BEA WebLogic® Integration

Oracle9*i* Database Tuning Guide

Version 10.2
Revised: March 2008

Contents

CHAPTER 1

Introduction

BEA WebLogic Integration (WLI) enables developers to rapidly create complex enterprise business integration applications. Performance and scalability of these applications is closely tied to the performance and efficiency of the database used for managing persistence.

The default Oracle database installation provides average performance to a wide variety of typical database applications. This default configuration can be modified to meet the specific needs of an application such as WLI to achieve better performance and efficiency. This guide provides best practices and guidelines for configuring and tuning the Oracle9*i* database for optimized performance with WLI.

This guide is organized into two sections:

- Database Tuning, which details the specific techniques used to set up and tune the Oracle9*i* database for use with WLI.
- WLI Schema Tuning, which identifies specific target areas within the WLI database schema that tend to be sensitive to database performance issues.

A brief introduction to identifying database performance issues with Oracle Statspack is also provided in Appendix A, "Oracle Statspack".

Target Audience

The target audience of this guide includes:

- Senior application developers using WLI
- Professional services personnel implementing WLI

- Senior DBAs looking for guidance on how to configure Oracle for use with WLI.

Some of the suggestions made in this guide are advanced and should only be implemented by knowledgeable professionals who can accurately gauge the effect(s) such changes would have on the overall system.

References

Oracle9i Database Reference:
http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96536/toc.htm

Oracle9i Database Concepts:
http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96524/toc.htm

Oracle9i Database Performance Tuning Guide and Reference:
http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96533/toc.htm

ORACLE-BASE - Oracle9i Recovery Enhancements:
http://www.oracle-base.com/articles/9i/RecoveryEnhancements9i.php

Partitioning in Oracle. What Why When Who Where How:
http://www.devarticles.com/c/a/Oracle/Partitioning-in-Oracle/1/

CHAPTER 2

Database Tuning

This chapter includes the following sections:

Initialization Parameters

Oracle database makes use of initialization parameters to

- set limits on user and database processes
- enable or disable features
- optimize resource utilization

Some of these parameters can have a significant impact on the performance of a WLI application. This section describes how some of the more important parameters can be tuned to get better performance with WLI.

COMPATIBLE

The `compatible` initialization parameter can be used to set down the compatibility of the instance to a prior release of Oracle. Compatibility should be set to 10.2.0 for use with this tuning guide.

DB_nK_CACHE_SIZE

This parameter sets the cache size for data contained in multiple block size tablespaces. Applications can make use of multiple block size tablespaces to reduce I/O for larger data objects like large objects (LOBs) and index segments. Values for n (in nK) can be 4, 8, 16, or 32 and must be a multiple of the default block size set by DB_BLOCK_SIZE. This parameter should only be set when using multiple block size tablespaces. See Multiple Block Size Buffer Caches, or Caching JPD BLOB Data in WLI Schema Tuning for more information on setting this parameter.

DB_BLOCK_BUFFERS

This is a deprecated parameter. The DB_BLOCK_SIZE parameter should be used instead. Use of this parameter will disable the use of multiple block size tablespaces.

DB_BLOCK_SIZE

This parameter sets the default block size of the database. It can only be set during the creation of the Oracle database. The setting of this parameter for a WLI database should be based on the characteristics of the application created using WLI.

Table 2-1 WLI Application Characteristics and DB_BLOCK_SIZE

WLI Application Characteristics	Block Size
• Messaging (JMS) • Worklist heavy	2k
• Heavy usage of stateful JPDs • Database shared with other applications that require a larger block size	4k

Note: A small default block size can cause ORA-01450 errors for indexes with large key lengths. If the database is shared with another application that has an index with a large key length, the index will have to be moved to a multiple block-size tablespace with a block size large enough to accommodate the larger index key length.

DB_CACHE_SIZE

This parameter sets the size of the default buffer pool for the Oracle database. The default buffer pool is used to cache highly utilized data in memory for faster access. This area should be set appropriately to accommodate 90% of all requests for information from the database as measured by the buffer hit % ratio in Statspack. See Appendix A, “Oracle Statspack” for more information on using Statspack.

DB_FILE_MULTIBLOCK_READ_COUNT

This parameter sets the number of blocks Oracle will request from the I/O subsystem for a sequential read such as a full table scan. This value should be set to 16 for use with WLI. Values greater than 16 increase the likelihood of the Oracle optimizer choosing full scans over index lookups.

DB_KEEP_CACHE_SIZE

This parameter sets the size of the KEEP buffer pool. The KEEP buffer pool is an alternate buffer pool for default block size data objects. This buffer pool can be used to segregate cached data objects from the default buffer pool such as lookup tables that could possibly be aged out of the default buffer pool by more dynamic data. This parameter should be set only if using the KEEP pool. See Caching JPD BLOB Data in WLI Schema Tuning for more information on using alternate buffer pools with WLI.

DB_RECYCLE_CACHE_SIZE

This parameter sets the size of RECYCLE buffer pool. The RECYCLE buffer pool is an alternate buffer pool for default block size data objects. This buffer pool can be used to segregate cached data objects from the default buffer pool such as highly dynamic data that could possibly age data out of the default buffer pool. This parameter should be set only if using the RECYCLE pool. See Caching JPD BLOB Data in WLI Schema Tuning for more information on using this buffer pool with WLI.

