IBM Storage

Working on the chain gang: Using Oracle as off-chain storage

Document version 4.1

Contents

List of figures

1 Introduction

In the blockchain paradigm you can best understand the relationship to off-chain storage by considering the blockchain as the fact table in a star schema with the off-chain storage being the dimension tables. Figure 1 shows a simplified star schema.

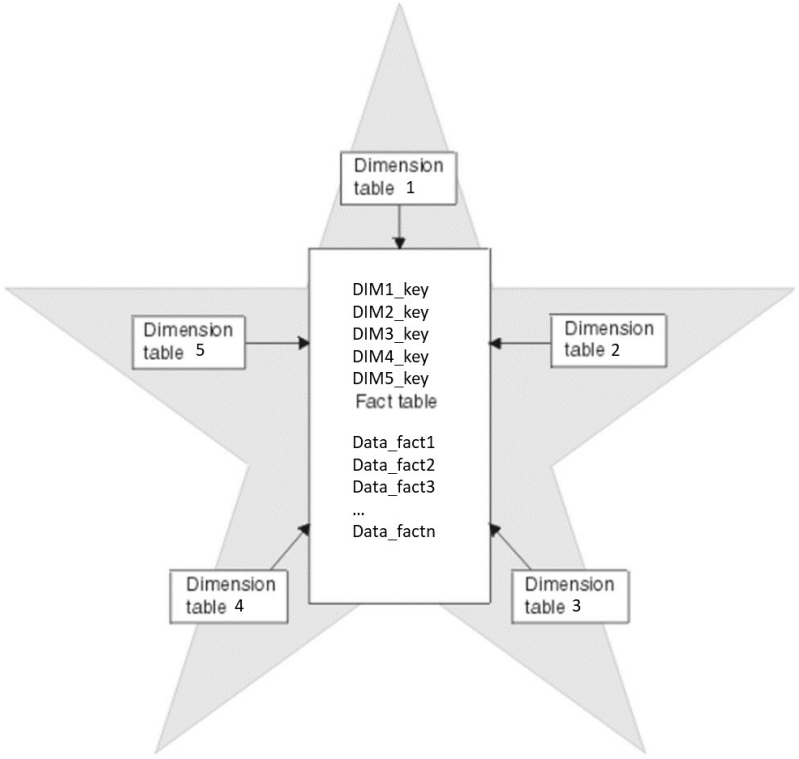


Figure 1: Example star schema

Similar to the fact table, which is a collection of the keys from the dimension tables and the data facts that result from that collection, the blockchain transactions are a collection of hash or signature values along with a few key data items such as amounts, sums, and other aggregations that are pertinent to the combination of hashes contained in the transactions. The transaction data, along with hash pointers to the previous and next blocks in the chain will also include hash pointers to other data items. Each stored transaction in the block is equivalent to a tuple in a star schema fact table. This is shown in Figure 2.

