



OPTIMIZING MYSQL™ DATABASE APPLICATION PERFORMANCE WITH SOLARIS™ DYNAMIC TRACING

Luojia Chen, ISV Engineering

Sun BluePrints™ On-Line

Part No 820-7239-10
Revision 1.0, 1/6/09

Table of Contents

Introduction	1
Approaching MySQL Database Application Tuning	3
The Advantages of Solaris Dynamic Tracing	3
Simplifying and Speeding Performance Tuning Efforts	6
Analyzing Query Loads	6
Probing the Cost of File Sort Operations	9
Profiling the Use of Stored Procedures	11
Observing Slave Queries	12
Optimizing Use of the MySQL Database Query Cache	14
Putting it all Together	17
For More Information	18
About the Author	18
Related Resources	18
Ordering Sun Documents	18
Accessing Sun Documentation Online	18

Chapter 1

Introduction

Today business is increasingly done on the Web, and thousands of new people, applications, and services are coming online daily. In fact, Wiki pages, mashups, social networking sites, and online stores are at the forefront of Web 2.0 technologies. As more companies, services, and sites go online and gain in popularity, enterprises must deal with the massive increases in data, as well as collected community knowledge and shared information.

When information is readily available and secure, it can help make the organization smarter, and more effective at solving business challenges. As a result, efficient and flexible environments that can scale and adapt, deploy new services quickly, and keep valuable information safe are paramount. To support this effort, Web 2.0 companies need easy access to an open, integrated platform that can help developers build and deploy high-performance, reliable Web services and applications fast. By using a complete SAMP (Solaris™ Operating System (OS), Apache, MySQL™ database, Perl) application stack along with high-performance servers and storage systems, organizations are better positioned to create environments that are capable of supporting rapidly evolving, high traffic Web sites.

Part of a series, this Sun BluePrints™ article describes how taking advantage of Solaris Dynamic Tracing (DTrace) probes can help simplify MySQL database application tuning. Through examples, this document shows some of the specific aspects of MySQL database server operation that can be observed through DTrace probes. This article assumes a basic understanding of the Solaris OS, DTrace, the MySQL database server, and database application development while addressing the following topics:

- Chapter 2, “Approaching MySQL Database Application Tuning”, discusses the importance of evaluating database performance and introduces DTrace.
- Chapter 3, “Simplifying and Speeding Performance Tuning Efforts”, provides a detailed review of utilizing DTrace probes to help analyze various aspects of MySQL database application performance.
- Chapter 4, “Putting it all Together”, summarizes the main points presented in this paper.
- Chapter 5, “For More Information”, includes links to references.

Developers looking to deploy and optimize environments with SAMP technology can also take advantage of the information in other articles in the series, including:

- *Addressing Systemic Qualities in SAMP Architectures*, an article that defines the set of basic systemic qualities (performance, scalability, availability, extensibility, interoperability, and security) and introduces best practices for optimizing Web 2.0 solutions accordingly.
- *Deploying the SAMP Stack as Software as a Service*, an article that explains how to deploy the SAMP stack on the Solaris OS in a virtualized environment.
- *Running MySQL Enterprise Database in Solaris Containers*, an article that describes the process of deploying the MySQL database in virtualized environments using Solaris Zones.

Chapter 2

Approaching MySQL Database Application Tuning

The MySQL database continues to gain popularity across a wide variety of organizations. Many types of Web 2.0 applications including social-networking, video sharing, wiki, and blog sites are powered by MySQL software. Given the interactive nature of online applications, performance and responsiveness for MySQL database applications is critical to capturing and keeping the attention of users. For example:

- Long delays while searching for a particular Web entry may prompt users to seek the information elsewhere.
- Extended download times for video clips can minimize site popularity and expansion efforts.
- Stalled queries on a social-networking site can leave users with the impression that the site is broken or difficult to use.

Rapid growth, new features, or changes in usage patterns of database applications can all lead to performance levels that fall short of expectations. Careful problem analysis before attempting software changes or configuration adjustments often facilitates faster problem resolution. MySQL database log files and built-in command line tools can help get the trouble shooting process started. However, these tools generally point to high-level problem characteristics rather than underlying root causes.

