKS					
18371	32	94%	32	94%	0	0%	0	0%	0%
18381	1	3%	1	3%	0	0%	0	0%	0%
18556	1	3%	1	3%	0	0%	0	0%	0%

To invoke the Load Session screen, use the `s` command from the Load subsystem or the `/load s` command from other subsystems.

When you invoke this screen, its counters are set to zero.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTLOAD	An indicator of how the sessions are sorted. The sessions are sorted in descending order by the value in the column marked with the asterisk. In the example above, sessions are sorted by their ROLLBK WORK values. For more information, see the SET SORTLOAD command.
PID	The HP-UX process identification number of the DBEnvironment session.
BEGIN WORK	The number of BEGIN WORK commands the session has completed, including those that that sessions issued explicitly and those that ALLBASE/SQL issued.
COMMIT WORK	The number of COMMIT WORK commands the session has completed.
ROLLBK WORK	The number of ROLLBACK WORK commands the session has completed. This number includes the ROLLBACK WORK commands the session issued and the ROLLBACK WORK commands ALLBASE/SQL issued against the session to resolve deadlocks.
DEADLOCKS	The number of times the session was deadlocked and then rolled back.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SORTLOAD	Sorts sessions in descending order according to the value of the specified column.	SET SORTLOAD 2
SET TOP	Limits the number of sessions displayed.	SET TOP 10

Lock Screen

This screen displays lock activity data for the entire DBEnvironment.

REFRESH = 10		LOCK	LOCKFILTER = SU/TPR/GWC/SXR _{sxr6v} /1	
G	OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	LOCK QUEUE	
T	DBCORE.MARSCH		ss	
T	DBCORE.MARSCS		sss	
T	DBCORE.MARSINDX		sss	
T	DBCORE.MARSREL		sss	
T	HPRDBSS.COLUMN		ss	
T	HPRDBSS.DBFILESET		ss	
T	HPRDBSS.INDEX		ss	
T	HPRDBSS.SECTION		s	
T	HPRDBSS.SPECAUTH		ss	
T	HPRDBSS.TABAUTH		ss	
T	HPRDBSS.TABLE		ss	
T	PURCHDB.PARTS		S	
T	PURCHDB.VENDORS		S	
T	STOREDSECT.SYSTEM		s	
P	DBCORE.MARSCH	0:0:2:0	ss	
P	DBCORE.MARSCS	0:0:3:0	sss	
P	DBCORE.MARSINDX	0:0:5:0	sss	
P	DBCORE.MARSREL	0:0:1:0	sss	
P	HPRDBSS.COLUMN	0:0:100:0	S	
P	HPRDBSS.COLUMN	0:0:103:0	S	

Continue? ([YES],NO) █

To invoke the Lock screen, use the **l** command from the Lock subsystem or the **/loc** command from other subsystems.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
LOCKFILTER	The current setting of the LOCKFILTER variable, which determines the type of lock information that appears on the screen. For more information, see the SET LOCKFILTER command.
G	The granularity of the lock, either table (T), page (P), or row (R).
OWNER.TABLE [/CONSTRAINT]	The name of the table or referential constraint that is locked.
PAGE/ROW ID	The identifier of the page or row that is locked. If the lock is a table lock, this field is blank.
LOCK QUEUE	A list of characters. Each character represents a lock that a DBEnvironment session has requested. If a character appears in inverse video, the session is waiting for a lock. If a character appears without inverse video, the lock has been granted. If a character is in inverse video and underlined, the session is converting the lock to a stronger mode.

The character indicates the mode of the lock, as listed below:

- S Share
- X Exclusive
- s Intent share
- x Intent exclusive
- 6 Share + intent exclusive
- R Recovery exclusive
- r Recovery intent exclusive
- v Recovery share + intent exclusive

When a session is waiting to acquire or convert a lock, the character indicates the lock mode the session is attempting to acquire.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET LOCKFILTER	Filters the lock information provided.	SET LOCKFILTER /U/TP/W/SXx/5
SET LOCKTABFILTER	Filters lock information by object name.	SET LOCKTABFILTER PurchDB.Orders

Lock Impede Screen

This screen identifies the locks granted to a particular session that are causing other sessions to wait.

```
REFRESH = 10                                LOCK IMPEDE
PID:18381          STATUS:Idle                XID:33794      ISO: RR
LOGIN NAME: DGREEN          LABEL:          PRI:127
PROGRAM NAME: isql
G | OWNER.TABLE[/CONSTRAINT]          PAGE/ROW ID | GWC | MOD | NEW | WAITING:
T | PURCHDB.VENDORS                    | G | S | x | PID: 18827
```

To invoke the Lock Impede screen, use the `i` command from the Lock subsystem or the `/loc i` command from other subsystems.

When you invoke the screen, you must specify the process identification number of a session. You can specify the process identification number on the command line, like this:

```
SQLMONITOR OVERVIEW => /loc i 18381
```

You can also specify the number in response to an SQLMON prompt, for example:

```
SQLMONITOR OVERVIEW => /loc i
```

```
PID: 18381
```

The information in the `NEW` and `WAITING` fields, which is displayed in inverse video, applies to the sessions that are waiting for locks. The other fields on the screen apply to the session identified by the field in the upper left corner.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
PID	The HP-UX process identification number of the DBEnvironment session you specified when you invoked this screen.

STATUS	The status of the DBCore call the session has made. This field is equivalent to the STATUS column of the SYSTEM.CALL pseudotable. The possible values are listed below: Running DBCore is processing a call from a session. Waiting for The session has made a DBCore call, but is waiting until <i>resource</i> it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT. Idle DBCore is not processing a call.
XID	The transaction identifier, equivalent to the XID column of the SYSTEM.TRANSACTION pseudotable.
ISO	The isolation level, equivalent to the ISOLATION LEVEL column of the SYSTEM.TRANSACTION pseudotable. See the “Concurrency Control through Locks and Isolation Levels” chapter in the <i>ALLBASE/SQL Reference Manual</i> for more information.
LOGIN NAME	The HP-UX login name for the DBEnvironment session.
LABEL	The transaction label, equivalent to the LABEL column of the SYSTEM.TRANSACTION pseudotable. To assign a label to a transaction, use a BEGIN WORK or a SET TRANSACTION statement.
PRI	The transaction priority, equivalent to the PRIORITY column of the SYSTEM.TRANSACTION pseudotable. The lowest transaction priority is 255, and the highest is 0.
PROGRAM NAME	The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment.
G	The granularity of the lock, either table (T), page (P), or row (R).
OWNER. TABLE [/CONSTRAINT]	The name of the table or referential constraint that is locked.
PAGE/ROW ID	The identifier of the page or row that is locked. If the lock is a table lock, this field is blank.
GWC	The status of the lock, always set to G to indicate that the lock has been granted to the session indicated by the PID field in the upper left corner of the screen.
MOD	The mode of lock that has been granted, as follows: S Share X Exclusive s Intent share x Intent exclusive 6 Share + intent exclusive R Recovery exclusive r Recovery intent exclusive v Recovery share + intent exclusive
NEW	The lock mode the waiting session is attempting to acquire. If the lock has been granted, this field is blank.

Lock Impede Screen

`WAITING` The HP-UX process identifier for the DBEnvironment session that is waiting because an incompatible lock has been granted to the session identified by the `PID` field in the upper left corner of the screen.

Related SET Commands

Use the `SET REFRESH` command to modify the refresh rate of the screen.

Lock Memory Screen

This screen lists the number of locks allocated to each session according to the lock granularity.

REFRESH = 10		LOCK MEMORY			SESSIONS = 3	
SORTLOCK	(1)	(2)	(3)	(4)	(*5)	
PID	TABLE	PAGE	ROW	TOTAL	MAXTOTAL	
18381	11	10	15	36	80	
18371	11	10	15	36	40	
18827	6	8	6	20	23	

To invoke the Lock Memory screen, use the `m` command from the Lock subsystem or the `/loc m` command from other subsystems.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
SORTLOCK	An indicator of how the sessions are sorted. The sessions are sorted in descending order by the value in the column indicated by the asterisk. In the example above, the sessions are sorted by MAXTOTAL values. For more information, see the description of the SET SORTLOCK command.
PID	The HP-UX process identification number of the DBEnvironment session.
TABLE	The number of table locks allocated to the session. This number includes locks that have been granted, locks the session is waiting to acquire, and locks the session is waiting to convert.
PAGE	The number of page locks allocated to the session. This number includes locks that have been granted, locks the session is waiting to acquire, and locks the session is waiting to convert.
ROW	The number of row locks allocated to the session. This number includes locks that have been granted, locks the session is waiting to acquire, and locks the session is waiting to convert.

Lock Memory Screen

TOTAL The total number of locks allocated to the session, calculated as follows:

$$\text{TOTAL} = \text{TABLE} + \text{PAGE} + \text{ROW}$$

MAXTOTAL The maximum number of **TOTAL** locks allocated to the session since it was attached to the DBEnvironment.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SORTLOCK	Sorts sessions in descending order according to the value of the specified column.	SET SORTLOCK 2
SET TOP	Limits the number of sessions displayed.	SET TOP 10

Lock Object Screen

This screen identifies the sessions in the lock queue for a particular table, page, or row.

REFRESH = 10				LOCK OBJECT		LOCKFILTER = U/TPR/GWC/SXRSXR6V/1			
GRN	OWNER	TABLE[/CONSTRAINT]	PAGE/ROW ID						
CWC	MOD	NEW	PID	LOGIN NAME	XID	ISO	PRI	LABEL	
				PROGRAM NAME					
T PURCHDB.PARTS									
G	6		22556	SUPPORT	72049	RR	127		
				isql					
W		S	22772	SUPPORT	72064	RR	127	transact1	
				isql					
P PURCHDB.PARTS									
					0:3:3:0				
G	X		22556	SUPPORT	72049	RR	127		
				isql					

To invoke the Lock Object screen, use the `o` command from the Lock subsystem or the `/loc o` command from other subsystems.

By default, the `LOCKOBJECT` variable is set to `ALL`. Therefore, by default, the `LOCK OBJECT` screen displays all the lock objects that qualify under the `LOCKFILTER` and `LOCKTABFILTER` variables.

If `LOCKOBJECT` is set to `OFF`, you must specify a lock or `ALL` when you invoke the Lock Object screen. For example, to view information about all the qualified locks, you can enter

```
SQLMONITOR OVERVIEW => /loc o ALL
```

To view information about a single lock on a table, you must provide the table name:

```
SQLMONITOR OVERVIEW => /loc o PurchDB.Parts
```

If the lock is on a page or row, you must provide the table name *and* the page or row id, separating the two fields with a slash:

```
SQLMONITOR OVERVIEW => /loc o PurchDB.Parts/0:1:5:28
```

```
SQLMONITOR OVERVIEW => /loc o PurchDB.Parts/0 1 5 28
```

As shown in the last two examples, you can use either spaces or colons to delimit the numbers in the page or row id.

Lock Object Screen

Field Definitions

REFRESH	The screen refresh rate, in seconds.
LOCKFILTER	If the lock object is ALL, the current setting of the LOCKFILTER variable. The LOCKFILTER variable determines the type of lock information that is provided. For more information, see the description of the SET LOCKFILTER command.
G	The granularity of the lock, either table (T), page (P), or row (R).
OWNER.TABLE [/CONSTRAINT]	The name of the table or referential constraint that is locked.
PAGE/ROW ID	The identifier of the page or row that is locked. If the lock object is a table, this field is blank.
GWC	The lock status, either granted (G), waiting (W), or converting (C) to a stronger mode.
MOD	The mode of lock that has been granted, as listed below: S Share X Exclusive s Intent share x Intent exclusive 6 Share + intent exclusive R Recovery exclusive r Recovery intent exclusive v Recovery share + intent exclusive If the session is waiting for a lock, this field is blank. If the session is converting a lock, this field displays the old value.
NEW	The lock mode the waiting session is attempting to acquire. If the lock has been granted, this field is blank.
PID	The HP-UX process identification number.
LOGIN NAME	The HP-UX login.
XID	The transaction identifier, equivalent to the XID column of the SYSTEM.TRANSACTION pseudotable.
ISO	The isolation level, equivalent to the ISOLATION LEVEL column of the SYSTEM.TRANSACTION pseudotable. See the “Concurrency Control through Locks and Isolation Levels” chapter in the <i>ALLBASE/SQL Reference Manual</i> for more information.
PRI	The transaction priority, equivalent to the PRIORITY column of the SYSTEM.TRANSACTION pseudotable. The lowest transaction priority is 255 and the highest is 0.
LABEL	The transaction label, equivalent to the LABEL column of the SYSTEM.TRANSACTION pseudotable. To assign a label to a transaction, use a BEGIN WORK or a SET TRANSACTION statement.

PROGRAM NAME The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment.

Display Conventions

Sessions waiting for locks appear on the screen in inverse video. If the session is converting the lock to a stronger mode, the fields are in inverse video and underlined. The absence of inverse video and underlining indicates that the lock has been granted.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET LOCKOBJECT	Specifies the lock object displayed. When set to ALL, the screen displays all lock objects that qualify under the LOCKFILTER and LOCKTABFILTER variables.	SET LOCKOBJECT PurchDB.Parts
SET LOCKFILTER	Filters the lock information provided.	SET LOCKFILTER /U/TP/W/SXx/5
SET LOCKTABFILTER	Filters lock information by object name.	SET LOCKTABFILTER PurchDB.Orders

Lock Session Screen

This screen displays lock activity data for a single session.

REFRESH = 10		LOCK SESSION		LOCKFILTER = SU/TPR/GWC/SXR _{sxr6v} /-			
PID:18827		STATUS:Waiting for LOCK		XID:33812		ISO: RR	
LOGIN NAME: DGREEN				LABEL:		PRI:127	
PROGRAM NAME: pasex71.r							
G	OWNER.TABLE[/CONSTRAINT]	PAGE/ROW ID	GWC	MOD	NEW	WAITS FOR:	
T	DBCORE.MARSCS		G	s			
T	DBCORE.MARSINDX		G	s			
T	DBCORE.MARSREL		G	s			
T	HPRDBSS.SECTION		G	s			
T	PURCHDB.VENDORS		C	s	x	PID: 18381	
T	STOREDSECT.SYSTEM		G	s			
P	DBCORE.MARSCS	0:0:3:0	G	s			
P	DBCORE.MARSINDX	0:0:5:0	G	s			
P	DBCORE.MARSREL	0:0:1:0	G	s			
P	HPRDBSS.SECTION	0:0:114:0	G	S			
P	PURCHDB.VENDORS	0:1:2:0	G	S			
P	STOREDSECT.SYSTEM	0:0:116:0	G	S			
P	STOREDSECT.SYSTEM	0:0:117:0	G	S			
P	STOREDSECT.SYSTEM	0:0:118:0	G	S			
R	DBCORE.MARSCS	0:0:3:0	G	S			
R	DBCORE.MARSINDX	0:0:5:2	G	S			
Continue? ([YES],NO) █							

To invoke the Lock Session screen, use the **s** command from the Lock subsystem or the **/loc s** command from other subsystems.