DML_LOCKS

This parameter sets the maximum number of simultaneous DML operations that can occur from all concurrent transactions in the database. On very-high-volume transactional database systems (such as WLI), the default value (4 X TRANSACTIONS) may not be enough and can be set to a higher static value. This limit can alternatively be removed completely by setting the value to 0 (zero) but it has the consequence of disabling DROP TABLE, CREATE INDEX, and explicit LOCK statements.

For most WLI implementations, the default value for this parameter is adequate. On systems where a high number of enqueue waits are observed and all other methods of tuning for enqueue waits have been exhausted, this value should be altered. Consult the DBA before altering the default value.

FAST_START_IO_TARGET

This parameter is deprecated in Oracle9*i* and should not be set. Setting this parameter overrides the use of FAST_START_MTTR_TARGET, which is the Oracle-recommended method of limiting instance recovery time in Oracle9*i*.

FAST_START_MTTR_TARGET

This parameter limits the mean time to recovery (MTTR) after a database instance crash. Use of this feature (although seemingly advantageous) can hinder performance on some systems due to the increased contention for I/O while dirty buffers are continuously flushed to disk. On some WLI databases, where I/O is identified as a performance problem, lowering the value for FAST_START_MTTR_TARGET can enhance the performance to a great extent.

HASH_JOIN_ENABLED

WLI performance is slightly improved when hash joins are disabled. This parameter should be set to FALSE when not needed by another application running in the same database instance.

LOG_BUFFER

This parameter sets the amount of memory Oracle uses to buffer entries written to the online REDO log. WLI applications that have a high volume of transactions should set the value of this parameter higher than the default of 512 KB. Values of 1 - 2 MB provide good performance for high volume WLI applications.

LOG_CHECKPOINT_INTERVAL

Setting this parameter will interfere with the correct operation of FAST_START_MTTR_TARGET (the Oracle recommended method of limiting instance recovery time). This parameter should be set to 0 (zero) to allow the checkpoint interval to be controlled by FAST_START_MTTR_TARGET.

LOG_CHECKPOINT_TIMEOUT

Setting this parameter will interfere with the correct operation of FAST_START_MTTR_TARGET (the Oracle recommended method of limiting instance recovery time). This parameter should be set to 0 (zero) to the allow checkpoint interval to be controlled by FAST_START_MTTR_TARGET.

OPTIMIZER_MODE

The Oracle optimizer is responsible for generating the most efficient access paths to data. It can operate in a number of modes including: CHOOSE, RULE, FIRST_ROWS, and ALL_ROWS. WLI performance is greatly improved when the optimizer runs in the CHOOSE mode and

database statistics have been gathered on all database objects. See Database Statistics for more information on gathering database statistics.

PGA_AGGREGATE_TARGET

This parameter sets the target memory size for the Program Global Area (PGA) in Oracle. Use of this parameter in conjunction with WORK_AREA_SIZE_POLICY set to AUTO can increase performance dramatically for memory-intensive SQL operations such as `sort` and `group by`.

In order to have Oracle manage this area of memory automatically, the following parameters must be unset: BITMAP_MERGE_AREA_SIZE, CREATE_BITMAP_AREA_SIZE, HASH_AREA_SIZE, and SORT_AREA_SIZE.

Common values of this parameter for WLI are 32 MB and 64 MB. The value of this parameter can be fine tuned by looking at the PGA Memory Advisory section of the Oracle Statspack report. For information about running and using Oracle Statspack, see Appendix A, “Oracle Statspack”.

PROCESSES

This parameter sets the max number of operating system user processes in Oracle and should be set to a minimum value of 600 for WLI database applications.

SHARED_POOL_SIZE

This parameter sets the amount of memory Oracle dedicates to caching shared cursors, stored procedures and control structures. A common setting of this parameter for WLI is 32 MB. To tune this parameter for optimal performance, see the Shared Pool Advisory section of the Oracle Statspack report. For more information on Oracle Statspack, see Appendix A, “Oracle Statspack”.

UNDO_RETENTION

This parameter sets the amount, in seconds, of UNDO information to be retained in UNDO tablespaces. The retention of large amounts of undo information on a heavily loaded WLI database can place a substantial additional strain on the I/O subsystem. WLI does not use undo retention. Unless the database is being shared with other applications that do make use of this feature, it should be turned off (set to 0).

WORKAREA_SIZE_POLICY

The default setting of this parameter is AUTO. Oracle recommends that this parameter be left as default to allow for the use of automatic SQL work area memory management.

Database Statistics

The Oracle database uses an optimizer to create the most efficient access plans for retrieving data. The ability of the optimizer to select the best plan is strongly influenced by the amount of information (statistics) Oracle has about the underlying data and the performance of the system that will access the data. To give the optimizer the best chance of creating efficient data access plans, statistics should be gathered at the database, schema, and system levels.

This section details various database statistics that can be gathered.

Database Level Statistics

Statistics gathered at database level capture information about the data structures and data for the entire database, including the SYSTEM and SYS schemas. Database level statistics should be gathered after database creation and periodically over the lifetime of the database.

