Tuning Oracle Reports 6i

An Oracle Technical White Paper
February 2001

INTRODUCTION

There are a number of distinct areas to focus on when tuning Oracle Reports. It's helpful to have a clear understanding of these areas which can provide both perceived and measured improvements. Consideration must be given to the costs involved, computing environment complexity and the trade-offs that may occur in improving performance in a single area such as reports. Investigation of some of these areas can result in significant performance improvements, some may result in minor performance improvements and yet others may have no effect on the actual report performance, but can improve the perceived execution time.

The first step is to determine where Oracle Reports is spending its time. Once this is accomplished, there are several performance tools available to evaluate the query, review database optimization and examine specific pieces of code used by the report for efficiency. The second area of performance tuning involves examining the formatting and layout of the report information. Finally, there are some runtime parameters to maximize performance and distribution of reports.

Changes should be made incrementally to isolate performance bottlenecks, optimize data access, streamline format processing and maximize runtime efficiencies. Changes made in one area may have a performance impact in another area.

The suggestions given in this paper are general in nature and not all suggestions will apply to all cases. However, by implementing some or all of the points pertinent to a given application environment, the performance of a report execution (both real and perceived) should be enhanced. This paper assumes that the user is familiar with building reports using Oracle Reports.

PERFORMANCE ANALYSIS TOOLS

The first step that should always be taken when attempting to tune a report is to determine where the report is spending the majority of its time in execution. That is, does the report spend a large proportion of its time within the database retrieving the actual data, within the formatting process once the data has been retrieved or waiting for runtime resources/distribution. The report may have the most streamlined and tuned layout model possible, but it is of little consequence if the vast majority of time is spent retrieving the data due to inefficient SQL.

REPORTS PROFILE

One of the first steps is to determine the how much time is spent on fetching the data versus the amount of time spent formatting the report. The Reports profile option, when set, produces a log file that shows how the report spent its processing time. This may help you identify performance bottlenecks.

Profile can be set in the report builder (*Tools -> Preferences >Runtime Settings>Profile*) or as a command line argument. The preferred method is to set the *TRACEFILE* (i.e. *TRACEFILE=<filename>*, with the additional command line arguments of *TRACEMODE* and *TRACEOPTS*. *TRACEMODE* indicates whether to replace or append to the existing trace log file. *TRACEOPTS* is used to build the event list within the log file with *TRACE_PRF* being the specific profile option. See Appendix A or the Oracle Reports builder on-line help for the syntax and complete list of *TRACEFILE*, *TRACEMODE* and *TRACEOPTS* options.

The following command line:

```
c:\rwrun60 report=emp.rdf userid=scott/tiger@orcl desformat=pdf
tracemode=replace tracefile=emp.lis traceopts=trace_prf
```

was used to produced the trace file for the Reports Profile Statistics:

```
+-----------------------------------+
|  Reports Profiler statistics      |
+-----------------------------------+
```

Total Elapsed Time:	29.00 seconds
Reports Time:	24.00 seconds (82.75% of TOTAL)
Oracle Time:	5.00 seconds (17.24% of TOTAL)
UPI:	1.00 seconds
SQL:	4.00 seconds

TOTAL CPU Time used by process: N/A

From the profile, it is possible to see the execution time (total elapsed time) for the report, the amount of time that was spent formatting the retrieved data (Reports Time) and the amount of time spent waiting for the data to be retrieved (Oracle Time). UPI time is the time spent establishing the database connection and parsing and executing the SQL. The SQL time is the time spent while the database server fetches the data (percent of time spent executing SRW.DO_SQL() statements, EXEC_SQL statements, PL/SQL Cursors, etc.).

In this example, the profile shows that the majority of the time was spent formatting (laying out) the data rather than querying and fetching it, so that is where we should concentrate our tuning efforts.

Note that the older option to obtain the reports profile using the command line *PROFILE*=<filename> option, where <filename> is the name of the required log file is still available but not recommended.

