DEPT. OF CSE, KUET.

OVerview of Oracle Data Types

CSE 3110

DR. K.M.Azharul Hasan

Oracle Data Types

This chapter discusses the Oracle built-in datatypes, their properties, and how they map to non-Oracle datatypes. This chapter includes the following topics:

- Introduction to Oracle Datatypes
- Character Datatypes
- Numeric Datatypes
- DATE Datatype
- LOB Datatypes
- RAW and LONG RAW Datatypes
- ROWID and UROWID Datatypes
- ANSI, DB2, and SQL/DS Datatypes
- XML Datatypes
- URI Datatypes
- Object Datatypes and Object Views
- Data Conversion

Introduction to Oracle Datatypes

Each column value and constant in a SQL statement has a datatype, which is associated with a specific storage format, constraints, and a valid range of values. When you create a table, you must specify a datatype for each of its columns. Oracle provides the following categories of built-in datatypes:

- Character Datatypes
- Numeric Datatypes
- DATE Datatype
- LOB Datatypes
- RAW and LONG RAW Datatypes
- ROWID and UROWID Datatypes

Note:PL/SQL has additional datatypes for constants and variables, which include `BOOLEAN`, reference types, composite types (collections and records), and user-defined subtypes.

The following sections that describe each of the built-in datatypes in more detail.

Character Datatypes

The character datatypes store character (alphanumeric) data in strings, with byte values corresponding to the character encoding scheme, generally called a character set or code page.

The database's character set is established when you create the database. Examples of character sets are 7-bit ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), Code Page 500, Japan Extended UNIX, and Unicode UTF-8. Oracle supports both single-byte and multibyte encoding schemes.

This section includes the following topics:

- CHAR Datatype
- VARCHAR2 and VARCHAR Datatypes
- Length Semantics for Character Datatypes
- NCHAR and NVARCHAR2 Datatypes
- Use of Unicode Data in Oracle Database
- LOB Character Datatypes
- LONG Datatype

CHAR Datatype

The CHAR datatype stores fixed-length character strings. When you create a table with a CHAR column, you must specify a string length (in bytes or characters) between 1 and 2000 bytes for the CHAR column width. The default is 1 byte. Oracle then guarantees that:

- When you insert or update a row in the table, the value for the CHAR column has the fixed length.
- If you give a shorter value, then the value is blank-padded to the fixed length.
- If a value is too large, Oracle Database returns an error.

Oracle Database compares CHAR values using blank-padded comparison semantics.

VARCHAR2 and VARCHAR Datatypes

The VARCHAR2 datatype stores variable-length character strings. When you create a table with a VARCHAR2 column, you specify a maximum string length (in bytes or characters) between 1 and 4000 bytes for the VARCHAR2 column. For each row, Oracle Database stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle Database returns an error. Using VARCHAR2 and VARCHAR saves on space used by the table.

For example, assume you declare a column VARCHAR2 with a maximum size of 50 characters. In a single-byte character set, if only 10 characters are given for the VARCHAR2 column value in a particular row, the column in the row's row piece stores only the 10 characters (10 bytes), not 50.

Oracle Database compares VARCHAR2 values using nonpadded comparison semantics.

VARCHAR Datatype

The VARCHAR datatype is synonymous with the VARCHAR2 datatype. To avoid possible changes in behavior, always use the VARCHAR2 datatype to store variable-length character strings.

NCHAR and NVARCHAR2 Datatypes

NCHAR and NVARCHAR2 are Unicode datatypes that store Unicode character data. The character set of NCHAR and NVARCHAR2 datatypes can only be either AL16UTF16 or UTF8 and is specified at database creation time as the national character set. AL16UTF16 and UTF8 are both Unicode encoding.

- The NCHAR datatype stores fixed-length character strings that correspond to the national character set.
- The NVARCHAR2 datatype stores variable length character strings.

When you create a table with an NCHAR or NVARCHAR2 column, the maximum size specified is always in character length semantics. Character length semantics is the default and only length semantics for NCHAR or NVARCHAR2.

