NEW SQL CAPABILITIES IN ORACLE DATABASE 10*G*: REGULAR EXPRESSIONS, NATIVE NUMBERS, LOB PERFORMANCE, AND MORE

Geoff Lee, Oracle Corporation
Peter Linsley, Oracle Corporation
Jonathan Gennick, O'Reilly & Associates

INTRODUCTION

Oracle's SQL Engine is the underpinning of all Oracle Database applications. Oracle SQL continually evolves to meet the growing demands of increasingly more sophisticated database applications, and to enable emergent standard-based computing architectures, APIs, and network protocols.

Oracle Database 10*g* features several new SQL capabilities, making Oracle the best implementation of one of the most popular and enduring standards in computing. This paper looks at *regular expressions*, *native numbers*, LOB performance, and other new features and enhancements to SQL in Oracle Database 10*g*. These newly integrated SQL capabilities are essential for developing high performance and scalable Life Sciences, Business Intelligence, Content Management, E-Business and other database applications.

In the next sections, we will cover the following topics in depth,

- The introduction of IEEE/POSIX standard native regular expression support to SQL in Oracle Database 10*g* revolutionizes the ability to search for and manipulate text within the database. Regular expressions are especially useful for dealing with loosely formatted, free-form textual data.
- The new native floating-point datatype based on the *IEEE 754 Standard for Binary Floating-Point Arithmetic* in Oracle Database 10*g* dramatically improves performance of floating-point processing common in XML and Java standards while substantially reducing storage space.
- To meet the demands of unstructured data processing, performance of LOB datatypes has been greatly improved in Oracle Database 10*g* throughout the stack (SQL layer and its APIs). Furthermore, the size limit of a LOB has been expanded well beyond the previous 4 gigabytes, to terabytes.
- There are numerous enhancements of collection types to provide versatile ANSI SQL standard Multiset operations and flexible storage options for nested-structured data prevalent in XML, Java, Business Intelligence, and Spatial applications.

ORACLE REGULAR EXPRESSIONS

Regular expressions provide a powerful means of identifying a pattern within a body of text. A pattern describes what the text to identify looks like and could be something quite simple, such as that which describes any three-letter word, to something quite complex, like that which describes an email address.

Pattern matching with regular expressions can be applied to all kinds of real word problems. They are used heavily in web applications to verify, parse, manipulate, and format data coming to and from the middle tier; a surprisingly large amount of middle tier processing is taken up with such string processing. They are used in bioinformatics to assist with identifying DNA and protein sequences. Linguists use regular expressions to aid research of natural languages. Server configuration is often done in terms of regular expressions such as in a mail server to help identify potential spammers, and are perhaps also used by the spammers themselves to effortlessly collect innocent victims email addresses from Internet based data stores. Many protocol and language standards accept regular expressions as filters and validation constructs. In short, it is hard to imagine an application that could not benefit from the functionality that regular expressions offer.

Oracle Regular Expressions remove the limitations of existing solutions by providing a sophisticated set of metacharacters for describing more complex patterns than previously possible, all native to the database available in both SQL and PL/SQL. Leveraging their power, an application will not only run more efficiently but it can have an improved data flow and be more robust.

Enhancing the Data Flow

While middle tier technologies have long had the ability to perform regular expression searching, support in the backend database is a valuable and often overlooked consideration. The introduction of Oracle Regular Expressions brings the database closer to the Internet by providing enhanced string manipulation features right within the database, providing the flexibility to perform regular expression based string manipulation at any tier.

Oracle Regular Expressions can be used in many data manipulation scenarios such as updating, selecting, and formatting for presentation. These scenarios are described in the following sections in terms of the data flow.

Database to Client

Regular expressions are especially helpful in a web application where data from the database needs to be filtered and formatted for presentation. As an example, consider an application that selects a phone number stored within a CHAR(10) column as a series of 10 digits in the format XXXXXXXXXX. The requirement is to format this column as (XXX) XXX-XXXX for presentation to the end user. Should this processing occur in the middle tier then other clients who have access to the same data will need to duplicate the formatting logic. A DBA querying the table through SQL*Plus, for example, will only be able to view the format as stored on disk without incorporating their own means of formatting. This forces all clients of the data to have special knowledge of how the telephone number is stored within the database; when clients have such an expectation it becomes difficult to change the backend format. As a means to resolve this it would be a trivial task to create a database view that uses the regular expression enabled replace function (REGEXP_REPLACE) to reformat the telephone number. All clients of the data will then simply need to query the view and will benefit from the centralized logic providing them only the pre-formatted version of the telephone number.

Regular expressions are often used in a client to filter and refine a result set to the desired rows. With the regular expression enabled LIKE condition (REGEXP_LIKE) this filtering can now easily occur within the database directly on the data minimizing network data flow and putting the job of getting the correct data right where it ought to be, in the hands of the database. Moving complex logic closer to the data in this manner creates a more robust environment as data not targeted for the end user does not have to leave the database and client processing is reduced.

Database Update

Oracle Database 10*g* allows you to perform updates in terms of regular expressions. Where traditionally it would have been easiest to select the query set, perform regular expression based updates within a client and write the results back to the database, it is now possible to perform all of this in a single update statement. The data never has to leave the database bringing benefits such as reduction in network traffic, tighter security, and improved performance.

Client to Database

Where regular expressions normally play a part in data flowing from client to database is in validating and formatting data for storage.

We may want to validate that a credit card number or an email address matches a certain pattern or we may want to reformat a user provided telephone number into a format that will make more sense when stored in the database. Much like the flow from database to client, while it is possible to perform such manipulation on the middle tier, only clients of the middle tier would have access to this logic.

When a regular expression describes the required format for data in a column, the regular expression itself is a property of the data and therefore should not be externalized in client logic. With column constraints defined in terms of regular expressions, we can bulletproof our database so that only data matching a certain pattern will be allowed into a table irrespective of the source that the data originates from, be it an external client or even an internal PL/SQL routine.