REPORTS TRACE

The reports trace option produces a file that describes the series of steps completed during the execution of the report. The trace option can be set so that either all events are logged in the file or only a subset of those events (e.g., only SQL execution steps). The trace file can provide an

abundance of information that is not only useful for performance tuning, but also helping to debug reports.

The trace option can be set either from the main menu (*Tools -> Trace*) or from the command line argument *TRACEFILE* (i.e. *TRACEFILE=<filename>*. Additional command line arguments for this option include *TRACEMODE* and *TRACEOPTS*. *TRACEMODE* specifies whether to replace or append to the existing trace log file. *TRACEOPTS* is used to specify the events to record within the log file. See Appendix A for the syntax and Appendix B for an example of a trace file.

EFFICIENT SQL

Oracle Reports uses the Structured Query Language (SQL) to retrieve data from the relational database. It is essential for anyone tuning reports to have a good working knowledge of SQL and to understand how the database is going to execute these statements. For those who are less proficient in SQL, Oracle Reports provides Wizards to help build queries; however, the wizards cannot prevent inefficient SQL from being created (e.g., SQL that does not use available indexes).

An invaluable aid to tuning the SQL in the report is the SQL trace functionality provided by the database. SQL trace allows the user to determine the SQL statement sent to the database as well as the time taken to parse, execute and fetch data. Once a trace file has been generated, the *TKPROF* database utility can be used to generate an *EXPLAIN PLAN*, which is a map of the execution plan used by the Oracle Optimizer. The *EXPLAIN PLAN* shows, for example, where full table scans have been used, which may prompt the creation of an index (depending on the performance hit). In Oracle Reports 6 users can turn on SQL trace without modifying the report simply by using the command line option *SQLTRACE=YES|NO*.

The following explain plan was generated by using SQL trace and EXPLAIN PLAN. Further information can be found in the Oracle Server SQL Language Documentation.

Example

The statement being executed is as follows:

```
Select e.ename, d.
From emp e, dept d
Where e.empno(+) = d.deptno
```

The explain plan generated is as follows:

```
OPERATION          OPTIONS     OBJECT_NAM        POSITION
------------------ ----------- ----------------- --------
SELECT STATEMENT
  MERGE JOIN       OUTER                         1
    SORT           JOIN                          1
      TABLE ACCESS FULL        DEPT              1
    SORT           JOIN                          2
      TABLE ACCESS FULL        EMP               1
```

In tuning the data for Oracle Reports, it is important to understand the Oracle RDBMS provides two optimizers, the cost-based and the rule-based. Whilst use of the cost-based optimizer removes much of the complexity of tuning SQL, an understanding of the rules used by the rule- based (heuristic) optimizer allows the developer to choose the preferred method. The cost-based optimizer will generally give an optimal execution plan, however a developer who understands the spread of the data and the rules governing the optimizer has much greater control over the execution plan and hence efficiency. If the cost-based optimizer has not been activated then the use of inefficient SQL (from an optimizer point of view) can cripple a report.

Note

For especially large queries dependent on the optimizer, it is imperative to either:

Activate the cost-based optimizer by either running *ANALYZE* on the tables or setting the appropriate init.ora parameter

or

Optimize all SQL following the rules laid out for the rule-based optimizer.

More information on optimizer database functionality can be found in the Oracle Server Documentation.

The ORA_PROF built-in package is used for performance tuning that focuses on how much time a specific piece of code takes to run. The following commands are available using the Ora_Prof built-in package within Oracle Reports, which lets you control when and where the code will be measured:

Create_Timer

Destroy_Timer

Elapsed_Time

Reset_Timer

Start_Timer

Stop_Timer

More information on the ORA_PROF built-in can be found using the Oracle Reports builder on-line help.

PL/SQL

PL/SQL that perform a significant amount of database operations will perform better if it is implemented as stored database procedures. These stored procedures will run on the Oracle database and so they can perform the operations more quickly than PL/SQL that is local to the report (which would use the PL/SQL parser in reports, then the SQL parser in the database and include a network trip as well). The opposite is true if the PL/SQL does not involve any database operations. In this case the PL/SQL should be coded as locally as possible using the program units node in the report object navigator. This also has a performance advantage over executing PL/SQL in an external PL/SQL library. The performance overhead of PL/SQL libraries is only outweighed when the benefits of code sharing can be utilized.