Database level statistics can be gathered using the following script by a user with the SYSDBA system privilege.

```
-- gather database level statistics


begin
   dbms_stats.gather_database_stats
      (
         estimate_percent  => dbms_stats.auto_sample_size,
         block_sample      => FALSE,
         method_opt        => 'FOR ALL INDEXED COLUMNS SIZE AUTO',
         degree            => NULL,
         granularity       => 'ALL',
         cascade           => TRUE,
         stattab           => NULL,
```

```
            statid            => NULL,
            options           => 'GATHER',
            statown           => NULL,
            gather_sys        => TRUE,
            no_invalidate     => FALSE,
            gather_temp       => FALSE
         );
end;
/
```

Schema Level Statistics

Statistics gathered at the schema level only collect statistics on the objects within the target schema. Statistics should be gathered frequently for the WLI schema: at least once per week on low-volume systems and once daily on high volume systems. The need for frequent statistics gathering in the WLI schema is due to the highly dynamic nature of some WLI data structures.

Schema-level statistics can be gathered using the following script by the WLI schema owner or another user with the privileges.

```
-- gather schema level statistics


begin
   dbms_stats.gather_schema_stats
      (
         ownname           => 'WLI_SCHEMA',
         estimate_percent  => dbms_stats.auto_sample_size,
         block_sample      => FALSE,
         method_opt        => 'FOR ALL INDEXED COLUMNS SIZE AUTO',
         degree            => NULL,
         granularity       => 'ALL',
```

```
         cascade           => TRUE,
         stattab           => NULL,
         statid            => NULL,
         options           => 'GATHER',
         statown           => NULL,
         no_invalidate     => FALSE,
         gather_temp       => FALSE
      );
end;
/
```

System Level Statistics

Statistics gathered at the system level collect information about the performance characteristics of the database host OS and its subsystems. In particular, statistics are gathered on I/O performance; CPU performance; and system utilization. These statistics should be gathered while the database is under a typical workload using WLI.

System level statistics can be gathered using the following script by a user with the SYSDBA system privilege.

```
begin
   dbms_stats.gather_system_stats
      (
         gathering_mode    => 'INTERVAL',
         interval          => 60,  -- time in minutes
         stattab           => NULL,
         statid            => NULL,
         statown           => NULL
      );
end;
```

/

Disk I/O

Normally, the slowest part of an Oracle database is its access to persisted data - disk I/O. To increase performance and concurrency for disk I/O, Oracle recommends separating I/O with distinct access characteristics onto separate disks or using high performance storage subsystems that have very high I/O bandwidth. This section addresses these recommendations.

Separating I/O

Oracle recommends separating I/O into seven distinct I/O channels by data type:

1. system data
2. temporary data
3. UNDO and rollback segment data
4. application data
5. application index data
6. REDO log data
7. archive log data

This recommendation would require a minimum of seven disks to run Oracle. Adding redundancy would double this number. This recommendation is a not always practical. Smaller database systems do not generally have more than four disks. Many database systems do not exhibit the same need for these seven distinct I/O channels.

A better way of separating I/O is to identify the distinct access patterns of a database system based on the application running on it. WLI applications have access patterns that are similar to Online Transaction Processing (OLTP) systems with most of the requests for data being small and answered by in-memory data buffers. These data buffers are loaded into memory at first request and then remain in memory until they are aged out by other more frequently used data. This behavior does not put much stress on read I/O for application data or application indexes.

However, WLI does stress the I/O subsystem in the number and type of data writes. These writes are for REDO logs (and archive logs when running in archive log mode), LOB data, UNDO data, application data and application indexes.

For WLI, it is recommended the following data types be stored on physically separate disks or logical units (LUNs) when possible. They are listed in order of importance for separation. Systems that have fewer disks should attempt to separate the earlier data types first.

Table 2-2 Data Types and Disk Separation

Data Type	Importance of Separation
REDO Log Data	WLI applications can produce high volumes of REDO log data. This is the most important item for separation.
Archive Log Data	When the database is operating in ARCHIVE LOG MODE, archive log data will be produced at the same rate as REDO log data. It should be placed on a separate disk whenever possible.
LOB Data	WLI applications that use business process logic will make heavy use of tables that have LOB datatype columns. These datatypes should be stored separately from table data in their own tablespace. When possible, this tablespace should be stored separately on another disk.
UNDO Data	WLI can produce a large amount of UNDO data. UNDO tablespaces should be stored separately when possible.
Application Data	WLI application data is write heavy and can contend with other persisted data. When possible, it should be stored separately.
Application Index Data	WLI application index data is write heavy and can contend with other persisted data. When possible, it should be stored separately.

High Performance Storage Systems

There are a variety of high performance storage subsystems that can increase performance of the Oracle database. These systems achieve very high I/O bandwidth rates by using large striped arrays (RAID 0); redundancy (RAID 1); high-speed connections like Fibre channel; and advanced load balancing algorithms within the storage system. Recommendations for these storage systems are beyond the scope of this document. However, WLI applications have shown increased performance when using an Oracle database with a high-performance storage subsystem. I/O performance can be evaluated in the “File I/O Stats” section of an Oracle Statspack report. See Appendix A, “Oracle Statspack” for more information on using Oracle Statspack.

Reverse Key Indexes

Many database tables have primary or unique keys based on a sequence. These keys are usually indexed by b-tree indexes which, by nature, store the indexed values in order. This behavior of sequential storage gives this type of index the name of “monotonic” or “right-growing” index. These types of indexes can become performance bottlenecks on high-volume transactional systems because of serialization that occurs when inserting values into the leaf-blocks of these indexes.