With the above scenarios in mind, it is evident that the combination of Oracle Regular Expressions and SQL is an extremely powerful one that can drastically change the way in which string manipulation and pattern matching is performed within an application.

ORACLE REGULAR EXPRESSIONS KEY FEATURES

Most regular expression implementations are based to some extent on the documented behaviour of Basic and Extended Regular Expressions (BRE and ERE) as described in the POSIX standard, often referred to as UNIX style regular expressions. The standard leaves plenty of room for extensions that most regular expression implementations readily take advantage of which means that in general, no two implementations are alike. It is because of this incompatibility issue that Oracle Regular Expressions are based on the POSIX ERE definition. This assures that any Oracle Regular Expression will have the same behaviour on similarly conformant implementations.

INTERFACES

Oracle Regular Expressions are implemented by the following interfaces available in both SQL and PL/SQL:

SQL Function	**Description**
REGEXP_LIKE	Determine whether pattern matches
REGEXP_SUBSTR	Determine what string matches the pattern
REGEXP_INSTR	Determine where the match occurred in the string
REGEXP_REPLACE	Search and replace a pattern

For detailed information on the valid arguments and syntax, refer to the sections on Conditions and Functions in the SQL Reference.

METACHARACTERS

For a complete list of supported metacharacters please refer to the Appendix C of the SQL Reference. They are listed here for reference and will not be described further.

Syntax	**Description**	**Classification**
.	Match any character	Dot
a?	Match ‘a’ zero or one time	Quantifier
a*	Match ‘a’ zero or more times	Quantifier
a+	Match ‘a’ one or more times	Quantifier
a|b	Match either ‘a’ or ‘b’	Alternation
a{m}	Match ‘a’ exactly m times	Quantifier
a{m,}	Match ‘a’ at least m times	Quantifier
a{m,n}	Match ‘a’ between m and n times	Quantifier
[abc]	Match either ‘a’ or ‘b’ or ‘c’	Bracket Expression
(…)	Group an expression	Subexpression
\n	Match nth subexpression	Backreference
[:cc:]	Match character class in bracket expression	Character Class
[.ce.]	Match collation element in bracket expression	Collation Element
[=ec=]	Match equivalence class in bracket expression	Equivalence Class

Locale Support

Locale support refers to how the regular expression will behave under the properties of a given character set, language, territory, and sort order. As regular expressions are for matching text, and text is not limited to English, it stands to reason that they should be able to handle text in any language or character set and that the unique properties of that locale should be honoured. Most standards are rather vague when it comes to supporting different locales. POSIX, while stating that there are some considerations that should be taken to support locales, does not match up to the locale definition that Oracle provides.

Oracle Regular Expression pattern matching is sensitive to the underlying locale defined by the session environment. This affects all aspects of matching including whether it will be case or accent insensitive, whether a character is considered to fall within a range, what collation elements are considered valid, and so on. The engine is also strictly character based, as an example the dot (.) will match a single character in the current character set and never a single byte of the data.

USING REGULAR EXPRESSIONS IN ORACLE

Function Overview

This section introduces the functions that provide Oracle Regular Expressions support and shows some simple usage scenarios. All functions have similar signatures and support CHAR, VARCHAR2, CLOB, NCHAR, NVARCHAR, and NCLOB datatypes.

REGEXP_LIKE

The REGEXP_LIKE condition is a little different to the other REGEXP functions in that it only returns a Boolean indicating whether the pattern matched in the given string or not. No details on how or where the match occurred is provided. There is only one optional argument being the match option, position and occurrence are redundant.

Consider a usage example where you were given the task to write an expression that could search for rows containing common inflections of the verb 'fly'. The following regular expression would do the job nicely matching fly, flying, flew, flown, and flies.

```
SELECT c1 FROM t1 WHERE REGEXP_LIKE(c1, 'fl(y(ing)?|(ew)|(own)|(ies))');
```

REGEXP_SUBSTR

This function returns the actual data that matches the specified pattern. In cases where it is not obvious how the pattern matched, REGEXP_INSTR can be called to locate the exact character offsets. Using the scenario above:

```
SELECT  REGEXP_SUBSTR(
  'the bird flew over the river',
  'fl(y(ing)?|(ew)|(own)|(ies))') FROM dual;
→ flew
```

REGEXP_INSTR

The REGEXP_INSTR function performs a regular expression match and returns the character position of either the beginning or end of the match. Unlike INSTR, REGEXP_INSTR is not able to work from the end of a string. When analyzing the return value it often helps to view the match position as being the position right before the character count returned. Special care must be taken when using the output of this function as input to other SQL functions as they might not interpret the values in the same manner. Some common scenarios regarding the return values are described below.

REGEXP_REPLACE

The power of Oracle Regular Expressions really becomes evident when coupled with the ability to replace the pattern matched. This function works by looking for an occurrence of a regular expression and replacing it with the contents

of a supplied text literal. The replacement text literal can also contain backreferences to subexpressions included in the match giving extremely granular control over your search and replace operations.

Simple HTML Filter

With REGEXP_REPLACE it is simple to filter out certain parts of data. This example shows how a very simple HTML filter could be written.

```
SELECT REGEXP_REPLACE (c1, '<[^>]+>') FROM t1;
```

Using Backreferences

References to matched subexpressions are valid within the replacement string and are identified as \n where n refers to the nth subexpression. In order to specify the backslash (\) to be part of the replace expression, it must be escaped with the backslash as in '\\'.