The Advantages of Solaris Dynamic Tracing

DTrace is a dynamic tracing framework that helps organizations simplify the process of identifying the source of intermittent and sustained application performance problems. With DTrace, administrators and application developers can instrument a live operating system kernel and running applications without rebooting the kernel or recompiling — or even restarting — applications. Instrumentation can be activated as desired, leaving no overhead when tracing is turned off.

The characteristics and features of DTrace ease adoption of this performance troubleshooting approach. DTrace scripts are written in a language similar to C, simplifying the learning process for most system administrators and application developers. Within DTrace scripts, probes record data and set timestamps at locations of interest throughout execution. The Solaris 10 OS includes nearly 40,000 built-in probes that are points of instrumentation in the Solaris kernel. Embedded DTrace probes are also available for instrumentation of specific applications, including MySQL database version 6.0. To further speed analysis, DTrace scripts can collect information into a table and return aggregated values such as the average, minimum, maximum, or sum for a specific variable.

Utilizing custom and embedded DTrace probes significantly improves MySQL database application observability and helps administrators speed the process of identifying the root cause of performance issues. Powerful DTrace probes can help an administrator clarify the amount of time spent on each aspect of a database operation including processing the query, waiting on locks and latches, performing disk I/O, and transporting data. Custom DTrace probes can be created to dive even further and measure the cost of individual processing steps.

Available embedded DTrace probes for MySQL database version 6.0.8 include:

```
provider mysql {
    probe command__start(unsigned long conn_id, int command,char *user,
        char *host);
    probe command__done(int status);
    probe insert__row__start(char *db, char *table);
    probe insert__row__done(int status);
    probe update__row__start(char *db, char *table);
    probe update__row__done(int status);
    probe delete__row__start(char *db, char *table);
    probe delete__row__done(int status);
    probe handler__rdlock__start(char *db, char *table);
    probe handler__wrlock__start(char *db, char *table);
    probe handler__unlock__start(char *db, char *table);
    probe handler__rdlock__done(int status);
    probe handler__wrlock__done(int status);
    probe handler__unlock__done(int status);
    probe filesort__start(char *db, char *table);
    probe filesort__done(int status, unsigned long rows);
    probe select__start(char *query);
    probe select__done(int status, unsigned long rows);
    probe insert__start(char *query);
    probe insert__done(int status, unsigned long rows);
    probe insert__select__start(char *query);
    probe insert__select__done(int status, unsigned long rows);
    probe update__start(char *query);
    probe update__done(int status, unsigned long rowsmatches,
        unsigned long rowschanged);
    probe multi__update__start(char *query);
    probe multi__update__done(int status, unsigned long rowsmatches,
        unsigned long rowschanged);
    probe delete__start(char *query);
    probe delete__done(int status, unsigned long rows);
    probe multi__delete__start(char *query);
    probe multi__delete__done(int status, unsigned long rows);
    probe query__cache__hit(char *query, unsigned long rows);
    probe query__cache__miss(char *query);
    probe query__start(char *query,unsigned long connid,char *db_name,
        char *user, char *host);
    probe query__done(int status);
    probe query__parse__start(char *query);
    probe query__parse__done(int status);
    probe query__exec__start(char *query, unsigned long connid,
        char *db_name,char *user,
        char *host,int exec_type);
    probe query__exec__done(int status);
    probe net__read__start();
    probe net__read__done(int status, unsigned long bytes);
```

```
probe net__write__start(unsigned long bytes);  
probe net__write__done(int status);  
  
};
```

To make the above DTrace probes available, include `--enable-dtrace` as a configuration option during the MySQL database build. After starting the MySQL database, the list of embedded DTrace probes can be obtained with the following command:

```
#dtrace -l |grep mysql
```

Chapter 3

Simplifying and Speeding Performance Tuning Efforts

Custom and embedded MySQL database DTrace probes help guide optimization efforts. By writing simple DTrace scripts, administrators can gain application performance insights that can not easily be obtained by other means. Data points gathered by DTrace probes help with answers to questions such as:

- Which queries impose the heaviest load and can be targeted for optimization?
- Are any queries executing with enough frequency to generate a sizeable cumulative load?
- How much time is a particular query spending on file sorting?
- Do any stored procedures generate excessive load? If so, which SQL statements are executing within that stored procedure?
- Which queries contribute the most to replication lag?
- Can the application take better advantage of cache and how?