To avoid this serialization, reverse-key indexes can be used. A reverse-key index stores indexed values in reverse-bit order. So, where the values (234, 235, 236) are stored sequentially and contiguously in a normal b-tree index, they are stored out of sequence and non-adjacent (236, 234, 235) for the reverse-key index (see Table 2-3). Over a larger set, this reversing of the key distributes the indexed values across the leaf-node blocks of the index, thereby eliminating the serialization on sequential inserts.

Table 2-3 Normal and Reverse B-Tree Index

Decimal Representation	Binary Representation	Order
Normal B-Tree Index		
Index Key		
234	11101010	1st
235	11101011	2nd
236	11101100	3rd
Reverse B-Tree Index		
Index Key		
Decimal Representation	Reverse Binary Representation	Order
234	01010111	2nd
235	11010111	3rd
236	00110111	1st

Note: Some caution should be used when choosing to use reverse-key indexes. Once an index is built in REVERSE, it can not be used for index range scans. This means that Oracle

will have to use table scans to answer predicates that define a range of values, as in the following SQL statement:

```
WHERE          salary > 100,000
         AND   salary < 200,000
/
```

To create a reverse-key index, the REVERSE keyword must be used to create or rebuild the index.

The following code sample shows how to create and rebuild an index.

```
-- create the index
CREATE UNIQUE INDEX table_pk
   ON table (column)
   REVERSE
   COMPUTE STATISTICS
/


--rebuild the index
ALTER INDEX table_pk
   REBUILD
   REVERSE
   COMPUTE STATISTICS
```

See WLI_PROCESS_INSTANCE_INFO Table in WLI Schema Tuning for more information on using reverse-key indexes with WLI.

Multiple Block Size Tablespaces

Oracle9*i* introduced a new feature that allowed a single instance of the database to have data structures with multiple block sizes. This feature is useful for databases that need the flexibility of using a small block size for transaction processing applications (OLTP); and a larger block size to support batch processing applications, decision support systems (DSS), or data warehousing. It can also be used to support more efficient access to larger data types like LOBs.

To create a multiple block size tablespace, the keyword BLOCKSIZE must be used when creating the tablespace, as shown in the following code sample.

```
-- create wli_lob_data tablespace
```

```
CREATE TABLESPACE wli_lob_data
   LOGGING
   DATAFILE '/oracle/oradata/perfdb01/wli_lob_data_01.dbf'
      SIZE 1000M REUSE
   BLOCKSIZE 16K
   EXTENT MANAGEMENT LOCAL
   UNIFORM SIZE 50M
   SEGMENT SPACE MANAGEMENT AUTO
/
```

See JPD Tables in WLI Schema Tuning for more information on using multiple block size tablespaces with WLI.

Note: A multiple block size buffer cache must be created before a multiple block size tablespace can be created. See Multiple Block Size Buffer Caches for information on multiple block size buffer caches.

Multiple Block Size Buffer Caches

To cache multiple block size data, Oracle9*i* has multiple block size buffer caches. These caches are used to buffer reads for data contained in multiple block size tablespaces.

Multiple block size caches can be created by running the following statement by a user with privileges, as shown in the following code sample.

```
-- create 16K block size cache

ALTER SYSTEM
   SET db_16k_cache_size = 64M
   SCOPE = BOTH
/
```

See JPD Tables in WLI Schema Tuning for more information on using multiple block size tablespaces with WLI.

LOB Tuning

LOB tuning includes tuning caching and setting appropriate physical storage parameters.

Caching

By default, LOB data is not cached in Oracle. Caching LOB data can have a significant positive effect on LOB access performance. However, caching LOB data in the DEFAULT pool can cause other application data to be quickly aged out. It is recommended that you cache LOB data in an alternate pool such as the RECYCLE or KEEP pools, or in a multiple block size cache when using a multiple block size tablespace.

LOB caching can be enabled by creating or altering a table to use it, as shown in the following code sample.

```
-- create table foo with LOB caching
CREATE TABLE foo
   (
      bar     NUMBER(16),
      baz     BLOB
   )
   TABLESPACE wli_data
   LOB (baz)
      STORE AS
         (
            CACHE
         )
/

-- alter table foo to use LOB caching
ALTER TABLE foo
   MODIFY LOB (baz) (CACHE)
/
```

Physical Storage Parameters

Setting the CHUNK parameter and disabling STORAGE IN ROW improve the database performance.

CHUNK

The CHUNK parameter sets the amount of data to be operated on at one time for a LOB in bytes. This value has to be set to a multiple of the block size for the LOB. Depending on the average data size of the LOBs stored, this value should be set as large as possible or until it exceeds the average size of the data stored for the LOB column.

To find the average LOB length for a table, use the following SQL statement.

```
-- get the average length (in bytes) of the bas LOB column in table foo

SELECT AVG(DBMS_LOB.GETLENGTH(baz)) avg_lob_len

FROM foo

/


AVG_LOB_LEN

-----------

  13171.712
```

In the preceding example, the average LOB length for table foo is 13171.712 bytes, or ~13K. Setting the CHUNK size to 16K would make the average number of I/Os per request for LOB data from foo ~1.

DISABLE STORAGE IN ROW

LOB data can be stored in-line with the table's row (in the same segment) or can be stored in its own segment. Storing LOB data in its own segment can increase the efficiency and performance of data access, particularly when coupled with the storage of LOB data in a larger block size tablespace.