For example, if national character set is UTF8, then the following statement defines the maximum byte length of 90 bytes:

```
CREATE TABLE tab1 (col1 NCHAR(30));
```

This statement creates a column with maximum character length of 30. The maximum byte length is the multiple of the maximum character length and the maximum number of bytes in each character.

This section includes the following topics:

- NCHAR
- NVARCHAR2

NCHAR

The maximum length of an NCHAR column is 2000 bytes. It can hold up to 2000 characters. The actual data is subject to the maximum byte limit of 2000. The two size constraints must be satisfied simultaneously at run time.

NVARCHAR2

The maximum length of an NVARCHAR2 column is 4000 bytes. It can hold up to 4000 characters. The actual data is subject to the maximum byte limit of 4000. The two size constraints must be satisfied simultaneously at run time.

Use of Unicode Data in Oracle Database

Unicode is an effort to have a unified encoding of every character in every language known to man. It also provides a way to represent privately-defined characters. A database column that stores Unicode can store text written in any language.

Oracle Database users deploying globalized applications have a strong need to store Unicode data in Oracle Databases. They need a datatype which is guaranteed to be Unicode regardless of the database character set.

Oracle Database supports a reliable Unicode datatype through NCHAR, NVARCHAR2, and NCLOB. These datatypes are guaranteed to be Unicode encoding and always use character length semantics. The character sets used by NCHAR/NVARCHAR2 can be either UTF8 or AL16UTF16, depending on the setting of the national character set when the database is created. These datatypes allow character data in Unicode to be stored in a database that may or may not use Unicode as database character set.

LOB Character Datatypes

The LOB datatypes for character data are CLOB and NCLOB. They can store up to 8 terabytes of character data (CLOB) or national character set data (NCLOB).

LONG Datatype

Note: Do not create tables with LONG columns. Use LOB columns (CLOB, NCLOB) instead. LONG columns are supported only for backward compatibility.

Oracle also recommends that you convert existing LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. Further, LOB functionality is enhanced in every release, whereas LONG functionality has been static for several releases.

Columns defined as LONG can store variable-length character data containing up to 2 gigabytes of information. LONG data is text data that is to be appropriately converted when moving among different systems.

LONG datatype columns are used in the data dictionary to store the text of view definitions. You can use LONG columns in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements.

Numeric Datatypes

The numeric datatypes store positive and negative fixed and floating-point numbers, zero, infinity, and values that are the undefined result of an operation (that is, is "not a number" or NAN).

NUMBER Datatype

The NUMBER datatype stores fixed and floating-point numbers. Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems operating Oracle Database, up to 38 digits of precision.

The following numbers can be stored in a NUMBER column:

- Positive numbers in the range 1×10^{-130} to $9.99...9 \times 10^{125}$ with up to 38 significant digits
- Negative numbers from -1×10^{-130} to $9.99...99 \times 10^{125}$ with up to 38 significant digits
- Zero
- Positive and negative infinity (generated only by importing from an Oracle Database, Version 5)

For numeric columns, you can specify the column as:

```
column_name NUMBER
```

Optionally, you can also specify a precision (total number of digits) and scale (number of digits to the right of the decimal point):

```
column_name NUMBER (precision, scale)
```

If a precision is not specified, the column stores values as given. If no scale is specified, the scale is zero.

Oracle guarantees portability of numbers with a precision equal to or less than 38 digits. You can specify a scale and no precision:

```
column_name NUMBER (*, scale)
```

In this case, the precision is 38, and the specified scale is maintained.

When you specify numeric fields, it is a good idea to specify the precision and scale. This provides extra integrity checking on input.

Table 26-1 shows examples of how data would be stored using different scale factors.

If you specify a negative scale, Oracle Database rounds the actual data to the specified number of places to the left of the decimal point. For example, specifying (7,-2) means Oracle Database rounds to the nearest hundredths, as shown in Table 26-1.

For input and output of numbers, the standard Oracle Database default decimal character is a period, as in the number 1234.56. The decimal is the character that separates the integer and decimal parts of a number. You can change the default decimal character with the initialization parameter NLS_NUMERIC_CHARACTERS. You can also change it for the duration of a session with the ALTER SESSION statement. To enter numbers that do not use the current default decimal character, use the TO_NUMBER function.