SRW.DO_SQL() should be used as sparingly as possible, because each call to SRW.DO_SQL() necessitates parsing and binding the command and the opening of a new cursor (just as with a normal query). Unlike the query, however, this operation will occur each time the object owning the SRW.DO_SQL() fires. For example, if a block of PL/SQL in a formula column calls SRW.DO_SQL() and the data model group where the formula resides returns 100 records, then the parse/bind/create cursor operation will occur 100 times. It is therefore advisable to only use SRW.DO_SQL() for operations that cannot be performed within normal SQL (for example, to create a temporary table, or any other form of DDL) and to use it in places where it will be executed as few times as possible (for example, in triggers that are only fired once per report).

The primary reason to use SRW.DO_SQL is to perform DDL operations, such as creating or dropping temporary tables. For example, you can have SRW.DO_SQL create a table whose name is determined by a parameter entered on the runtime parameter form:

Example

```
SRW.DO_SQL (`CREATE TABLE' || :tname || `(ACCOUNT NUMBER
    NOT NULL PRIMARY KEY, COMP NUMBER (10,2))');
```

JAVA STORED PROCEDURES

With stored procedures, you can implement business logic at the server level, thereby improving application performance, scalability and security. Oracle 8i now allows both PL/SQL and Java Stored Procedures to be stored in the database. Typically, SQL programmers who want procedural extensions favor PL/SQL and Java programmers who want easy access to Oracle data favor Java. This now opens up the Oracle database to all Java programmers.

For more information on Java Stored Procedures, please refer to the Oracle8i Java Stored Procedures Developer's Guide

ACCESSING THE DATA

If the performance tools show that a report is spending a large amount of time accessing data from the database, then it is often beneficial to review the structure of the data and how it is being used. A bad schema design can have a dramatic effect on the performance of a reporting system. For example, an overly normalized data-model can result in many avoidable joins or queries.

INDEXES

Columns used in the WHERE clause should be indexed. The impact of indexes used on the columns in the master queries of a report will be minor as these queries will only access the database once. However, for a significant performance improvement indexes should be used on any linked columns in the detail query.

A lack of appropriate indexes can result in many full-table scans which can have a major impact on performance.

CALCULATIONS

When performing calculations within a report (either through summary or formula columns), the general rule of thumb is the more calculations that can be performed within the SQL of the report queries, the better. If the calculations are included in the SQL, then they are performed before the data is retrieved by the database, rather than the performed on the retrieved data by the Report. Database-stored user-defined PL/SQL functions can also be included in the query select list. This is more efficient then using a local PL/SQL function (e.g. in a formula column), since the calculated data is returned as part of the result set from the database.

Example

The following PL/SQL function can be stored in the database:

```
CREATE OR REPLACE FUNCTION CityState (
  p_location_id  world_cities.location_id%TYPE)
  RETURN VARCHAR2 is
  v_result  VARCHAR2(100);
BEGIN
   SELECT city || ‘,‘||state
    INTO v_result
    FROM world_cities
    WHERE location_id = p_location_id;
  RETURN v_result;
END CityState;
```

This function will return the city separated by a space and then the state. This formatting is done at the database level and passed back to the report to display.

In the report the query would look like:

```
Select location_id, citystate(location_id)”City & State” from
world_cities
```

The result would look like this:

```
LOCATION_ID CITY & STATE
----------- -------------------------
          1 Redwood Shores,California
          2 Seattle,Washington
          3 Los Angeles,California
          4 New York,New York
```

REDUNDANT QUERIES

Ideally a report should have no redundant queries (queries which return data which is not required in the report), since they will clearly have an effect on performance. In general, the fewer queries you have the faster your report will run. So, single query data models tend to execute more quickly

than multi-query data models. However, situations can arise where a report not only needs to produce a different format for different users, but also needs to utilize different query statements. Clearly this could be achieved by producing two different reports, but it may be desirable to have a single report for easier maintenance. In this instance, the redundant queries should be disabled by use of the SRW.SET_MAXROW() procedure.