You must specify the session's process identification number when you invoke the screen. You can specify a process identification number on the command line or in response to an SQLMON prompt:

```
SQLMONITOR OVERVIEW => /loc s 18827
```

```
SQLMONITOR OVERVIEW => /loc s
```

```
PID: 18827
```

Field Definitions

REFRESH	The screen refresh rate, in seconds.
LOCKFILTER	The current setting of the LOCKFILTER variable, which determines the type of lock information that is provided. For more information, see the description of the SET LOCKFILTER command.
PID	The HP-UX process identification number of the DBEnvironment session.

STATUS	The status of the DBCore call the session has made. This field is equivalent to the STATUS column of the SYSTEM.CALL pseudotable. The possible values are listed below: Running DBCore is processing a call from a session. Waiting for The session has made a DBCore call, but is waiting until <i>resource</i> it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT. Idle DBCore is not processing a call.
XID	The transaction identifier, equivalent to the XID column of the SYSTEM.TRANSACTION pseudotable.
ISO	The isolation level, equivalent to the ISOLATION LEVEL column of the SYSTEM.TRANSACTION pseudotable. For more information, see the chapter “Concurrency Control through Locks and Isolation Levels” in the <i>ALLBASE/SQL Reference Manual</i> .
LOGIN NAME	The HP-UX login name of the DBEnvironment session.
LABEL	The transaction label, equivalent to the LABEL column of the SYSTEM.TRANSACTION pseudotable. To assign a label to a transaction, use either a BEGIN WORK or a SET TRANSACTION statement.
PRI	The transaction priority, equivalent to the PRIORITY column of the SYSTEM.TRANSACTION pseudotable. The lowest transaction priority is 255, and the highest is 0.
PROGRAM NAME	The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment.
G	The granularity of the lock, either table (T), page (P), or row (R).
OWNER.TABLE [/CONSTRAINT]	The name of the table or referential constraint that is locked.
PAGE/ROW ID	The identifier of the page or row that is locked. If the lock object is a table, this field is blank.
GWC	The status of the lock, either granted (G), waiting (W), or converting (C) to a stronger mode.
MOD	The mode of lock that has been granted, as listed below: S Share X Exclusive s Intent share x Intent exclusive 6 Share + intent exclusive R Recovery exclusive r Recovery intent exclusive v Recovery share + intent exclusive If the session is waiting for a lock, this field is blank. If the session is converting a lock, the old value is displayed.

Lock Session Screen

- NEW** The lock mode the waiting session is attempting to acquire, just as listed under MOD. If the lock has been granted, this field is blank.
- WAITS FOR** The process identifier of a session that has access to the lock that the current session is waiting for. Other sessions might also have access to the lock. For a complete list of the sessions in the lock queue, you can access the Lock Object screen.

Display Conventions

If the session appears in inverse video, it is waiting for a lock. If the session appears in inverse video and underlined, it is converting a lock to a stronger mode. If the session appears without inverse video or underlining, the lock has been granted.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET LOCKFILTER	Filters the lock information provided.	SET LOCKFILTER /U/TP/W/SXx/5
SET LOCKTABFILTER	Filters lock information by object name.	SET LOCKTABFILTER PurchDB.Orders

Lock TabSummary Screen

This screen displays summarized information about the locks a session holds. The locks are grouped by granularity, that is, by table locks, page locks, and row locks. You can display lock information for a single session or for all sessions connected to the DBEnvironment.

REFRESH = 10		LOCK TABSUMMARY	
PID:18827		STATUS:Waiting for LOCK	
LOGIN NAME: DGREEN		XID:33812	
PROGRAM NAME: pasex71.r		ISO: RR	
		LABEL:	
		PRI:127	
TOTAL LOCKS	G	OWNER.TABLE [/CONSTRAINT]	
1	T	DBCORE.MARSCS	
1	T	DBCORE.MARSINDX	
1	T	DBCORE.MARSREL	
1	T	HPRDBSS.SECTION	
1	T	PURCHDB.VENDORS	
1	T	STOREDSECT.SYSTEM	
3	P	STOREDSECT.SYSTEM	
1	P	DBCORE.MARSCS	
1	P	DBCORE.MARSINDX	
1	P	DBCORE.MARSREL	
1	P	HPRDBSS.SECTION	
1	P	PURCHDB.VENDORS	
3	R	DBCORE.MARSREL	
2	R	DBCORE.MARSINDX	
1	R	DBCORE.MARSCS	

To invoke the Lock screen, use the `t` command from the Lock subsystem or the `/loc t` command from other subsystems.

When you invoke the screen, you must specify either a process identification number or the keyword ALL, as shown in these examples:

```
SQLMONITOR OVERVIEW => /loc t all
```

```
SQLMONITOR LOCK => t 18827
```

The keyword ALL designates all sessions that are connected to the DBEnvironment.

You can also specify the process identification number in response to a prompt generated by SQLMON, for example:

```
SQLMONITOR OVERVIEW => /loc t
```

```
PID [or ALL]: 18827
```

Lock Tab Summary Screen

Field Definitions

REFRESH	The screen refresh rate, in seconds.						
PID	The HP-UX process identification number you specified when you invoked the screen. If you specified ALL when you invoked the screen, the value of PID is ALL.						
STATUS	<p>The status of the DBCore call the session has made. This field is equivalent to the STATUS column of the SYSTEM.CALL pseudotable. The possible values are listed below:</p> <table><tr><td>Running</td><td>DBCore is processing a call from a session.</td></tr><tr><td>Waiting for <i>resource</i></td><td>The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT.</td></tr><tr><td>Idle</td><td>DBCore is not processing a call.</td></tr></table> <p>If the value of PID is ALL, this field is not displayed.</p>	Running	DBCore is processing a call from a session.	Waiting for <i>resource</i>	The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT.	Idle	DBCore is not processing a call.
Running	DBCore is processing a call from a session.						
Waiting for <i>resource</i>	The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT.						
Idle	DBCore is not processing a call.						
XID	The transaction identifier, equivalent to the XID column of the SYSTEM.TRANSACTION pseudotable. If the value of PID is ALL, this field is not displayed.						
ISO	<p>The isolation level, equivalent to the ISOLATION LEVEL column of the SYSTEM.TRANSACTION pseudotable. If the value of PID is ALL, this field does not appear.</p> <p>For more information, see the chapter “Concurrency Control through Locks and Isolation Levels” in the <i>ALLBASE/SQL Reference Manual</i>.</p>						
LOGIN NAME	The HP-UX login name for the DBEnvironment session you specified when you invoked the screen. If the value of PID is ALL, this field is not displayed.						
LABEL	The transaction label, equivalent to the LABEL column of the SYSTEM.TRANSACTION pseudotable. To assign a label to a transaction, use either the BEGIN WORK or SET TRANSACTION statement. If the value of PID is ALL, this field is not displayed.						
PRI	The transaction priority, equivalent to the PRIORITY column of the SYSTEM.TRANSACTION pseudo-table. The lowest transaction priority is 255 and the highest is 0. If the value of PID is ALL, this field is not displayed.						
PROGRAM NAME	<p>The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment.</p> <p>If the value of PID is ALL, this field is not displayed.</p>						
TOTAL LOCKS	The total number of lock control blocks.						
G	The granularity of the lock control blocks, either table (T), page (P), or row (R).						

OWNER.TABLE The name of the table or referential constraint that is locked.
[/CONSTRAINT]

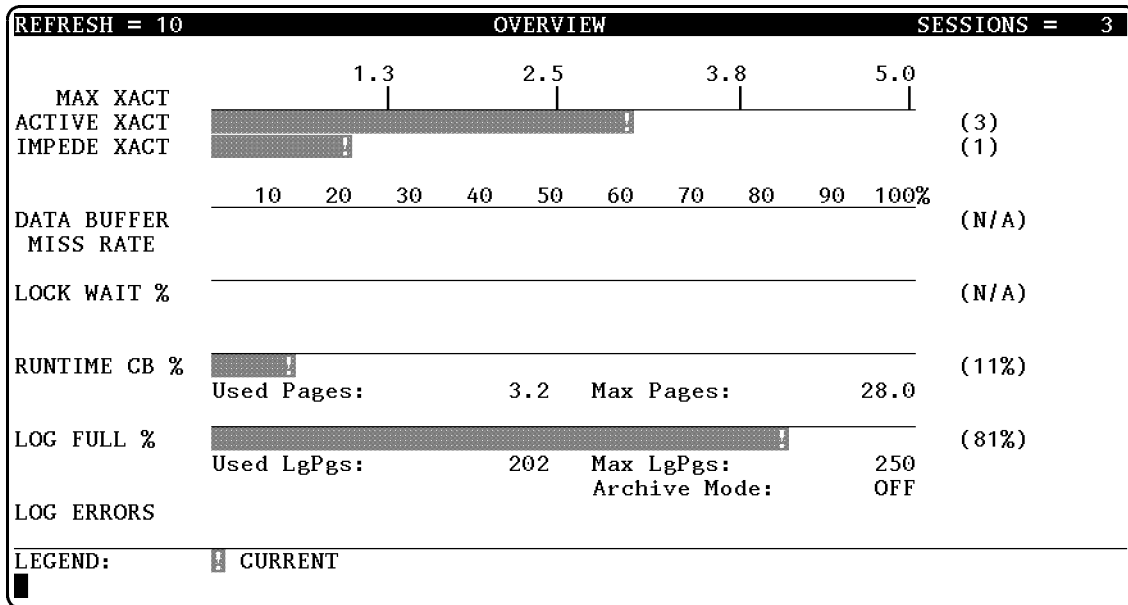
Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET TOP	Limits the number of objects displayed at each granularity level.	SET TOP 10

Overview Screen

This screen displays important aspects of the DBEnvironment's performance, such as the data buffer pool miss rate and the amount of available runtime control block space.



To invoke the Overview screen, enter the `o` command from the Overview subsystem or the `/o` command from other subsystems.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
MAX XACT	A scale showing the number of transactions that can be active at one time. The maximum appears at the far right of the scale, and the first three numbers on the scale are 1/4, 1/2, and 3/4 of the maximum.
ACTIVE XACT	The number of transactions that have started but have not terminated (that is, a <code>BEGIN WORK</code> statement has completed but a <code>COMMIT WORK</code> statement has not yet begun.)
IMPEDE XACT	The number of active transactions that are blocked because they are waiting for a lock owned by another session. To determine the amount of lock contention, compare the <code>IMPEDE XACT</code> value with the <code>ACTIVE XACT</code> and <code>SESSIONS</code> values.
DATA BUFFER MISS RATE	The percentage of pages that are not in the data buffer pool at request time. The operating system must fetch these pages, either from the operating system buffer pool or by a physical disk read.
LOCK WAIT %	The percentage of lock requests postponed because another session held an incompatible lock.

RUNTIME CB %	<p>The percentage of runtime control block space that is occupied, calculated as follows:</p> $\text{RUNTIME CB \%} = (\text{Used Pages} / \text{Max Pages}) * 100$ <p>Each runtime control block page holds 4096 bytes. Lock management is the single greatest user of runtime control block space.</p>
Used Pages	The number of runtime control block pages in use.
Max Pages	<p>The maximum number of runtime control block pages. The <i>ControlBlockPages</i> parameter that you specify in the START DBE, START DBE NEW, or START DBE NEWLOG statements is used to calculate MAX PAGES. MAX PAGES is usually smaller than <i>ControlBlockPages</i>, and if <i>ControlBlockPages</i> is too small, MAX PAGES is set to a certain minimum value.</p>
LOG FULL %	<p>The percentage of log file space that is occupied, as calculated by the following formula:</p> $\text{LOG FULL \%} = (\text{Used LgPgs} / \text{Max LgPgs}) * 100$ <p>A log page contains 512 bytes.</p>
Used LgPgs	The number of log pages that have been written.
Max LgPgs	The maximum number of log pages. To change this value, issue either the SQL START DBE or the SQLUtil ALTDBE command.
Archive Mode	Either ON or OFF, to indicate archive or nonarchive logging.
LOG ERRORS	<p>The entire LOG ERRORS bar is displayed in inverse video whenever an internal log counter is set to a nonzero value. Two internal log counters are used, one for each log file. With single logging, the log file is corrupt if a log error occurs. With dual logging, the logs are corrupt only if errors occur when writing to both log files. To prevent data loss, you should immediately store the DBEnvironment whenever a log error is encountered.</p>

Display Conventions

The numbers listed on the right side of the screen are also represented graphically with half-brightness inverse video bars. The exclamation point represents activity that has occurred during the most recent refresh interval.

Related SET Commands

Use the SET REFRESH command to modify the refresh rate of the screen.

Overview Program Screen

This screen displays session information for each program.

REFRESH = 10		OVERVIEW PROGRAM					SESSIONS = 3	
PID	LOGIN NAME	STATUS	XID	ISO	PRI	LABEL		
PROGRAM NAME = isql								
18371	DGREEN	Idle	33827	RU	127	GetParts		
18381	DGREEN	Idle	33794	RR	127			
PROGRAM NAME = pasex71.r								
18827	DGREEN	Wait	33812	RR	127			

To invoke the Overview Program screen, use the `p` command from the Overview subsystem or the `/o p` command from other subsystems.

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SESSIONS	The number of DBEnvironment sessions.
PID	The HP-UX process identification number of the DBEnvironment session.
LOGIN NAME	The HP-UX login name of the DBEnvironment session.
STATUS	The status of the DBCore call the session has made. This field is equivalent to the STATUS column of the SYSTEM.CALL pseudotable. The possible values are listed below: Running DBCore is processing a call from a session. Waiting for <i>resource</i> The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT. Idle DBCore is not processing a call.
XID	The transaction identifier, equivalent to the XID column of the SYSTEM.TRANSACTION pseudotable.
ISO	The isolation level, equivalent to the ISOLATION LEVEL column of the SYSTEM.TRANSACTION pseudotable. For more information, see the

chapter “Concurrency Control through Locks and Isolation Levels” in the *ALLBASE/SQL Reference Manual*.

PRI	The transaction priority, equivalent to the PRIORITY column of the SYSTEM.TRANSACTION pseudotable. The lowest transaction priority is 255, and the highest is 0.
LABEL	The transaction label, equivalent to the LABEL column of the SYSTEM.TRANSACTION pseudotable. To assign a label to a transaction, use a BEGIN WORK statement or a SET TRANSACTION statement.
PROGRAM NAME	The name of the program being run, usually the parent process of the process actually connected to the DBEnvironment. All of the sessions running a program are listed beneath the PROGRAM NAME.

Display Conventions

When a session is displayed in inverse video, it is waiting for a resource, probably a lock.

Related SET Commands

Use the SET REFRESH command to modify the refresh rate of the screen.

Overview Session Screen

This screen identifies all sessions connected to the DBEnvironment.

REFRESH = 10		OVERVIEW SESSION					SESSIONS = 3
PID	LOGIN NAME	STATUS	XID	ISO	PRI	LABEL	
18371	DGREEN	Idle	33827	RU	127	GetParts	
18381	DGREEN	Idle	33794	RR	127		
18827	DGREEN	Wait	33812	RR	127		

To invoke the Overview Session screen, use the `s` command from the Overview subsystem or the `/o s` command from other subsystems.