See JPD Tables in WLI Schema Tuning for more information on using LOB tuning with WLI.

Partitioning

The Oracle9*i* database has a feature whereby tables can be partitioned into smaller manageable pieces. Each of these pieces is stored in a separate physical data segment. Partitioning is transparent to the application and partitioned tables can be treated the same as standard non-partitioned tables. There are three basic methods by which a table can be partitioned: range, hash, and list. Only hash partitioning will be described in this document.

With hash partitioning, a table is sub-divided into a specified number of partitions by the hash of a key value found in the table. This partitioning, with the selection of a good value for the hash, serves to equally distribute data across the partitions of the table. In a busy table that suffers resource contention problems (high row lock waits, buffer busy waits) this type of tuning can have a very positive effect on performance.

To partition an existing table, a new partitioned table must be created and the data from the old table must be copied to the new table, as shown in the following code sample.

```
-- create non-partitioned table
CREATE TABLE foo
   (
      bar      NUMBER(16),
      baz      BLOB,
      CONSTRAINT foo_pk
         PRIMARY KEY (bar)
   )
   TABLESPACE users
/


-- create new partitioned version of foo with data from foo
CREATE TABLE new_foo
   (
      bar,
      baz,
```

```
      CONSTRAINT new_foo_pk
         PRIMARY KEY (bar)
   )
   TABLESPACE users
   PARTITION BY HASH (bar)
   PARTITIONS 32
   AS SELECT * FROM foo
/

-- drop the original foo table
DROP TABLE foo
/

-- rename new_foo table to foo
RENAME new_foo TO foo
/
```

See WLI_PROCESS_INSTANCE_INFO Table in WLI Schema Tuning for more information on using partitioning with WLI.

WLI Schema Tuning

In this section, we highlight typical areas within the WLI schema that can potentially perform better with tuning, depending on the application architecture of the target WLI application. The tuning techniques described in this section impose incremental costs in terms of database system resources. While benefits from these tuning techniques usually outweigh the additional cost of resources, these techniques should only be applied when a specific performance problem is identified. See Oracle Statspack for a list of commonly identifiable database performance issues found with WLI applications.

This chapter includes the following sections:

JPD Tables

In WLI, business process logic is implemented using Java Process Definitions (JPDs). JPD information is persisted in JPD_PROCESS tables in the WLI schema. These tables can be tuned to accommodate a much higher level of concurrency and throughput by applying some database tuning techniques.

One of the main areas where JPD_PROCESS tables derive a large performance increase is from modifying the storage characteristics for the BLOB data column, CG_DATA. This column contains the serialized byte array representing the JPD instance.

BLOB storage in a JPD table suffers from two common problems with LOBs in Oracle: not caching the BLOB data and storing BLOB data in-line with table data. To remove these bottlenecks, the JPD table's BLOB column should be cached and stored in a separate tablespace.

Caching JPD BLOB Data

JPD BLOB caching should be enabled in WLI database where a JPD table has been identified as a bottleneck. BLOB data can be cached in Oracle's DEFAULT pool, KEEP pool, RECYCLE pool, or an alternate block size cache. Caching LOB data in the DEFAULT pool can have a negative effect on performance because it will compete for space with the most commonly cached data in the database. For this reason, LOB data should be cached in one of the alternate buffer pools.

To cache JPD BLOBs, a target buffer pool must be identified or created, and the JPD table must be created or altered to use the cache.

The following code samples show how to create a RECYCLE pool, and to alter and create the `jpd_process_table`.

```
-- create RECYCLE pool
ALTER SYSTEM
   SET db_recycle_cache_size = 64M
   SCOPE = BOTH
/


-- alter existing JPD table to use RECYCLE pool


ALTER TABLE jpd_processes_table
   MODIFY LOB (cg_data)
      (
         CACHE
         STORAGE
            (
               BUFFER_POOL RECYCLE
```

```
            )
      )
/

-- create new JPD table to use RECYCLE pool

CREATE TABLE jpd_processes_table
   (
      cg_id                VARCHAR2(768 byte)      NOT NULL,
      last_access_time     NUMBER(19),
      cg_data              BLOB,
      CONSTRAINT jpd_proceses_table_pk
         PRIMARY KEY(cg_id)
            USING INDEX TABLESPACE wli_index
   )
   TABLESPACE wli_data
   LOB(cg_data) STORE AS
      (
         CACHE
         STORAGE
            (
               BUFFER_POOL RECYCLE
            )
      )
/
```

See LOB Tuning in Database Tuning for more information on LOB caching.

Separate Tablespace for BLOBs

A dedicated tablespace for WLI LOB data should be created. This tablespace can be created with either the default database block size or an alternate larger block size. Larger block sizes can increase performance for LOB data access.

