B

Blockchain Tables Reference

You can independently verify the hash value and signature of a row by using its row content.

You can use the data format for the row content and column content to create procedures or functions that verify the hash value and user signature for a row.

- Blockchain Tables Column Content
  The column content of a row consists of the column metadata and the column data.
- Blockchain Tables Row Content
  A predefined format is used to compute the row content of rows in the blockchain table.
- Format of the Signed Digest in Blockchain Tables
  The signed digest consists of metadata and data about the last row in each chain of a blockchain table.

B.1 Blockchain Tables Column Content

The column content of a row consists of the column metadata and the column data.

The **data format** for column content is a platform-neutral sequence of bytes, in a predefined format, that is based on the column metadata and the column data. To understand how the data format for column content is computed, consider a row in the `bctab` table. The table contains two columns `bank` and `amount`. A row in this table contains the values 'Chase' and 1000 respectively.

Data Format for Column Data

Data format is computed for both user-defined and hidden columns. The column data is platform-neutral and can be obtained by using the SQL DUMP function.

The following example, assuming that the database character set is ANSI ASCII, gets the data format for column data as:

```
SELECT REGEXP_REPLACE(REGEXP_SUBSTR(DUMP(bank_name, 16), '[^ ]+',1, 3), ',',
'') "Data Value"
    FROM examples.bank_ledger WHERE bank_name = 'MyBank';

Data Value
--------------------------------------------------------------------------------
--
4368617365
```

Data Format for Column Metadata

The column metadata is a 20-byte structure in the following format:

```
typedef struct col_meta_data
{
     ub2  version;                /* VALUE IS ALWAYS 1/
     ub2  col_position;
```

```
     ub2  col_type;
     ub1  is_col_null;
     ub1  reserved1;               /*VALUE IS ALWAYS 0*/
     ub8  col_len;
     ub4  spare;
} col_meta_data;
```

where:

- `ub1` is an unsigned single byte value
- `ub2` is an unsigned short, 2 bytes value
- `ub4` is an unsigned long, 4 bytes value
- `ub8` is unsigned long long, 8 bytes value

`ub2`, `ub4`, and `ub8` use the little-endian format.

The attributes of the `col_meta_data` structure describe information about the column, such as its position in the table, data type, whether the column value is `NULL` or not, and the length of the column data (in bytes).

In the example, the column metadata for the value 'Chase' in ASCII and using the structure defined above is:

```
01 00
01 00
01 00
00
00
05 00 00 00 00 00 00 00
00 00 00 00
```

The first line, 01 00, is a 2-byte representation of the data format version, which is 1 in this release. The second line, 01 00, is a 2-byte representation of the column position of `BANK_NAME`, which is 1. The third line, 01 00, is a 2-byte representation of the internal code for the data type of the `BANK_NAME` column. The data type is `VARCHAR2`, which has an internal code of 1. The fourth line, 00, has a byte that specifies whether the column value is NULL (01) or not NULL (00). The column value is "Chase" and is hence not NULL. The fifth line, 00, is a reserved byte and must be zero in this release. The sixth line, 05 00 00 00 00 00 00 00, is an 8-byte length. The value "Chase" is 5 bytes in the database character set. The seventh line, 00 00 00 00, is four reserved bytes, each of which must be zero in this release.

The data format for column content for the column value 'Chase' is the concatenation of bytes from its column metadata and its column data. Therefore, the data format for the column content is the following value:

```
01 00
01 00
01 00
00
00
05 00 00 00 00 00 00 00
00 00 00 00
43 68 61 73 65
```

Example B-1 Creating a User-Defined Function to Construct a Column Metadata Value

The procedure `little_endian_ubx` is a useful utility procedure. It converts a number to a `ub2`, `ub4`, or a `ub8` value in little-endian format. The parameter `x` represents the data type of the column. Use the value 2 for `ub2` input, 4 for `ub4` input, and 8 for `ub8` input.

```
CREATE OR REPLACE FUNCTION little_endian_ubx(value IN NUMBER, x IN NUMBER)
RETURN RAW IS
  format VARCHAR2(16);
  string VARCHAR2(16);
  result RAW(8);
BEGIN
  format := RPAD('0', 2*x-1, '0') || 'X';       -- conversion format is all
zeroes and a final X
  string := SUBSTR(TO_CHAR(value, format), 2);  -- use SUBSTR to strip
leading space
  dbms_output.put_line('string is ' || string);
  result := utl_raw.reverse(HEXTORAW(string));
  dbms_output.put_line('result is ' || RAWTOHEX(result));
  RETURN result;
END little_endian_ubx;
```

B.2 Blockchain Tables Row Content

A predefined format is used to compute the row content of rows in the blockchain table.

The **row content** of a row is a contiguous sequence of bytes that is based on the row data and the hash value of the previous row in the chain. The ***data format*** for the row content defines the order and sequence of bytes.

The data format for row content is different when computing the hash value and when computing the row signature.

Data Format for Row Content When Computing Hash Value

The data format for row content of a hash value is computed based on the user columns, some hidden columns, and the hash value of the previous row in the chain.

To understand the data format for row content when computing a hash value, consider a blockchain table `bctab` that contains two columns `bank` and `amount`. The second row in this table contains the values 'Chase' and 1000 respectively. The data format for the row content, when computing the row hash value, is obtained by concatenating the byte values of the following, in the order listed:

- Data format for bank column, containing value 'Chase'
- Data format for amount column, containing the value 1000
- Data format for hidden column `ORABCTAB_INST_ID$`
- Data format for hidden column `ORABCTAB_CHAIN_ID$`
- Data format for hidden column `ORABCTAB_SEQ_NUM$`
- Data format for hidden column `ORABCTAB_CREATION_TIME$`
- Data format for hidden column `ORABCTAB_USER_NUMBER$`
- Data format for hash value of the first row in the table (assuming both rows are in the same chain)

The order of columns is dictated by the `INTERNAL_COLUMN_ID` column in the view `ALL_TAB_COLS`.

Data Format for Row Content When Computing Row Signature

The data format for row content of a row signature is computed based on the hash value of the row.

B.3 Format of the Signed Digest in Blockchain Tables

The signed digest consists of metadata and data about the last row in each chain of a blockchain table.

The data format for the signed digest contains a header and an array of row information.

The structure of the header is as follows:

```
{
    ub1    version;
                /* 1 in 19.x */
    ub1    reserved_1;
    ub1    reserved_2;
    ub1    reserved_3;
    ub4    reserved_4;
    ub8    total_length;
                /* total length of signature content buffer, except version,
reserved_% and total_length fields */
    ub1    pdb_guid[16];
                /* 16 bytes long PDB GUID */
    ub4    owner_schema_objn;
    ub4    blockchain_table_objn;
    ub4    signature_algorithm;
    ub4    number_of_rows;
 }
```

The structure of the row information is as follows:

```
{
    ub4   instance_id ;
    ub4   chain_id
    ub8   sequence_number;
    ub4   user_number;
    ub1   row_creation_time[16];
                /* UTC format that Oracle uses has 13 bytes; padded 3 bytes */
    ub4   crypto_hash_len;
    ub1  *crypto_hash;
                /* padded to 4 byte boundary */
    ub4   user_columns_count;
                /* always 0 in 19.x
                 * padded to 8 byte boundary */
    ub8   user_columns_data_len;
                /* always 0 in 19.x */
}
```

where:

- `ub1` is an unsigned single byte value
- `ub4` is an unsigned long, 4 bytes value
- `ub8` is an unsigned long, 8 bytes value

`ub4` and `ub8` use the little-endian format.

Padding is done by appending binary zeros.