Analyzing Query Loads

Tuning inefficient queries can return responsiveness and viability to an application. Prioritizing tuning efforts by identifying queries that impose the greatest system load can lead to gaining the highest rewards. The MySQL database has a built-in feature to help administrators identify good candidates for optimization — the slow query log. This log creates entries for queries that take longer to process than the user-definable variable `long_query_time`. However, limitations exist with this approach, including:

- The slow query log lists the slow queries in executing order — not actual processing time — making it difficult for administrators to identify the greatest consumers and properly prioritize the order of optimization efforts.
- In some cases, queries that run frequently generate a greater load and impose more significant performance delays than the single longest running query. The slow query log only creates entries for queries that surpass `long_query_time`, ignoring the cumulative impact of queries that execute frequently.

The embedded MySQL database DTrace probes `query-start` and `query-end` can help administrators overcome these two limitations and efficiently tackle the prioritization issue. Utilizing these probes, administrators can create scripts that count the number of times a query is utilized as well as the total processing time for each query.

The following script, `query_load.d`, generates a report that lists queries in order of least to most frequently executed:

```
#cat query_load.d
#!/usr/sbin/dtrace -s
#pragma D option quiet
dtrace::BEGIN
{
    printf("Tracing... Hit Ctrl-C to end.\n");
}

mysql*:::query-start
{
    this->query = copyinstr(arg0);
    this->who   = strjoin(copyinstr(arg3),strjoin("@",copyinstr(arg4)));
    this->connid = arg1;
}

mysql*:::query-done
{
    @queries[this->who,this->connid,this->query] = count();
}

dtrace::END
{
    printf(" %-18s %5s %s %s\n", "USER@HOST","CONNECT_ID", "COUNT",
"QUERY");
    printa(" %-18s %6d %@5d %s\n", @queries);
}
}
```

Sample output of the `query_load.d` script is shown here:

```
# ./query_load.d
Tracing... Hit Ctrl-C to end.

USER@HOST  CONNECT_ID  COUNT  QUERY
root@localhost4  247  SELECT * FROM city_huge ORDER BY population DESC LIMIT 3
root@localhost4  249  INSERT INTO city_huge (Name, CountryCode, District, Population)
SELECT Name, CountryCode, District, Population FROM City LIMIT 1
root@localhost4  249  UPDATE city_huge SET population=population+1 WHERE id = last_insert_id()
root@localhost4  500  SET @var = floor(rand()) * (SELECT max(id) FROM city_huge)
root@localhost4  500  UPDATE city_huge SET population = population + 1 WHERE id = @var
root@localhost4  502  SELECT * FROM city_huge WHERE id = @var
root@localhost4  749  COMMIT
root@localhost4  750  BEGIN
```

By utilizing timestamps and the DTrace aggregation `sum` function, the `query_stime.d` script generates a report that lists queries in order of cumulative processing time:

```
#cat query_stime.d
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
    printf("Tracing... Hit Ctrl-C to end.\n");
}

mysql*:::query-start
{
    this->query = copyinstr(arg0);
    this->who   = strjoin(copyinstr(arg3),strjoin("@",copyinstr(arg4)));
    this->connid = arg1;
    this->querystart = timestamp;
}

mysql*:::query-done
{
    this->elapsed = (timestamp - this->querystart) /1000000;
    @time[this->who,this->connid,this->query] = sum(this->elapsed);
}

dtrace:::END
{
    printf(" %-18s %5s %10s %10s\n", "USER@HOST","CONNECT_ID", "QUERY ms",
"QUERY");
    printa(" %-18s %6d %@5d %s\n", @time);
}
}
```