The following code samples show how to create default and alternate block size table space:

```
-- create default block size tablespace
CREATE TABLESPACE wli_lob_data
   DATAFILE '/u03/app/oracle/oradata/wlidb1/wli_lob_data01.dbf' SIZE 1000M
   EXTENT MANAGEMENT LOCAL
   UNIFORM SIZE 50M
   SEGMENT SPACE MANAGEMENT AUTO
/

-- create alternate block size tablespace
CREATE TABLESPACE wli_lob_data
   DATAFILE '/u03/app/oracle/oradata/wlidb1/wli_lob_data01.dbf' SIZE 1000M
   BLOCKSIZE 16K
   EXTENT MANAGEMENT LOCAL
   UNIFORM SIZE 50M
   SEGMENT SPACE MANAGEMENT AUTO
/
```

To store BLOB data from the JPD tables in a separate tablespace, the JPD table must be created or moved with the TABLESPACE storage parameter set to the alternate tablespace.

The following code samples show how to create a JPD table and alter an existing table:

```
-- create the a new JPD table

CREATE TABLE jpd_processes_table
   (
```

```
      cg_id                VARCHAR2(768 byte)      NOT NULL,
      last_access_time     NUMBER(19),
      cg_data              BLOB,
      CONSTRAINT jpd_proceses_table_pk
         PRIMARY KEY(cg_id)
         USING INDEX TABLESPACE wli_index
   )
   TABLESPACE wli_data
   LOB(cg_data) STORE AS
      (
         TABLESPACE wli_lob_data
         DISABLE STORAGE IN ROW
         CACHE
      )
/

-- alter an existing JPD table
ALTER TABLE jpd_processes_table
   MOVE
   LOB(cg_data)
      STORE AS
         (
            DISABLE STORAGE IN ROW
            TABLESPACE wli_lob_data
            CACHE
         )
/
```

See Multiple Block Size Tablespaces in Database Tuning for more information on multiple block size tablespaces.

WLI_PROCESS_INSTANCE_INFO Table

The WLI_PROCESS_INSTANCE_INFO table is updated on every persistent change in the JPD. In some applications built with WLI, this table can become a performance bottleneck due to a large number of concurrent inserts. Two tuning techniques that have had a positive effect on the performance of this table are: adding a reverse-key index and partitioning the table by hash.

Reverse Key Index

The primary key index of the WLI_PROCESS_INSTANCE_INFO table is populated by a sequence. This sequential population causes the index on the primary key to be right-growing and suffer performance problems when under heavy concurrent load. Reversing the index on the primary key alleviates this problem by removing the serialization that occurs in the index.

To reverse the index for the primary key of the WLI_PROCESS_INSTANCE_INFO table, the index has to be altered or the table has to be recreated.

The following code sample shows how to reverse the index.

```
-- rebuild the index reverse


ALTER INDEX pk_wli_process_instance_info
   REBUILD
   REVERSE
   COMPUTE STATISTICS
/
```

See Reverse Key Indexes in Database Tuning for more information on using reverse-key indexes.

Partitioning

As the concurrent demand for access to the WLI_PROCESS_INSTANCE_INFO table grows in a high-volume WLI application, contention can begin to occur at the block level. Partitioning the table decreases this contention by distributing table data across many physical partitions, thereby reducing the likelihood that concurrent transactions will try to access the same physical block.

To partition the WLI_PROCESS_INSTANCE_INFO table, it has to be recreated. Partition values should be set to powers of two. Good performance has been observed with partition values of 32 and 64.

The following code sample shows how to partition the WLI_PROCESS_INSTANCE_INFO table.

```
-- create partitioned WLI_PROCESS_INSTANCE_INFO table

CREATE TABLE WLI_PROCESS_INSTANCE_INFO

   (

      PROCESS_INSTANCE        VARCHAR(768) NOT NULL,

      PROCESS_TYPE            VARCHAR(200) NOT NULL,

      PROCESS_LABEL           VARCHAR(1000),

      PROCESS_STATUS          SMALLINT     NOT NULL,

      PROCESS_START_TIME      NUMBER       NOT NULL,

      PROCESS_END_TIME        NUMBER,

      SLA_EXCEED_TIME         NUMBER,

      SEQUENCE_ID             INTEGER      NOT NULL,

      CONSTRAINT PK_WLI_PROCESS_INSTANCE_INFO

         PRIMARY KEY(PROCESS_INSTANCE)

            USING INDEX TABLESPACE wli_index

   )

   PARTITION BY HASH (PROCESS_INSTANCE) PARTITIONS 64

   TABLESPACE wli_data

/
```

See Partitioning in Database Tuning for more information on partitioning tables.

WLI_PROCESS_EVENT Table

The WLI_PROCESS_EVENT table contains detailed tracking information that describes the events that occurred within a JPD. At the end of a JPD transaction, all the events generated during that transaction are sent to a JMS queue and are written to the WLI_PROCESS_EVENT table.

The number of events actually generated depends on the complexity of the JPD and the TrackingLevel set through the OA&M console. With the TrackingLevel set at its most verbose setting, contention for access to this table can degrade system performance. This performance degradation can be alleviated by partitioning the WLI_PROCESS_EVENT table.

Partitioning

As the number of events being tracked increases, contention for access to the WLI_PROCESS_EVENT table at the block level also increases. Partitioning the table decreases this contention by distributing table data across many physical partitions, thereby reducing the likelihood that concurrent transactions will try to access the same physical block.

To partition the WLI_PROCESS_EVENT table, it has to be recreated. Partition values should be set to powers of two. Good performance has been observed with partition values of 32 and 64.