Using with DDL

Constraints

You can use Oracle Regular Expressions to filter data that is allowed to enter a table by using constraints. The following example shows how a column could be configured to allow only alphabetical characters within a VARCHAR2 column. This will disallow all punctuation, digits, spacing elements, and so on, from entering the table.

```
CREATE TABLE t1 (c1 VARCHAR2(20), CHECK (REGEXP_LIKE(c1, '^[[:alpha:]]+$')));

INSERT INTO t1 VALUES ('newuser');
→ 1 row created.

INSERT INTO t1 VALUES ('newuser1');
→ ORA-02290: check constraint violated

INSERT INTO t1 VALUES ('new-user');
→ ORA-02290: check constraint violated
```

As the description of allowable data is tied in with the column definition, this acts as a shield to all incoming data so it is no longer required to filter such data on the client before inserting into the database.

Indexes

It is normal to improve performance when accessing a table by creating an index on frequently accessed and easily indexable columns. Oracle Regular Expressions are not easily able to make use of these indexes as they rely on knowing how the data within the column begins and do not lend well to functions that seek for columns with an abstract pattern. It is possible, however, to make use of functional indexes for cases where the same expression is expected to be issued on a column within a query. Functional indexes are created based on the results

```
CREATE INDEX t1_ind ON t1 (REGEXP_SUBSTR(c1, 'a'));

SELECT c1 FROM t1 WHERE REGEXP_SUBSTR(c1, 'a') = 'a';
```

Views

Views are a great mechanism for query subsetting or formatting data before it is presented to the end user. In this example we show how combining views with regular expressions makes it easy to reformat data. Suppose there was a requirement to mangle an email address, perhaps in order to avoid automatic detection but remain readable. One way we could do this would be to insert a space between each character. With REGEXP_REPLACE and backreferences, we are able to replace every letter by itself followed by a space:

```
CREATE VIEW v1 AS
  SELECT empno,
  REGEXP_REPLACE(email, '(.)', '\1 ‘) email
  FROM emp;

SELECT email FROM v1 WHERE empno = 7369;
→ j d o e @ s o m e w h e r e . c o m
```

USING WITH PL/SQL

The Oracle Regular Expression functions do not have to be part of a SQL statement and are fully supported as a PL/SQL built-in function.

As an example, to create a function that performs several regular expression operations on the provided data, the code would look something like the following:

```
src := REGEXP_REPLACE (src, ‘<regexp_1>’);
src := REGEXP_REPLACE (src, ‘<regexp_2>’);
src := REGEXP_REPLACE (src, ‘<regexp_3>’);
```

PL/SQL can also be used to enhance regular expression functionality. The following shows some PL/SQL that could be used to create a function that returns the nth subexpression:

```
CREATE FUNCTION regexp_subx (
  input VARCHAR2,
  regx  VARCHAR2,
  subx  NUMBER) RETURN VARCHAR2
IS
  ret VARCHAR2(4000);
BEGIN
  ret := REGEXP_SUBSTR (input, regx);
  ret := REGEXP_REPLACE (ret, regx, ‘\’ || subx);

  RETURN (ret);
END regexp_subx;
/
```

PERFORMANCE CONSIDERATIONS

Due to the inherent complexity of the compile and match logic, regular expression functions can perform slower than their non-regular expression counter parts. When an expression is provided, internally a compiler begins to process the string to try to convert it to an internal format, this process also ensures that the regular expression is well formed and contains no errors. This cost alone can be expensive especially when compared to parsing the arguments of LIKE for which it is impossible to construct a badly formed expression. It is also possible that the regular expression functions run faster as the compiled regular expression is highly optimized at run time. This is especially true when a complex regular expression would have to be written in a number of different SQL conditions rather than a single regular expression based function. Bear in mind that the regular expression functions are not able to make use of normal indexes.

The compilation is a little different for REGEXP_LIKE as it is optimized to work in a manner where it is not required that Oracle proves where a match occurs, we only need to prove whether the pattern occurs in the string or not. For this reason, REGEXP_LIKE can be considerably faster than other regular expression functions and may be used as a preprocessing phase to the other, more expensive, regular expression functions.

For complex expressions that would require several LIKE statements or possibly PL/SQL logic, Oracle Regular Expressions can be considerably faster at performing the same logic within a single function but it is difficult to quantify how much faster as it depends not only on the regular expression, but also on the data being matched against.

Sometimes the combination of regular expression and a particular segment of text can provide many thousands of alternatives that Oracle has to process in order to prove that a match exists, or in the worst case, that it does not exist as every possible alternative must be exhausted.

NATIVE FLOATING POINT DATATYPES

Oracle Database 10*g* introduces two new native floating point datatypes. The `BINARY_FLOAT` and `BINARY_DOUBLE` datatypes store floating-point data in the 32-bit IEEE 754 format and the double precision 64-bit IEEE 754 format respectively. Compared to the Oracle `NUMBER` datatype, arithmetic operations on floating-point data is usually faster for `BINARY_FLOAT` and `BINARY_DOUBLE`. Also, values with significant precision will require less space when stored as `BINARY_FLOAT` and `BINARY_DOUBLE`.

Arithmetic operations on **`BINARY_FLOAT`** and `BINARY_DOUBLE` datatypes, and related client interfaces supported by the database, are performed by the native instruction set supplied by the hardware vendor. Compliance with the IEEE 754 standard ensures common results across supported platforms. In addition, the native floating-point dataypes has many salient features,

- Create tables with columns of type binary_float or binary_double.
- Include binary_float/binary_double columns in a select list.
- Create indexes on binary_float/binary_double columns.
- Aggregation is supported for binary_float/binary_double columns.
- binary_float/binary_double are supported in *order by* and *group by* clauses.
- Storage of binary_float/binary_double is platform independent.

In the past, Oracle Number is the only datatype for numeric values in an Oracle RDBMS and all arithmetic operations are performed using this datatype. The benefits of Oracle Number include:

- Oracle Number is portable because it is implemented in software.
- Oracle Number uses a decimal representation. Precision is not lost when strings are converted to numeric values, and rounding is done on decimal values. Many applications require this behavior.
- By limiting the RDBMS to use one numeric datatype, most of the arithmetic operators and math functions are not overloaded. Excluding date datatypes, arithmetic is Oracle Number arithmetic and math functions only support Oracle Number. When more numeric datatypes are supported, arithmetic operators and math functions are overloaded, and the rules for resolving overloaded operators and functions introduce complexity into the implementation and description of expression evaluation.