The output of the `query_stime.d` script lists queries from shortest to longest cumulative processing time:

```
# ./query_stime.d
Tracing... Hit Ctrl-C to end.

USER@HOST  CONNECT_ID  ms  QUERY
root@localhost4  0  BEGIN
root@localhost4  0  COMMIT
root@localhost4  1  INSERT INTO city_huge (Name, CountryCode, District, Population)
SELECT Name, CountryCode, District, Population FROM City LIMIT 1
root@localhost4  1247  UPDATE city_huge SET population=population+1 WHERE id = last_insert_id()
root@localhost4  1500  SELECT * FROM city_huge WHERE id = @var
root@localhost4  1501  SET @var = floor(rand() * (SELECT max(id) FROM city_huge))
root@localhost4  1502  UPDATE city_huge SET population = population + 1 WHERE id = @var
root@localhost4  2748  SELECT * FROM city_huge ORDER BY population DESC LIMIT 3
```

Checking the reports generated by the previous two scripts helps easily identify the more expensive queries. In the example, the script with the highest processing time is: `SELECT * FROM city_huge ORDER BY population DESC LIMIT 3`. Optimization of this query may lead to the greatest return on efforts.

Probing the Cost of File Sort Operations

Large dataset sizes or inadequate indexing can result in data sorts that consume enough resources to dramatically slow application response time. For example, sorting the results of a query with matches to millions of table rows can be time intensive. DTrace can help administrators understand when adjustments to sorting parameters are necessary by quantifying the time spent on file sort operations.

Data sorts within MySQL databases rely on a built-in `filesort` algorithm. Utilizing the MySQL database `explain` command can help identify queries that make use of `filesort`. For example:

```
mysql> explain select * from city_huge ORDER by population DESC LIMIT 3 \G;
***** 1. row *****
      id: 1
    select_type: SIMPLE
          table: city_huge
           type: ALL
possible_keys: NULL
           key: NULL
        key_len: NULL
           ref: NULL
          rows: 1627800
     Extra: Using where; Using filesort
1 row in set (0.00 sec)
```

Since `Using filesort` appears in the `Extra` field of the `explain` tool report, the query does indeed utilize `filesort`. Before determining if tuning is necessary, further information regarding how much time is spent on the `filesort` operation must be gathered. To measure the time spent in the `filesort` algorithm, DTrace offers two useful probes — `filesort-start` and `filesort-done`. The following script (`filesort.d`) demonstrates how to instrument a timer:

```

#cat filesort.d
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
    printf("Tracing... Hit Ctrl-C to end.\n");
}

mysql*:::query-start
{
    this->query = copyinstr(arg0);
    this->who   = strjoin(copyinstr(arg3),strjoin("@",copyinstr(arg4)));
    this->connid = arg1;
}

mysql*:::filesort-start
{
    self->filesortstart = timestamp;
}

mysql*:::filesort-done
{
    this->filesort= (timestamp - self->filesortstart) /1000000;
    @timefilesort[this->who,this->connid,this->query] = sum(this->filesort);
}

dtrace:::END
{
    printf(" %-18s %5s %15s %s\n", "USER@HOST","CONNECT_ID", "FILESORT ms",
"QUERY");
    printa(" %-18s %6d %@15d %s\n", @timefilesort);
}

```

Executing the `filesort.d` script provides the following results:

```

#./filesort.d
Tracing... Hit Ctrl-C to end.

USER@HOST   CONNECT_ID   FILESORT ms  QUERY
root@localhost4  2640        SELECT * FROM city_huge ORDER BY population DESC LIMIT 3

```

In the previous section of this document titled “Analyzing Query Loads”, the script `query_stime.d` shows that the long running query `SELECT * FROM city_huge ORDER BY population DESC LIMIT 3` takes a total of 2748 milliseconds to execute. Now, the `filesort.d` script reports that 2640 milliseconds or 96% of the execution time for the query is spent in the `filesort` algorithm. Armed with the knowledge that tuning the sort method of this query can

result in significant performance gains, an administrator can add a special index for the “population” sorting column. After adding the index, `filesort` is no longer utilized and query execution time drops to 93 milliseconds (from 2748 milliseconds).

The `filesort-start` and `filesort-done` probes also include the parameters `database`, `table`, `error message`, and `number of rows to execute the query`. Printing this additional information from the `filesort.d` script can help provide further insight on how MySQL executes the query.