The following code sample shows how to partition the WLI_PROCESS_EVENT table.

```
-- create partitioned WLI_PROCESS_EVENT table

CREATE TABLE wli_process_event

   (

      process_type            VARCHAR(200) NOT NULL,

      process_event_id        VARCHAR(60)  NOT NULL,

      process_instance        VARCHAR(768) NOT NULL,

      deployment_id           INTEGER      NOT NULL,

      event_time              INTEGER      NOT NULL,

      activity_id             SMALLINT     NOT NULL,

      event_type              SMALLINT     NOT NULL,

      event_data              BLOB,

      process_label           VARCHAR(1000),

      is_rolled_back          SMALLINT     NOT NULL,

      event_elapsed_time      NUMBER,

      start_event_id          VARCHAR(60),

      event_count             INT,

      CONSTRAINT pk_wli_process_event
```

```
      PRIMARY KEY (process_instance, process_event_id)
         USING INDEX TABLESPACE wli_index
   )
   PARTITION BY HASH (process_instance, process_event_id) PARTITIONS 64
   TABLESPACE wli_data
/
```

APPENDIX A

Oracle Statspack

Statspack is a performance tuning tool provided by Oracle with the Oracle9*i* database distribution. With minimal effort, it can be installed on any Oracle9*i* database to quickly gather detailed analysis of the performance of that database instance. This appendix describes in brief: Installing the Statspack, Collecting Snapshots, Generating Reports, and identifying the Top WLI Database Bottlenecks.

Installing the Statspack

Installation of the Oracle Statspack tool is a relatively simple process. The following is a step-by-step guide to the process of installing Oracle Statspack on a UNIX system.

1. Navigate to the `$ORACLE_HOME/rdbms/admin` directory as follows:

   ```
   # cd $ORACLE_HOME/rdbms/admin/
   ```

2. Start the Statspack install script, `spcreate.sql`, as follows:

   ```
   # sqlplus "/ as sysdba" @spcreate.sql
   ```

3. Enter a password for the PERFSTAT user when prompted.
4. Enter the default tablespace (tools) for the PERFSTAT user when prompted.
5. Enter the temporary tablespace (temp) for the PERFSTAT user when prompted.
6. Exit sqlplus as follows:

   ```
   SQL> exit
   ```

Collecting Snapshots

Once the Oracle Statspack tool is installed, snapshots must be collected to evaluate database performance. Snapshots are moment-in-time collections of all of the database statistics that the Oracle database continuously collects. Once two snapshots are collected, they can be compared to identify the activity that occurred during the interval between the two snapshots.

Snapshots can be collected a various levels, each increasing level collecting a greater amount of information about the database. As the levels go higher, each level is inclusive of the information collected at the levels below it.

Table A-1 Levels of Statistics

Level	Information Collected
0	General Performance Statistics
5	Addition Data: SQL Statements
6	Addition Data: SQL Plans and SQL Plan Usage
7	Addition Data: Segment Level Statistics
10	Addition Data: Parent and Child Latches

To collect statistics

1. Connect to the database as the PERFSTAT user as follows:

   ```
   sqlplus perfstat/<password>
   ```

2. Create a snapshot with the statspack package as follows:

   ```
   SQL> execute statspack.snap(i_snap_level=>7);
   ```

3. Exit SQLPLUS as follows:

   ```
   SQL> exit
   ```

Generating Reports

Oracle Statspack comes with a comprehensive reporting script called `spreport.sql`. When this script is run, it outputs a list of available snapshots, asks the user for two snapshot IDs and a name for the report, and then outputs a text report of the results.

To run a Statspack report.

1. Navigate to the `$ORACLE_HOME/rdbms/admin` directory as follows:

   ```
   # cd $ORACLE_HOME/rdbms/admin/
   ```

2. Run the standard Statspack report as follows:

   ```
   # sqlplus perfstat/<password> @spreport
   ```

   – Enter a beginning snapshot ID.

   – Enter an ending snapshot ID.

   – Enter a name for the report or accept the default.

   – Exit SQLPLUS as follows:

   ```
   SQL> exit
   ```

Top WLI Database Bottlenecks

Oracle Statspack is capable of identifying all of the common database performance bottlenecks that have been observed with WLI. This section describes the top WLI database performance bottlenecks and how they are identified in the Oracle Statspack report, and provides recommendations to work around them

Enqueue Waits

Enqueues are local locks that serialize access to various resources. This wait event indicates a wait for a lock that is held by another session (or sessions) in an incompatible mode to the requested mode.

The action to take to reduce enqueue waits depends on the lock type that is causing the wait.

Types of Locks

There are three types of locks that predominantly cause enqueue waits - TX, TM, and ST.

- TX (Transaction Lock): The TX lock is acquired when a transaction initiates its first change and is held until the transaction does a COMMIT or ROLLBACK. It is used mainly as a queuing mechanism so that other resources can wait for a transaction to complete.
- TM (DML Enqueue): This lock/enqueue is acquired when performing an insert, update, or delete on a parent or child table.

- ST (Space management Enqueue): These enqueues are caused if a lot of space management activity is occurring on the database (such as small extent size, several sortings occurring on the disk).

Identification and Recommendations

Enqueue waits and their types can be identified by looking at the “Enqueue activity” section of the Statspack report.

For the WLI application, enqueue waits are primarily found for indexed monotonic keys and data block access on the WLI_PROCESS_INSTANCE_INFO table. Enqueue waits can be reduced on these objects by using reverse-key indexes and by partitioning the WLI_PROCESS_INSTANCE_INFO table. See WLI Schema Tuning for more information on using reverse-key indexes and partitioning.