However, Java and XML are prevalent in database applications today. These languages support *IEEE 754 Standard for Binary Floating-Point Arithmetic*. Furthermore, many database applications require extensive floating point computations (e.g., Life Sciences, OLAP, Data Mining, etc.). Therefore, a wide range of applications will benefit from improved performance, reduced storage space, and greater functionality. **`BINARY_FLOAT`** and `BINARY_DOUBLE` do not replace Oracle Number. They are alternatives to Oracle Number that provide the following benefits:

- binary_float/binary_double match the datatypes used by RDBMS clients. Both Java and XML Schema support datatypes that are equivalent to IEEE 754 datatypes. Currently, numeric data must be stored as an Oracle Number and the conversion to Oracle Number may lose precision and may raise an error. Accuracy might be lost because Oracle Number uses base 10 and binary_float/binary_double use base 2. Also, the set of values that can be represented by Oracle Number is neither a subset nor a superset of the set of values that can be represented by binary_float or binary_double.
- Arithmetic is faster for binary_float/binary_double. It can be between 5 and 10 times faster than it is for Oracle Number.
- binary_float/binary_double may use fewer bytes to store values on disk. binary_float (double) requires 5 (9) bytes, including the length byte. Oracle Number uses from 1 to 22 bytes.

- IEEE 754 provides more functionality necessary for writing numerical algorithms and many of these features are not provided by Oracle Number.

Floating-Point Number System Concepts

The floating-point number system is a common way of representing and manipulating numeric values in computer systems. A floating-point number is characterized by three components: a *sign*, a signed *exponent* and a *significand*, and assumes a fixed *base*. Its value is the signed product of its significand and the base raised to the power of its exponent:

$(-1)^{sign} \cdot significand \cdot base^{exponent}$

A floating-point number format specifies how the three components of a floating-point number are represented. The choice of representation determines the range and precision of the values the format can represent. By definition, the range is the interval bounded by the smallest and the largest values the format can represent and the precision is the number of digits in the significand.

Formats for floating-point values support neither infinite precision nor infinite range. There are a finite number of bits to represent a number and only a finite number of values that a format can represent. A floating-point number that uses more precision than available with a given format will be rounded.

Floating-Point Formats

The table below shows the range and precision of the required formats in the IEEE 754 standard and those of Oracle `NUMBER`.

Range and Precision of IEEE 754 formats

Range and Precision	**Single-precision 32-bit**[Foot 1]	**Double-precision 64-bit1**	**Oracle NUMBER**
Max positive normal number	3.40282347e+38	1.7976931348623157e+308	10e125
Min positive normal number	1.17549435e-38	2.2250738585072014e-308	1e-130
Max positive subnormal number	1.17549421e-38	2.2250738585072009e-308	not applicable
Min positive subnormal number	1.40129846e-45	4.9406564584124654e-324	not applicable
Precision (decimal digits)	6 - 9	15 - 17	38 - 40

Comparison Operators for Native Floating-Point Datatypes

Comparison operators are defined for *equal to*, *not equal to*, *greater than*, *greater than or equal to*, *less than*, *less than or equal to*, and *unordered*. There are special cases:

- Comparisons ignore the sign of 0 (-0 is equal to +0 and -0 is not less than +0).
- All comparisons are false when at least one of the operands is *NaN* (i.e., Not a Number) and the comparison operator is any operator except for not equal.
- All comparisons are true when at least one of the operands is NaN and the comparison operator is not equal.

Arithmetic Operators for Native Floating-Point Datatypes

Arithmetic operators are defined for multiplication, division, addition, subtraction, remainder and square root. The rounding mode used to round the result of the operation can be defined. Exceptions can be raised when operators are performed. Exceptions can also be disabled.

Java, until recently, required floating-point arithmetic to be exactly reproducible. IEEE 754 does not require such behavior. IEEE 754 allows for the result of operations, including arithmetic, to be delivered to a destination that uses a range greater than that used by the operands to the operation. The result of a double-precision multiply can computed in an extended double-precision destination. When this is done, the result must be rounded as if the destination were single-precision or double-precision. However, the range of the result (the number of bits used for the exponent) can use the range supported by the wider (extended double-precision) destination. This may result in a double-rounding error in which the last significand bit of the result is incorrect.

This can only occur for double-precision multiplication and division on hardware that implements the IA-32 and IA-64 instruction set architecture. Thus, with the exception of this case and hardware bugs, arithmetic for these datatypes will be reproducible across platforms. When the result of a computation is NaN, all platforms will produce a value for which `expr IS NAN` is true. However, all platforms do not have to produce the same bit pattern.

Conversion Functions for Native Floating-Point Datatypes

Conversion functions are defined that convert between floating-point formats and string formats which use decimal precision. Precision may be lost during the conversion. Exceptions can be raised during conversion. The following conversions can be done:

- float to double
- double to float
- float/double to decimal (string)
- decimal (string) to float/double
- float/double to integer valued float/double

Numeric Functions Supporting Native Floating-Point Datatypes

SQL built-in numeric functions (e.g., ABS, ACOS, etc.) now support binary_float and binary_double datatypes. For example,

```
SQL> select ceil(1.2343243e2F) from dual;

CEIL(1.2343243E2F)
------------------
         1.24E+002
```

Aggregate Functions Supporting Native Floating-Point Datatypes

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in `ORDER BY` and `HAVING` clauses. They are commonly used with the `GROUP BY` clause in a `SELECT` statement, where Oracle divides the rows of a queried table or view into groups. In a query containing a `GROUP BY` clause, the elements of the select list can be aggregate functions, `GROUP BY` expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

With the introduction of native floating-point datatypes, a number of built-in aggregate functions (e.g., AVG, CORR, MAX, MIN, STDDEV, etc.) now support these new datatypes.