Profiling the Use of Stored Procedures

While stored procedures generally improve performance, these routines can also come into play during the application tuning process. Abnormalities such as repetitive execution of a stored procedure or queries that spend excessive time within the stored procedure may require attention. Unfortunately, the MySQL database slow query log offers little assistance in the debugging process. While the slow query log creates entries for stored procedure calls, the log does not trace which SQL statement executed in a stored procedure. A sample slow query log entry for a stored procedure is shown here:

```
# Query_time: 3.000040 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0
SET timestamp=1217959609;
call innodb_inserts(2,2);
```

Debugging and finding performance bottlenecks related to stored procedures depends on knowing what SQL statements spend time in the stored procedure. The simple DTrace script, `query_time.d`, can be used as a starting point for profiling queries by type. To make the script adjustments, add the `this->type` parameter to the aggregation function as follows:

```
@time[this->type,this->who,this->connid,this->query]=sum(this->elapsed)
```

Through this separation by type, the query time also now reflects time specifically spent as a general query, prepared statement, cursor statement, or in a stored procedure. Types are defined as follows:

- 0: Executed general query
- 1: Executed prepared statement
- 2: Executed cursor statement
- 3: Executed query in stored procedure

The new `Type` field is shown on the far left in this output:

TYPE	USER@HOST	CONNECT_ID	QUERY ms	QUERY
3	root@localhost	204	52	select * from city_huge

The specific example above shows that the query `select * from city_huge` executed in a stored procedure — type 3 — for 52 ms.

Observing Slave Queries

The ability to run statements in parallel on MySQL database master servers can help keep performance levels high — despite long running queries. Unfortunately, MySQL database slave servers can not operate under the same model. To protect data integrity, replication threads on slave servers run in series.

The difference in progress between the master MySQL database server and the slave server is known as replication lag. Assuming that master and slave servers each offer equal processing power, a query that takes two minutes to execute on the master server is likely to take two minutes to complete on the slave. During the processing of this lengthy query, the propagation of all subsequent updates are delayed. Replication lag can have serious consequences for failover scenarios and architectures that support direct client access to slave servers.

Tuning queries that execute slowly on slave servers can help reduce replication lag. Utilizing traditional tools to find performance issues on slave servers can be challenging. For example, the slow query log does not log slave server queries that cross the `long_query_time` threshold.

The following statement is executed on the master:

```
mysql>update city_huge SET population = population + 1;
Query OK, 362461 rows affected (1.34 sec)
Rows matched: 362461 Changed: 362461 Warnings: 0
```

When the actual update takes place on the slave, an administrator can observe that execution takes 872 milliseconds to complete. Even though `long_query_time` is set to zero, the query is not logged in the slow query log on the slave server.

By using the following DTrace script (`query_etime.d`), long replication queries can be readily identified:

```
#cat query_etime.d
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
    printf("%-20s %-6s %-6s %8s %8s %-s \n", "DB", "ConnID", "Query ms",
        "Matched Rows", "Changed Rows", "Query");
}
mysql*:::query-start
{
    self->query = copyinstr(arg0);
    self->db     = copyinstr(arg2);
    self->connid = arg1;
    self->rowsm  = 0;
    self->rowsu  = 0;
    self->querystart = timestamp;
}
mysql*:::query-done
{
    this->elapsed = (timestamp - self->querystart) /1000000;
    printf("%-20s %6d %6d %8d %8d %-s\n", self->db,
        self->connid, this->elapsed, self->rowsm, self->rowsu, self->query);
}

mysql*:::update-done
{
    self->rowsm = arg1;
    self->rowsu = arg2;
}

mysql*:::insert-done
{
    self->rowsm = arg0;
}

mysql*:::delete-done
{
    self->rowsm = arg1;
}

mysql*:::select-done
{
    self->rowsm = arg1;
}
```

The output of `query_etime.d` is shown below. A connection ID identifies the query as executing on a particular slave server. The script also reports the number of rows scanned during processing of the query.

```
# ./query_etime.d
DB      ConnID  Query ms  Matched RowsChanged Rows Query
world   13      872      362461   362461   update city_huge SET population = population + 1
```

The length of this query can likely be attributed to the number of rows that are affected. To reduce replication lag, this query can be broken down to distribute updates across multiple queries.