Table 26-1 How Scale Factors Affect Numeric Data Storage

Input Data	Specified As	Stored As
7,456,123.89	NUMBER	7456123.89
7,456,123.89	NUMBER(*,1)	7456123.9
7,456,123.89	NUMBER(9)	7456124
7,456,123.89	NUMBER(9,2)	7456123.89
7,456,123.89	NUMBER(9,1)	7456123.9
7,456,123.89	NUMBER(6)	(not accepted, exceeds precision)
7,456,123.89	NUMBER(7,-2)	7456100

Internal Numeric Format

Oracle Database stores numeric data in variable-length format. Each value is stored in scientific notation, with 1 byte used to store the exponent and up to 20 bytes to store the mantissa. The resulting value is limited to 38 digits of precision. Oracle Database does not store leading and trailing zeros. For example, the number 412 is stored in a format similar to 4.12×10^2, with 1 byte used to store the exponent(2) and 2 bytes used to store the three significant digits of the mantissa(4,1,2). Negative numbers include the sign in their length.

Taking this into account, the column size in bytes for a particular numeric data value NUMBER(p), where p is the precision of a given value, can be calculated using the following formula:

```
ROUND((length(p)+s)/2))+1
```

where s equals zero if the number is positive, and s equals 1 if the number is negative.

Zero and positive and negative infinity (only generated on import from Oracle Database, Version 5) are stored using unique representations. Zero and negative infinity each require 1 byte; positive infinity requires 2 bytes.

Floating-Point Numbers

Oracle Database provides two numeric datatypes exclusively for floating-point numbers: BINARY_FLOAT and BINARY_DOUBLE. They support all of the basic functionality provided by the NUMBER datatype. However, while NUMBER uses decimal precision, BINARY_FLOAT and BINARY_DOUBLE use binary precision. This enables faster arithmetic calculations and usually reduces storage requirements.

BINARY_FLOAT and BINARY_DOUBLE are approximate numeric datatypes. They store approximate representations of decimal values, rather than exact representations. For example, the value 0.1 cannot be exactly represented by either BINARY_DOUBLE or BINARY_FLOAT. They are frequently used for scientific computations. Their behavior is similar to the datatypes FLOAT and DOUBLE in Java and XMLSchema.

BINARY_FLOAT Datatype

BINARY_FLOAT is a 32-bit, single-precision floating-point number datatype. Each BINARY_FLOAT value requires 5 bytes, including a length byte.

BINARY_DOUBLE Datatype

BINARY_DOUBLE is a 64-bit, double-precision floating-point number datatype. Each BINARY_DOUBLE value requires 9 bytes, including a length byte.

DATE Datatype

The DATE datatype stores point-in-time values (dates and times) in a table. The DATE datatype stores the year (including the century), the month, the day, the hours, the minutes, and the seconds (after midnight).

Oracle Database can store dates in the Julian era, ranging from January 1, 4712 BCE through December 31, 9999 CE (Common Era, or 'AD'). Unless BCE ('BC' in the format mask) is specifically used, CE date entries are the default.

Oracle Database uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

For input and output of dates, the standard Oracle date format is DD-MON-YY, as follows:

```
'13-NOV-92'
```

You can change this default date format for an instance with the parameter NLS_DATE_FORMAT. You can also change it during a user session with the ALTER SESSION statement. To enter dates that are not in standard Oracle date format, use the TO_DATE function with a format mask:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