Log File Sync

When a user session COMMITs (or rolls back), session REDO information needs to be flushed to the REDO log file. The user session will post the log writer (LGWR) to write all REDO information required from the log buffer to the REDO log file. When the LGWR has finished, it posts the user session. The user session waits on this wait event while waiting for LGWR to post it back to confirm all the REDO changes are safely on disk.

Identification and Recommendations

Waits on log file sync can be identified by looking at the “Top 5 Timed Events” or “Wait Events” section of the Statspack report.

These waits can be reduced by moving log files to the faster disks or by reducing COMMIT frequency by performing batch transactions.

Buffer Busy Waits

Buffer busy waits happen when a session needs to access a database block in the buffer cache but cannot, because the buffer is “busy”. The two main cases where this can occur are:

- Another session is reading the block into the buffer.
- Another session holds the buffer in an incompatible mode to this request.

Identification and Recommendations

Segments with high buffer busy waits can be identified by looking in the “Top 5 Buf. Busy Waits per Segment” section of the Statspack report.

Buffer busy waits can be reduced by using reverse-key indexes for busy indexes and by partitioning busy tables. See WLI Schema Tuning for more information on using reverse-key indexes and partitioning.

Log File Parallel Writes

Log file parallel write waits occur when waiting for writes of REDO records to the REDO log files to complete. The wait occurs in log writer (LGWR) as part of normal activity of copying records from the REDO log buffer to the current online log.

The actual wait time is the time taken for all the outstanding I/O requests to complete. Even though the writes may be issued in parallel, LGWR needs to wait for the last I/O to be on disk before the parallel write is considered complete. Hence the wait time depends on the time it takes the OS to complete all requests.

Identification and Recommendations

Waits for log file parallel writes can be identified by looking at the “Top 5 Timed Events” or “Wait Events” section of the Statspack report.

Log file parallel write waits can be reduced by moving log files to the faster disks and/or separate disks where there will be less contention.

DB File Sequential Reads

DB file sequential read waits signify a wait for an I/O read request to complete. This call differs from ‘DB file scattered reads’ in that a sequential read reads data into contiguous memory (whereas a scattered read reads multiple blocks and scatters them into different buffers in the SGA). If the time spent waiting for reads is significant, then it can be helpful to determine which segments Oracle is performing the reads against.

Identification and Recommendations

Segments that are excessive on reads can be identified by looking at the “Top 5 Physical Reads per Segment” and “SQL ordered by Reads” sections of the Statspack report.

Block reads are fairly inevitable so the aim should be to minimize unnecessary I/O. I/O for sequential reads can be reduced by tuning SQL calls that result in full table scans and using the partitioning option for large tables.

DB File Scattered Reads

DB file scattered read waits happens when a session is waiting for a multi-block I/O to complete. This typically occurs during full table scans or index fast full scans.

Identification and Recommendations

Segments that are excessive on reads can be identified by looking at the “Top 5 Physical Reads per Segment” and “SQL ordered by Reads” sections of the Statspack report.

Ideally, applications should not repeatedly perform full table scans of the online portions of application data when there is a faster and more selective way to retrieve the data. Query tuning should be used to optimize online SQL to use indexes.

Buffer Hit Ratio

The buffer hit ratio metric shows how often processes are finding data blocks in memory vs. retrieving them from disk.

Identification and Recommendations

Buffer hit ratio can be found in the “Instance Efficiency Percentages” section for the Statspack report.

The exact value of the buffer hit ratio is of less importance than the ability to monitor it over time and notice any significant changes in the profile of activity on the database. If the ratio falls below 80%, then more memory should be allocated to the database by increasing the value of the DB_CACHE_SIZE parameter.

In some cases, the ratio can be low due to poorly performing SQL statements. In this case, the buffer hit ratio may not increase after increasing DB_CACHE_SIZE. These SQL statements should be tuned to avoid excessive physical I/O.

Row Lock Waits

Row lock waits occur when a process requests an incompatible lock for a row that is currently locked by another process. These lock waits can usually be attributed to high volume inserts on a table with a primary key index.

Identification and Recommendations

Segments where performance suffers from excessive row lock waits can be identified in the “Top 5 Row Lock Waits per Segment” section of the Statspack report.

These waits can be avoided by partitioning tables or by using reverse-key indexes. For WLI, these waits can be found on the WLI_PROCESS_INSTANCE_INFO table and on the primary key index of this tables. See WLI Schema Tuning for more information on using reverse-key indexes and partitioning.

Library Hit Ratio

The library cache hit ratio indicates how often Oracle retrieves a parsed SQL or PL/SQL statement from the library cache. When an application makes a SQL or stored procedure call, Oracle checks the library cache to determine if a parsed version of the statement is already stored there. If the parsed statement is stored in the library cache, Oracle executes the statement immediately. If not, Oracle parses the statement and allocates a shared SQL area within the library cache for it. A low library cache hit ratio can result in additional parsing, which decreases performance and increases CPU consumption for the database.

Identification and Recommendations

The library hit ratio can be found in the “Instance Efficiency Percentages” section of the Statspack report.

If this ratio falls below 80%, increasing the size of shared pool area can help. This can be done by changing the value of the SHARED_POOL_SIZE parameter.