Analytic Functions Supporting Native Floating-Point Datatypes

Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. The group of rows is called a **window** and is defined by the analytic clause. For each row, a "sliding" window of rows is defined. The window determines the range of rows used to perform the calculations for the "current row". Window sizes can be based on either a physical number of rows or a logical interval such as time.

Analytic functions are the last set of operations performed in a query except for the final `ORDER BY` clause. All joins and all `WHERE`, `GROUP BY`, and `HAVING` clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or `ORDER BY` clause. Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

With the introduction of native floating-point datatypes, a number of built-in analytic functions (e.g., AVG, CORR, MAX, MIN, STDDEV, etc.) now support these new datatypes.

Constraints on Native Floating-Point Columns

Use one of the *constraints* to define an **integrity constraint--**a rule that restricts the values in a database. Oracle Database 10*g* lets you create six types of constraints for columns of floating-point datatypes.

- A **NOT NULL constraint** prohibits a database value from being null.
- A **unique constraint** prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.
- A **primary key constraint** combines a `NOT NULL` constraint and a unique constraint in a single declaration. That is, it prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null.
- A **foreign key constraint** requires values in one table to match values in another table.
- A **check constraint** requires a value in the database to comply with a specified condition.
- A `REF` column by definition references an object in another object type or in a relational table. A **REF constraint** lets you further describe the relationship between the `REF` column and the object it references.

The example below shows how these constraints can be created for floating-point datatypes.

```
SQL> create table floating_point_table1 (
      fltNnull binary_float constraint flt_null not null,
      dblNnull binary_double constraint dbl_null not null,
      fltUnq   binary_float constraint flt_unq unique,
      dblUnq   binary_double constraint dbl_unq unique,
      fltChk   binary_float constraint
                        flt_chk check ( fltChk is not nan ) ,
      dblChk   binary_double constraint
                        dbl_chk check ( dblChk is not infinite) ,
      fltPrm   binary_float constraint flt_prm primary key);

Table created.

SQL> create table floating_point_table2 (
      dblPrm  binary_double constraint
                  dbl_prm primary key,
      fltFrn  binary_float constraint flt_frn
                  references floating_point_table1(fltPrm)
                  on delete cascade);

Table created.
```

CLIENT INTERFACES FOR NATIVE FLOATING-POINT DATATYPES

Support for native floating-point datatypes is implemented in a number of client interfaces, inlcuding SQL , PL/SQL, OCI, OCCI, Pro*C/C++, JDBC, and XML (XML Schema mapping).

SQL NATIVE FLOATING-POINT DATATYPES

The SQL datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` implement native floating-point datatypes in the SQL environment. A number of SQL functions are also provided that operate on these datatypes. `BINARY_FLOAT` and `BINARY_DOUBLE` are supported wherever an expression (`expr`) appears in SQL syntax.

OCI NATIVE FLOATING-POINT DATATYPES SQLT_BFLOAT AND SQLT_BDOUBLE

The Oracle Call Interface (OCI) application programming interface implements the IEEE 754 single precision and double precision native floating-point datatypes with the datatypes `SQLT_BFLOAT` and `SQLT_BDOUBLE` respectively.

Conversions between these types and the SQL types `BINARY_FLOAT` and `BINARY_DOUBLE` are exact on platforms that implement the IEEE 754 standard for the C datatypes `float` and `double`.

NATIVE FLOATING-POINT DATATYPES SUPPORTED IN ORACLE OBJECT TYPES

The SQL datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` are supported as attributes of Oracle `OBJECT` types.

LOB ENHANCEMENTS

TEMPORARY LOB PERFORMANCE IMPROVEMENT

With Oracle Database 10*g* release, the performance and the storage requirement of temp LOBs have been greatly improved by a newly implemented *Reference on Read and Copy on Write* mechanism. The basic idea is that no deep copy of LOB values will occur after assignment. Instead, for each LOB, a reference count is used to keep track of the number of references on a temp LOB. Whenever a new copy is made from the LOB, the count is increased by one; when a copy of the LOB disappears, we decrement the count by one. When the count drops to zero, the temp LOB is deleted. As long as there exist referers, the temp LOB data is kept valid. For read operations the above reference model works fine. However, to ensure value semantics, once a LOB is modified, a new copy of the LOB needs to be made. The new LOB is completely disengaged from the previous copy, and the reference count for the new LOB is reset to one.

Applications using mostly read operations on temporary LOBs will no longer incur the performance and storage space cost of deep copies during variable assignments and passing function call parameters.

UNLIMITED-SIZE LOB SUPPORT

In the past, the maximum size of the LOB datatype (BLOB, CLOB and NCLOB) is 4294967295 (4 gigabytes -1, or 2^32-1), the size limit of a 4-byte unsigned integer (ub4). The new max size of LOBs will be the size limit of (4 gigabytes - 1) * (db_block_size). So the actual LOB max size will depend on the block size of the database. Given the currently allowed db block size range of 2k to 32k, the size limit ranges from 8 to 128 terabytes.

Unlimited-size LOBs, that is, LOBs 4 gigabytes or larger in size, are supported in the following programmatic environments:

- PL/SQL using the DBMS_LOB Package
- Java using JDBC (Java Database Connectivity)
- C using OCI (Oracle Call Interface)

MAXIMUM STORAGE LIMIT FOR UNLIMITED-SIZE LOBS

In supported environments, you can create and manipulate LOBs that are up to the maximum storage size limit for your database configuration. The maximum allowable storage limit for your configuration depends on the database block size setting, the value of the DB_BLOCK_SIZE initialization parameter, and is calculated as (4 gigabytes - 1) times the value of the DB_BLOCK_SIZE parameter. With the current allowable range for the database block size from 2k to 32k, the storage limit ranges from 8 terabytes to 128 terabytes.

