

Oracle9i Performance and Scalability  
in DSS Environments  
*An Oracle Technical White Paper*  
*April 2001*

# Oracle9i Performance and Scalability in DSS Environments

Executive Overview.....	3
Introduction.....	3
List Partitioning .....	3
Intra-Partition Parallelism for Inserts .....	5
Merge.....	8
Multi-table Insert.....	11
Data Block Prefetching.....	12
Index Skip Scan .....	13
Pipelined Table Functions .....	14
Automatic Memory Management in PGA.....	18
Conclusion .....	21

## Executive Overview

The changing landscape of the IT industry and the ever increasing dependency on the internet has produced more challenges for database management software than ever before. The exponential growth of companies relying on the internet for increasing their market penetration quickly and effectively has created an explosion of data that needs to be stored and managed in data warehouses. As a consequence, the demands for DBMS scalability and performance are greater than ever.

Oracle has always been a leader in developing features that meet the changing demands of businesses, and this leadership has made it the most successful of all database vendors. The latest release of the Oracle database, Oracle9i, is designed to deliver the scalability and performance that are critical for the quality of service requirements of e-business and traditional businesses.

This white paper describes in technical detail the performance benefits of new features in the Oracle9i database that pertain to DSS and Data Warehousing environments. Many of these features will be useful in traditional OLTP environments as well. Topics related to the Oracle9i Application Server and features in the Oracle9i database which are not directly related to DSS environments are not covered in this document.

## Introduction

This paper describes the performance benefits of new DSS features of the Oracle9i database. The performance claims are supported with results and graphs from experiments conducted to mimic real world applications. The results clearly illustrate the performance gain over the best practices that would have been implemented using Oracle8i to achieve the same functionality. Apart from the inherent performance gain, some of these features also add ease of use.

Almost all of the experiments were conducted on a Sun (sparc) Ultra-Enterprise 4500 machine with 3 GB RAM, 10 CPUs running at 168 MHz, and running Solaris 5.6. For each feature the relevant system configuration parameters are mentioned in its respective test description. Some of the features introduce new SQL syntax. A skeletal form of the syntax is listed wherever necessary. For exact and complete SQL syntax of the new features, please refer to the Oracle9i SQL Reference.

## List Partitioning

A new partitioning method called List Partitioning is added in Oracle9i to the set of partitioning methods already supported (Range, Hash and Composite). List partitioning allows explicit control over how rows map to partitions. This is done by specifying list of discrete values for the partitioning column in the description for each partition. This is different from RANGE partitioning where a range of values is associated with a partition and from HASH partitioning where the user has no control of the row to partition mapping. This partitioning method has been specifically added to model data distributions that follow discrete values. Partitioning along discrete values cannot be easily done Oracle8i using:

- Range partitioning, because this method assumes a natural range of values for the partitioning column, and it is impossible to group together out-of-range values partitions.
- Hash partitioning, because the user has no control over the distribution of data (since these are distributed over the various partitions via the system hash function) and hence again cannot logically group together discrete values for the partitioning columns into partitions.

LIST partitioning allows for the distribution of data, based on discrete column values. Unordered and unrelated sets of data can be grouped and organized together very naturally using LIST partitioning.

### **Test Case - Updating with LIST Partitioning:**

Consider a database table that contains the subscriber information for a telephone company with area codes 408, 415, 510, 650, and 707. A new area code 925 has to be added, and roughly half of the 510 customers' phone numbers (those with phone numbers greater than 4330000) have to undergo this area code change.

***8i Implementation:*** Using Oracle8i, we would have probably implemented the table in this way:

```
CREATE TABLE SUBSCRIBERS (AREA_CODE NUMBER,
                           LAST_NAME VARCHAR2(30),
                           ...)
PARTITION BY RANGE (AREA_CODE)
(PARTITION SUB_408 VALUES LESS THAN (409),
 PARTITION SUB_415 VALUES LESS THAN (416),
 ...
 PARTITION SUB_707 VALUES LESS THAN (MAXVALUE));
ALTER TABLE SUBSCRIBERS ENABLE ROW MOVEMENT;
```

Hence, when following statement is issued,

```
UPDATE SUBSCRIBERS SET AREA_CODE = 925 WHERE AREA_CODE=510 AND TELEPHONE# > 4330000;
```

the partitioning key gets updated and the rows have to be moved to a new partition. Row movement can be quite an expensive operation and as the number of rows to be moved increases the performance deteriorates. Moreover this can lead to skew in data distribution too.

***9i Implementation:*** Using Oracle9i List partitioning, we would have

```
CREATE TABLE SUBSCRIBERS (AREA_CODE NUMBER,
                           LAST_NAME VARCHAR2(30),
                           ...)
PARTITION BY LIST (AREA_CODE)
(PARTITION SUB_408 VALUES (408),
 PARTITION SUB_415 VALUES (415),
 ...
 PARTITION SUB_707 VALUES (707));
```

To update, the following statements will be issued,

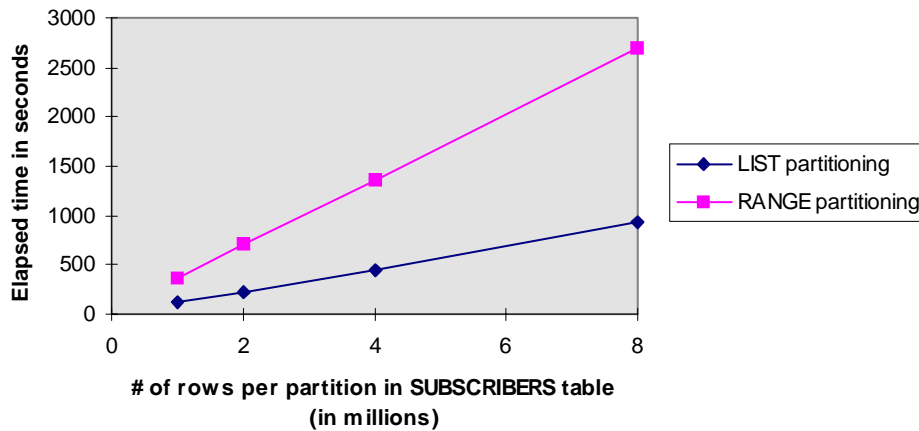
```
ALTER TABLE SUBSCRIBERS MODIFY PARTITION SUB_510 ADD VALUES(925);
UPDATE SUBSCRIBERS SET AREA_CODE = 925 WHERE AREA_CODE=510 AND TELEPHONE# > 4330000;
```

The first statement is a DDL that updates just the data dictionary, a very low cost operation. The second statement updates the rows in the partition SUB\_510 of the table SUBSCRIBERS. Therefore, there is no cost of row movement here.