Field Definitions

REFRESH	The screen refresh rate, in seconds.						
SESSIONS	The number of DBEnvironment sessions.						
PID	The HP-UX process identification number for the DBEnvironment session.						
LOGIN NAME	The HP-UX login name for the DBEnvironment session.						
STATUS	The status of the DBCore call the session has made. This field is equivalent to the STATUS column of the SYSTEM.CALL pseudotable. The possible values are listed below: <table><tr><td>Running</td><td>DBCore is processing a call from a session.</td></tr><tr><td>Waiting for <i>resource</i></td><td>The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT.</td></tr><tr><td>Idle</td><td>DBCore is not processing a call.</td></tr></table>	Running	DBCore is processing a call from a session.	Waiting for <i>resource</i>	The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT.	Idle	DBCore is not processing a call.
Running	DBCore is processing a call from a session.						
Waiting for <i>resource</i>	The session has made a DBCore call, but is waiting until it can acquire a <i>resource</i> that is currently unavailable. The <i>resource</i> can be LOCK, LATCH, BUFFER, THROTTLE, or XACT.						
Idle	DBCore is not processing a call.						
XID	The transaction identifier, equivalent to the XID column of the SYSTEM.TRANSACTION pseudotable.						
ISO	The isolation level, equivalent to the ISOLATION LEVEL column of the SYSTEM.TRANSACTION pseudotable. See the chapter “Concurrency						

Control through Locks and Isolation Levels” in the *ALLBASE/SQL Reference Manual*.

PRI	The transaction priority, equivalent to the PRIORITY column of the SYSTEM.TRANSACTION pseudotable. The lowest transaction priority is 255, and the highest is 0.
LABEL	The transaction label, equivalent to the LABEL column of the SYSTEM.TRANSACTION pseudotable. To assign a label to a transaction, use a BEGIN WORK or a SET TRANSACTION statement.

Display Conventions

When a session is displayed in inverse video, the session is waiting for a resource, probably a lock.

Related SET Commands

Use the SET REFRESH command to modify the refresh rate of the screen.

SampleIO Screen

This screen displays DBEFile I/O information.

REFRESH = 10		SAMPLEIO		
DBEFILESET	DBEFILE	(1) SWAPIN	(2) SWAPOUT	(*3) TOTALIO
SYSTEM	PARTSDBE0	68	2	70
INVOICEFS	INVOICDATAF1	9	9	18
PURCHFS	PURCHDATAF1	2	0	2
WAREHFS	WAREHDATAF1	2	0	2
WAREHFS	WAREHINDEXF1	1	0	1

To invoke the SampleIO screen, use the `s` command from the SampleIO subsystem or the `/sa` command from other subsystems. For more information about using the SAMPLEIO subsystem, see the chapter “Troubleshooting with SQLMON.”

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SortSAMPLEIO	An indicator of how the DBEFiles are sorted. The DBEFiles are sorted in descending order by the value in the column indicated by the asterisk. In the example above, the DBEFiles are sorted by the TOTALIO values. For more information, see the description of the SET SortSAMPLEIO command.
DBEFILESET	The name of the DBEFileSet to which the DBEFile has been added.
DBEFILE	The name of the DBEFile belonging to the DBEFileSet.
SWAPIN	An approximation of read I/O for the DBEFile.
SWAPOUT	An approximation of write I/O for the DBEFile.
TOTALIO	The sum of the SWAPIN and SWAPOUT values. This value approximates the total I/O.

Display Conventions

When the DISPLAYSAMPLES variable is set to ON, a row displayed in inverse video indicates that DBEFile I/O has occurred during the most recent refresh interval.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET DISPLAYSAMPLES	Determines whether a SampleIO screen is displayed when samples are collected.	SET DISPLAYSAMPLES ON
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SAMPLING	Enables sampling of the data buffer pool.	SET SAMPLING ON
SET SORTSAMPLEIO	Sorts DBEFiles in descending order according to the value of the specified column.	SET SORTSAMPLEIO 3
SET TOP	Limits the number of DBEFiles displayed.	SET TOP 10

SampleIO Indexes Screen

This screen provides index and referential constraint I/O information.

REFRESH = 10		SAMPLEIO INDEXES		
DBEFILESET	INDEX, CONSTRAINT	(1) SWAPIN	(2) SWAPOUT	(*3) TOTALIO
PURCHFS				
PURCHDB.VENDORS	VENDORNUMINDEX	1	0	1
WAREHFS				
PURCHDB.PARTS	PARTNUMINDEX	1	0	1

To invoke the SampleIO Indexes screen, use the `i` command from the SampleIO subsystem or the `/sa i` command from other subsystems. For more information about using the SAMPLEIO subsystem, see the chapter “Troubleshooting with SQLMON.”

Field Definitions

REFRESH	The screen refresh rate, in seconds.
DBEFILESET	The name of the DBEFileSet. The indexes and referential constraints contained within a DBEFileSet are listed below the DBEFileSet name.
SORTSAMPLEIO	An indicator of how the indexes and referential constraints are sorted. The indexes and referential constraints are sorted in descending order by the value in the column indicated by the asterisk. In the above example, the indexes and referential constraints are sorted by TOTALIO values. For more information, see the description of the SET SORTSAMPLEIO command.
OWNER.TABLE	The name of the table upon which the index or referential constraint is defined.
INDEX, CONSTRAINT	The name of the index or referential constraint.
SWAPIN	An approximation of read I/O for the index or referential constraint.
SWAPOUT	An approximation of write I/O for the index or referential constraint.
TOTALIO	The sum of the SWAPIN and SWAPOUT values. This value approximates total I/O.

Display Conventions

When the DISPLAYSAMPLES variable is set to ON, a row displayed in inverse video indicates that index I/O has occurred during the most recent refresh interval.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET DBEFILESET	Limits the objects displayed to those in the DBEFileSet specified.	SET DBEFILESET PurchFS
SET DISPLAYSAMPLES	Determines whether a SampleIO screen is displayed when samples are collected.	SET DISPLAYSAMPLES ON
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SAMPLING	Enables sampling of the data buffer pool.	SET SAMPLING ON
SET SORTSAMPLEIO	Sorts objects in descending order according to the value of the specified column.	SET SORTSAMPLEIO 3
SET TOP	Limits the number of objects displayed.	SET TOP 2

SampleIO Objects Screen

This screen lists the database objects currently residing in the data buffer pool.

REFRESH = 10		SAMPLEIO OBJECTS	
DBEFILESET			
OWNER.TABLE[/INDEX,CONSTRAINT]	CURRENT PGS	TOTALIO	
INVOICEFS			
0:0:2:10 (PAGE TABLE PAGE)	1	???	
0:0:2:12 (PAGE TABLE PAGE)	1	???	
0:0:2:9 (PAGE TABLE PAGE)	1	???	
PURCHDB.CUSTOMER	8	11	
PURCHDB.CUSTOMER/ININDEX	1	2	
PURCHDB.INVOICE	17	20	
PURCHDB.SALESDATA	22	22	
PURCHFS			
0:0:2:1 (PAGE TABLE PAGE)	1	???	
0:0:2:2 (PAGE TABLE PAGE)	1	???	
PURCHDB.SUPPLYPRICE	1	1	
PURCHDB.SUPPLYPRICE/PARTONUMINDEX	1	1	
PURCHDB.SUPPLYPRICE/PARTTOVENDINDEX	1	1	
PURCHDB.SUPPLYPRICE/VENDPARTINDEX	1	1	
PURCHDB.VENDORS	1	1	
PURCHDB.VENDORS/VENDORNUMINDEX	1	1	
Continue? ([YES],NO) █			

To invoke the SampleIO Objects screen, use the `o` command from the SampleIO subsystem or the `/sa o` command from other subsystems. For more information about the SampleIO subsystem, see the chapter “Troubleshooting with SQLMON.”

Field Definitions

REFRESH	The screen refresh rate, in seconds.
DBEFILESET	The name of the DBEFileSet. The database objects within the DBEFileSet are listed below the DBEFileSet name.
OWNER.TABLE [/INDEX, CONSTRAINT]	The name of the database object residing in the data buffer pool. A database object can be a table, index, referential constraint, or page table page. Because temporary tables and page table pages do not have names, their tuple identifiers are listed instead. Long column data is stored in internal ALLBASE/SQL tables which have no entries in SYSTEM.TABLE. To construct a name for long column tables, SQLMON appends the column number to the table name.
CURRENT PGS	The number of pages in the data buffer pool that the database object currently occupies.
TOTALIO	An approximation of total I/O for the object since the SET DBENVIRONMENT command was issued. The characters ??? appear when no I/O data is available for the object.

Related SET Commands

The SET commands in the following table affect this screen:

Command	Description	Example
SET DBEFILESET	Limits the objects displayed to those contained in the DBEFileSet specified.	SET DBEFILESET PurchFS
SET DISPLAYSAMPLES	Determines whether a SampleIO screen is displayed when samples are collected.	SET DISPLAYSAMPLES ON
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SAMPLING	Enables sampling of the data buffer pool.	SET SAMPLING ON

SampleIO TabIndex Screen

This screen displays I/O information about a specific table, its indexes, and its referential constraints.

REFRESH = 10	SAMPLEIO TABINDEX			
OWNER . TABLE	INDEX, CONSTRAINT	(1) SWAPIN	(2) SWAPOUT	(*3) TOTAL IO
PURCHDB . VENDORS		1	1	2
	VENDORNUMINDEX	1	1	2

To invoke the SampleIO TabIndex screen, use the `tabi` command from the SampleIO subsystem or the `/sa tabi` command from other subsystems.

You must specify a table name when invoking the SampleIO TabIndex screen, either on the command line or at an SQLMON prompt:

```
SQLMONITOR OVERVIEW => /sa tabi PurchDB.Vendors
```

```
SQLMONITOR OVERVIEW => /sa tabi
```

```
OWNER . TABLE: PurchDB.Vendors
```

Field Definitions

REFRESH	The screen refresh rate, in seconds.
SORTSAMPLEIO	An indicator of how the indexes and referential constraints are sorted. The indexes and referential constraints are sorted in descending order by the value in the column indicated by the asterisk. In the above example, the indexes and referential constraints are sorted by TOTALIO values. For more information, see the SET SORTSAMPLEIO command.
OWNER . TABLE	The name of the table.
INDEX, CONSTRAINT	The index or referential constraint name.

SWAPIN	An approximation of read I/O for the object since the SET DBENVIRONMENT command was issued.
SWAPOUT	An approximation of write I/O for the object since the SET DBENVIRONMENT command was issued.
TOTALIO	The sum of the SWAPIN and SWAPOUT values. This value approximates total I/O.

Display Conventions

When the DISPLAYSAMPLES variable is set to ON, a value displayed in inverse video indicates that I/O on the database object has occurred during the most recent refresh interval.

Related SET Commands

The SET commands in the following table affect this screen:

Command	Description	Example
SET DISPLAYSAMPLES	Determines whether a SampleIO screen is displayed when samples are collected.	SET DISPLAYSAMPLES ON
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SAMPLING	Enables sampling of the data buffer pool.	SET SAMPLING ON
SET SORTSAMPLEIO	Sorts objects in descending order according to the value of the specified column.	SET SORTSAMPLEIO 2
SET TOP	Limits the number of objects displayed.	SET TOP 4

SampleIO Tables Screen

This screen provides table I/O information.

REFRESH = 10		SAMPLEIO TABLES		
DBEFILESET	SORTSAMPLEIO	(1)	(2)	(*3)
OWNER. TABLE		SWAPIN	SWAPOUT	TOTALIO
FILEFS				
PURCHDB.REPORTS(3)		1	0	1
INVOICEFS				
1:0:1:70		8	0	8
PURCHF5				
PURCHDB.SUPPLYPRICE		1	1	2
PURCHDB.VENDORS		1	1	2
SYSTEM				
STOREDSECT.SYSTEM		4	0	4
UNKNOWN				
2:0:1:70		12	2	14
WAREHFS				
PURCHDB.PARTS		1	0	1

To invoke the SampleIO Tables screen, use the `tabl` command from the SampleIO subsystem or the `/sa tabl` command from other subsystems. For more information about using the SampleIO subsystem, see the chapter “Troubleshooting with SQLMON.”

Field Definitions

REFRESH	The screen refresh rate, in seconds.
DBEFILESET	The name of the DBEFileSet. The tables contained within the DBEFileSet are listed below the DBEFileSet name.
SORTSAMPLEIO	An indicator of how the tables are sorted. The tables are sorted in descending order by the value in the column indicated by the asterisk. In the example above, the tables are sorted by the TOTALIO values. For more information, see the description of the SET SORTSAMPLEIO command.
OWNER.TABLE	The table name. Long column data is stored in internal ALLBASE/SQL tables which have no entries in SYSTEM.TABLE. To construct a name for these tables, SQLMON appends the column number of the long column to the table name. For example, the PURCHDB.REPORTS(3) entry represents the table that holds the long column data of the third column of the PURCHDB.REPORTS table.
SWAPIN	An approximation of read I/O for the table since the SET DBENVIRONMENT command was issued.
SWAPOUT	An approximation of write I/O for the table since the SET DBENVIRONMENT command was issued.

TOTALIO The sum of the SWAPIN and SWAPOUT values. This value approximates the total I/O.

Display Conventions

When the DISPLAYSAMPLES variable is set to ON, a value displayed in inverse video indicates that table I/O has occurred during the most recent refresh interval.

Related SET Commands

The SET commands in the following table affect this screen.

Command	Description	Example
SET DBEFILESET	Limits the tables displayed to those contained in the DBEFileSet specified.	SET DBEFILESET PurchFS
SET DISPLAYSAMPLES	Determines whether a SampleIO screen is displayed when samples are collected.	SET DISPLAYSAMPLES ON
SET REFRESH	Controls the refresh rate of the screen.	SET REFRESH 5
SET SAMPLING	Enables sampling of the data buffer pool.	SET SAMPLING ON
SET SORTSAMPLEIO	Sorts tables in descending order according to the value of the specified column.	SET SORTSAMPLEIO 3
SET TOP	Limits the number of tables displayed.	SET TOP 10

Static Screen

This screen lists information about indexes, referential constraints, and hash structures for each table contained in a DBEFileSet.

PartsDBE		STATIC			
DBEFILESET	HASH?	IMAGE?	NUMIDX	TYPE	OWNER.TABLE
FILEFS					
			0	PUBLIC	STOREDSECT.FILEFS
			0	PUBLIC	PURCHDB.REPORTS(3)
INVOICE3FS					
			0	PUBLIC	STOREDSECT.INVOICE3FS
INVOICEFS					
			0	PUBLIC	STOREDSECT.INVOICEFS
			1	PUBLIC	PURCHDB.CUSTOMER
			0	PUBLICROW	PURCHDB.INVOICE
*			0	PUBLIC	PURCHDB.SALESDATA
ORDERFS					
			0	PUBLIC	STOREDSECT.ORDERFS
			1	PUBLIC	PURCHDB.ORDERITEMS
			2	PUBLIC	PURCHDB.ORDERS
			0	PUBLIC	PURCHDB.REPORTS

Continue? ([YES],NO) █

To invoke the Static screen, use the `st` command from the Static subsystem or the `/st` command from other subsystems.