Example

The following code in the Before Report trigger will disable either *Query_Emp* or *Query_Dept* depending on a user parameter:

```
IF :Parameter_1 = 'A' then
      SRW.SET_MAXROW('Query_Emp',0);
ELSE
      SRW.SET_MAX_ROW('Query_Dept',0);
END IF;
```

Note

The only meaningful place to use SRW.SET_MAXROW() is in the Before Report trigger (after the query has been parsed). If SRW.SET_MAXROW() is called after this point then the SRW.MAXROW_UNSET packaged exception is raised.

The query will still be parsed and bound, but no data will be returned to the report.

BREAK GROUPS

Limiting the number of break groups can improve the performance of a report. For each column in the data model that has the break order property set (except the lowest child group), Oracle Reports appends this as an extra column to the ORDER BY clause for the appropriate query. Clearly, the fewer columns in the ORDER BY clause, the less work the database has to do before returning the data in the required order. The creation of a break group may make an ORDER BY clause defined as part of the query redundant. If this is the case then the redundant ORDER BY should be removed, since this will require extra processing on the database.

If the report requires the use of break groups, then care should be taken to ensure that the break

order property is set for as few columns in the break group as possible. A break order column is indicated by a small triangle or arrow to the left of the column name in the group in the data model. Each break group above the lowest child group of a query requires at least one column within in to have the break order property set. Simply taking the break order off columns where sorting is not required can also increase performance.

Try to limit break groups to a single column. These columns should be as small as possible and be database columns (as opposed to summary or formula columns) wherever feasible. Both of these conditions can help the local caching that Oracle Reports does before the data is formatted for maximum efficiency. Clearly, these conditions cannot always be met, but will increase efficiency when utilized.

The following profile output was generated by the same report that generated the profile output on page 2. With the original run, no break orders were set, to generate the following profile all columns had break order set:

```
+-----------------------------------+
|  Reports Profiler statistics      |
+-----------------------------------+
```

Total Elapsed Time:	34.00 seconds
Reports Time:	26.00 seconds (76.47% of TOTAL)
Oracle Time:	8.00 seconds (23.52% of TOTAL)
UPI:	1.00 seconds
SQL:	7.00 seconds

TOTAL CPU Time used by process: N/A

The SQL took about twice as long to run with break order set. As background, this report is a single query tabular report that retrieves about 20 records from about 500000 present in the table, where the table resides in a well-structured and well-indexed database.

GROUP FILTERS

The main use for group filters in the database is to reduce the number of records retrieved from

the database. When using a group filter, the query is still passed to the database and all the data is returned to reports, where the filtering will take place. Therefore, even if the filter is defined to only displays the top five records, the result set returned to reports will contain all the records returned by the query.

For this reason, it is usually more efficient to incorporate the functionality of the group filter into the query WHERE clause, or into the maximum rows property of the query wherever possible. This will restrict the data returned by the database.

TO LINK OR NOT TO LINK

As with most operations in Reports, there are a number of ways to create data models that include more than one table. Consider, for example, the standard case of the dept/emp join, i.e., the requirement is to create a report that lists all the employees in each department in the company. In Reports the user can either use the following in a single query:

```
Select d.dname, e.ename
From emp e, dept d
Where e.deptno(+) = d.deptno
```

Or they can create two queries

```
Select deptno, dname from dept
Select deptno, ename from emp
```

and create a column link between the two on deptno.

When designing the data model in the report, it is preferable to minimize the actual number of queries by using fewer, larger (multi-table) queries, rather than several simpler (single-table) queries. Each time a query is issued, Oracle Reports needs to parse, bind and execute a cursor. A single query report is therefore able to return all the required data in a single cursor rather than many. Also be aware that with master-detail queries, the detail query will be re-parsed, re-bound and re-executed for each master record retrieved. In this instance it is often more efficient to merge the two queries and use break groups to create the master-detail effect.