Some experiments were conducted for the above mentioned problem by varying the number of rows in each partition. The results, which are very encouraging, are given below:

Rows per partition in SUBSCRIBERS table	Time taken for Update using LIST partitioning	Time taken for Update using RANGE partitioning
1 million	127.03 s	359.74 s
2 million	230.33 s	702.78 s
4 million	453.22 s	1364.36 s
8 million	922.86 s	2696.19 s

### Update on Partitioning Key Column



#### Test Case - Adding a New Partition and Loading Data with LIST Partitioning

If a new partitioning has to be added and then loaded with data, with LIST partitioning we can issue the following low cost DDL:

```
ALTER TABLE SUBSCRIBERS ADD PARTITION SUB_925 VALUES(925);
```

followed by a call to SQL\*Loader to load the partition.

In contrast, with RANGE partitioning we will have to issue:

```
ALTER TABLE SUBSCRIBERS SPLIT PARTITION SUB_707 AT (708) INTO (PARTITION SUB_707 , PARTITION SUB_925);
```

followed by a call to SQL\*Loader to load the partition.

The RANGE partitioning (`SPLIT PARTITION`) is a much more expensive operation as all the rows need to be scanned. Here are the results for of using ADD partition in Oracle9*i* versus SPLIT partition in Oracle8*i*:

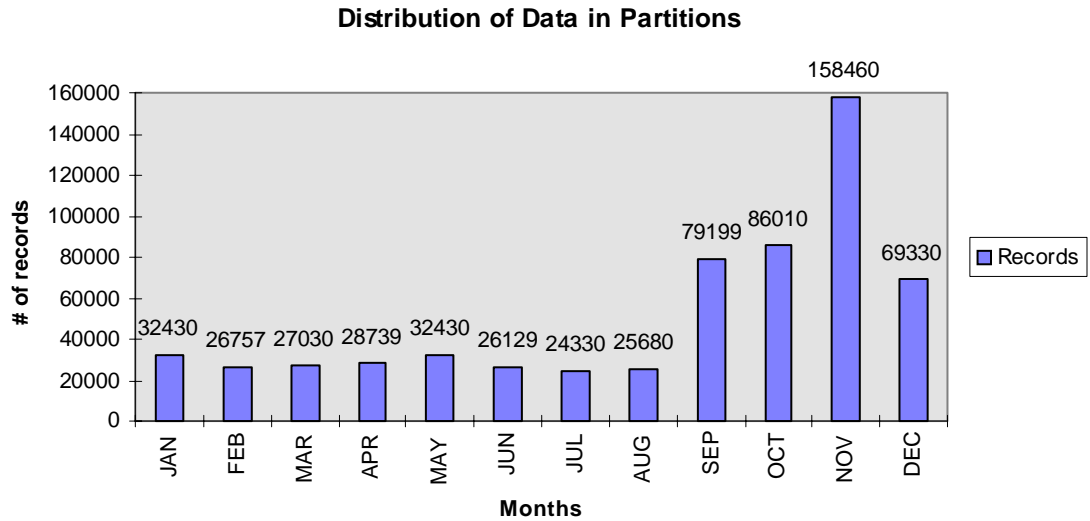
Rows per partition in SUBSCRIBERS table	Time taken for ADD partition using LIST partitioning	Time taken for SPLIT PARTITION using RANGE partitioning
1 million	0.09 s	34.17 s
2 million	0.11 s	63.50 s
4 million	0.12 s	125.17 s
8 million	0.14 s	245.09 s

#### **Intra-Partition Parallelism for Inserts**

In Oracle8*i*, one partition of a table is acted upon by only one slave during parallel insert (PDML). As a result, load balancing was not possible if there was a data skew among the partitions. For example, in many applications the table is partitioned by range on a date column, and the rows are mainly inserted in the last partition. Due to this data distribution, the slave working on the last partition needs to do much more work than other slaves. Oracle9*i* introduces intra-partition parallelism for inserts. Allowing multiple slaves to work on a partition helps to alleviate this performance bottleneck. It takes advantage of the dynamic load balancing capabilities of parallel execution. This is all done automatically when Parallel DML is enabled. The benefits of intra-partition parallelism of Inserts are maximum when there is a lot of data skew. If the data to be inserted is uniform then there is little or no advantage over regular parallel inserts.

**Test Case for Intra-Partition Parallelism of INSERT:**

Rows from an Operational Data Store table SALES\_ODS have to inserted into a fact table SALES\_1999 that is partitioned by month. The data distribution is skewed and is shown in the graph below:



**Implementation:** In both Oracle8i and Oracle9i, this will be implemented as follows:

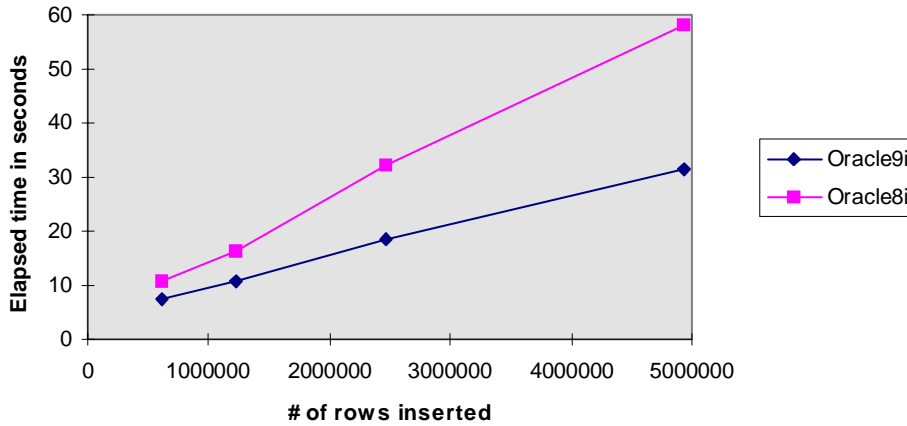
```
ALTER SESSION ENABLE PARALLEL DML;

INSERT INTO SALES_1999 /*+ parallel(SALES_1999, 10) */
SELECT * from SALES_ODS;
```

The following table shows the elapsed time comparisons for parallel inserts with increasing data volumes. Note that the first row of the table shows an insertion for the total of rows shown in the graph above. The second through fourth rows of the table represent insertion using data with the same distribution as the set shown in the graph above.