Oracle Database stores time in 24-hour format—HH:MI:SS. By default, the time in a date field is 00:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as in:

```
INSERT INTO birthdays (bname, bday) VALUES
    ('ANDY',TO_DATE('13-AUG-66 12:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

LOB Datatypes

The LOB datatypes BLOB, CLOB, NCLOB, and BFILE enable you to store and manipulate large blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) in binary or character format. They provide efficient, random, piece-wise access to the data. Oracle recommends that you always use LOB datatypes over LONG datatypes. You can perform parallel queries (but not parallel DML or DDL) on LOB columns.

LOB datatypes differ from LONG and LONG RAW datatypes in several ways. For example:

- A table can contain multiple LOB columns but only one LONG column.

- A table containing one or more LOB columns can be partitioned, but a table containing a LONG column cannot be partitioned.
- The maximum size of a LOB is 128 terabytes depending on database block size, and the maximum size of a LONG is only 2 gigabytes.
- LOBs support random access to data, but LONGs support only sequential access.
- LOB datatypes (except NCLOB) can be attributes of a user-defined object type but LONG datatypes cannot.
- Temporary LOBs that act like local variables can be used to perform transformations on LOB data. Temporary internal LOBs (BLOBs, CLOBs, and NCLOBs) are created in a temporary tablespace and are independent of tables. For LONG datatypes, however, no temporary structures are available.
- Tables with LOB columns can be replicated, but tables with LONG columns cannot.

SQL statements define LOB columns in a table and LOB attributes in a user-defined object type. When defining LOBs in a table, you can explicitly specify the tablespace and storage characteristics for each LOB.

LOB datatypes can be stored inline (within a table), out-of-line (within a tablespace, using a LOB locator), or in an external file (BFILE datatypes). With compatibility set to Oracle9i or higher, you can use LOBs with SQL VARCHAR operators and functions.

BLOB Datatype

The BLOB datatype stores unstructured binary data in the database. BLOBs can store up to 128 terabytes of binary data.

BLOBs participate fully in transactions. Changes made to a BLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, BLOB locators cannot span transactions or sessions.

CLOB and NCLOB Datatypes

The CLOB and NCLOB datatypes store up to 128 terabytes of character data in the database. CLOBs store database character set data, and NCLOBs store Unicode national character set data. Storing varying-width LOB data in a fixed-width Unicode character set internally enables Oracle Database to provide efficient character-based random access on CLOBs and NCLOBs.

CLOBs and NCLOBs participate fully in transactions. Changes made to a CLOB or NCLOB value by the DBMS_LOB package, PL/SQL, or the OCI can be committed or rolled back. However, CLOB and NCLOB locators cannot span transactions or sessions. You cannot create an object type with NCLOB attributes, but you can specify NCLOB parameters in a method for an object type.

BFILE Datatype

The BFILE datatype stores unstructured binary data in operating-system files outside the database. A BFILE column or attribute stores a file locator that points to an external file containing the data. The amount of BFILE data that can be stored is limited by the operating system.

BFILEs are read only; you cannot modify them. They support only random (not sequential) reads, and they do not participate in transactions. The underlying operating system must maintain the file integrity, security, and durability for BFILEs. The database administrator must ensure that the file exists and that Oracle Database processes have operating-system read permissions on the file.

RAW and LONG RAW Datatypes

Note:

The LONG RAW datatype is provided for backward compatibility with existing applications. For new applications, use the BLOB and BFILE datatypes for large amounts of binary data.

Oracle also recommends that you convert existing LONG RAW columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. Further, LOB functionality is enhanced in every release, whereas LONG RAW functionality has been static for several releases.

The RAW and LONG RAW datatypes are used for data that is not to be interpreted (not converted when moving data between different systems) by Oracle Database. These datatypes are intended for binary data or byte strings. For example, LONG RAW can be used to store graphics, sound, documents, or arrays of binary data. The interpretation depends on the use.

RAW is a variable-length datatype like the VARCHAR2 character datatype, except Oracle Net Services (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Oracle Net Services and Import/Export automatically convert CHAR, VARCHAR2, and LONG data between the database character set and the user session character set, if the two character sets are different.

When Oracle Database automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

LONG RAW data cannot be indexed, but RAW data can be indexed.

ROWID and UROWID Datatypes

Oracle Database uses a ROWID datatype to store the address (rowid) of every row in the database.

- Physical rowids store the addresses of rows in ordinary tables (excluding index-organized tables), clustered tables, table partitions and subpartitions, indexes, and index partitions and subpartitions.
- Logical rowids store the addresses of rows in index-organized tables.

A single datatype called the universal rowid, or UROWID, supports both logical and physical rowids, as well as rowids of foreign tables such as non-Oracle tables accessed through a gateway.

A column of the UROWID datatype can store all kinds of rowids. The value of the COMPATIBLE initialization parameter (for file format compatibility) must be set to 8.1 or higher to use UROWID columns.

The ROWID Pseudocolumn

Each table in an Oracle database internally has a pseudocolumn named ROWID. This pseudocolumn is not evident when listing the structure of a table by executing a SELECT * FROM ... statement, or a DESCRIBE ... statement using SQL*Plus, nor does the pseudocolumn take up space in the table. However, each row's address can be retrieved with a SQL query using the reserved word ROWID as a column name, for example:

```
SELECT ROWID, last_name FROM employees;
```

You cannot set the value of the pseudocolumn ROWID in INSERT or UPDATE statements, and you cannot delete a ROWID value. Oracle Database uses the ROWID values in the pseudocolumn ROWID internally for the construction of indexes.

You can reference rowids in the pseudocolumn ROWID like other table columns (used in SELECT lists and WHERE clauses), but rowids are not stored in the database, nor are they database data. However, you can create tables that contain columns having the ROWID datatype, although Oracle does not guarantee that the values of such columns are valid rowids. The user must ensure that the data stored in the ROWID column truly is a valid ROWID.

Physical Rowids

Physical rowids provide the fastest possible access to a row of a given table. They contain the physical address of a row (down to the specific block) and allow you to retrieve the row in a single block access.

Every row in a nonclustered table is assigned a unique rowid that corresponds to the physical address of a row's row piece (or the initial row piece if the row is chained among multiple row pieces). In the case of clustered tables, rows in different tables that are in the same data block can have the same rowid.

After a rowid is assigned to a row piece, the rowid can change in special circumstances. For example, if row movement is enabled, then the rowid can change because of partition key updates, Flashback Table operations, shrink table operations, and so on. If row movement is disabled, then a rowid can change if the row is exported and imported using Oracle Database utilities.

A physical rowid datatype has one of two formats:

- The extended rowid format supports tablespace-relative data block addresses and efficiently identifies rows in partitioned tables and indexes as well as nonpartitioned tables and indexes. Tables and indexes created by an Oracle8i (or higher) server always have extended rowids.
- A restricted rowid format is also available for backward compatibility with applications developed with Oracle Database Version 7 or earlier releases.

Extended Rowids

Extended rowids use a base 64 encoding of the physical address for each row selected. The encoding characters are A–Z, a–z, 0–9, +, and /. For example, the following query:

```
SELECT ROWID, last_name FROM employees WHERE department_id = 20;
```

can return the following row information:

```
ROWID              LAST_NAME
------------------ ----------
AAAAaoAATAAABrXAAA BORTINS
AAAAaoAATAAABrXAAE RUGGLES
AAAAaoAATAAABrXAAG CHEN
AAAAaoAATAAABrXAAN BLUMBERG
```

An extended rowid has a four-piece format, OOOOOOFFFBBBBBBRRR:

- OOOOOO: The data object number that identifies the database segment (AAAAao in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.

- FFF: The tablespace-relative datafile number of the datafile that contains the row (file AAT in the example).
- BBBBBB: The data block that contains the row (block AAABrX in the example). Block numbers are relative to their datafile, not tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- RRR: The row in the block.

You can retrieve the data object number from data dictionary views USER_OBJECTS, DBA_OBJECTS, and ALL_OBJECTS. For example, the following query returns the data object number for the employees table in the SCOTT schema:

```
SELECT DATA_OBJECT_ID FROM DBA_OBJECTS
    WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMPLOYEES';
```

You can also use the DBMS_ROWID package to extract information from an extended rowid or to convert a rowid from extended format to restricted format (or vice versa).

Restricted Rowids

Restricted rowids use a binary representation of the physical address for each row selected. When queried using SQL*Plus, the binary representation is converted to a VARCHAR2/hexadecimal representation. The following query:

```
SELECT ROWID, last_name FROM employees
    WHERE department_id = 30;
```

can return the following row information:

```
ROWID              ENAME
------------------ ----------
00000DD5.0000.0001 KRISHNAN
00000DD5.0001.0001 ARBUCKLE
00000DD5.0002.0001 NGUYEN
```

As shown, a restricted rowid's VARCHAR2/hexadecimal representation is in a three-piece format, block.row.file:

- The data block that contains the row (block DD5 in the example). Block numbers are relative to their datafile, not tablespace. Two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- The row in the block that contains the row (rows 0, 1, 2 in the example). Row numbers of a given block always start with 0.
- The datafile that contains the row (file 1 in the example). The first datafile of every database is always 1, and file numbers are unique within a database.

Examples of Rowid Use

You can use the function SUBSTR to break the data in a rowid into its components. For example, you can use SUBSTR to break an extended rowid into its four components (database object, file, block, and row):

```
SELECT ROWID,
       SUBSTR(ROWID,1,6) "OBJECT",
       SUBSTR(ROWID,7,3) "FIL",
```

```
       SUBSTR(ROWID,10,6) "BLOCK",
       SUBSTR(ROWID,16,3) "ROW"
       FROM products;

ROWID               OBJECT  FIL  BLOCK   ROW
------------------  ------  ---  ------  ----
AAAA8mAALAAAAQkAAA  AAAA8m  AAL  AAAAQk  AAA
AAAA8mAALAAAAQkAAF  AAAA8m  AAL  AAAAQk  AAF
AAAA8mAALAAAAQkAAI  AAAA8m  AAL  AAAAQk  AAI
```

Or you can use `SUBSTR` to break a restricted rowid into its three components (block, row, and file):

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FILE",
       SUBSTR(ROWID,1,8) "BLOCK",
       SUBSTR(ROWID,10,4) "ROW"
       FROM products;

ROWID               FILE  BLOCK     ROW
------------------  ----  --------  ----
00000DD5.0000.0001  0001  00000DD5  0000
00000DD5.0001.0001  0001  00000DD5  0001
00000DD5.0002.0001  0001  00000DD5  0002
```

Rowids can be useful for revealing information about the physical storage of a table's data. For example, if you are interested in the physical location of a table's rows (such as for table striping), the following query of an extended rowid tells how many datafiles contain rows of a given table:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID,7,3))) "FILES" FROM tablename;