Optimizing Use of the MySQL Database Query Cache

The MySQL database query cache stores each `SELECT` statement and the corresponding results. Modifying SQL statements to take better advantage of the query cache can lead to significant performance gains. Applications that utilize cache can significantly reduce time spent on parsing, optimizing, and executing identical queries. As shown in the following sample output, the command `SHOW STATUS LIKE 'Qcache%'` can be used to report query cache usage statistics.

```
mysql> show status like 'Qcache%';
-----
Variable_name          Value
-----
Qcache_free_blocks    46
Qcache_free_memory    134997
Qcache_hits            76780
Qcache_inserts        23065
Qcache_lowmem_prunes  2115
Qcache_not_cached     33212
Qcache_queries_in_cache 512
Qcache_total_blocks   934
-----
```

While the `STATUS` command provides information about the number of cache hits and queries in cache, more information is needed to optimize use of this valuable resource. Utilizing the `query-cache-hit` and `query-cache-miss` DTrace probes can help administrators and application developers gain even more detailed information about query cache usage. As shown below, the `query_etime.d` script can be modified to identify query cache hits or misses:

```

#cat query_etime.d
...
mysql*:::query-start
{...
  this->hitormiss=0;
...
}
mysql*:::query-done
{
  this->elapsed = (timestamp - self->querystart) /1000000;
  printf("%-20s %2d %6d %6d %8d %8d %s\n", self->db,
this->hitormiss, self->connid, this->elapsed, self->rowsm, self->rowsu,
self->query);
}

mysql*:::query-cache-hit
{
  this->hitormiss = 1;
}

mysql*:::query-cache-miss
{
  this->hitormiss = 0;
}
...

```

In the example output below, the `query_etime.d` script identifies two identical queries that fail to utilize cache and take near 1.5 seconds to complete. Careful examination of the SQL statements in the DTrace output, reveals that one SQL statement uses lower case “from” and the other uses the upper case “FROM” in the query.

```

./query_etime.d
DB      Hit/MissConnID      Query ms      Matched RowsChanged Rows      Query
world  0      2864      1464      231      0      SELECT distinct Country.name FROM Country WHERE
code IN (SELECT CountryCode FROM city_huge WHERE population > 200)

world  0      2864      1458      231      0      SELECT distinct Country.name from Country WHERE
code IN (SELECT CountryCode FROM city_huge WHERE population > 200)

```

After re-writing the statements to use the exact same syntax, the query can take advantage of the cached results and improve performance significantly. The new results are shown here:

```
# ./query_etime.d
DB      Hit/MissConnID      Query ms      Matched RowsChanged Rows      Query
world  1      2864          0             231      0      SELECT distinct Country.name FROM Country WHERE
code IN (SELECT CountryCode FROM city_huge WHERE population > 200)
```

Some additional aspects to consider in trying to make the most of the query cache include:

- If the `query_etime.d` DTrace script reveals that many identical queries are failing to utilize cache, the cache-hit ratio may be improved by increasing the `query_cache_size` configuration parameter.
- A certain amount of CPU overhead is imposed by the process of checking query cache. If the `query_etime.d` DTrace script shows that tables are modified between identical queries, reducing or disabling the query cache may improve performance. In fact, Sun testing has shown that disabling the query cache can result in performance improvements as high as 10% for certain workload types.

Chapter 4

Putting it all Together

As shown in this document, simple DTrace scripts can become very powerful tuning and debugging tools. MySQL database support for DTrace probes facilitates retrieval of useful timing and argument details from within MySQL database functions. As a result, developers and administrators can speed resolution of performance issues related to bottlenecks within the database design, application SQL statements, and MySQL database configurations.

Chapter 5

For More Information

About the Author

Luojia Chen is a software engineer in Sun's ISV Engineering organization. Working on the open source team, Luojia specializes in MySQL software adoption of key Sun technologies. Currently, she is focused on MySQL database benchmarks, performance monitoring, optimization, and scalability in order to understand how to make MySQL software run at peak performance on Sun platforms.

Related Resources

MySQL Database

<http://sun.com/software/products/mysql>

Solaris Dynamic Tracing Data Sheet

<http://sun.com/software/solaris/ds/dtrace.pdf>

Solaris Dynamic Tracing Guide

<http://docs.sun.com/app/docs/doc/817-6223>

Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is

<http://docs.sun.com/>

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at:

<http://www.sun.com/blueprints/online.html>