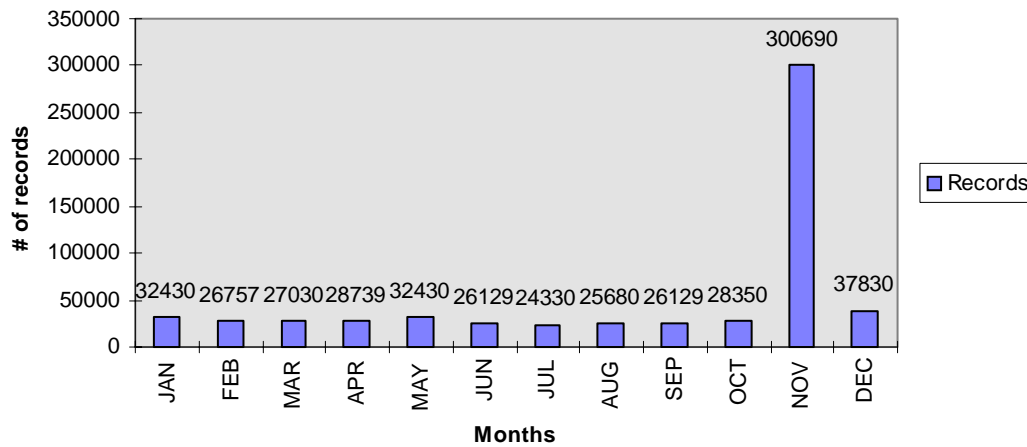
Rows to be inserted in SALES_1999 table from SALES_ODS table	Oracle8i (without intra-partition parallelism)	Oracle9i (with intra-partition parallelism)
616,524	10.68 s	7.40 s
1,233,048	16.37 s	10.68 s
2,466,096	32.31 s	18.46 s
4,932,192	58.00 s	31.66 s

### Parallel Inserts



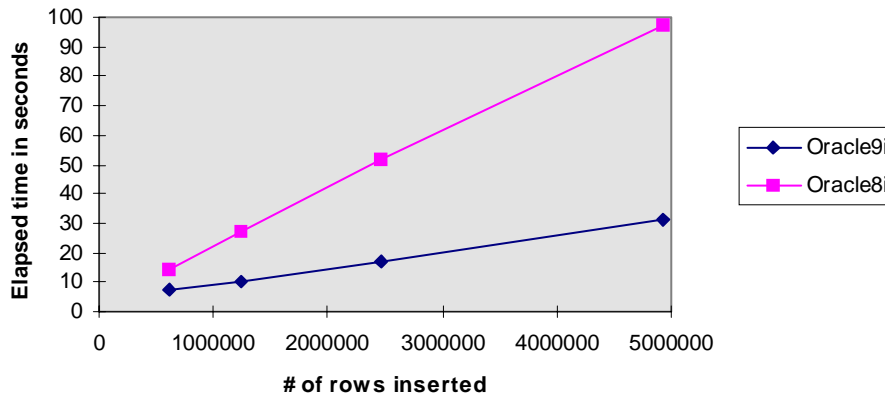
If we increase the data skew to have an extreme spike for the month of NOVEMBER 1999, we can see even more performance gain. For some industrial sectors, such as consumer retailing for seasonal products, spikes are normal patterns. The performance of parallel inserts in Oracle8 *i* deteriorates with the increase in skew whereas performance with Oracle9*i* is more or less constant. The following results are for the more skewed distribution of data (graph of distribution given below).

### Distribution of Data in Partitions



Rows to be inserted in SALES_1999 table from SALES_ODS table	Oracle8i (without intra-partition parallelism)	Oracle9i (with intra-partition parallelism)
616,524	14.17 s	7.34 s
1,233,048	27.13 s	10.43 s
2,466,096	51.66 s	16.96 s
4,932,192	97.16 s	31.22 s

### Parallel Inserts (more skew)



Intra-partition parallelism of inserts in Oracle9i achieves the following to give better performance over Oracle8i Parallel Insert (PDML).

1. More CPU Usage - In Oracle9i this operation is almost completely CPU bound whereas in Oracle8i it is not
2. Better Use of IO bandwidth - In Oracle9i there are more transactions per second (tps) than Oracle8i
3. More IO throughput - In Oracle9i there is more Mbps of IO transferred than Oracle8i.

### Merge

The MERGE statement is a new SQL statement introduced in Oracle9i that provides the ability to update or insert rows conditionally into a table or view. In a data warehousing environment, tables need to be periodically refreshed with the new data arriving from on-line systems. This new data (from a Source Table) might contain changes to the existing rows of the warehouse table (Destination table) or might introduce a new row altogether. Thus, depending on the existence of the key for this row in the data warehousing table a suitable insert or update needs to be generated. This is exactly what is achieved by the MERGE statement. This statement is a convenient way to combine these two operations and avoid the need to use multiple INSERT and UPDATE statements. It is particularly useful in Materialized View refreshes which can be done more efficiently with MERGE.

Prior to Oracle9i, the same functionality could be achieved by one of the following two methods:

1. PL/SQL programs that use cursors, to UPDATE or INSERT each row
2. A sequence of DML statements, i.e., UPDATE of a JoinView followed by an INSERT

The first approach suffers from important drawbacks: PL/SQL will be a little slower than SQL, and it cannot be parallelized. The second approach requires two joins of the Source and the Destination table when only one should be required.

MERGE statement can be parallelized and the performance is better as it used one outer-join internally. Hence, the performance gain is of the order of  $\text{Scan}(\text{Source table}) + \text{Scan}(\text{Destination table}) + \text{JoinCost}(\text{Source, Destination})$ .

### **Test Case for MERGE:**

The fact table SALES\_FACT in a data warehouse needs to be periodically updated with sales data coming in from the on-line systems. If there is a new store that is added, then the new data needs to be inserted into the SALES\_FACT table.