This storage limit applies to all LOB types in environments that support unlimited-size LOBs; however, note that CLOB and NCLOB types are sized in characters while the BLOB type is sized in bytes.

USING UNLIMITED-SIZE LOBS WITH PL/SQL, JDBC, AND OCI

Unlimited-size LOBs are now supported with all APIs in the DBMS_LOB PL/SQL package. The DBMS_LOB.GET_STORAGE_LIMIT function returns the storage limit for your database configuration. Oracle JDBC classesalso support unlimited-size LOBs with all LOB APIs in JDBC.

The Oracle Call Interface API provides a set of functions specifically for operations on unlimited-size LOBs. For example, user-defined read and write callback functions for inserting or retrieving data provide an alternative to the polling methods for streaming LOB. These functions are implemented by you and registered with OCI through the OCILobRead(), OCILobWriteAppend(), and OCILobWrite()calls. These callback functions are called by OCI whenever required. For LOBs of size greater than 4 GB, OCILobRead2(), OCIWriteAppend2(), and OCILobWrite2() have callback functions which have the parameter lengths defined as oraub8.

CONVERSION BETWEEN CLOB AND NCLOB

Converting data between Unicode and the database national language character set is becoming more frequent. Explicit conversion between `CLOB` and `NCLOB` is already available in SQL and in PL/SQL via the `TO_CLOB` and `TO_NCLOB` functions. Oracle Database 10*g* introduces implicit conversion for SQL IN and OUT bind variables in

queries and DML operations, as well as for PL/SQL function and procedure parameter passing, and PL/SQL variable assignment. For example, conversion is completely transparent in the following scenarios:

Example 1

```
CREATE TABLE my_table (nclob_col NCLOB);

DECLARE
  clob_var CLOB;
  nclob_var NCLOB;
BEGIN
  clob_var := 'clob data'; -- initialize the CLOB value;
  -- Bind a CLOB into an NCLOB column
  INSERT INTO my_table VALUES (clob_var);
  SELECT nclob_col
    INTO clob_var
  FROM my_table; -- Define an NCLOB column as a CLOB var
END;
/
```

Example 2

```
CREATE FUNCTION TRY_ME (a IN CLOB)
RETURN NCLOB is
BEGIN
  RETURN a;
END;
/

DECLARE
  clob_var CLOB;
  nclob_var NCLOB;
BEGIN
  nclob_var:= 'nclob data';
  /* Pass an NCLOB into a function which takes a CLOB
    Return an NCLOB variable to a CLOB variable. */
  clob_var:=TRY_ME(nclob_var);
end;
/
```

Loading BFILES into CLOB/NCLOB With Character Set Conversion

Loading external BFILE data into a database LOB can be tricky. Data can become unreadable because of character set or character width restrictions between the source and the destination. The new LOADFROMFILE2() procedure offers improved functionality for loading character data into a CLOB or NCLOB. The procedure is available in the DBMS_LOB package as well as in OCI.

The LOADFROMFILE2() procedure allows you to specify the character set of the BFILE through the csid (character set id) parameter. In the case of CLOB or NCLOB, the new procedure converts the data from the external BFILE character set you specify, to the database character set used for the CLOB or the database national language character set for NCLOB. If the source (BFILE) and destination (internal LOB) are the same fixed width character set, no conversion is performed. If the destination character set is varying in width, data will be converted to UCS2 (2 byte Unicode) format because varying width data is stored in UCS2 format in the database. If you do not specify the character set, the CLOB uses the current database csid and NCLOB uses the database national language csid by default

Source BFILE CharacterSet1	Destination LOB CharacterSet2	Character Set Conversion from -> to
Fixed width	Fixed width	cs1->cs2 if (cs1!=cs2)
Varying width	Fixed width	cs1->cs2
Fixed width	Varying width	cs1->UCS2
Varying width	Varying width	cs1->UCS2

TRANSPORTABLE TABLESPACE SUPPORT FOR LOBS / VARYING WIDTH LOB STORAGE

Cross platform transportable tablespace feature introduced in Oracle Database 10*g* allows a tablespace to be moved across independent databases on different platforms.

COLLECTION ENHANCEMENTS

SPECIFYING A TABLESPACE WHEN STORING A NESTED TABLE

A nested table can be stored in a different tablespace than its parent table. In the following SQL statement, the nested table is stored in the `users` tablespace:

```
CREATE TABLE people_tab (
    people_column people_typ )
    NESTED TABLE people_column STORE AS people_column_nt (TABLESPACE users);
```

If the `TABLESPACE` clause is not specified, then the storage table of the nested table is created in the tablespace where the parent table is created. For multilevel nested tables, Oracle creates the child table in the same tablespace as its immediate preceding parent table.

The user can issue `ALTER TABLE MOVE` statement to move a table to a different tablespace. If the user issues `ALTER TABLE MOVE` statement on a table with nested table columns, it only moves parent table, no action is taken on the nested table's storage tables. If the user wants to move a nested table s storage table to a different tablespace, issue `ALTER TABLE MOVE` on the storage table. For example:

```
ALTER TABLE people_tab MOVE TABLESPACE users;
ALTER TABLE people_column_nt MOVE TABLESPACE example;
```

Now the `people_tab table` is in `users` tablespace and nested table is stored in the `example` tablespace.

ANSI SQL STANDARD MULTISET OPERATIONS FOR NESTED TABLES

New in Oracle Database 10*g*, a number of Multiset operators are now supported for the Nested Table collection type. Real world applications use collection types to model containment relationships. Comparison and Set operators for collection types provide powerful tools for these applications. Oracle supports two collection datatypes, VARRAYs and Nested Tables.