It should be noted, however, that the larger and more complex a query gets, the more difficult it can be to maintain. Each site needs to decide at what point to balance the performance versus the maintenance requirements.

FORMATTING THE DATA

Once the data has been retrieved from the database, Reports needs to format the output following the layout model that the user has created. The time taken to generate the layout is dependent on a number of factors, but in general it comes down to how much work is involved in preventing an object from being overwritten by another object and the efficiency of any calculations or functions performed in the format triggers.

LAYOUT

When generating a default layout Oracle Reports puts a frame around virtually every object, so that it is protected from being overwritten when the report is run. At runtime, every layout object (frames, fields, boilerplate, etc.) is examined to determine the likelihood of that object being overwritten. In some situations (for example boilerplate text column headings), there is clearly no risk of the objects being overwritten and hence the immediately surrounding frame can be removed. This reduces the number of objects that Oracle Reports has to format and hence improves performance.

Similarly, when an object is defined as having an undefined size (variable, expanding or contracting in either or both the horizontal and vertical directions) then extra processing is required since Oracle Reports must determine that instance of the object's size before formatting that object and those around it. If this sizing can be set to fixed then this additional processing is not required, since the size and positional relationships between the objects is already known.

Furthermore, instead of truncating a character string from a field in the Report Builder Layout, it is better to use the SUBSTR function in the report query to truncate the data at the database level.

This reduces unnecessary processing and formatting after the data retrieval.

FORMAT TRIGGERS

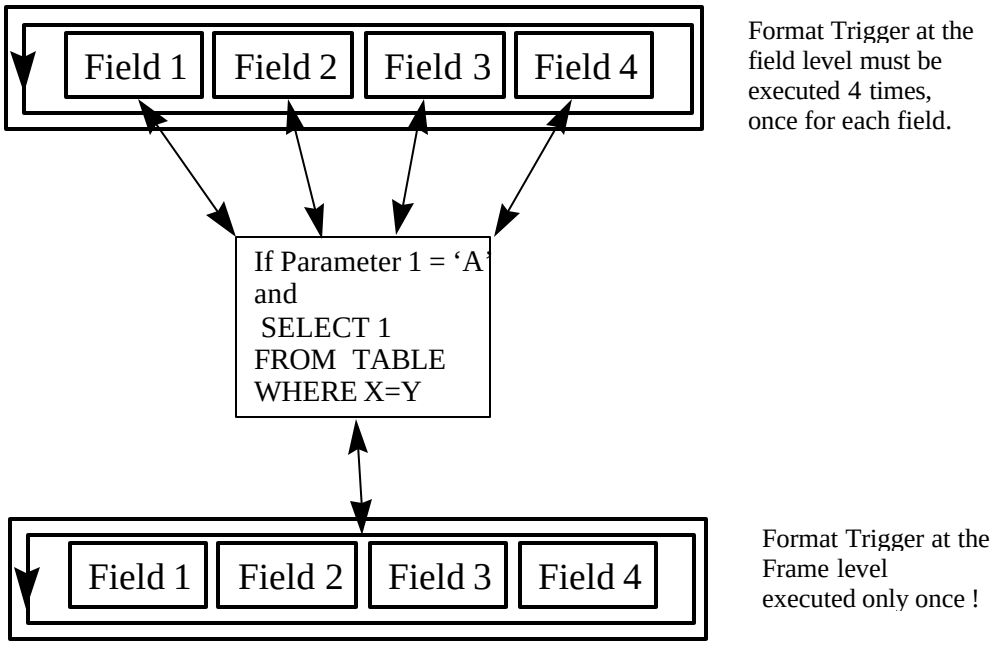
Format triggers have two major purposes:

- Dynamically disable and enable objects at runtime.
- Dynamically change the appearance of an object at runtime.

Care should always be exercised when using format triggers, since the triggers do not only fire for every instance of their associated object produced, but every time the object is formatted at runtime.