Field Definitions

DBEFILESET	The name of the DBEFileSet.
HASH?	An asterisk in this field indicates a hashed table. A table is hashed if the <code>UNIQUE HASH ON</code> or <code>HASH ON CONSTRAINT</code> clauses are included in the <code>CREATE TABLE</code> statement.
IMAGE?	An asterisk in this field indicates that the table is stored in a TurboIMAGE data set instead of in ALLBASE/SQL DBEFileSets.
NUMIDX	The number of indexes and referential constraints defined on the table.
TYPE	The table type, as defined by the <code>CREATE TABLE</code> statement. You can use the <code>ALTER TABLE</code> statement to change the table type.
OWNER.TABLE	The names of the tables contained within the DBEFileSet. Long column data is stored in internal ALLBASE/SQL tables which have no entries in <code>SYSTEM.TABLE</code> . To construct a name for these tables, SQLMON appends the column number of the long column to the table name. For example, the <code>PURCHDB.REPORTS(3)</code> entry represents the table that holds the long column data of the third column of the <code>PURCHDB.REPORTS</code> table.

Static Cluster Screen

This screen provides information about the clustering of indexes and referential constraints in a DBEFileSet.

PartsDBE		STATIC CLUSTER			
DBEFILESET	OWNER. TABLE	UNLOAD/LOAD SUGGESTD	%	TABLE PAGES	TOTAL ROWS CCOUNT
(CONST)	EVENTS_FK	?	7%		2
	RECDB.MEMBERS			1	37
(CONST)	MEMBERS_FK		0%		1
(INDEX)	MEMBERS_PK		0%		1
SYSTEM					
WAREHFS					
	MANUFDB.SUPPLYBATCHES			1	10
(INDEX)	SQLCON_00000003300		0%		1
	MANUFDB.TESTDATA			1	13
(CONST)	SQLCON_00000003301	?	8%		2
	PURCHDB.INVENTORY			1	22
(INDEX)	INVPARTNUMINDEX		0%		1
	PURCHDB.PARTS			1	22
(INDEX)	PARTNUMINDEX		0%		1

Continue? ([YES],NO) █

To invoke the Static Cluster screen, use the `c` command from the Static subsystem or the `/st c` command from other subsystems.

Field Definitions

DBEFILESET	The name of the DBEFileSet.
	The keyword <code>DETACHED</code> means that the DBEFileSet is detached. The keywords <code>DETACHED DBEFILES</code> mean that the DBEFileSet is attached, but some of the DBEFiles within it are detached.
OWNER. TABLE	The name of each table in the DBEFileSet that has an index or referential constraint defined upon it. Indexes and referential constraints are listed below the table name, and are preceded by the following labels:
(INDEX)	A nonclustering index, created by using the <code>CREATE INDEX</code> statement or by specifying a unique constraint.
(INDEX*)	A clustering index, created by using the <code>CREATE CLUSTERING INDEX</code> statement or by specifying a unique constraint with the <code>CLUSTERING ON CONSTRAINT</code> clause in the <code>CREATE TABLE</code> statement.
(CONST)	A nonclustering PCR (parent-child relationship), created by defining a referential constraint.

Static Cluster Screen

(CONST*)	A clustering PCR (parent-child relationship), created by defining a referential constraint with the CLUSTERING ON CONSTRAINT clause in the CREATE TABLE statement.
TABLE PAGES	The number of pages containing data for the table.
TOTAL ROWS	The total number of rows in the table.
UNLOAD/LOAD SUGGESTED %	The percentage of rows in the table that are not physically stored in index order, calculated as follows: $\left(\frac{\text{CCOUNT} - \text{TABLE PAGES}}{\text{TOTAL ROWS} - \text{TABLE PAGES}} \right) * 100$ When the index is well clustered, CCOUNT equals TABLE PAGES, and the percentage is equal to zero. As the clustering of the index degrades, CCOUNT approaches TOTAL ROWS, and the percentage increases. Performance is best when CCOUNT is equal to TABLE PAGES. As CCOUNT approaches TOTAL ROWS, performance degrades. When the percentage is high, additional I/O is required to perform an index scan over the table. To improve performance, UNLOAD the data in sorted order and LOAD it back in again with ISQL.
CCOUNT	The total number of pages touched when a complete scan of the table is made in index order. CCOUNT is incremented each time the next row to be accessed is on a different page than the previous row. If the data is not physically stored in index order, the same page is touched more than once.

Display Conventions

The full bright, inverse video bars represent a value of 100%. The actual percentage is indicated by the half bright, inverse video bars, which are delimited on the right with exclamation points.

Related SET Commands

Use the SET DBEFILESET command to improve performance and display only those tables contained in a particular DBEFileSet.

Static DBEFile Screen

This screen lists the file capacity of each DBEFile in a DBEFileSet.

PartsDBE		STATIC DBEFILE				
DBEFILESET	DBEFILE TYP BD	DBEFILESET	FULLNESS	%	FSUSED PAGES	FSMAX PAGES
		DBEFILE	FULLNESS	%	USED PAGES	MAX PAGES
FILEFS				4%	2	50
	FILEDATA TBL			4%	2	50
INVOICE3FS				5%	1	22
	INVOICE3DATAF1 MIX			5%	1	22
INVOICEFS				84%	47	56
	INVOICEDATAF1 MIX			92%	11	12
	INVOICEDATAF2 TBL			64%	14	22
	INVOICEDATAF4 MIX *			100%	22	22
ORDERFS						100
	ORDERDATAF1 TBL			DETACHED		50
	ORDERINDEXF1 IDX			DETACHED		50
PURCHF5				8%	8	100
	PURCHDATAF1 TBL			6%	3	50
	PURCHINDEXF1 IDX			10%	5	50

Continue? ([YES],NO) █

To invoke the Static DBEFile screen, use the `d` command from the Static subsystem or the `/st d` command from other subsystems.

Field Definitions

DBEFILESET The name of the DBEFileSet.

The keyword `DETACHED` means that the DBEFileSet is detached. The keywords `DETACHED DBEFILES` mean that the DBEFileSet is attached, but some of the DBEFiles within it are detached.

DBEFILESET FULLNESS % The percentage of file space in use in the DBEFileSet, calculated as follows:

$$\text{DBEFILESET FULLNESS \%} = (\text{FSUSED PAGES} / \text{FSMAX PAGES}) * 100$$

FSUSED PAGES The number of pages in use in the DBEFileSet. This value is equivalent to the value of the `DBEFSUPAGES` column of `SYSTEM.DBEFILESET` plus the number of page table pages in the DBEFileSet. A used page might not be full.

FSMAX PAGES The maximum number of pages in the DBEFileSet. This value is equal to the `DBEFSMPAGES` column of `SYSTEM.DBEFILESET`.

DBEFILE The name of the DBEFile contained in the DBEFileSet. If the DBEFile has been detached with the `SQLUtil DETACHFILE` command, the keyword `DETACHED` appears.

Static DBEFile Screen

TYP	The type of the DBEFile, as listed below: TBL Table data pages, including hash structures and long column data IDX Index or referential constraint pages MIX Mixed, indicating either table data, index, or referential constraint pages
BD	Whether a DBEFile is bound, indicated by an asterisk. When a table is defined as hashed, a group of primary pages from up to 16 DBEFiles is allocated for it. The DBEFiles are bound, that is, they are unavailable for any other table, index, or referential constraint. You can only insert data for the hashed table itself into these DBEFiles. The last bound DBEFile might have unused space, which ALLBASE/SQL can use for overflow pages for the hash structure.
DBEFILE FULLNESS %	The percentage of file space in use in the DBEFile, calculated as follows: $\text{DBEFILE FULLNESS \%} = (\text{USED PAGES} / \text{MAX PAGES}) * 100$
USED PAGES	The number of pages in use in the DBEFile. This value is equivalent to the DBEFUPAGES column of SYSTEM.DBEFILE plus the number of page table pages in the DBEFile.
MAX PAGES	The maximum number of pages in the DBEFile. This value is equivalent to the DBEFMPAGES column of SYSTEM.DBEFILE and is specified with the CREATE DBEFILE statement.

Display Conventions

The full bright, inverse video bars represent the total amount of file space. The quantity of space occupied is indicated by the half bright, inverse video bars, which are delimited on the right with exclamation points.

Related SET Commands

Use the SET DBEFILESET command to improve performance and display only those tables contained in a particular DBEFileSet.

Static Hash Screen

This screen provides information about the primary and overflow pages of hashed tables.

PartsDBE		STATIC HASH						
DBEFILESET	OWNER. TABLE	PRIMPAGES	PRIMDATA	PRIMOVERF	OVERPAGES	OVERFLOW	CHAIN LNPTH	MAXOVERFLOW
						12345678901234567890		AVGOVERFLOW
INVOICEFS								
PURCHDB.SALES	DATA	11	11	11	11			1
								1

To invoke the Static Hash screen, use the `h` command from the Static subsystem or the `/st h` command from other subsystems.

Field Definitions

DBEFILESET	The name of the DBEFileSet. The keyword <code>DETACHED</code> means that the DBEFileSet is detached. The keywords <code>DETACHED DBEFILES</code> mean that the DBEFileSet is attached, but some of the DBEFiles within it are detached.
OWNER. TABLE	The name of each hashed table in the DBEFileSet.
PRIMPAGES	The number of primary pages allocated for the hash structure. This value is equivalent to the <code>PRIMPAGES</code> column of <code>SYSTEM.HASH</code> and is specified by the <code>UNIQUE HASH</code> clause of the <code>CREATE TABLE</code> statement.
PRIMDATA	The number of primary pages that currently contain table data. This value is equivalent to the <code>NPAGES</code> column of <code>SYSTEM.HASH</code> .
PRIMOVERF	The number of primary pages containing table data that have overflow pages. This value is equivalent to the <code>NOVERFLOW</code> column of <code>SYSTEM.HASH</code> .
OVERPAGES	The number of overflow pages allocated for the hash structure.

Static Hash Screen

OVERFLOW CHAIN LNTH	The maximum and average lengths of the overflow chains, displayed graphically. The maximum length is represented by full bright, inverse video bars. The average length is indicated by half bright, inverse video bars and is delimited on the right with an asterisk.
MAXOVERFLOW	The length of the longest overflow chain in the table. This value is one less than the MAXLEN column of the SYSTEM.HASH pseudotable, because MAXLEN includes the primary page in the chain length.
AVGOVERFLOW	The average length of the overflow chains, calculated as follows: $\text{AVGOVERFLOW} = \text{round} (\text{OVERPAGES} / \text{PRIMOVERF})$

Related SET Commands

Use the SET DBEFILESET command to improve performance and to display only those tables contained in a particular DBEFileSet.

Static Indirect Screen

This screen displays information about the indirect rows in each table of a DBEFileSet.

PartsDBE		STATIC INDIRECT			
DBEFILESET	OWNER.TABLE	TABLE	INDIRECT	ROW %	TOTAL ROWS
FILEFS					
	STOREDSECT.FILEFS			0%	0
	PURCHDB.REPORTS(3)			0%	1
INVOICE3FS					
	STOREDSECT.INVOICE3FS			0%	0
INVOICEFS					
	STOREDSECT.INVOICEFS			0%	0
	PURCHDB.CUSTOMER			0%	20
	PURCHDB.INVOICE			0%	5005
	PURCHDB.SALESDATA			0%	3000
ORDERFS		DETACHED			
PURCHFS					
	STOREDSECT.PURCHFS			0%	0
	PURCHDB.SUPPLYPRICE			0%	69
	PURCHDB.VENDORS			0%	13
Continue? ([YES],NO) █					

To invoke the Static Indirect screen, use the `i` command from the Static subsystem or the `/s i` command from other subsystems.

Field Definitions

DBEFILESET	The name of the DBEFileSet.
	The keyword <code>DETACHED</code> means that the DBEFileSet is detached. The keywords <code>DETACHED DBEFILES</code> mean that the DBEFileSet is attached, but some of the DBEFiles within it are detached.
OWNER.TABLE	The name of each table in the DBEFileSet. Long column data is stored in internal ALLBASE/SQL tables that have no entries in <code>SYSTEM.TABLE</code> . To construct a name for these tables, SQLMON appends the column number of the long column to the table name. For example, the <code>PURCHDB.REPORTS(3)</code> entry represents the table that holds the long column data of the third column of the <code>PURCHDB.REPORTS</code> table.
TABLE INDIRECT ROW %	The percentage of indirect rows within the table. To access an indirect row, ALLBASE/SQL must perform two page fetches. The first fetch obtains the row's address, and the second fetch acquires the row itself. Indirect rows degrade performance by increasing the amount of I/O.
TOTAL ROWS	The total number of rows in the table. This value is equivalent to the <code>NROWS</code> column of <code>SYSTEM.TABLE</code> .

Static Indirect Screen

Related SET Commands

Use the SET DBEFILESET command to improve performance and to display only those tables contained in a particular DBEFileSet.

Static Size Screen

This screen provides information about the size of tables, indexes, and referential constraints in a DBEFileSet.

PartsDBE		STATIC SIZE		
DBEFILESET	DBEFILESET	FULLNESS	%	FSUSED PAGES
OWNER, TABLE	TABLE	TABLE	PAGES INDEX PAGES	TOTAL PAGES
FILEFS			4%	2
PAGE TABLE PAGES				1
STOREDSECT. FILEFS			0	0
PURCHDB.REPORTS(3)			1	1
INVOICE3FS			5%	1
PAGE TABLE PAGES				1
STOREDSECT. INVOICE3FS			0	0
INVOICEFS			84%	47
PAGE TABLE PAGES				3
STOREDSECT. INVOICEFS			0	0
PURCHDB.CUSTOMER			1	2
(INDEX) INVINDEX				1
PURCHDB.INVOICE			20	20
PURCHDB.SALESDATA			22	22
ORDERFS			DETACHED	

Continue? ([YES],NO) █

To invoke the Static Size screen, use the `s` command from the Static subsystem or the `/st s` command from other subsystems.

Field Definitions

DBEFILESET The name of the DBEFileSet.

The keyword `DETACHED` means that the DBEFileSet is detached. The keywords `DETACHED DBEFILES` mean that the DBEFileSet is attached, but one or more of the DBEFiles within it are detached.

TEMPORARY PAGES A label that appears just after the DBEFileSet name and indicates that the DBEFileSet has temporary pages. Temporary pages are usually allocated from the `SYSTEM` DBEFileSet, and are used primarily for sorting table data.

PAGE TABLE PAGES The number of page table pages in the DBEFileSet. A page table page is an internal `ALLBASE/SQL` directory that holds information on up to 252 pages in a DBEFile. Every DBEFile contains at least one page table page.

DBEFILESET FULLNESS % The percentage of file space in use in the DBEFileSet, calculated as follows:

$$\text{DBEFILESET FULLNESS \%} = (\text{FSUSED PAGES} / \text{FSMAX PAGES}) * 100$$

`FSMAX PAGES` is the maximum number of pages in the DBEFileSet.

Static Size Screen

FSUSED PAGES	The number of pages in use in the DBEFileSet. This value includes all pages used for table data, indexes, referential constraints, page table pages, and temporary pages.
OWNER.TABLE	The names of the tables contained within the DBEFileSet. Indexes and referential constraints are listed below the table name, and are preceded by the following labels: (INDEX) A nonclustering index, created with the CREATE INDEX statement or by specifying a unique constraint. (INDEX*) A clustering index, created with the CREATE CLUSTERING INDEX statement or by specifying a unique constraint with the CLUSTERING ON CONSTRAINT clause in the CREATE TABLE statement. (CONST) A nonclustering PCR (parent-child relationship), created by defining a referential constraint. (CONST*) A clustering PCR (parent-child relationship), created by defining a referential constraint with the CLUSTERING ON CONSTRAINT clause in the CREATE TABLE statement. ALLBASE/SQL stores long column data in internal tables which have no entries in SYSTEM.TABLE. To construct a name for these tables, SQLMON appends the column number of the long column to the table name. For example, the PURCHDB.REPORTS(3) entry represents the table that holds the long column data of the third column of the PURCHDB.REPORTS table.
TABLE PAGES	The number of pages allocated for the table.
INDEX PAGES	The number of pages allocated for the index or referential constraint.
TOTAL PAGES	The total number of pages allocated for the table and all indexes and referential constraints defined upon it.