**Oracle8i Implementation:** The sequence of DML statements can be implemented as follows:

```
UPDATE /*+ BYPASS_UJVC */
(SELECT
  S.TIME_ID ,S.STORE_ID ,S.REGION_ID
 ,S.PARTS s_parts ,S.SALES_AMT s_sales_amt ,S.TAX_AMT s_tax_amt ,S.DISCOUNT s_discount
 ,D.PARTS d_parts ,D.SALES_AMT d_sales_amt ,D.TAX_AMT d_tax_amt ,D.DISCOUNT d_discount
FROM SALES_NOV99 S, SALES_FACT D
WHERE  D.TIME_ID = S.TIME_ID
AND D.STORE_ID = S.STORE_ID
AND D.REGION_ID = S.REGION_ID) JV
  SET d_parts = d_parts + s_parts
    , d_sales_amt = d_sales_amt + s_sales_amt
    , d_tax_amt = d_tax_amt + s_tax_amt
    , d_discount = d_discount + s_discount
;
commit;
INSERT INTO SALES_FACT (
  TIME_ID,STORE_ID ,REGION_ID
 ,PARTS ,SALES_AMT ,TAX_AMT ,DISCOUNT)
SELECT
  S.TIME_ID ,S.STORE_ID ,S.REGION_ID
 ,S.PARTS ,S.SALES_AMT ,S.TAX_AMT ,S.DISCOUNT
FROM SALES_NOV99 S
WHERE  (S.TIME_ID, S.STORE_ID, S.REGION_ID) NOT IN (SELECT /*+ HASH_AJ */
  D.TIME_ID, D.STORE_ID, D.REGION_ID
  FROM SALES_FACT D
  )
;
commit;
```

**Oracle9i Implementation:** In Oracle9i, the MERGE statement can replace the above sequence of DMLs

```
MERGE INTO SALES_FACT D
  USING SALES_NOV99 S
  ON (D.TIME_ID = S.TIME_ID
    AND D.STORE_ID = S.STORE_ID
    AND D.REGION_ID = S.REGION_ID)
  WHEN MATCHED THEN
  UPDATE
  SET d_parts = d_parts + s_parts
    , d_sales_amt = d_sales_amt + s_sales_amt
    , d_tax_amt = d_tax_amt + s_tax_amt
    , d_discount = d_discount + s_discount
  WHEN NOT MATCHED THEN
  INSERT (D.TIME_ID ,D.STORE_ID ,D.REGION_ID
    ,D.PARTS ,D.SALES_AMT ,D.TAX_AMT ,D.DISCOUNT)
  VALUES (
    S.TIME_ID ,S.STORE_ID ,S.REGION_ID
    ,S.PARTS ,S.SALES_AMT ,S.TAX_AMT ,S.DISCOUNT);
commit;
```

Experiments were conducted using the above example and the performance of MERGE was measured in serial and parallel execution and compared against the sequence of DMLs (SEQDML). The percentage of the number of rows in the Source table (SALES\_NOV99) eligible for update was varied from 0 (all inserts case) , to 100 (all updates) for both serial and parallel execution.

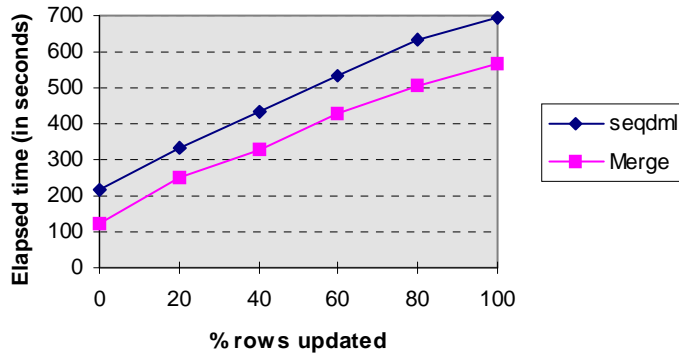
Serial Execution: SALES\_FACT contained 5 million rows and SALES\_NOV99 contained 1 million rows  
Parallel Execution: SALES\_FACT contained 50 million rows (10 partitions), and SALES\_NOV99 contained 5 million rows. The degree of parallelism (DOP) was 10.

The following tables have the results of these tests:

**Serial Execution (Elapsed times in seconds)**

	%Rows in SALES_NOV99 eligible for UPDATE in SALE_FACT table					
	0	20	40	60	80	100
seqdml	217.027	331.583	432.687	536.003	631.687	693.543
merge	120.097	251.69	330.273	425.69	504.49	568.55
<b>% gain</b>	<b>44.66</b>	<b>24.09</b>	<b>23.669</b>	<b>20.58</b>	<b>20.14</b>	<b>18.02</b>

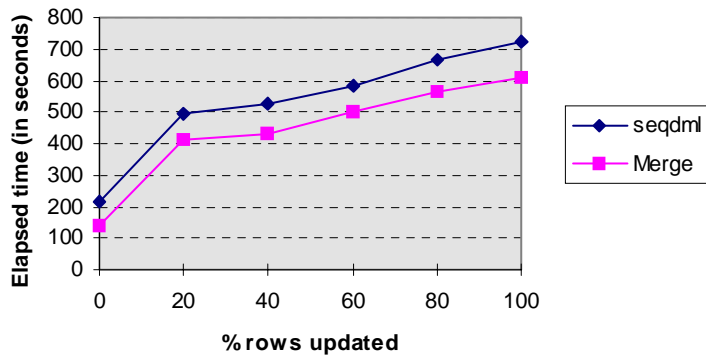
**MERGE vs SEQDML(serial)**



**Parallel Execution (Elapsed time in seconds)**

	%Rows in SALES_NOV99 eligible for UPDATE in SALE_FACT table					
	0	20	40	60	80	100
seqdml	213.643	495.83	527.953	581.75	664.683	721.203
merge	139.97	411.007	433.427	501.443	567.873	607.613
<b>% gain</b>	<b>34.48</b>	<b>17.11</b>	<b>17.904</b>	<b>13.8</b>	<b>14.56</b>	<b>15.75</b>

**MERGE vs SEQDML(parallel)**



The results show that MERGE is a performance enhancement that will simplify and speed up data warehousing applications both in serial and parallel environments.

## Multi-table Insert

The `INSERT . . . SELECT` statement in Oracle8*i* provides several performance benefits when dealing with large amounts of data. Chief among these are the direct-load access path and the ability to parallelize the insert operation. The existing statement, however, is limited to inserting rows into a single target table. In Oracle9*i* a new feature called **multi-table insert** is introduced by which data can be inserted into multiple tables in a single DML statement in SQL.

In Oracle8*i*, if data needed to be inserted into multiple tables then multiple `INSERT...SELECT` statements would have to be issued and the source data scanned once for each target table as opposed to just once for the entire operation in Oracle9*i*. In addition, if the `SELECT` statement itself describes a complex transformation (which it usually does, e.g., `GROUP BY`), then the transformed data is either recomputed for each scan or materialized beforehand in a temporary table, which consumes large amounts of space and is clearly undesirable. The performance gain with multi-table insert is thus directly proportional to the amount and complexity of the source data involved. In case of a single target table, there is no performance gain.

A DSS system which tends to deal with large amounts of data in fewer "long" transactions will benefit from this feature. The multi-table insert feature specifically focuses on enhancing performance during the critical ETL (extraction, transformation, and load) process by allowing data to be moved into multiple tables in one operation. This is expected to have broad applicability within the data warehousing environment.