These two scenarios may sound the same, but consider the following situation: A tabular report includes a single repeating frame that can expand vertically and has page protect set on. As this report is formatted, there is room for one more line at the bottom of the first page. Reports starts to format the next instance of the repeating frame and fires its associated format trigger. One of the objects inside the repeating frame is found to have expanded and this instance of the repeating frame is therefore moved to the following page and the format trigger for the repeating frame is fired again. Hence, although the repeating frame only appears once (at the top of the second page), the format trigger has fired twice. Because you can not be sure how many times a format trigger will fire for a particular object, DML should not be performed in a format trigger. With the example above, had the format trigger contained an INSERT statement then two rows of data would have been inserted.

Format triggers should also be placed at the highest level possible in the object/frame hierarchy, so that the trigger fires at the lowest possible frequency. For example:



When defining any triggers or PL/SQL program units within Oracle Reports, it is important to maximize the efficiency of the code. For example, if the display attributes of a field are to change dynamically (for example, to draw attention to values outside the norm), then the attributes should be changed using the individual built-ins such as SRW.SET_TEXT_COLOR. For general PL/SQL tuning issues, please refer to the PL/SQL Users Guide and Reference.

Reports performance can be improved by giving layout objects (e.g., frames and repeating frames) a transparent border and fill pattern. Transparent objects do not need to be rendered in a bitmapped file. As a result, processing is faster when objects are transparent.

GENERAL LAYOUT GUIDELINES

The following guidelines can improve performance when creating or changing a report layout:

- Make your non-graphical layout objects (e.g. boilerplate text or fields with text) fixed in size -- that is, set theVertical and Horizontal Elasticity property of the field to Fixed. In particular, making repeating frames and their contents fixed in size can improve performance. Non-graphical objects that are variable in size require more processing because Report Builder must determine their size before formatting them. Non-graphical objects that are fixed in size do not require this additional processing because their size is already known.
- Make your graphical layout objects (e.g., images and Oracle Graphics objects) variable in size -- that is, Vertical and Horizontal Elasticity of Variable. Graphical objects that are fixed in size usually need to have their contents scaled to fit inside of the object. Scaling an object's contents requires more processing. If the object is variable in size, it can grow or shrink with the contents and scaling is not necessary.
- Specify Reduce Image Resolution for image objects whose size you reduce. (This option is available as a drawing option under the Format menu). When you reduce the size of an image, it requires less information to display it than when it was larger. Reduce Image Resolution eliminates the unnecessary information and reduces the amount of space needed to store the image. This can be particularly useful for large, multi-colored images.
- Make fields that contain text one line long and ensure that their contents fit within their specified width (e.g., by using the SUBSTR function). If a field with text spans more than one line, then Report Builder must use its word-wrapping algorithm to format the field. Ensuring that a field only takes one line to format avoids the additional processing of the word-wrapping algorithm.
- Minimize the use of different formatting attributes (e.g., fonts) within the same field or boilerplate text. If text in a field or boilerplate object contains numerous different formatting attributes, it requires longer to format.

FETCHING AHEAD

Oracle Reports provides the report developer with the ability to display data such as total number of pages, or grand totals in the report margins, or on the report header pages. This is an extremely useful function, but has the requirement that the entire report is 'fetched ahead', therefore the entire report is processed before the first page can be output. The usual Oracle Reports model is to format pages on an *as-needed* basis. Using the read-ahead functionality will not affect the overall time that the report takes to generate, but it does affect the amount of temporary storage required and the amount of time taken before the first page can be viewed in the Live Previewer or Previewer (if running to Screen). This is an example of the perceived performance as opposed to the actual performance. If the report is going to be run to the screen in the production environment then read-ahead should be avoided unless the performance is deemed acceptable.

BURSTING AND DISTRIBUTION

With the introduction of report bursting in Reports 6.0, a report layout can be made up of three distinct sections: a header, body and trailer. A report can now comprise all three of these sections or it could be viewed as three separate reports within one report. This is made possible by a set of properties each individual section. The performance gain is evident when bursting is used in conjunction with another feature new to Reports 6.0 - Distribution. This allows each section of a report to have multiple different formats and be sent to multiple destinations. Once the distribution options have been set the report needs only to be run once to be output to multiple destinations with a single execution of the query(s) where previously it would have required the report to be run multiple times.