Display Conventions

The full bright, inverse video bars represent the total amount of file space in the DBEFileSet. The quantity of space occupied is indicated by the half bright, inverse video bars, which are delimited on the right with exclamation points.

Related SET Commands

Use the SET DBEFILESET command to improve performance and to display only those tables contained in a particular DBEFileSet.

SQLMON Command Reference

This chapter describes the SQLMON commands and gives syntax and examples for each. Within this chapter, the commands are arranged alphabetically. You will find a table summarizing the commands at the end of the chapter.

You can execute these commands from an SQLMON subsystem prompt, but not from within the help facility or from an SQLMON screen. In general, the commands are not case sensitive; however, some of them contain case sensitive parameters.

EXIT

Leaves SQLMON

Scope

SQLMON Only

SQLMON Syntax

E[XIT]

Description

- This command is equivalent to the QUIT command.

Example

```
SQLMONITOR OVERVIEW => EXIT
```

HELP

Invokes the SQLMON online help facility

Scope

SQLMON Only

SQLMON Syntax

$$\left\{ \begin{array}{l} \text{H[ELP]} \\ ? \end{array} \right\} \left[\begin{array}{l} \left[\begin{array}{l} \text{ScreenName} \left[\begin{array}{l} \text{CONTROL} \\ \text{INFO} \\ \text{SUBSYSTEM} \\ \text{TUNE [Number]} \end{array} \right] \\ \text{MAIN} \\ \text{SetCommand} \end{array} \right] \end{array} \right]$$

Parameters

<i>ScreenName</i>	Displays help text for <i>ScreenName</i> .
CONTROL	Displays information about the SET commands affecting the current screen.
INFO	Describes each field on the current screen.
SUBSYSTEM	Describes the subsystem.
TUNE	Displays a list of performance tuning topics relevant to <i>ScreenName</i> .
<i>Number</i>	Provides detailed information on the tuning topic designated by <i>Number</i> .
MAIN	Displays general information about SQLMON.
<i>SetCommand</i>	Describes <i>SetCommand</i> .

Description

- The help information is context sensitive. For example, if you specify TUNE from the Lock subsystem, SQLMON displays tuning hints about locks.
- You remain in the help facility after SQLMON displays the information you requested.
- You can enter any of the parameters from the Help prompt. For example, if you enter


```
SQLMONITOR HELP LOCK = info
```

 SQLMON displays information about each field on the Lock screen.
- To leave the help facility, you can enter any of the following commands:

```
//
EXIT
QUIT
```

HELP

Examples

To invoke the help facility for the last screen displayed, issue the HELP command without options, as in

```
SQLMONITOR OVERVIEW => HELP
```

You can use a question mark instead of the HELP command, as in:

```
SQLMONITOR OVERVIEW => ?
```

To get general information on SQLMON from within the help facility, use the following command:

```
SQLMONITOR HELP OVERVIEW => MAIN
```

In the last example, you do not need to enter the keyword HELP, since you are already in the help facility.

```
SQLMONITOR OVERVIEW => HELP MAIN
```

The following command displays information about the fields on the IO screen:

```
SQLMONITOR OVERVIEW => HELP IO INFO
```

To get help on the SET DBENVIRONMENT command, you can issue:

```
SQLMONITOR OVERVIEW => HELP SET DBENV
```

To exit the help facility and return to the subsystem prompt, you would issue

```
SQLMONITOR HELP OVERVIEW => EXIT
```


QUIT

Leaves SQLMON

Scope

SQLMON Only

SQLMON Syntax

```
Q[UIT]
```

Description

- This command is equivalent to the EXIT command.

Examples

```
SQLMONITOR OVERVIEW => QUIT
```

SET

Displays the current settings of the SQLMON variables

Scope

SQLMON Only

SQLMON Syntax

```
SET
```

Description

- Each setting corresponds to a SET command. For example, the CYCLE setting is specified by the SET CYCLE command.

Example

To view the current settings, issue the SET command, as in

```
SQLMONITOR OVERVIEW => SET
```

SQLMON responds with a list of the current settings, like this one:

```
C[CYCLE]                OFF
DBEF[ILESET]           OFF
DBEN[VIRONMENT]       PartsDBE
DBEC[ONNECT]          ON
DBEI[NITPROG]         ON
E[CHO]                OFF
LOCKF[ILTER]          SU/TPR/GWC/SXRsr6v/1
LOCKO[BJECT]          OFF
LOCKT[ABFILTER]       OFF
M[ENU]                ON
OUTP[UT]              OFF
R[EFRESH]             10
SA[MPLING]            ON
DI[SPLAYSAMPLES]     OFF
SORTIOD[ATA]          3
SORTIOL[OG]           OFF
SORTLOA[D]            3
SORTLOC[K]            5
SORTS[AMPLEIO]        3
T[OP]                 OFF
U[SERTIMEOUT]         5
```

SET CYCLE

Specifies the number of refresh cycles for which screens are displayed

Scope

SQLMON Only

SQLMON Syntax

```
SET C[CYCLE] [ NumCycles ]
              [ OFF ]
```

Parameters

<i>NumCycles</i>	Specifies that SQLMON displays screens for <i>NumCycles</i> - 1 refresh cycles before returning you to the prompt. An integer value.
OFF	Specifies that SQLMON displays screens until you press Return. The logical equivalent of a <i>NumCycles</i> value of infinity. The default value.

Description

- This command is useful in batch jobs. If you use SQLMON interactively, you can set *NumCycles* to OFF and then exit screens by pressing Return.
- See “Creating Batch Reports” in the chapter “Getting Started with SQLMON.”

Examples

When you issue the following command, SQLMON displays a screen once and refreshes it twice:

```
SQLMONITOR OVERVIEW => SET CYCLE 3
```

If you omit the integer after SET CYCLE, SQLMON prompts for it, as follows:

```
SQLMONITOR OVERVIEW => SET CYCLE

NUMBER OF CYCLES <OFF=INFINITY>: 3
```

SET DBECONNECT

Modifies the behavior of the SET DBENVIRONMENT command

Scope

SQLMON Only

SQLMON Syntax

```
SET DBEC [ONNECT] [ ON  
OFF ]
```

Parameters

ON	When you issue SET DBENVIRONMENT, SQLMON connects to the DBEnvironment and accesses the system catalog. The default value.
OFF	When you issue SET DBENVIRONMENT, SQLMON does not connect to the DBEnvironment or access the system catalog.

Description

- Use SET DBECONNECT OFF if the prompt does not return after you issue SET DBENVIRONMENT. SET DBENVIRONMENT might execute indefinitely if the DBEnvironment has a locking problem.

If SET DBENVIRONMENT executes indefinitely, do the following:

- Use the HP-UX `kill` command to terminate SQLMON.
- Run SQLMON again.
- Issue SET DBECONNECT OFF.
- Issue SET DBENVIRONMENT.
- Use the screens in the Lock subsystem, because the DBEnvironment probably has a lock problem.

Example

```
SQLMONITOR OVERVIEW => SET DBECONNECT OFF
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR OVERVIEW => SET DBECONNECT  
ON/OFF: OFF
```

SET DBEFILESET

Determines which DBEFileSets are included on a screen

Scope

SQLMON Only

SQLMON Syntax

```
SET DBEF [ ILESET ] [ DBEFileSetName ]
                OFF
```

Parameters

DBEFileSetName Displays information only on *DBEFileSetName*.

OFF Displays information on all DBEFileSets.

Description

- SET DBEFILESET affects the following screens:
 - SampleIO
 - SampleIO Indexes
 - SampleIO Objects
 - SampleIO Tables
 - Static Cluster
 - Static DBEFile
 - Static Hash
 - Static Indirect
 - Static Size
- When you are interested in a single DBEFileSet, you can use the SET DBEFILESET command to remove unwanted information and make the screens easier to read.
- You can use the SET DBEFILESET command in the Static subsystem to improve SQLMON's performance. When you access a screen in the Static subsystem, SQLMON performs multiple serial scans on each DBEFileSet, which can be very time-consuming. However, when you specify a DBEFileSet with SET DBEFILESET, SQLMON scans only the DBEFileSet you name.

Examples

In the following example, the information displayed is limited to the OrderFS fileset:

```
SQLMONITOR STATIC => SET DBEF OrderFS
```

If you omit the DBEFileSetName, SQLMON prompts for it, as follows:

```
SQLMONITOR STATIC => SET DBEFILESET
```

```
DBEFILESET NAME <or OFF>: OrderFS
```

SET DBEINITPROG

Modifies the behavior of the SET DBENVIRONMENT command

Scope

SQLMON Only

SQLMON Syntax

```
SET DBEINITPROG [ ON  
                  OFF ]
```

Parameters

ON	SQLMON gathers program information for each session when you issue SET DBENVIRONMENT. The default value.
OFF	SQLMON defers gathering program information for each session until you invoke a screen that displays the information.

Description

- If SET DBENVIRONMENT pauses for too long to gather program information, issue the command SET DBEINITPROG OFF. SET DBENVIRONMENT might pause if it is gathering information for a large number of sessions.

Example

```
SQLMONITOR OVERVIEW => SET DBEINITPROG OFF
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR OVERVIEW => SET DBEINITPROG
```

```
ON/OFF: OFF
```

SET DBENVIRONMENT

Specifies which DBEnvironment SQLMON monitors

Scope

SQLMON Only

SQLMON Syntax

```
SET DBEN[VIRONMENT] [ DBEnvironmentName [ MAINT=Maintenance Word ]
OFF
```

Parameters

<i>DBEnvironmentName</i>	The name of the DBEnvironment to be monitored. <i>DBEnvironmentName</i> is case sensitive.
<i>Maintenance Word</i>	The maintenance word of the DBEnvironment, required if you are not the DBEnvironment creator or superuser.
OFF	The default value, indicating that no DBEnvironment has been specified.

Description

- You cannot display an SQLMON screen until you issue SET DBENVIRONMENT.
- SQLMON can monitor just one DBEnvironment at a time.
- The SET DBECONNECT and SET DBEINITPROG commands affect the behavior of the SET DBENVIRONMENT command.

Examples

The following example shows how to monitor the PartsDBE DBEnvironment:

```
SQLMONITOR OVERVIEW => SET DBENV PartsDBE
```

If you omit the DBEnvironment name or the maintenance word, SQLMON prompts for them, as follows:

```
SQLMONITOR OVERVIEW => SET DBENVIRONMENT
```

```
DBENVIRONMENT NAME <or OFF>: PartsDBE
```

```
MAINTENANCE WORD:
```

SET DISPLAYSAMPLER

Determines whether a SampleIO screen or a sampling scale is displayed when samples are collected from the data buffer pool

Scope

SQLMON Only

SQLMON Syntax

```
SET DI [ DISPLAYSAMPLER ] [ ON ]  
                                [ OFF ]
```

Parameters

- | | |
|-----|---|
| ON | When SAMPLING is ON and you access a SampleIO screen, SQLMON displays the screen for the number of refresh cycles that you have set with SET CYCLE. |
| OFF | When SAMPLING is ON and you access a SampleIO screen, SQLMON displays a scale instead of the screen. Each time SQLMON completes a refresh cycle, it displays a period beneath the scale. The default value. |

Description

- If SAMPLING is OFF and you access a Sample IO screen, SQLMON displays the screen showing activity since the last SET DBENVIRONMENT command.
- The following is an example scale:

```
SQLMONITOR SAMPLEIO => /sa
```

```
SAMPLING = ON  
REFRESH  = 10      (One set of samples will be taken every 10 seconds).  
CYCLE    = 5       (A total of 5 sets of samples will be taken, then you  
will automatically return to the SQLMONITOR prompt).
```

```
1020304050  
12345678901234567890123456789012345678901234567890  
.....
```

- The scale is only printed when SAMPLING is ON, DISPLAYSAMPLER is OFF, and you access a SampleIO subsystem screen. The scale helps you to identify the number of samples that have been taken.
- The REFRESH variable determines the number of seconds that SQLMON pauses for each refresh cycle. The data buffer pool is sampled 25 times during each refresh cycle.

SET DISPLAYSAMPLES

- The CYCLE variable determines the number of refresh cycles that will be executed before returning to the SQLMON prompt. If CYCLE is OFF, you must press return to break out of the sampling loop:

```
SQLMONITOR SAMPLEIO => /sa
```

```
SAMPLING = ON
REFRESH  = 10      (One set of samples will be taken every 10 seconds).
CYCLE    = OFF     (No limit has been set for the number of samples to
                  be taken. You must press RETURN when you wish to
                  return to the SQLMONITOR prompt.)
```

```
1020304050
12345678901234567890123456789012345678901234567890
.....RETURN
```

Examples

To view screens in the SampleIO subsystem, first issue the following command:

```
SQLMONITOR SAMPLEIO => SET DISPLAYSAMPLES ON
```

You can then invoke the screen you wish to display.

If you omit the last parameter, SQLMON prompts you for it, as follows:

```
SQLMONITOR SAMPLEIO => SET DISPLAYSAMPLES
```

```
ON/OFF: ON
```

SET ECHO

Echoes commands to standard output

Scope

SQLMON Only

SQLMON Syntax

```
SET E[CHO] [ ON  
OFF ]
```

Parameters

ON	Causes your input to appear on standard output.
OFF	Does not cause your input to appear on standard output. The default value.

Description

- When ECHO is set to ON, SQLMON displays on standard output all of the commands you enter.
- SET ECHO is primarily used in batch jobs.
- See “Creating Batch Reports” in the chapter “Getting Started with SQLMON.”

Examples

After you enter the following command, SQLMON commands you enter appear on standard output:

```
SQLMONITOR OVERVIEW => SET ECHO ON
```

You can also use SET ECHO in a batch file to display the commands in the file as they execute. In the example, the Static screen is copied to an output file named Report1:

```
SET ECHO ON  
SET DBENVIRONMENT PartsDBE  
SET CYCLE 1  
SET OUTPUT Report1  
STATIC  
EXIT
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR OVERVIEW => SET ECHO
```

```
ON/OFF: ON
```

SET LOCKFILTER

Filters lock information so that only certain information is displayed

Scope

SQLMON Only

SQLMON Syntax

```
SET LOCKF [ILTER] [ TableType ] [ ... ] / [ Granularity ] [ ... ] / [ LockStatus ] [ ... ]
/ [ LockMode ] [ ... ] / [ QueueLength ] [ ... ]
```

Parameters

<i>TableType</i>	<p>Displays only the locks related to the table types you specify. The allowed values are</p> <ul style="list-style-type: none"> S System tables, those tables whose owner is HPRDBSS, DBCORE, or STOREDSECT U User tables, those tables that are not system tables
<i>Granularity</i>	<p>Limits the display of locks according to the granularity of the lock object. The allowed values are</p> <ul style="list-style-type: none"> T Table P Page R Row
<i>LockStatus</i>	<p>Displays the locks that have the status you specify. The allowed values are</p> <ul style="list-style-type: none"> G Locks that have been granted W Locks for which at least one session is waiting C Locks that are being converted to a stronger mode
<i>LockMode</i>	<p>Limits the display of locks according to the mode of the lock. The allowed values are</p> <ul style="list-style-type: none"> S Share X Exclusive s Intent share x Intent exclusive 6 Share + intent exclusive R Recovery exclusive r Recovery intent exclusive v Recovery share + intent exclusive <p>This parameter is case sensitive.</p>
<i>QueueLength</i>	<p>Represents the number of sessions in the lock object queue.</p>

SET LOCKFILTER

Description

- SET LOCKFILTER affects only the Lock, Lock Object, and Lock Session screens.
- The default value for LOCKFILTER is

```
SU/TPR/GWC/SXRsr6v/1
```

This value causes all locks to be displayed.