### Test Case for Multi-Table Insert:

Consider a situation where two source tables are used to load three target tables. The `LINEITEM` and `ORDERS` tables are read and used to load the "major" orders (defined as orders with quantity > 110) into one table, "minor" orders (defined as orders with quantity < 100) into another and the rest ("medium") in a third table. This example uses a TPC-H schema.

**Oracle8i Implementation:** The following sequence of SQL statements will be issued

```
INSERT INTO ORDERS_QUANT_MINOR (ORDERKEY,QUANT_SUM)
SELECT O_ORDERKEY ORDERKEY,SUM(L_QUANTITY) QUANT_SUM
FROM LINEITEM,ORDERS
WHERE LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
GROUP BY O_ORDERKEY
HAVING SUM(L_QUANTITY) < 100;

INSERT INTO ORDERS_QUANT_MAJOR (ORDERKEY,QUANT_SUM)
SELECT O_ORDERKEY ORDERKEY,SUM(L_QUANTITY) QUANT_SUM
FROM LINEITEM,ORDERS
WHERE LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
GROUP BY O_ORDERKEY
HAVING SUM(L_QUANTITY) > 110;

INSERT INTO ORDERS_QUANT_MEDIUM (ORDERKEY,QUANT_SUM)
SELECT O_ORDERKEY ORDERKEY,SUM(L_QUANTITY) QUANT_SUM
FROM LINEITEM,ORDERS
WHERE LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
GROUP BY O_ORDERKEY
HAVING SUM(L_QUANTITY) BETWEEN 100 AND 110;
```

**Oracle9i Implementation:** The following statement will be issued

```
INSERT FIRST
WHEN QUANT_SUM < 100 THEN
  INTO ORDERS_QUANT_MINOR VALUES (ORDERKEY,QUANT_SUM)
WHEN QUANT_SUM > 110 THEN
  INTO ORDERS_QUANT_MAJOR VALUES (ORDERKEY,QUANT_SUM)
ELSE
  INTO ORDERS_QUANT_MEDIUM VALUES (ORDERKEY,QUANT_SUM)
SELECT O_ORDERKEY ORDERKEY,SUM(L_QUANTITY) QUANT_SUM
FROM LINEITEM,ORDERS
WHERE LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
GROUP BY O_ORDERKEY;
```

Experiments were conducted using Conventional Inserts, Direct Load Inserts (serial) and Direct Load Inserts (parallel degree 8) and it was found that Multi Table Insert showed a tremendous improvement over its Oracle8 *i* counterpart of a series of multiple INSERT... .SELECT statements.

The number of records in LINEITEM is 6,001,215 and ORDERS is 1,500,000.

The following table presents the results of the test case:

Statistic	Conventional Serial		Direct Serial		Direct Parallel	
	Oracle8 <i>i</i> Multiple INSERT...S ELECT	Oracle9 <i>i</i> MTI	Oracle8 <i>i</i> Multiple INSERT...SELEC T	Oracle9 <i>i</i> MTI	Oracle8 <i>i</i> Multiple INSERT...S ELECT	Oracle9 <i>i</i> MTI
Elapsed Time	1768	623	1784	649	181	70
Physical Reads	477 K	162 K	472 K	162 K	426 K	142 K

### Data Block Prefetching

Data Block prefetching is an internal optimization that is introduced in Oracle9 *i* which can improve response times of queries significantly in some cases. Data block prefetching is used by table lookup. When an index access path is chosen and the query can not be satisfied by the index alone, the data rows pointed to by the rowid also need to be fetched. This rowid to data row access (table lookup) is improved using data block prefetching which involves reading an array of blocks which are pointed by an array of qualifying rowids. Block prefetching allows better utilization of the I/O capacity and reduction in response time by issuing reads in parallel (whenever possible).

Data block prefetching will be useful when the index is poorly clustered and the table access tends to involve random disk accesses with an overall poor buffer cache hit ratio. In such cases, the query can easily become I/O bound, waiting for these single blocks to be read into the cache synchronously, even though there may be available I/O bandwidth on the system. Data block pre-fetching delays reading in a table block until a number of rowids satisfying the query have been accumulated from the underlying index. Waiting much less often for an I/O to complete helps the database instance better utilize the CPU resources.

### Test Case with Data Block Prefetching:

Query 17 from the TPC-H specification is a classic example where data block prefetch can be used. For a given brand and a given container type, what would be the average yearly gross loss in revenue if the orders for those parts with a quantity less than 20% of the average were no longer received? The database contains data for 7 years.

**Implementation:** The following query (DOP 16) will be used for implementing the above problem in both Oracle8*i* and Oracle9*i*:

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    parts,
    lineitem l1
where
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container = 'MED BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem l2
        where
            l_partkey = p_partkey);
```

Time taken for the Query to execute on **Oracle8i** = **264 seconds**

Time taken for the Query to execute on **Oracle9i** with Data Block Prefetch = **180 seconds**, which is an improvement of **31.82%**

## Index Skip Scan

In prior releases, a composite index would only be used if the index prefix column was included in the predicate of the statement. With Oracle9i, the optimizer can use a composite index even if the predicate column does not contain the prefix column. The optimizer uses an algorithm called skip scanning to retrieve the row address (ROWID) for values that do not use the prefix column. Index Skip Scan is based on the fact that it is faster to scan index blocks than to scan table's data blocks. Skip scans reduce the need to add another index (thereby saving space) to support occasional queries which do not reference the prefix column of an existing index. This can be useful when high levels of DML activity would be degraded by the existence of too many indexes used to support infrequent queries.

While not as fast as a direct index look up, the skip scan algorithm is faster than a full table scan in cases where the number of distinct values in the prefix column is relatively low. The optimizer uses statistics to determine whether a skip scan retrieval would be more efficient than a full table scan, or other possible retrieval paths, when parsing SQL statements. It is called skip scan because, while searching for the non prefix column in the composite index, it can skip a sub-tree lookup depending on the low and high values for the non-prefix column at the current node.

### Test Case for Index Skip Scan:

Assume that a national registry of motor vehicles includes a table holding registered owners of all the vehicles, called REGISTERED\_OWNERS (described below) which has a composite index on (STATE, REGISTRATION#). You want to find out the details of the owner of a particular REGISTRATION# and the STATE is not known