  FILES
--------
     2
```

How Rowids Are Used

Oracle Database uses rowids internally for the construction of indexes. Each key in an index is associated with a rowid that points to the associated row's address for fast access. End users and application developers can also use rowids for several important functions:

- Rowids are the fastest means of accessing particular rows.
- Rowids can be used to see how a table is organized.
- Rowids are unique identifiers for rows in a given table.

Before you use rowids in DML statements, they should be verified and guaranteed not to change. The intended rows should be locked so they cannot be deleted. Under some circumstances, requesting data with an invalid rowid could cause a statement to fail.

You can also create tables with columns defined using the `ROWID` datatype. For example, you can define an exception table with a column of datatype `ROWID` to store the rowids of rows in the database that violate integrity constraints. Columns defined using the `ROWID` datatype behave like other table columns: values can be updated, and so on. Each value in a column defined as datatype `ROWID` requires six bytes to store pertinent column data.

Logical Rowids

Rows in index-organized tables do not have permanent physical addresses—they are stored in the index leaves and can move within the block or to a different block as a result of insertions. Therefore their row identifiers cannot be based on physical addresses. Instead, Oracle provides index-organized tables with logical row identifiers, called logical rowids, that are based on the table's primary key. Oracle Database uses these logical rowids for the construction of secondary indexes on index-organized tables.

Each logical rowid used in a secondary index includes a physical guess, which identifies the block location of the row in the index-organized table at the time the guess was made; that is, when the secondary index was created or rebuilt.

Oracle Database can use guesses to probe into the leaf block directly, bypassing the full key search. This ensures that rowid access of nonvolatile index-organized tables gives comparable performance to the physical rowid access of ordinary tables. In a volatile table, however, if the guess becomes stale the probe can fail, in which case a primary key search must be performed.

The values of two logical rowids are considered equal if they have the same primary key values but different guesses.

Comparison of Logical Rowids with Physical Rowids

Logical rowids are similar to the physical rowids in the following ways:

- Logical rowids are accessible through the `ROWID` pseudocolumn.

  You can use the `ROWID` pseudocolumn to select logical rowids from an index-organized table. The `SELECT ROWID` statement returns an opaque structure, which internally consists of the table's primary key and the physical guess (if any) for the row, along with some control information.

  You can access a row using predicates of the form `WHERE ROWID = value`, where `value` is the opaque structure returned by `SELECT ROWID`.

- Access through the logical rowid is the fastest way to get to a specific row, although it can require more than one block access.
- A row's logical rowid does not change as long as the primary key value does not change. This is less stable than the physical rowid, which stays immutable through all updates to the row.
- Logical rowids can be stored in a column of the `UROWID` datatype

One difference between physical and logical rowids is that logical rowids cannot be used to see how a table is organized.

Note:

An opaque type is one whose internal structure is not known to the database. The database provides storage for the type. The type designer can provide access to the contents of the type by implementing functions, typically 3GL routines.

Guesses in Logical Rowids

When a row's physical location changes, the logical rowid remains valid even if it contains a guess, although the guess could become stale and slow down access to the row. Guess information cannot be updated dynamically. For secondary indexes on index-organized tables, however, you can rebuild the index to obtain fresh guesses. Note that

rebuilding a secondary index on an index-organized table involves reading the base table, unlike rebuilding an index on an ordinary table.

Collect index statistics with the `DBMS_STATS` package or `ANALYZE` statement to keep track of the staleness of guesses, so Oracle Database does not use them unnecessarily. This is particularly important for applications that store rowids with guesses persistently in a `UROWID` column, then retrieve the rowids later and use them to fetch rows.

When you collect index statistics with the `DBMS_STATS` package or `ANALYZE` statement, Oracle Database checks whether the existing guesses are still valid and records the percentage of stale/valid guesses in the data dictionary. After you rebuild a secondary index (recomputing the guesses), collect index statistics again.

In general, logical rowids without guesses provide the fastest possible access for a highly volatile table. If a table is static or if the time between getting a rowid and using it is sufficiently short to make row movement unlikely, logical rowids with guesses provide the fastest access.

Rowids in Non-Oracle Databases

Oracle Database applications can be run against non-Oracle database servers using SQL*Connect. The format of rowids varies according to the characteristics of the non-Oracle system. Furthermore, no standard translation to `VARCHAR2`/hexadecimal format is available. Programs can still use the `ROWID` datatype. However, they must use a nonstandard translation to hexadecimal format of length up to 256 bytes.

Rowids of a non-Oracle database can be stored in a column of the `UROWID` datatype.

ANSI, DB2, and SQL/DS Datatypes

SQL statements that create tables and clusters can also use ANSI datatypes and datatypes from IBM's products SQL/DS and DB2. Oracle Database recognizes the ANSI or IBM datatype name that differs from the Oracle datatype name, records it as the name of the datatype of the column, and then stores the column's data in an Oracle datatype based on the conversions.