CALLING REPORTS FROM FORMS

In an application built using Oracle Forms and Oracle Reports, it's often the case that some kind of report is required on data that has already been retrieved/updated by the Oracle Forms section of the application. The tight product integration between the Oracle Forms and Oracle Reports

components allows for the passing of blocks of data between the associated products, thus removing the need for subsequent queries to the database. Between Forms and Reports, this passing of data is achieved by use of record groups and data parameters, then using the Run_Product built-in procedure. Run_Product is being replaced by Run_Report_Object for calling Reports from Forms. This paper will not discuss the details of converting Run_Product to Run_Report_Object, except to say that it is advisable to migrate towards this new built-in when reviewing performance efficiencies. For more information on calling Reports from a Form, refer to The Forms Server 6i Report Integration White Paper.

Note

Unless these data parameters are reasonably large, or the queries are particularly complicated then the perceived performance improvements would be negligible. It should also be noted that only top-level groups in a report can accept data parameters passed from Forms.

When calling a report from Forms, Reports could re-query the data, but it is more efficient to have Forms create a record group to hold the data and pass that along as a parameter to Reports. This technique is referred to as query partitioning. This means that Oracle Reports will do no work on querying and will only be responsible for the formatting of the data. This means that any dynamic alteration of queries (via triggers and lexical parameters) will be ignored.

The first time that a report is called from a form using run_product on a Windows platform then the Reports background engine will be loaded into memory. Putting the background engine executable in the start-up group so that it is initialized every time Windows is booted up can hide this start-up cost.

RUNNING THE REPORT

Having designed a report to run efficiently, is it possible to further effect the overall performance of a report by setting specific runtime arguments.

By default, Oracle Reports automatically runs error checks on the layout and bind variables within the report every time the report is run. This is very useful during the design phase, but is generally superfluous when the report is running in a production environment. Setting the runtime parameter *RUNDEBUG = NO* will turn off this extra error checking at runtime.

Oracle Reports is able to take advantage of Oracle's array processing capabilities for data fetching. This allows records to be fetched from the database in batches instead of one at a time and results in significantly fewer calls to the database. The downside of array processing is that more space (memory) is required on the execution platform for storing the arrays of records returned. If the load on the network becomes a major bottleneck in the production environment, then the *ARRAYSIZE* command line parameter (defined in kilobytes) can be set to as large a value as possible for the execution environment to reduce the number of network trips made to fetch the data for the report.

Similarly, if the report makes use of the LONG, CLOB or BLOB datatypes to retrieve large amounts of data, the *LONGCHUNK* parameter can be set to a large a value as possible to reduce the number of increments it takes Reports to retrieve long values. If you are using the Oracle8 or Oracle 8i server, it is more efficient to use the CLOB or BLOB datatypes instead of LONG or LONG RAW.

Finally, if a parameter form or on-line previewing of the report is not required, then these functions can be bypassed by setting the *PARAMFORM* and *BATCH* system parameters NO and YES respectively.

When printing to PostScript, use the COPIES parameter carefully. If COPIES is set to something greater than 1 for a PostScript report, Report Builder must save the pages in temporary storage in order to collate them. This can significantly increase the amount of temporary disk space used by Report Builder and the overhead of writing additional files can slow performance.

CONCLUSION

When developing Oracle Reports modules the developer must look at the report not in isolation, but rather in the context of:

- The application requirements
- The 'correctness' of the underlying data model
- The environment in which this report is going to run (e.g., Client/Server, the Web, network implications, etc.)
- The degree of user interaction required.

Once these confines have been identified, then the tuning process becomes one of optimizing the issued SQL, minimizing calls to the database and minimizing the amount of unnecessary format processing required for the layout of the returned data.