```
REGISTERED_OWNERS
(
  STATE          VARCHAR2(2)  NOT NULL,
  REGISTRATION#  VARCHAR2(10) NOT NULL,
  FIRST_NAME     VARCHAR2(30),
  LAST_NAME      VARCHAR2(30),
  MAKE           VARCHAR2(30),
  MODEL          VARCHAR2(15),
  YEAR_MFG       NUMBER,
  CITY           VARCHAR2(30),
  ZIP            NUMBER
)
```

**Implementation:** The above query can be formulated in both Oracle8i and Oracle9i as

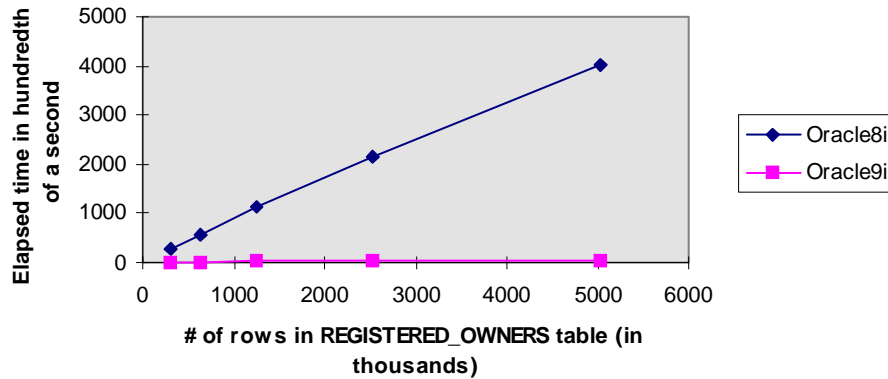
```
SELECT first_name, last_name, zip
FROM REGISTERED_OWNERS WHERE registration# = '4FBB428'
```

In Oracle8i, the optimizer will choose a full table scan for the query (unless there exists another index with REGISTRATION# as the prefix). But in Oracle9i, the optimizer (depending on the statistics) will use the composite index on (STATE, REGISTRATION#) because the cardinality of the STATE column is fairly low and return all the rows that match the registration# 4FBB428. This is much faster than full table scan.

This new feature is transparent to users and applications. The Oracle9 i optimizer uses statistics to determine whether a skip scan retrieval would be more efficient than a full table scan. Tests were conducted with five states in the REGISTERED\_OWNERS table by varying the number of rows and running the above query. The test results shown in the table and chart below prove the tremendous speed advantage of the new index skip scan algorithm. Further, the results show that the relative speed advantage grows as the table size increases.

Number of records in REGISTERED_OWNERS table	Oracle8i (hundredth of a second)	Oracle9i (Index Skip scan) (hundredth of a second)
315 K	287	12
630 K	585	16
1260 K	1135	21
2520 K	2142	24
5040 K	4035	25

### Query Performance (SkipScan vs. No Skip Scan)



### Pipelined Table Functions

Table functions are defined as functions written in PL/SQL, Java, or C that can produce a set of rows as output. Additionally, table functions can take a set of rows or cursors as input. SQL can be used to iterate over the set of rows returned by a table function. Specifically, a table function can be queried by using the TABLE keyword when it is placed in the FROM clause. Oracle9i supports pipelined table function and parallel execution of a table function.

Since the table functions can accept and produce rows of input and output, they can perform complex transformations on data sets without ever needing to load the data into intermediate staging tables. By avoiding the disk access needed to create staging tables, the table functions can save a huge amount of IO time. Oracle refers to the combination of parallel processing without intermediate staging tables as pipelined table functions. In a data warehouse environment, table functions will be a great benefit for ETL processing.

### Test Case for Table Functions:

Since table functions are most used in ETL tasks, the description of this test case is relatively lengthy. Consider a data warehouse which receives sales records in non-normalized form, with each record holding seven days worth of data:

```
sourceT
(
  product_id      NUMBER,
  store_id        NUMBER,
  weekly_start_date DATE,
  sales_sun       NUMBER,
  sales_mon       NUMBER,
  sales_tue       NUMBER,
  sales_wed       NUMBER,
  sales_thu       NUMBER,
  sales_fri       NUMBER,
  sales_sat       NUMBER
);
```

In the data warehouse, records are stored in the SALES table in a normalized form, with one day per row:

```
sales
(
  product_id      NUMBER,
  store_id        NUMBER,
  time_id         DATE,
  sales_amount    NUMBER,
  sales_bin_name  VARCHAR2(7)
);
```

Therefore, we need one transformation such that each record in the input stream is converted into seven records for a temporary (ODS table) table TEMP\_SALES. This transformation is commonly referred to as *pivoting*.

We also want to assign each sales value to a "bin," a category indicating its relative magnitude. Assigning values to bins, called "binning," is an important part of preparing data for analysis. Therefore, we need another transformation to find the SALES\_BIN\_NAME value for each row of the resulting table. This transformation called *binning* could merely involve a table lookup into a bin table (say PRODUCT\_BIN - described below) for every row of the TEMP\_SALES table.

```
temp_sales
(
  product_id      NUMBER,
  store_id        NUMBER,
  time_id         DATE,
  sales_amount    NUMBER
);

product_bin
(
  product_id      NUMBER,
  sales_bin_name  VARCHAR2(7),
  min_sales_amount NUMBER,
  max_sales_amount NUMBER
);
```

In Oracle8i, the transformations were implemented using multi-stage data transformations. Pivoting and binning will consist of the following steps: transform the source table (in the non relational form) to the format of TEMP\_SALES table, materialize the TEMP\_SALES table, join the TEMP\_SALES table with the PRODUCT\_BIN table and move the final result to SALES table. Therefore in multi-stage data transformation, there is a serial process of transform-materialize- transform.

In Oracle9i, the transformations can be pipelined using table functions. In pipelined data transformation, we no longer need to materialize the intermediate staging table. Instead, we can do transform-while-transforming. The whole process becomes more efficient, more scalable and non interruptive.

### **Oracle8i Implementation:**

```
CREATE TABLE temp_sales (product_id number, store_id number, time_id date, sales_amount number);

DECLARE
  CURSOR c1 IS
    SELECT product_id, store_id, weekly_start_date, sales_sun, sales_mon, sales_tue,
           sales_wed, sales_thu, sales_fri, sales_sat
    FROM sourceT
BEGIN
  FOR crec IN c1 LOOP
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date, crec.sales_sun);
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date+1, crec.sales_mon);
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date+2, crec.sales_tue);
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date+3, crec.sales_wed);
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date+4, crec.sales_thu);
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date+5, crec.sales_fri);
    INSERT INTO temp_sales VALUES (crec.product_id, crec.store_id,
                                   crec.weekly_start_date+6, crec.sales_sat);
```

```

END LOOP;
COMMIT;
END;

```

Once the TEMP\_SALES table is materialized, binning can be done by a SQL query as follows.