- You must use all four delimiting slashes in the parameter list, but you can omit any or all of the parameters. If you omit all of the parameters, you set the lock filter to the default value. If you omit some of the parameters, the settings for the missing parameters are unchanged.

Examples

To set the lock filter to the default value, you would enter

```
SQLMONITOR LOCK => SET LOCKFILTER ////
```

To display only those locks for which one or more sessions are waiting, enter

```
SQLMONITOR LOCK => SET LOCKFILTER //WC//
```

The next example displays only those locks that

- are for user tables
- are table locks or page locks
- are held by sessions that are waiting to acquire or convert locks
- are either exclusive, intent exclusive, or share plus intent exclusive

```
SQLMONITOR LOCK => SET LOCKFILTER U/TP/WC/Xx6/1
```

Since you can enter the characters within a filter parameter in any order, the two commands in the following example are equivalent:

```
SQLMONITOR LOCK => SET LOCKFILTER ///SX6/
```

```
SQLMONITOR LOCK => SET LOCKFILTER ///6SX/
```

SET LOCKFILTER

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR LOCK => SET LOCKFILTER
```

PRESS <RETURN> TO SELECT ALL ITEMS FOR EACH PROMPT

```
TABLES      <(S)ystem      (U)ser              > : U
GRANULARITY <(T)able      (P)age      (R)Row      > : T P
LOCK STATUS <(G)ranted    (W)aiting    (C)onverting > : W
LOCK MODES  <(S)Share    (X)Exclusive (R)ecovry Excl
            (s)IS      (x)IX        (r)RIX
            (6)SIX    (v)RSIX      > :
QUEUE LENGTH <integer value greater than 0 > : 2
```

The current setting for the LOCKFILTER is:

```
TABLES      : (U)SER
GRANULARITY : (T)ABLE      (P)AGE
LOCK STATUS : (W)AITING
LOCK MODES  : (S)SHARE    (X)EXCLUSIVE    (R)ECOVRY EXCL
            (s)IS      (x)IX          (r)RIX
            (6)SIX    (v)RSIX
QUEUE LENGTH : 2
```

SET LOCKOBJECT

Specifies the objects that are displayed on the Lock Object screen

Scope

SQLMON Only

SQLMON Syntax

```
SET LOCKO [BJECT] [ Owner.Table [/Constraint] [/PageRowID] ]  
                [ ALL  
                [ OFF
```

Parameters

<i>Owner.Table</i>	Displays only table lock information about <i>Owner.Table</i> .
<i>Constraint</i>	Displays only constraint lock information about <i>Constraint</i> .
<i>PageRowID</i>	Displays only page lock or row lock information about the page or row you specify. <i>PageRowID</i> consists of four digits delimited by either blanks or colons, as in the following examples: 0 1 5 20 0:1:5:20
ALL	Displays all of the lock objects that qualify under LOCKFILTER and LOCKTABFILTER. The default value.
OFF	Causes SQLMON to prompt you for the locked object when you invoke the Lock Object screen.

Description

- This command affects only the Lock Object Screen.
- If LOCKOBJECT is set to OFF, SQLMON prompts you for parameters when you invoke the Lock Object screen.

Examples

To display only the sessions that are requesting a table level lock on PurchDB.Parts, you would enter

```
SQLMONITOR LOCK => SET LOCKOBJECT PurchDB.Parts
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR LOCK => SET LOCKOBJECT
```

```
OWNER.TABLE[/CONSTRAINT]: PurchDB.Parts
```

```
PAGE/ROW ID: 0 1 5 20
```

To display only the lock information for the Members_FK referential constraint, enter

```
SQLMONITOR LOCK => SET LOCKOBJECT RecDB.Members/Members_FK
```

SET LOCKTABFILTER

Filters lock information according to the object specified

Scope

SQLMON Only

SQLMON Syntax

```
SET LOCKT [ABFILTER] [ Owner.Table [/Constraint] ]  
OFF
```

Parameters

<i>Owner.Table</i>	Displays only table lock information about <i>Owner.Table</i> .
<i>Constraint</i>	Displays only constraint lock information about <i>Constraint</i> .
OFF	Default value.

Description

- This command affects the following screens:
 - Lock
 - Lock Object
 - Lock Session

Example

Only the locks associated with the PurchDB.Parts table are displayed:

```
SQLMONITOR LOCK => SET LOCKTABFILTER PurchDB.Parts
```

SQLMON prompts for the parameters when they are omitted, as follows:

```
SQLMONITOR LOCK => SET LOCKTABFILTER
```

```
OWNER.TABLE[/CONSTRAINT]: PurchDB.Parts
```

Display only those locks associated with the Members_FK referential constraint:

```
SQLMONITOR LOCK => SET LOCKTABFILTER RecDB.Members/Members_FK
```


SET MENU

Controls the display of SQLMON menus

Scope

SQLMON Only

SQLMON Syntax

```
SET M[ENU] [ ON
            OFF ]
```

Parameters

ON Displays menus at the SQLMON prompt. The default value.
 OFF Does not display menus.

Description

- Expert users may want to set the menu option to OFF.
- When MENU is ON, SQLMON displays menus such as the following:

SQLMONITOR SUBSYSTEMS (and abbreviations):

```

OVERVIEW      IO      LOAD      LOCK      SAMPLEIO      STATIC
  /o          /i      /loa     /loc          /sa            /st
```

CURRENT SUBSYSTEM SCREENS:

```

OVERVIEW      SESSION      PROGRAM
      o          s          p
```

Examples

To turn off the menus, issue the following command:

```
SQLMONITOR OVERVIEW => SET MENU OFF
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR OVERVIEW => SET MENU
```

```
ON/OFF: ON
```

SET OUTPUT

Saves SQLMON screen images in a file

Scope

SQLMON Only

SQLMON Syntax

```
SET OUTP[UT] [ SystemFileName ]  
                [ OFF ]
```

Parameters

SystemFileName Echoes the screens that SQLMON displays on standard output to *SystemFileName*.

OFF Does not echo screens to a file. The default value.

Description

- If you issue a SET OUTPUT command with a filename, and if the file does not exist, SQLMON creates it.
- If you issue a SET OUTPUT command with a filename, and if the file exists, SQLMON asks you if you want to overwrite it.
- SQLMON appends information to the file until you issue SET OUTPUT OFF or terminate the SQLMON session.
- For more information, see the “Examples” section or the section “Creating Batch Reports” in the chapter “Getting Started with SQLMON.”

Examples

If you issue the commands

```
SQLMONITOR OVERVIEW => SET OUTPUT Myfile
```

```
SQLMONITOR OVERVIEW => OVERVIEW
```

```
SQLMONITOR OVERVIEW => OVERVIEW SESSION
```

```
SQLMONITOR OVERVIEW => SET OUTPUT OFF
```

SQLMON creates Myfile and adds images of both the Overview and Overview Session screens to the file.

If you then issue the commands

```
SQLMONITOR OVERVIEW => SET OUTPUT Myfile
```

```
System Filename already exists. Overwrite? yes
```

```
SQLMONITOR OVERVIEW => LOCK
```

```
SQLMONITOR OVERVIEW => LOCK OBJECT
```

```
SQLMONITOR OVERVIEW => SET OUTPUT OFF
```

SQLMON overwrites Myfile and then appends images of the Lock and Lock Object screens to the file.

To prevent accidental data loss, SQLMON prompts you before it overwrites the file, as in

```
SQLMONITOR OVERVIEW => SET OUTPUT Myfile
```

```
SystemFileName already exists. Overwrite? ([NO],YES) no
```

```
SystemFileName not overwritten.
```

If you omit the last parameter, SQLMON prompts for it, as in

```
SQLMONITOR OVERVIEW => SET OUTPUT
```

```
SystemFileName <or OFF>: OFF
```

SET REFRESH

Determines the refresh rate of the SQLMON screens

Scope

SQLMON Only

SQLMON Syntax

```
SET R[EFRESH] [Seconds]  
              [OFF]
```

Parameters

Seconds The number of seconds SQLMON waits before it resamples data and refreshes a screen. The default value is 10.

Description

- This command affects all screens, except those in the Static subsystem.

Examples

To refresh a screen every five seconds, issue the following command:

```
SQLMONITOR OVERVIEW => SET REFRESH 5
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR OVERVIEW => SET REFRESH
```

```
NUMBER OF SECONDS: 3  
                          _
```

SET SAMPLING

Enables sampling of the data buffer pool

Scope

SQLMON Only

SQLMON Syntax

```
SET SA[MPLING] [ON
                OFF]
```

Parameters

ON	Specifies that SQLMON samples the data buffer pool each time a SampleIO subsystem screen is refreshed. The default value.
OFF	Specifies that SQLMON does not sample the data buffer pool. Improves the performance of SQLMON.

Description

- It is recommended that you issue SET SAMPLING ON, perform sampling, issue SET SAMPLING OFF, and then access the SampleIO screens to view the results. You may find it easier to use the SampleIO screens when sampling is not being performed, because CPU utilization is smaller.
- SQLMON samples the data buffer pool only when SAMPLING is ON and you access a SampleIO screen. In that case, SQLMON samples the data buffer pool 25 times during each refresh cycle.
- The SWAPIN, SWAPOUT, and TOTALIO counters on the SampleIO screens are incremented to reflect activity during sampling. The counters are initialized to 0 when you issue SET DBENVIRONMENT. For more information, see the chapter “Troubleshooting with SQLMON.”
- When sampling is ON, the DISPLAYSAMPLES variable determines whether you see a scale or a screen when samples are collected. For more information, see the command page for SET DISPLAYSAMPLES.

sequence. First, to turn on sampling, enter

```
SQLMONITOR SAMPLEIO => SET SAMPLING ON
```

SET SORTIODATA

Defines how items are sorted on the IO Data Program and IO Data Session screens

Scope

SQLMON Only

SQLMON Syntax

```
SET SORTIOD [ATA] [ SortColumn ]  
                OFF
```

Parameters

SortColumn The column by which items are sorted. *SortColumn* is an integer from 0 to 4 that represents a column on the screen, as follows:

- 0 same as OFF
- 1 BUFF ACCESS
- 2 DATA DISK RD
- 3 DATA DISK WR (default)
- 4 MISS RATE

The items are sorted in descending order according to the values in the column.

OFF Sorts the items by program name and PID on the IO Data Program screen and by PID on the IO Data Session screen.

Description

- This command affects only the IO Data Program and IO Data Session screens.
- On each of these screens, an asterisk appears beside the sort column number.
- For more information on the IO Data Program and IO Data Session screens, see the chapter “SQLMON Screen Reference.”

Examples

To sort the items on the IO Data Program and IO Data Session screens according to the values in the DATA DISK RD column, enter

```
SQLMONITOR IO => SET SORTIODATA 2
```

If you omit the last parameter, SQLMON prompts for it, as follows:

```
SQLMONITOR IO => SET SORTIODATA
```

```
<0=OFF,1=BUFF ACCESS,2=DATA DISK RD,3=DATA DISK WR,4=MISS RATE>: 2
```

SET SORTIOLOG

Defines how items are sorted on the IO Log Program and IO Log Session screens

Scope

SQLMON Only

SQLMON Syntax

```
SET SORTIOL[OG] [SortColumn]
                [OFF]
```

Parameters

SortColumn The column by which items are sorted. *SortColumn* is an integer from 0 to 3 that represents a column on the screen, as follows:

- 0 same as OFF
- 1 LOG BUFF WR
- 2 LOG DISK RD
- 3 LOG DISK WR (default)

The items are sorted in descending order according to the values in the column.

OFF Sorts the items by program name and PID on the IO Log Program screen and by PID on the IO Log Session screen.

Description

- This command affects only the IO Log Program and IO Log Session screens.
- On each of these screens, an asterisk appears beside the sort column number.
- For more information on the IO Log Program and IO Log Session screens, see the chapter “SQLMON Screen Reference.”

Examples

To sort the items on the IO Log Program and IO Log Session screens according to the value in the LOG DISK RD column, enter

```
SQLMONITOR IO => SET SORTIOLOG 2
```

If you omit the last parameter, SQLMON prompts for it, as in

```
SQLMONITOR IO => SET SORTIOLOG
```

```
<0=OFF,1=LOG BUFF WR,2=LOG DISK RD,3=LOG DISK WR>: 2
```

SET SORTLOAD

Defines how items are sorted on the Load Program and Load Session screens

Scope

SQLMON Only

SQLMON Syntax

```
SET SORTL [OAD] [ SortColumn ]  
                [ OFF ]
```

Parameters

SortColumn The column by which items are sorted. *SortColumn* is an integer from 0 to 4 that represents a column on the screen, as follows:

- 0 same as OFF
- 1 BEGIN WORK
- 2 COMMIT WORK
- 3 ROLLBK WORK (default)
- 4 DEADLOCKS

The items are sorted in descending order according to the values in the column.

OFF Sorts the items by program name and PID on the Load Program screen and by PID on the Load Session screen.

Description

- This command affects only the Load Program and Load Session screens.
- On each of these screens, an asterisk appears beside the sort column number.
- For more information on the Load Program and Load Session screens, see the chapter “SQLMON Screen Reference.”

Examples

To sort the items on the Load Program and Load Session screens according to the value in the COMMIT WORK column, you can enter

```
SQLMONITOR LOAD => SET SORTLOAD 2
```

If you omit the last parameter, SQLMON prompts for it, as in:

```
SQLMONITOR LOAD => SET SORTLOAD
```

```
<0=OFF,1=BEGIN WORK,2=COMMIT WORK,3=ROLLBK WORK,4=DEADLOCKS>: 2
```


SET SORTLOCK

Defines how sessions are sorted on the Lock Memory screen

Scope

SQLMON Only

SQLMON Syntax

```
SET SORTL [OAD] [ SortColumn ]
                [ OFF ]
```

Parameters

SortColumn The column by which items are sorted. *SortColumn* is an integer from 0 to 5 that represents a column on the screen, as follows:

0	same as OFF
1	TABLE
2	PAGE
3	ROW
4	TOTAL
5	MAXTOTAL (default)

The items are sorted in descending order according to the values in the column.

OFF Sorts the items by PID.

Description

- This command affects only the Lock Memory screen.
- On the screen, an asterisk appears beside the sort column number.
- For more information on the Lock Memory screen, see the chapter “SQLMON Screen Reference.”

Examples

To sort the sessions on the Lock Memory screen according to the value in the ROW column, you can enter