A nested table is an unordered set of data elements, all of the same datatype. No maximum is specified in the definition of the table and the order of the elements is not preserved.

Elements of a nested table are actually stored in a separate storage table that contains a column that identifies the parent table row or object to which each element belongs. A nested table has a single column, and the type of that column is a built-in type or an object type. If the column in a nested table is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

COMPARISONS OF NESTED TABLES

EQUAL AND NOT EQUAL COMPARISONS

The equal (=) and not equal (<>) conditions determine whether the input nested tables are identical or not, returning the result as a boolean value.

Two nested tables are equal if they have the same named type, have the same cardinality, and their elements are equal.

Elements are equal depending on whether they are equal by the elements own equality definitions, except for object types which require a map method.

For example:

```
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER );
/

CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
END;
/

CREATE TYPE people_typ AS TABLE OF person_typ;
/
CREATE TABLE students (
   graduation DATE,
   math_majors people_typ,
   chem_majors people_typ,
   physics_majors people_typ)
  NESTED TABLE math_majors STORE AS math_majors_nt
  NESTED TABLE chem_majors STORE AS chem_majors_nt
  NESTED TABLE physics_majors STORE AS physics_majors_nt;

INSERT INTO students (graduation) VALUES ('01-JUN-03');
UPDATE students
  SET math_majors =
        people_typ (person_typ(12, 'Bob Jones', '111-555-1212'),
                    person_typ(31, 'Sarah Chen', '111-555-2212'),
                    person_typ(45, 'Chris Woods', '111-555-1213')),
      chem_majors =
        people_typ (person_typ(51, 'Joe Lane', '111-555-1312'),
                    person_typ(31, 'Sarah Chen', '111-555-2212'),
                    person_typ(52, 'Kim Patel', '111-555-1232')),
   physics_majors =
        people_typ (person_typ(12, 'Bob Jones', '111-555-1212'),
                    person_typ(45, 'Chris Woods', '111-555-1213'))
WHERE graduation = '01-JUN-03';


SELECT p.name
  FROM students, TABLE(physics_majors) p
WHERE math_majors = physics_majors;

no rows selected
```

In this example, the nested tables contain `person_typ` objects which have an associated map method.

IN COMPARISONS

The `IN` condition checks whether a nested table is in a list of nested tables, returning the result as a boolean value. NULL is returned if the nested table is a null nested table.

For example:

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IN (math_majors, chem_majors);

no rows selected
```

SUBSET OF MULTISET COMPARISON

The SUBMULTISET [OF] condition checks whether a nested table is a subset of a another nested table, returning the result as a boolean value. The OF keyword is optional and does not change the functionality of SUBMULTISET.

This operator is implemented only for nested tables because this is a multiset function only.

For example:

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors SUBMULTISET OF math_majors;

      IDNO NAME
---------- ------------------------------
        12 Bob Jones
        45 Chris Woods
```

MEMBER OF A NESTED TABLE COMPARISON

The MEMBER [OF] or NOT MEMBER [OF] condition tests whether an element is a member of a nested table, returning the result as a boolean value. The OF keyword is optional and has no effect on the output.

For example:

```
SELECT graduation
  FROM students
WHERE person_typ(12, 'Bob Jones', '1-800-555-1212') MEMBER OF math_majors;

GRADUATION
----------
01-JUN-03
```

where person_typ (12, 'Bob Jones', '1-800-555-1212') is an element of the same type as the elements of the nested table math_majors.

EMPTY COMPARISON

The IS [NOT] EMPTY condition checks whether a given nested table is empty or not empty, regardless of whether any of the elements are NULL. If a NULL is given for the nested table, the result is NULL. The result is returned as a boolean value.

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS NOT EMPTY;

      IDNO NAME
---------- ------------------------------
        12 Bob Jones
        45 Chris Woods
```

SET COMPARISON

The IS [NOT] A SET condition checks whether a given nested table is composed of unique elements, returning a boolean value.

For example:

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS A SET;

      IDNO NAME
```

```
---------- ------------------------------
        12 Bob Jones
        45 Chris Woods
```

MULTISETS OPERATIONS

CARDINALITY

The `CARDINALITY` function returns the number of elements in a varray or nested table. The return type is `NUMBER`. If the varray or nested table is a null collection, `NULL` is returned.

For example:

```
SELECT CARDINALITY(math_majors)
  FROM students;

CARDINALITY(MATH_MAJORS)
------------------------
                       3
```

COLLECT

The `COLLECT` function is an aggregate function which would create a multiset from a set of elements. The function would take a column of the element type as input and create a multiset from rows selected. To get the results of this function you must use it within a `CAST` function to specify the output type of `COLLECT`.

MULTISET EXCEPT

The `MULTISET EXCEPT` operator inputs two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table. The input nested tables and the output nested table are all type name equivalent.

The `ALL` or `DISTINCT` options can be used with the operator. The default is `ALL`.

With the `ALL` option, for `ntab1 MULTISET EXCEPT ALL ntab2`, all elements in `ntab1` other than those in `ntab2` would be part of the result. If a particular element occurs *m* times in `ntab1` and *n* times in `ntab2`, the result will have (*m* - *n*) occurrences of the element if *m* is greater than *n* otherwise `0` occurrences of the element.

With the `DISTINCT` option, any element that is present in `ntab1` which is also present in `ntab2` would be eliminated, irrespective of the number of occurrences.

For example:

```
SELECT math_majors MULTISET EXCEPT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETEXCEPTPHYSICS_MAJORS(IDNO, NAME, PHONE)
--------------------------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '111-555-2212'))
```

MULTISET INTERSECTION

The `MULTISET INTERSECT` operator returns a nested table whose values are common in the two input nested tables. The input nested tables and the output nested table are all type name equivalent.