```

create table SALES NOLOGGING as
select s1.product_id, s1.store_id, s1.time_id, s1.sales_amount, bin.sales_bin_name
from temp_sales s1, product_bin bin
where s1.product_id = bin.product_id and
      s1.sales_amount >= bin.min_sales_amount and
      s1.sales_amount < bin.max_sales_amount;

```

**Oracle9i Implementation:** In Oracle9i, one can define a pipelined table function SalesPivot that takes a cursor of the sourceT table as input, and output a set of sales table records.

```

create or replace type sourceT_obj as object (
product_id      NUMBER,
store_id        NUMBER,
weekly_start_date DATE,
sales_sun       NUMBER,
sales_mon       NUMBER,
sales_tue       NUMBER,
sales_wed       NUMBER,
sales_thu       NUMBER,
sales_fri       NUMBER,
sales_sat       NUMBER);

CREATE TYPE sales_t AS OBJECT
( product_id      NUMBER,
  store_id        NUMBER,
  time_id         DATE,
  sales_amount    NUMBER);

CREATE TYPE sales_set_t AS TABLE OF sales_t;

CREATE OR REPLACE PACKAGE RefCur_pack AS
type rec is record (t sourceT_obj);
type cur IS REF CURSOR RETURN rec;
type ret_rec is record (t1 sales_t);
END RefCur_pack;

CREATE FUNCTION SalesPivot(p RefCur_pack.cur) return sales_set_t
parallel_enable (partition p by any)
pipelined
is
rec RefCur_pack.rec;
ret_rec RefCur_pack.ret_rec;
begin
LOOP
FETCH p INTO rec;
EXIT WHEN (p%NOTFOUND);
ret_rec.t1 := sales_t(NULL, NULL, NULL, NULL);

ret_rec.t1.product_id := rec.t.product_id;
ret_rec.t1.store_id := rec.t.store_id;
ret_rec.t1.time_id := rec.t.weekly_start_date;
ret_rec.t1.sales_amount := rec.t.sales_sun;
pipe ROW(ret_rec.t1);

ret_rec.t1.time_id := rec.t.weekly_start_date+1;
ret_rec.t1.sales_amount := rec.t.sales_mon;
pipe ROW(ret_rec.t1);

ret_rec.t1.time_id := rec.t.weekly_start_date+2;
ret_rec.t1.sales_amount := rec.t.sales_tue;
pipe ROW(ret_rec.t1);

ret_rec.t1.time_id := rec.t.weekly_start_date+3;
ret_rec.t1.sales_amount := rec.t.sales_wed;
pipe ROW(ret_rec.t1);

ret_rec.t1.time_id := rec.t.weekly_start_date+4;
ret_rec.t1.sales_amount := rec.t.sales_thu;
pipe ROW(ret_rec.t1);

```

```

ret_rec.t1.time_id := rec.t.weekly_start_date+5;
ret_rec.t1.sales_amount := rec.t.sales_fri;
pipe ROW(ret_rec.t1);

ret_rec.t1.time_id := rec.t.weekly_start_date+6;
ret_rec.t1.sales_amount := rec.t.sales_sat;
pipe ROW(ret_rec.t1);
end loop;
return;
end;

```

Once the table function SalesPivot is defined, the two transformations, pivoting and binning, can be integrated into one single SQL query:

```

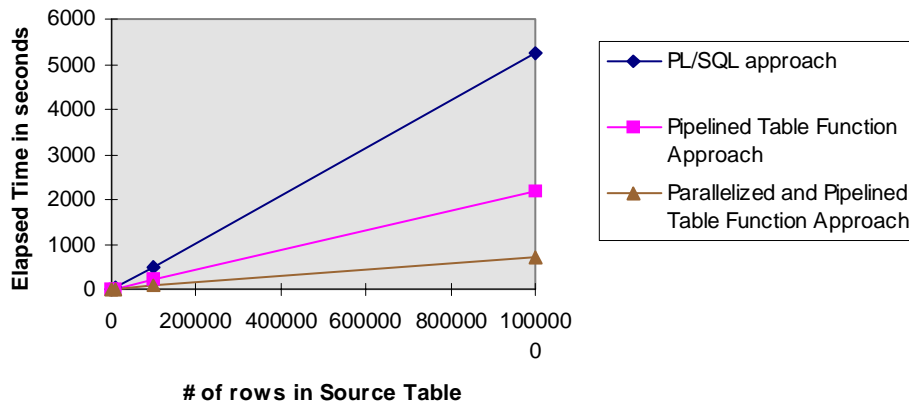
CREATE TABLE sales NOLOGGING AS
select s1.product_id, s1.store_id, s1.time_id, s1.sales_amount, bin.sales_bin_name
FROM TABLE(SalesPivot(cursor(select ... from sourceT))) s1,
product_bin bin
WHERE s1.product_id = bin.product_id and
s1.sales_amount >= bin.min_sales_amount and
s1.sales_amount < bin.max_sales_amount;

```

With a fixed small table product\_bin (10 rows) and the size of the sourceT table varying from 1,000 rows to 1,000,000 rows, experiments were conducted to compare the performance and the results are listed in the table below. All the elapsed times are in seconds

Number of Rows in SourceT table	Oracle8i PL/SQL (A)	Oracle9i Pipelined TF (B)	Oracle9i Parallel Pipelined TF (C)	%gain (A) -> (B)	%gain (A) -> (C)
1,000	5.36	2.37	1.31	55.78	75.56
10,000	49.54	21.8	9.72	56	80.38
100,000	503.85	214.14	74.93	57.5	85.13
1,000,000	5237.32	2195.67	702.66	58	86.58

### ETL Transformation



We can see that the pipelined table function approach gives more than a 56% improvement over the PL/SQL procedure in terms of elapsed time. With parallel degree 20, the performance gain is as high as 86% over the PL/SQL approach of Oracle8i.

**NOTE:** In the above example the second transformation (binning) was fairly simple using a join instead of another table function. It is possible to have more transformations, all of which are implemented as table functions, and these table functions could be nested in the SQL to create the resulting table.

### **Automatic Memory Management in PGA**

This feature introduces an automatic mode for allocating memory to working areas in Program Global Areas (PGA). It simplifies and improves the way PGA memory is allocated and tuned by Oracle. Before Oracle9i, existing parameters (like `sort_area_size`, `hash_area_size`, `bitmap_merge_area_size`, `bitmap_create_area_size`) used to control and limit PGA memory utilization could not be automatically adjusted to compensate for low or high memory usage. These parameters were defined once for all and could not vary when demand varies. New parameters introduced in Oracle9i allow database administrators to specify the policy for sizing work areas. In automatic mode, working areas used by memory-intensive operators can be automatically and dynamically adjusted, based on the PGA memory used by the system, the target aggregate PGA memory for the instance, and the requirement of each individual operator.

The new feature offers several performance and scalability benefits for DSS work loads or mixed workloads with complex queries, that is, queries where a large portion of the PGA runtime area is dedicated to working areas used by some memory intensive row sources such as sorts or hash-joins. The overall system performance is maximized, and the available memory is allocated more efficiently among queries to optimize both throughput and response time. In particular, improved use of memory translates to better throughput at high load.

This feature introduces two `init.ora` parameters. One, named **PGA\_AGGREGATE\_TARGET**, is a system parameter set by the DBA to specify the desired size of the pga memory for the instance. This is relatively easy for the DBA to compute (in most cases, `pga_aggregate_target = memory_available_for_oracle - sga_size`). The second parameter, **WORKAREA\_SIZE\_POLICY**, is a session and system level parameter which can take only two values, `MANUAL` or `AUTO`. This parameter allows users to select between automatic or manual tuning of work area sizes. The main motivation for the `MANUAL` value is backward compatibility with earlier versions of Oracle.

#### **Test Case for Automatic Memory Management:**

A query very similar to query 9 of TPC-H specification was run with multiple users in the following modes

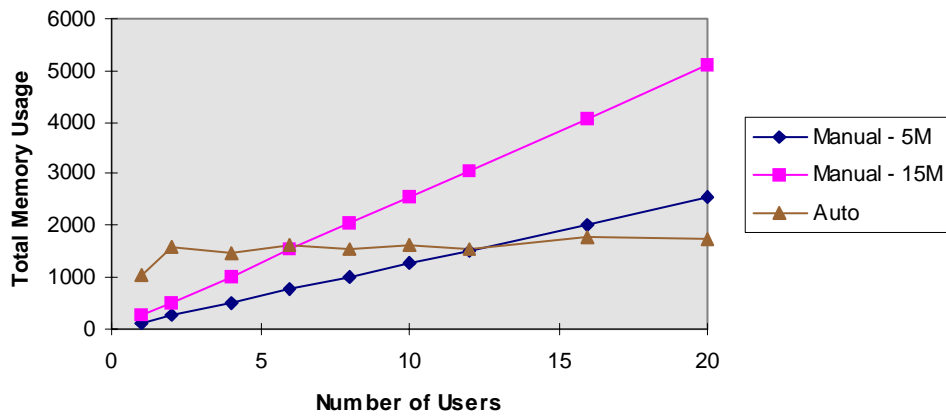
1. In `AUTO` mode,
2. In `MANUAL` mode with `SORT_AREA_SIZE` and `HASH_AREA_SIZE` as 5MB, and
3. In `MANUAL` mode with `SORT_AREA_SIZE` and `HASH_AREA_SIZE` as 15MB

The `PGA_AGGREGATE_TARGET` was set to 2GB in `AUTO` mode. The following observations were made for response time and the total memory consumption by the processes. The results show that automatic memory management adapts very well to a wide number of users and requests from processes.

MEMORY CONSUMPTION in MB

# of users	MANUAL 5MB	MANUAL 15 MB	AUTO
1	127	255	1054
2	254	510	1580
4	508	1020	1480
6	762	1530	1623
8	1016	2040	1550
10	1270	2550	1624
12	1524	3060	1550
16	2032	4080	1800
20	2540	5100	1750

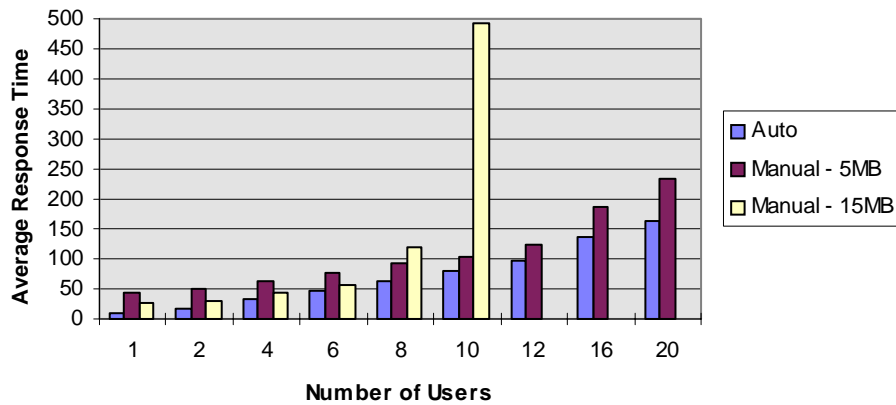
**Memory Consumption in MB**



RESPONSE TIME IN SECONDS

# of users	MANUAL 5MB	MANUAL 15 MB	AUTO
1	42	26	11
2	50	31	16
4	63	45	34
6	78	57	48
8	94	121	63
10	103	494	80
12	124		96
16	188		136
20	235		165

**Response Time in Seconds**



NOTE: In the response time measurements, in the MANUAL 15MB case, after 10 users, memory swapping started occurring. Hence we see a spike at 10 users and therefore measurements were not made for more users.

## Conclusion

The test cases described in this paper clearly show that the new features of Oracle9*i* greatly enhance performance for a broad range of important DSS tasks. The features discussed here include:

- List Partitioning
- Intra-Partition Parallelism
- Merge
- Multi-table Insert
- Data Block Prefetching
- Index Skip Scan
- Pipelined Table Functions
- Automatic Memory Management in PGA

The Oracle9*i* database is extremely scalable, delivers high performance and is significantly improved over the already powerful Oracle8*i* database. Oracle9*i* contains features that exceed the current requirements of the most demanding DSS environments and is flexible enough to accommodate the growing needs of businesses.

Applications currently running on Oracle8 *i* or prior versions will see major performance benefits when they leverage the new features of Oracle9*i*.



Oracle9i Performance and Scalability in DSS Environments

April 2001

Author: Neil Thombre

Contributing Authors: Linan Jiang, Sumant Sarkar

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

Worldwide Inquiries:  
Phone: +1.650.506.7000  
Fax: +1.650.506.7200  
[www.oracle.com](http://www.oracle.com)

Oracle Corporation provides the software  
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various  
product and service names referenced herein may be trademarks  
of Oracle Corporation. All other product and service names  
mentioned may be trademarks of their respective owners.

Copyright © 2000 Oracle Corporation  
All rights reserved