XML Datatypes

Oracle provides the `XMLType` datatype to handle XML data.

XMLType Datatype

`XMLType` can be used like any other user-defined type. `XMLType` can be used as the datatype of columns in tables and views. Variables of `XMLType` can be used in PL/SQL stored procedures as parameters, return values, and so on. You can also use XMLType in PL/SQL, SQL and Java, and through JDBC and OCI.

A number of useful functions that operate on XML content have been provided. Many of these are provided both as SQL functions and as member functions of `XMLType`. For example, function `extract` extracts a specific node(s) from an `XMLType` instance. You can use `XMLType` in SQL queries in the same way as any other user-defined datatypes in the system.

URI Datatypes

A URI, or uniform resource identifier, is a generalized kind of URL. Like a URL, it can reference any document, and can reference a specific part of a document. It is more general than a URL because it has a powerful mechanism for specifying the relevant part of the document. By using `UriType`, you can do the following:

- Create table columns that point to data inside or outside the database.
- Query the database columns using functions provided by `UriType`.

Object Datatypes and Object Views

Object types and other user-defined datatypes let you define datatypes that model the structure and behavior of the data in their applications. An object view is a virtual object table.

Built-in Datatype Summary

Datatype	Description
VARCHAR2(size [BYTE | CHAR])	Variable-length character string having maximum length size bytes or characters. Maximum size is 4000 bytes or characters, and minimum is 1 byte or 1 character. You must specify size for VARCHAR2. BYTE indicates that the column will have byte length semantics; CHAR indicates that the column will have character semantics.
NVARCHAR2(size)	Variable-length Unicode character string having maximum length size characters. The number of bytes can be up to two times size for AL16UTF16 encoding and three times size for UTF8 encoding. Maximum size is determined by the national character set definition, with an upper limit of 4000 bytes. You must specify size for NVARCHAR2.
NUMBER[(precision [, scale]])	Number having precision p and scale s. The precision p can range from 1 to 38. The scale s can range from -84 to 127.
LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes. Provided for backward compatibility.
DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is fixed at 7 bytes. This datatype contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It does not have fractional seconds or a time zone.
BINARY_FLOAT	32-bit floating point number. This datatype requires 5 bytes, including the length byte.
BINARY_DOUBLE	64-bit floating point number. This datatype requires 9 bytes, including the length byte.
TIMESTAMP [(fractional_seconds)]	Year, month, and day values of date, as well as hour, minute, and second values of time, where fractional_seconds_precision is the number of digits in the fractional part of the SECOND datetime field. Accepted values of fractional_seconds_precision are 0 to 9. The default is 6. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The sizes varies from 7 to 11 bytes, depending on the precision. This datatype contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It contains fractional seconds but does not have a time zone.
TIMESTAMP [(fractional_seconds)] WITH TIME ZONE	All values of TIMESTAMP as well as time zone displacement value, where fractional_seconds_precision is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is fixed at 13 bytes. This datatype contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, and TIMEZONE_MINUTE. It has fractional seconds and an explicit time zone.

Datatype	Description
TIMESTAMP [(fractional_seconds)] WITH LOCAL TIME ZONE	All values of TIMESTAMP WITH TIME ZONE, with the following exceptions: • Data is normalized to the database time zone when it is stored in the database. • When the data is retrieved, users see the data in the session time zone. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The sizes varies from 7 to 11 bytes, depending on the precision.
INTERVAL YEAR [(year_precision)] TO MONTH	Stores a period of time in years and months, where year_precision is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 2. The size is fixed at 5 bytes.
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds)]	Stores a period of time in days, hours, minutes, and seconds, where • day_precision is the maximum number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2. • fractional_seconds_precision is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6. The size is fixed at 11 bytes.
RAW(size)	Raw binary data of length size bytes. Maximum size is 2000 bytes. You must specify size for a RAW value.
LONG RAW	Raw binary data of variable length up to 2 gigabytes.
ROWID	Base 64 string representing the unique address of a row in its table. This datatype is primarily for values returned by the ROWID pseudocolumn.
UROWID [(size)]	Base 64 string representing the logical address of a row of an index-organized table. The optional size is the size of a column of type UROWID. The maximum size and default is 4000 bytes.
CHAR [(size [BYTE | CHAR])]	Fixed-length character data of length size bytes. Maximum size is 2000 bytes or characters. Default and minimum size is 1 byte. BYTE and CHAR have the same semantics as for VARCHAR2.
NCHAR[(size)]	Fixed-length character data of length size characters. The number of bytes can be up to two times size for AL16UTF16 encoding and three times size for UTF8 encoding. Maximum size is determined by the national character set definition, with an upper limit of 2000 bytes. Default and minimum size is 1 character.
CLOB	A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) * (database block size).
NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) * (database block size). Stores national character set data.
BLOB	A binary large object. Maximum size is (4 gigabytes - 1) * (database block size).
BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.