There are two options associated with the operator: `ALL` or `DISTINCT`. The default is `ALL`. With the `ALL` option, if a particular value occurs *m* times in `ntab1` and *n* times in `ntab2`, the result would contain the element `MIN`(*m*, *n*) times. With the `DISTINCT` option the duplicates from the result would be eliminated, including duplicates of `NULL` values if they exist.

For example:

```
SELECT math_majors MULTISET INTERSECT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';
```

```
MATH_MAJORSMULTISETINTERSECTPHYSICS_MAJORS(IDNO, NAME, PHONE)
-------------------------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))
```

MULTISET UNION

The `MULTISET UNION` operator returns a nested table whose values are those of the two input nested tables. The input nested tables and the output nested table are all type name equivalent.

There are two options associated with the operator: `ALL` or `DISTINCT`. The default is `ALL`. With the `ALL` option, all elements that are in `ntab1` and `ntab2` would be part of the result, including all copies of `NULL`s. If a particular element occurs *m* times in `ntab1` and *n* times in `ntab2`, the result would contain the element (*m* + *n*) times. With the `DISTINCT` option the duplicates from the result are eliminated, including duplicates of `NULL` values if they exist.

For example:

```
SELECT math_majors MULTISET UNION DISTINCT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETUNIONDISTINCTPHYSICS_MAJORS(IDNO, NAME, PHONE)
-------------------------------------------------------------------------------

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '111-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))

SELECT math_majors MULTISET UNION ALL physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETUNIONALLPHYSICS_MAJORS(IDNO, NAME, PHONE)
-------------------------------------------------------------------------------

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '111-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'),
           PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))
```

POWERMULTISET

The `POWERMULTISET` function generates all non-empty submultisets from a given multiset. The input to the `POWERMULTISET` function could be any expression which evaluates to a multiset. The limit on the cardinality of the multiset argument is 32.

For example:

```
SELECT * FROM TABLE(POWERMULTISET( people_typ (
           person_typ(12, 'Bob Jones', '1-800-555-1212'),
           person_typ(31, 'Sarah Chen', '1-800-555-2212'),
           person_typ(45, 'Chris Woods', '1-800-555-1213'))));

COLUMN_VALUE(IDNO, NAME, PHONE)
-------------------------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'))
PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'))
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'))
PEOPLE_TYP(PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
```

```
PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))

7 rows selected.
```

POWERMULTISET_BY_CARDINALITY

The `POWERMULTISET_BY_CARDINALITY` function returns all non-empty submultisets of a nested table of the specified cardinality. The output would be rows of nested tables.

`POWERMULTISET_BY_CARDINALITY(x, l)` is equivalent to `TABLE(POWERMULTISET(x)) p` where `CARDINALITY(value(p)) = l`, where *x* is a multiset and l is the specified cardinality.

The first input parameter to the `POWERMULTISET_BY_CARDINALITY` could be any expression which evaluates to a nested table. The length parameter should be a positive integer, otherwise an error will be returned. The limit on the cardinality of the nested table argument is 32.

For example:

```
SELECT * FROM TABLE(POWERMULTISET_BY_CARDINALITY( people_typ (
     person_typ(12, 'Bob Jones', '1-800-555-1212'),
     person_typ(31, 'Sarah Chen', '1-800-555-2212'),
     person_typ(45, 'Chris Woods', '1-800-555-1213')),2));

COLUMN_VALUE(IDNO, NAME, PHONE)
--------------------------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'), PERSON_TYP(31, 'Sarah
Chen', '1-800-555-2212'))

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'), PERSON_TYP(45, 'Chris
Woods', '1-800-555-1213'))

PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'), PERSON_TYP(45, 'Chris
 Woods', '1-800-555-1213'))
```

SET

The `SET` function converts a nested table into a set by eliminating duplicates, and returns a nested table whose elements are `DISTINCT` from one another. The nested table returned is of the same named type as the input nested table.

For example:

```
SELECT SET(physics_majors)
  FROM students
WHERE graduation = '01-JUN-03';

SET(PHYSICS_MAJORS)(IDNO, NAME, PHONE)
--------------------------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))
```

CONCLUSION

Oracle's SQL continues to evolve to help usher in the dawning of Grid computing. Fundamental to the successful evolution of Oracle SQL engine, Oracle Database 10*g* introduces native regular expressions, native floating point datatypes, LOB performance improvements, and collection type enhancements. As a result, we have further enriched the integration of SQL with critical data processing capabilities, Java, and XML in the Oracle Database 10*g* for Grid data provisioning, integration, and processing.

String searching, manipulation, validation, and formatting are at the heart of all applications that deal with text data;

regular expressions are considered the most sophisticated means of performing such operations. The introduction of native regular expression support to SQL and PL/SQL in the Oracle Database revolutionizes the ability to search for and manipulate text within the database by providing expressive power in queries, data definitions and string manipulations.

E-Business, Business Intelligence, Life Sciences, and numerous other enterprise applications require highly intensive data manipulation and floating-point computation of large data sets with structured and unstructured data stored in databases. Further consolidation of data processing capabilities (e.g, floating point datatypes, LOB processing) inside the database reduces network traffic congestion, promotes data sharing, and produces highly scalable Grid computing infrastructure.

Oracle's language binding APIs in Java and XML provide direct interfaces to database server. These comprehensive APIs support the most recent standards to provide wide-ranging Oracle database services through Java, XML, C/C++, and other programming languages.

In addition, Oracle Database 10*g* introduces enormous improvements in manageability, Grid computing infrastructure, Database Web Services, OLAP, Data Mining, and many other areas to meet the needs of global enterprises. Oracle's database technology provides the most comprehensive solution for the development, deployment, and management of enterprise applications.

Oracle has been the leader of industrial-strength SQL technology since its birth. Oracle will continue to meet the needs of our partners and customers with the best SQL technology. In short, new SQL capabilities in Oracle Database 10*g* provide the most comprehensive functionality for developing versatile, scalable, concurrent, and high performance database applications running in a Grid environment.