In addition, if execution of the report is using the reports server, some additional tuning may be done there. Please refer to the Advanced Techniques for the Oracle Repots Server White Paper. The suggestions given in this paper are general is nature and not all suggestions will apply to all sites. However, by implementing some of all of the points pertinent to a given application environment, the performance of a report execution (both real and perceived) should be enhanced.

APPENDIX A — TRACEFILE, TRACEMODE AND TRACEOPTS

TRACEFILE=<tracefile>

where tracefile is any valid file name in which the trace information will be logged.

TRACEMODE=<TRACE_APPEND | TRACE_REPLACE>

where TRACE_APPEND overwrites the contents of the tracefile.

TRACEOPTS=<TRACE_ERR | TRACE_PRF | TRACE_APP | TRACE_PLS

TRACE_SQL | TRACE_TMS | TRACE_DST | TRACE_ALL>

where TRACE_ERR - list error messages and warnings.

TRACE_PRF - performance statistics

TRACE_APP - information on all the report objects.

TRACE_PLS - information on all PL/SQL objects.

TRACE_SQL - Information on all SQL.

TRACE_TMS - timestamp for each entry in file.

TRACE_DST - distribution lists. This may be useful to determine which section was sent to which destination.

TRACE_ALL - all possible information. (DEFAULT)

Note

Trace cannot be used on a .rep file.

Options can be combined on the command line. For example the TRACEOPTS=(TRACE_APP, TRACE_PRF) means that the log file will contain information on all report objects and performance statistics.

APPENDIX B — SECTION OF SAMPLE REPORTS TRACE FILE

This is a look at a sample trace file generated on a group above report on the emp table with 14 rows retrieved with traceopts=all. Only the first page of this report is shown as an example.

```
LOG :
      Report: MODULE2
      Logged onto server:
      Username: scott



16:15:59  APP  (  Frame
16:15:59  APP . (  Text Boilerplate         B_OR$BODY_SECTION
16:15:59  APP . )  Text Boilerplate         B_OR$BODY_SECTION
16:15:59  APP . (  Generic Graphical Object B_1_SEC2
16:15:59  APP . )  Generic Graphical Object B_1_SEC2
16:15:59  APP . (  Text Boilerplate         B_DATE1_SEC2
16:15:59  APP . )  Text Boilerplate         B_DATE1_SEC2
16:15:59  APP . (  Text Boilerplate         B_PAGENUM1_SEC2
16:15:59  APP . )  Text Boilerplate         B_PAGENUM1_SEC2
16:15:59  APP . (  Text Field               F_DATE1_SEC2
16:15:59  APP .. (  Database Column          Name unknown
16:15:59  APP .. )  Database Column          Name unknown
16:15:59  APP . )  Text Field               F_DATE1_SEC2
16:15:59  APP  )  Frame
16:15:59  APP  (  Frame
16:15:59  APP . (  Frame                    M_G_DNAME_GRPFR
16:15:59  APP .. (  Repeating Frame          R_G_DNAME
16:15:59  APP ... (  Group                    G_dname  Local Break:  0
Global Break:  0
16:15:59  APP .... (  Query                    Q_1
16:15:59  SQL         EXECUTE QUERY : select d.dname, e.ename from emp
e, dept d where e.deptno(+)=d.deptno
16:15:59  APP .... )  Query                    Q_1
16:15:59  APP ... )  Group                    G_dname
16:15:59  APP ... (  Text Field               F_ENAME
16:15:59  APP .... (  Database Column          ename
16:15:59  APP .... )  Database Column          ename
16:15:59  APP ... )  Text Field               F_ENAME
16:15:59  APP ... (  Text Field               F_DNAME
16:15:59  APP .... (  Database Column          dname
16:15:59  APP .... )  Database Column          dname
16:15:59  APP ... )  Text Field               F_DNAME
16:15:59  APP ... (  Group                    G_dname  Local Break:  1
Global Break:  1
16:15:59  APP ... )  Group                    G_dname
16:15:59  APP ... (  Text Field               F_ENAME
16:15:59  APP .... (  Database Column          ename
16:15:59  APP .... )  Database Column          ename
16:15:59  APP ... )  Text Field               F_ENAME
```

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/