```
SQLMONITOR LOCK => SET SORTLOCK 3
```

If you omit the last parameter, SQLMON prompts for it, as in

```
SQLMONITOR LOCK => SET SORTLOCK
```

```
<0=OFF,1=TABLE,2=PAGE,3=ROW,4=TOTAL,5=MAXTOTAL>: 3
```

SET SORTSAMPLEIO

Defines how items are sorted on the SampleIO subsystem screens

Scope

SQLMON Only

SQLMON Syntax

```
SET SORTS [AMPLIO] [ SortColumn ]  
                [ OFF ]
```

Parameters

SortColumn The column by which items are sorted. *SortColumn* is an integer from 0 to 3 that represents a column on the screen, as follows:

- 0 same as OFF
- 1 SWAPIN
- 2 SWAPOUT
- 3 TOTALIO (default)

The items are sorted in descending order according to the values in the column.

OFF Sorts the items by object name.

Description

- This command affects the SampleIO, SampleIO Indexes, SampleIO TabIndex, and SampleIO Tables screens.
- On the screens, an asterisk appears beside the sort column number.
- For more information on the SampleIO screens, see the descriptions of each screen in the chapter “SQLMON Screen Reference.”

Examples

To sort the items on the SampleIO subsystem screens according to the value in the SWAPOUT column, enter

```
SQLMONITOR SAMPLEIO => SET SORTSAMPLEIO 2
```

If you omit the last parameter, SQLMON prompts for it, as in

```
SQLMONITOR SAMPLEIO => SET SORTSAMPLEIO
```

```
<0=OFF,1=SWAPIN,2=SWAPOUT,3=TOTALIO>: 2
```

SET TOP

Determines the number of items displayed on a screen

Scope

SQLMON Only

SQLMON Syntax

```
SET T[OP] [ NumItems
            OFF
```

Parameters

NumItems The number of items to display. An integer value.

OFF All items. The default value.

Description

This command affects the following screens:

- IO Data Program
- IO Data Session
- IO Log Program
- IO Log Session
- Load Program
- Load Session
- SampleIO
- SampleIO Indexes
- SampleIO TabIndex
- SampleIO Tables

Examples

To display the top ten items (sorted according to a defined value), you can enter

```
SQLMONITOR IO => SET TOP 10
```

The items on the screens are sorted according to the value of a sort command, as listed below:

- For the IO Data Program and IO Data Session screens, SET SORTIODATA
- For the IO Log Program and IO Log Session screens, SET SORTIOLOG
- For the Load Program and Load Session screens, SET SORTLOAD
- For the screens in the SampleIO subsystem, SET SORTSAMPLEIO
- For the Lock Memory screen, SET SORTLOCK

For more information on the commands listed above, see their command pages in this chapter.

If you omit the last parameter, SQLMON prompts for it, as in

```
SQLMONITOR IO => SET TOP
```

```
NUMBER OF ITEMS TO DISPLAY <OFF=ALL>: 5
```

SET USERTIMEOUT

Specifies how long SQLMON waits for a database resource that is unavailable

Scope

SQLMON Only

SQLMON Syntax

```
SET U[SERTIMEOUT] [ Seconds ]  
                   [ OFF ]
```

Parameters

- | | |
|----------------|--|
| <i>Seconds</i> | The number of seconds SQLMON waits. <i>Seconds</i> is an integer value greater than or equal to 0. The value 0 is equivalent to OFF. The default value is 5. |
| OFF | Specifies that SQLMON does not wait. |

Description

- The resources SQLMON might wait for include transaction slots and locks, as explained below:

Transaction Slots	If you issue SET DBENVIRONMENT and DBECONNECT is ON, SQLMON connects to the DBEnvironment and creates a transaction. If the transaction limit is reached, SQLMON waits for a transaction slot for <i>Seconds</i> seconds.
Locks	If SQLMON tries to lock an object that is already locked in a conflicting mode, SQLMON waits until the lock is granted or until <i>Seconds</i> seconds pass.
- If SQLMON cannot obtain the resource in the allotted time, it displays an error message.

Examples

To have SQLMON wait 10 seconds for a database resource, enter

```
SQLMONITOR OVERVIEW => SET USERTIMEOUT 10
```

If you omit the last parameter, SQLMON prompts for it, as in

```
SQLMONITOR OVERVIEW => SET USERTIMEOUT
```

```
NUMBER OF SECONDS: 10
```

!

Escapes temporarily to the operating system and (optionally) executes a single operating system command.

Scope

SQLMON Only

SQLMON Syntax

```
>> ! [CommandName];
```

Parameters

CommandName is the name of an HP-UX operating system command.

Description

- If you include a command name, control returns to SQLMON as soon as the command has been executed.
- If you omit the command name, use the *exit* command to return to SQLMON from the HP-UX shell.

Example

```
>> !ll;

total 5586
-rw-rw-r--  1 guest      guest      10626 Jul 12 11:06 +invfile
-rw-----  1 hpdb      guest      204800 Apr 20 15:20 OrderDF1
-rw-----  1 hpdb      guest      204800 Apr 20 15:20 OrderXF1
.
.
.
>>
```


Design for a High-Performance Interactive Table Editor

The following design shows how you can create interactive table editing for your applications while maintaining high concurrency using standard ALLBASE/SQL. The editor described supports very fast scrolling and very high concurrency by not holding any locks while terminal I/O takes place. High performance also derives from the use of BULK commands.

Example Table

The table described in the design is a Personnel table for all the employees in a company. The table consists of the Employee ID as the primary key, as well as Department Number and an Information field.

User Interface

The user interface consists of a main form which is a single table. The rows of the table contain all the fields of the employee record. All the fields are protected (i.e., display-only). Seven function keys are defined:

- **SELECT**, which causes display of a popup form asking for the department number. This popup form itself has two function keys: **OK** and **CANCEL**. **CANCEL** cancels the **SELECT** and makes the popup form disappear. **OK** selects the employees in the specified department and displays the rows in the main form after the popup form disappears.
- **CURSOR_UP** and **CURSOR_DOWN**, which position the cursor on the desired row in the table.
- **PAGE_UP** and **PAGE_DOWN**, which position the display table on the previous or next page of rows.
- **DELETE**, which deletes the current row. The row disappears from the table, and all the following rows are shifted up by one. A new row fills in the bottom of the table on the screen.
- **UPDATE**, which causes display of a popup form asking for the new values for the row. The popup form itself has two function keys: **OK** and **CANCEL**. **CANCEL** cancels the **UPDATE** and makes the popup form disappear. **OK** updates the employee specified and displays the rows in the main form after the popup form disappears. If the department number has not been changed, the new value of the row is displayed under the cursor; otherwise, the screen is handled as in the case of deletions.

Internal Algorithms

When the editor starts, it opens a file called the scroll file. It also allocates a 12K data buffer.

SELECT

When a SELECT is executed, the editor pulls in data from the table with BULK FETCH statements, 12K bytes at a time. If more than one BULK FETCH is needed, it appends the data from the previous FETCH to the scroll file. After the last FETCH in a sequence of FETCH statements, the data is appended to the scroll file, and the first 12K of data is read back into the data buffer. If all the data fits in the buffer, the scroll file is not used at all. Before control returns to the user, the transaction is ended.

The window pointer is then set to the first row in the buffer, and the first set of rows is displayed. Cursor positioning and page scrolling, since they are handled completely by the editor without accessing the database, are very fast.

DELETE

When a DELETE is executed, the tuple is fetched from the table based on its old primary key value. If the tuple is not found, then the following message is returned:

`Tuple Does Not Exist`

The row is marked as deleted in the memory buffer, and the table is redisplayed starting from the window pointer. All rows marked as deleted are skipped by the display manager.

If the tuple to be deleted is still found in the table, it is compared against the old value of the tuple. If they are identical, the tuple is deleted from the table, the row is marked as deleted in the memory buffer, and the table is redisplayed starting from the window pointer.

If the tuple in the database has changed, then the following message is returned:

`Tuple Has Changed`

The tuple's value in the buffer is updated, and the table is redisplayed starting from the window pointer.

In all three cases, the transaction is ended before control returns to the user. Locks are obtained and released before additional terminal I/O is requested.

UPDATE

When an UPDATE is executed, the tuple is fetched from the table based on its old primary key value. If the tuple is not found, then the following message is returned:

`Tuple Does Not Exist`

The row is marked as deleted in the memory buffer, and the table is redisplayed starting from the window pointer.

If the tuple is found, it is compared against the old value of the tuple. If they are identical, the tuple is updated, the tuple's value in the buffer is updated, and the table is redisplayed starting from the window buffer.

If the tuple in the database has changed, then the following message is returned:

Tuple Has Changed

The tuple's value in the buffer is updated, and the table is redisplayed starting from the window pointer.

In all three cases, the transaction is ended before control returns to the user.

Caution A dirty bit is kept for the whole data buffer. The bit is set when the contents of the buffer is modified (that is, becomes "dirty"). If set, the current contents have to be written to the scroll file before a new 12K block is read from it.

Index

A

- aborting programs
 - with `TERMINATE USER`, 5-2
- administration
 - DBA guidelines, 5-1
 - HP-UX guidelines, 5-15
 - system, 5-1
- aging
 - in data buffer calculations, 5-4
- arithmetic expressions
 - causing serial scans, 3-1
- authorization
 - effect on performance, 2-10

B

- balancing load
 - by separating files, 5-2
- batch reports
 - with `SQLMON`, 6-12
- `BEGIN WORK`
 - monitoring, 8-13
- `BETWEEN`
 - optimization of, 3-10
- B-tree
 - and indexes, 1-7
- buffer
 - calculating size of log buffer, 5-10
 - log data, 1-16
 - monitoring I/O, 8-2
 - types used by `ALLBASE/SQL`, 1-10
- `BULK` option
 - in queries, 3-12

C

- caching
 - of directory information, 1-15
 - of sections and `DBCORE` directory, 5-13
 - stored sections, 1-21
- calculations
 - overhead, 2-4
- checkpoints
 - monitoring, 8-3
- cluster count
 - and indexes, 2-7
 - defined, 1-20

- clustering indexes
 - monitoring, 8-51
 - using, 2-6
- `COMMIT WORK`
 - and log buffer, 1-15
 - causing I/O, 4-2
 - monitoring, 8-13
- compatibility
 - of data types, 3-2
- compression
 - of data pages, 1-6
- concurrency
 - improving, 2-13
- conjunctive normal form
 - in OR optimization, 3-8
- constraint
 - PCR, 1-9
- conversion
 - avoiding in `UNION` operations, 3-6
 - of data, 3-2
 - `SMALLINT` to `INTEGER`, 2-14
- correlated subqueries
 - effect on performance, 3-4
- cursor stability (CS)
 - using, 4-3

D

- data
 - conversion, 2-13, 3-2, 3-7
 - logical design, 2-1
 - physical design, 2-11
- data buffer
 - high miss rate, 7-3
 - insufficient space, 7-4
 - miss rate, 8-34
 - swapping, 7-19
- data buffer pages
 - adjusting with `SQLMON`, 5-9
 - calculating number of, 5-3
 - monitoring I/O, 8-2
 - monitoring objects, 8-44
 - monitoring swapping, 8-40
 - used by `ALLBASE/SQL`, 1-10
- data definition
 - reduces concurrency, 4-1
- data storage

- data page compression, 1-6
- DBA guidelines
 - for system administration, 5-1
- DBCORERR
 - deadlock, 7-11
 - lock allocation failure, 7-15
- DBEFile
 - defined, 2-13
 - monitoring capacity, 8-53
 - organization, 1-1
- DBEFileSet
 - defined, 2-11
 - determining contents, 8-50
 - full, 7-23
- DBEnvironment
 - specifying in SQLMON, 6-2
- DDL Enabled flag
 - and directory caching, 1-15
 - improving concurrency, 4-1
 - setting to NO, 5-2
- deadlock
 - and savepoints, 4-2
 - caused by locking, 4-2
 - detection, 1-17
 - example in SQLMON, 7-11
 - monitoring, 8-13
- default value
 - adding column with, 2-13
- deletions
 - tips on, 2-17
- design
 - logical, 2-1
 - of queries, 3-1
 - physical, 2-11
- detached files
 - identifying, 6-14
- directory caching
 - defined, 5-13
 - when DDL Enabled is set to NO, 1-15
- dirty reads
 - defined, 4-3
- disjunctive normal form
 - in OR optimization, 3-8
- disk space
 - increasing usage, 2-12
 - monitoring, 6-13
 - used for sorting, 5-12
- display options
 - setting for SQLMON, 6-9
- dynamic statements
 - performance of , 4-7

E

- echoing
 - SQLMON batch input, 9-14
- editor
 - high performance, for tables, A-1
- EXIT, 9-2

F

- factor
 - in optimization, 1-19
- file
 - load balancing, 5-2
 - monitoring capacity, 8-53
 - opening, 2-12
 - raw, 2-13
 - running out of space, 2-13
 - saving space, 2-13
- filtering SQLMON information
 - DBEFileSets, 9-9
 - lock mode, 9-15
 - lock object, 9-18
 - number of items displayed, 9-31
- filters
 - avoiding propagation by user, 3-11
- format
 - data page, 1-1
 - DBEFile, 1-1
 - page table page, 1-2
 - rows of user data on data pages, 1-3
 - TID, 1-4
 - tuple header, 1-3
- freezing
 - to examine deadlock, 7-11
 - to examine lock allocation failure, 7-15

G

- GENPLAN
 - analyzing queries with, 3-12
 - using to observe the optimizer's choices, 1-20

H

- hanging
 - SQLMON session, 9-8
- hashing
 - and TRUNCATE TABLE, 2-17
 - compared to B-tree index, 2-9
 - monitoring overflow pages, 7-24
 - monitoring tasks, 6-15
 - removing overflow pages, 2-16
 - Static Hash Screen, 8-55
 - storage on DBEFile pages, 1-5
 - using, 2-8
- HELP, 9-3
- host variables

- and scan buffer, 1-13
- hot spots
 - defined, 4-4
- HP-UX
 - setting parameters for, 5-15

I

IMAGE database

- storage of table, 8-50

index

- BETWEEN predicates, 3-10
- B-tree compared to hashing, 2-8, 2-9
- B-tree splits and logging, 1-16
- clustering, 2-6
- design of, 2-4
- LIKE predicates, 3-9
- maintaining, 2-6
- MIN/MAX predicates, 3-7
- monitoring clustering, 8-51
- monitoring I/O, 8-42
- monitoring tasks, 6-16
- NULL values, 2-14
- OR predicates, 3-8
- page splitting, 1-9
- placement on disks, 5-2
- poorly clustered, 7-23
- storage on DBEFile pages, 1-6
- table size, 1-19
- to improve I/O, 2-13
- UNION queries, 3-7
- USASCII columns, 2-13
- when needed, 2-14

indirect row

- avoiding, 7-24
- defined, 1-5
- monitoring, 8-57
- unloading and reloading to avoid, 2-16

INSTALL

- after preprocessing, 5-2

I/O

- balancing load of, 7-18
- caused by COMMIT WORK, 4-2
- data buffers, 1-10
- empty DBEFile space, 2-13
- fastest available disk, 2-6
- flooding with checkpoints, 5-10
- improving, 2-13
- incurred by inserts, 2-9
- log buffers, 5-10
- minimizing disk contention, 5-2
- monitoring buffers, 8-2
- monitoring data I/O, 6-18
- monitoring for DBEFile, 8-40
- monitoring for indexes, 8-42
- monitoring for logging, 8-8

- monitoring for program, 8-4
- monitoring for sessions, 8-6
- monitoring for tables, 8-48
- monitoring logging I/O, 6-19
- nested loop joins, 5-13
- shortening I/O path, 2-13
- small tables, 2-12
- system buffer pool, 1-13
- wasted, 2-6

IO Data Program Screen, 8-4

IO Data Session Screen, 8-6

IO Log Program Screen, 8-8

IO Log Session Screen, 8-10

IO Screen, 8-2

IO subsystem, 7-4

isolation level

- determining, 6-17
- improving concurrency with, 4-3
- KEEP CURSOR, 4-6
- locking of system catalog, 1-14
- locks, 1-17
- sorted query results, 4-3
- types of, 4-1

J

joins

- avoiding, 2-14
- compared with subqueries, 3-4
- denormalizing instead of, 2-2
- nested loop, 1-19, 5-13
- optimization of, 1-19
- sort/merge, 1-18, 1-19, 5-13

K

KEEP CURSOR

- use of, 4-1, 4-6

key columns

- updating, 3-10

L

latch

- use of, 1-17

leaf pages

- storage, 1-6

LIKE

- optimization of, 3-9

LOAD

- hashed table, 2-9
- initial, 2-15
- non-hash table, 2-14
- to avoid indirect rows, 2-16
- to improve cluster count, 2-1
- to recluster indexes, 2-8
- to remove overflow pages, 2-16

- load balancing
 - explained, 5-2
- Load Program Screen, 8-14
- Load Screen, 8-12
- Load Session Screen, 8-16
- Load subsystem, 7-7
- Lock Impede Screen, 8-20
- locking
 - allocation failures, 7-14
 - and deadlocks, 7-11
 - avoiding contention, 2-10
 - cause of waits, 8-20
 - causing delays, 7-9
 - causing transaction delays, 7-7
 - contention, 7-2
 - filtering information, 9-15
 - for KEEP CURSOR, 4-6
 - influenced by data definition, 4-1
 - monitoring contention, 7-8
 - monitoring tasks, 6-19
 - on the system catalog, 1-14
 - overview, 1-17
 - requests, 8-13
 - row level, 4-4
 - selecting types of, 4-1
 - sequence of events, 1-18
 - SQLMON screens, 8-18-33
 - subsystem in SQLMON, 7-9
 - wait %, 8-34
 - waits, 8-13
- Lock Memory Screen, 8-23
- lock object
 - defined, 4-4
- Lock Object Screen, 8-25
- Lock Screen, 8-18
- Lock Session Screen, 8-28
- Lock subsystem, 7-9
- Lock TabSummary Screen, 8-31
- logging
 - archive mode on IO screen, 8-3
 - choosing number and size of files, 5-10
 - choosing number of buffer pages, 5-10
 - definition of log, 1-15
 - file capacity, 7-3
 - file placement on disks, 5-2
 - insufficient buffer space, 7-5
 - monitoring errors, 8-35
 - monitoring file space, 8-35
 - monitoring I/O, 8-2
 - monitoring program I/O, 8-8
 - monitoring session I/O, 8-10
 - monitoring tasks, 6-19
 - using raw files, 5-16
- LRU (least recently used) algorithm
 - in calculating data buffer pages, 5-4

M

- memory
 - available real, 5-9
 - latches, 1-17
 - limit reached, 7-2
 - locks, 1-17
 - monitoring usage , 6-14
 - row level locks, 4-4
 - shared, calculating, 5-3
 - used in sorting, 5-13
- menus
 - displaying in SQLMON, 9-21
- MIN/MAX
 - optimization of, 3-7
- monitoring
 - with SQLMON , 6-13
- multiconnect
 - using, 5-14

N

- nested loop join
 - explained, 5-13
- network use
 - guidelines for, 5-15
- NLS data
 - USASCII columns, 2-13
- non-correlated subqueries
 - effect on performance, 3-4
- non-leaf pages
 - storage, 1-6
- normalization
 - in logical design, 2-1
 - pros and cons, 2-1
- NULL values
 - avoiding, 2-14
 - indirect rows, 1-5
 - tuple migration, 2-14

O

- operating system buffer
 - used by ALLBASE/SQL, 1-10
- optimization
 - BETWEEN predicates, 3-10
 - LIKE predicates, 3-9
 - MIN/MAX predicates, 3-7
 - OR predicates, 3-8
 - overview, 1-19
 - updates of key columns, 3-10
- optimizer
 - choice of join method, 5-13
 - observing results with GENPLAN, 1-20
 - overriding with SETOPT, 1-20
- OR

- optimization of, 3-8
- organization
 - DBEFile, 1-1
 - page, 1-1
- output
 - of SQLMON, 9-22
- overhead
 - generated by SQLMON, 6-12
- overnormalization
 - avoiding, 2-1
- Overview Program Screen, 8-36
- Overview Screen, 8-34
- Overview Session Screen, 8-38
- Overview subsystem, 7-1

P

- page
 - clean, 1-11
 - compression, 1-6
 - dirty, 1-11, 4-3
 - in DBEFiles, 1-1
 - no-log, 1-16
 - overflow , 1-5
- page splitting
 - of B-tree indexes, 1-9
- page table page
 - defined, 1-2
 - in DBEFiles, 2-13
- parallel serial scans
 - advantages of, 3-12
- PCR
 - defined, 1-9
- performance
 - basic concepts, 1-1
 - summary of topics, 2-1
- pin
 - use of, 1-18
- predicate
 - BETWEEN, 3-10
 - LIKE, 3-9
 - MIN/MAX, 3-7
 - OR, 3-8
- preprocessing
 - with a development DBE, 5-2
- procedures
 - and rules, 4-7
- program
 - monitoring data I/O, 8-4
 - monitoring load, 8-14
 - monitoring logging I/O, 8-8
 - monitoring status, 8-36
- propagation of filters
 - avoiding, 3-11
- PUBLIC ROW
 - overview, 4-4

Q

- queries
 - design of, 3-1
 - importance in design, 2-1

R

- raw files
 - and I/O, 2-13
 - using, 5-16
- read committed (RC)
 - using, 4-3
- read uncommitted (RU)
 - using, 4-3
- referential constraint
 - and index design, 2-4
 - and TRUNCATE TABLE, 2-17
 - enforced with a PCR, 1-9
 - monitoring clustering, 8-51
 - monitoring I/O, 8-42
 - monitoring tasks, 6-16
- REFETCH statement
 - and RC or RU, 4-6
- refresh
 - setting cycle , 9-7
 - setting rate, 9-24
- revalidation
 - avoiding at run time, 5-1
- ROLLBACK WORK
 - and transactions, 7-7
 - monitoring, 8-13
- row
 - how stored on DBEFile page, 1-2
- row level locking
 - overview, 4-4
- rules
 - and procedures, 4-7
- runtime control blocks
 - choosing size of, 5-9
 - latches, 1-17
 - locks, 1-17
 - monitoring, 8-35
 - row level locking, 4-4
 - running out of, 7-2

S

- SampleIO Indexes Screen, 8-42
- SampleIO Objects Screen, 8-44
- SampleIO Screen, 8-40
- SampleIO subsystem, 7-18
- SampleIO TabIndex Screen, 8-46
- SampleIO Tables Screen, 8-48
- sampling
 - setting display, 9-12

- setting with SQLMON, 9-25
- savepoint
 - defined, 4-2
- scan buffer
 - different from data buffer, 1-10
- scan type
 - avoiding serial scans, 3-1
 - overview of choices for optimization, 1-19
 - using parallel serial scans, 3-12
- screens
 - invoking in SQLMON, 6-3
- section caching
 - defined, 5-13
- sections
 - defined, 1-21
 - semi-permanent, 4-8
- SELECT
 - compared to FETCH with cursor, 4-1
 - design of, 3-1
- session
 - monitoring data I/O, 8-6
 - monitoring load, 8-16
 - monitoring locks, 8-28, 8-34
 - monitoring logging I/O, 8-10
 - monitoring status, 8-38
 - monitoring tasks, 6-17
- SET, 9-6
- SET CYCLE, 9-7
- SET DBECONNECT, 9-8
- SET DBEFILESET, 9-9
- SET DBEINITPROG, 9-10
- SET DBENVIRONMENT, 9-11
- SET DISPLAYSAMPLES, 9-12
- SET ECHO, 9-14
- SET LOCKFILTER, 9-15
- SET LOCKOBJECT, 9-18
- SET LOCKTABFILTER, 9-20
- SET MENU, 9-21
- SETOPT
 - modifying access optimization plan with, 3-13
 - using to override the optimizer's choices, 1-20
 - using to specify join method, 5-13
- SET OUTPUT, 9-22
- SET REFRESH, 9-24
- SET SAMPLING, 9-25
- SET SORTIODATA, 9-26
- SET SORTIOLOG, 9-27
- SET SORTLOAD, 9-28
- SET SORTLOCK, 9-29
- SET SORTSAMPLEIO, 9-30
- SET TOP, 9-31
- SET USERTIMEOUT, 9-32
- shared memory
 - calculating, 5-3
- shared tuple header

- explained, 1-3
- size of objects
 - monitoring, 8-59
- slot table
 - defined, 1-4
- sorting
 - large sorts, 5-13
 - methods used in ALLBASE/SQL, 5-11
 - overview, 1-18
- sort/merge join
 - explained, 5-13
- SQLMON
 - batch reports, 6-12
 - defined, 6-1
 - leaving, 6-2
 - monitoring tasks , 6-13
 - online help, 6-10
 - screen command summary, 6-6
 - starting, 6-1
 - troubleshooting with, 7-1
- SQLMON Commands
 - EXIT, 9-2
 - HELP, 9-3
 - SET, 9-6
 - SET CYCLE, 9-7
 - SET DBECONNECT, 9-8
 - SET DBEFILESET, 9-9
 - SET DBEINITPROG, 9-10
 - SET DBENVIRONMENT, 9-11
 - SET DISPLAYSAMPLES, 9-12
 - SET ECHO, 9-14
 - SET LOCKFILTER, 9-15
 - SET LOCKOBJECT, 9-18
 - SET LOCKTABFILTER, 9-20
 - SET MENU, 9-21
 - SET OUTPUT, 9-22
 - SET REFRESH, 9-24
 - SET SAMPLING, 9-25
 - SET SORTIODATA, 9-26
 - SET SORTIOLOG, 9-27
 - SET SORTLOAD, 9-28
 - SET SORTLOCK, 9-29
 - SET SORTSAMPLEIO, 9-30
 - SET TOP, 9-31
 - SET USERTIMEOUT, 9-32
- SQLMON Screens
 - IO, 8-2
 - IO Data Program, 8-4
 - IO Data Session, 8-6
 - IO Log Program , 8-8
 - IO Log Session , 8-10
 - Load, 8-12
 - Load Program, 8-14
 - Load Session, 8-16
 - Lock, 8-18

- Lock Impede, 8-20
- Lock Memory, 8-23
- Lock Object, 8-25
- Lock Session, 8-28
- Overview, 8-34
- Overview Program, 8-36
- Overview Session, 8-38
- SampleIO, 8-40
- SampleIO Indexes, 8-42
- SampleIO Objects, 8-44
- SampleIO TabIndex, 8-46
- SampleIO Tables, 8-48
- Static, 8-50
- Static Cluster, 8-51
- Static DBEFile, 8-53
- Static Hash, 8-55
- Static Indirect, 8-57
- Static Size, 8-59
- summary, 6-4
- Static Cluster Screen, 8-51
- Static DBEFile Screen, 8-53
- Static Hash Screen, 8-55
- Static Indirect Screen, 8-57
- Static Screen, 8-50
- Static Size Screen, 8-59
- Static subsystem, 7-23
- storage
 - B-tree indexes, 1-6
 - hash tables, 1-5
 - log data, 1-15
 - user table data on DBEFile pages, 1-3
- subqueries
 - compared with joins, 3-4
- subsystem
 - IO, 7-4
 - Load, 7-7
 - Lock, 7-9
 - navigating in SQLMON, 6-7
 - Overview, 7-1
 - SampleIO, 7-18
 - Static, 7-23
- swapping
 - monitoring, 8-40
 - pages, 1-11
- system administration
 - DBA guidelines, 5-1
 - HP-UX guidelines, 5-15
 - network guidelines, 5-15
- system catalog
 - defined, 1-14
 - DML only mode, 4-1
 - page locks on, 4-2
 - query results, 5-12, 5-13
 - sorting, 5-12
 - user data in, 2-11

SYSTEM.INDEX

- monitoring for cluster count, 2-7

T

- table
 - and DBEFileSets, 2-12
 - checking type, 8-50
 - indexes for large tables, 2-5
 - monitoring I/O, 8-48
 - monitoring locks, 8-31
 - monitoring tasks, 6-14
 - partitioning large tables, 2-3
 - size and row level locking, 4-4
 - storage on DBEFile pages, 1-3
 - suggestions on creation, 2-13
 - temporary, 8-44
- table editor
 - design for, A-1
- temporary space
 - controlling, 5-12
 - in sorting, 1-18, 5-11
 - reported by SQLMON, 8-59
 - used in system catalog, 5-13
- terminal reads
 - avoiding locks around, 4-6, A-2
- TERMINATE USER
 - use of, 5-2
- throttle wait queue
 - monitoring, 8-12
- throughput
 - monitoring, 8-12
- TID
 - and indexes, 1-7
 - defined, 1-4
 - on SampleIO Objects screen, 8-44
 - use of TID scans, 3-11
- timeout
 - and transaction limit, 7-1
 - setting for SQLMON, 9-32
 - using, 5-14
- transaction
 - delays, 7-7
 - design of, 4-1
 - impeded, 8-12
 - limit reached, 7-1
 - maximum number of, 7-7
 - monitoring tasks, 6-16
 - monitoring with Load Screen, 8-12
 - monitoring with Overview Screen, 8-34
 - using short transactions, 4-2
- troubleshooting
 - with SQLMON, 7-1
- TRUNCATE TABLE
 - use of, 2-17
- tuning hints, 9-3

- tuple
 - how stored on DBEFile page, 1-2
- tuple body
 - explained, 1-3
- tuple buffer
 - different from data buffer, 1-10
- tuple header
 - explained, 1-3
- TurboIMAGE
 - storage of table, 8-50
- types of data
 - compatibility of data types, 3-2

U

- UNION
 - and indexes, 3-6
 - avoiding conversions with, 3-6
- UNLOAD
 - to avoid indirect rows, 2-16
 - to improve cluster count, 2-8
 - to remove overflow pages, 2-16
- UPDATE STATISTICS

- cluster count, 2-7
- indexes, 2-10
- invalidating sections, 1-21
- large tables, 2-12
- VALIDATE, 2-10

V

- VALIDATE
 - and UPDATE STATISTICS, 2-10
- validation
 - avoiding at run time, 5-1
 - explained, 1-21
 - stored sections, 1-21
- VARCHAR data
 - and tuple migration, 2-14
- variable length data
 - avoiding, 2-14

W

- wait queue
 - monitoring, 8-12
 - timeouts, 5-14