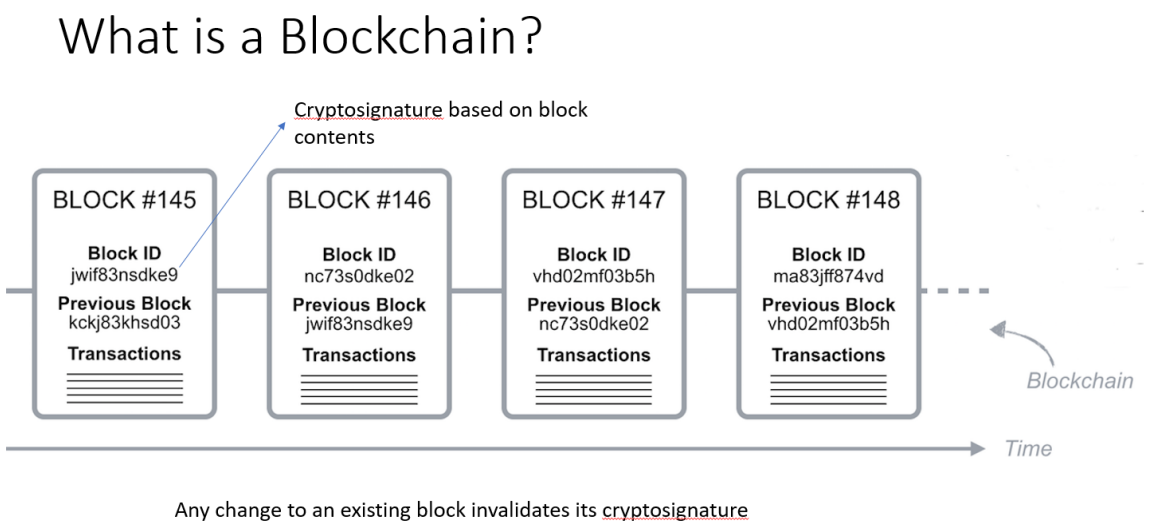


Figure 2: Example blockchain structure

But what happens when the data is large? For example, an image such as an X-ray or CAT scan? What about complex contracts or DOC files? These large data objects should not be stored in the blockchain as shown in Figure 3. How do you ensure that these items are not modified over time or replaced with fraudulent information?

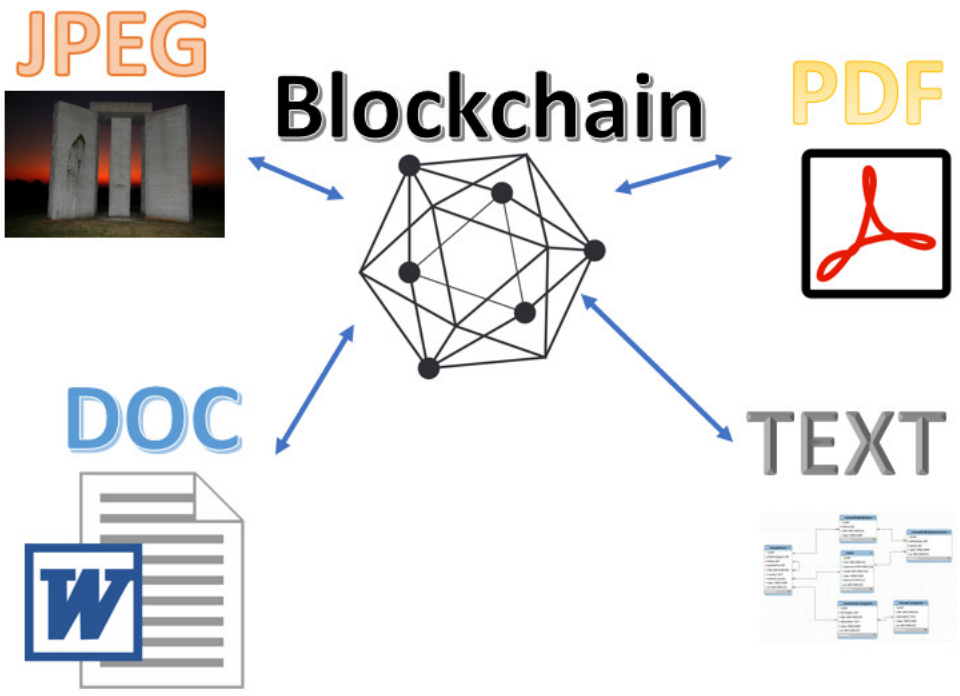


Figure 3: Example off-chain data

2 Identifying records

Each off-chain object pointed at by the blockchain transaction is identified by a hash, usually a DES 256 or MD5 generated identifier. The identifier might also be a MOD type value which means it will be a hash value encrypted by a supplied key. Each object referred to by a transaction must be re-verified against its hash or signature before being used. An example blockchain block containing hash pointers is shown in Figure 4.

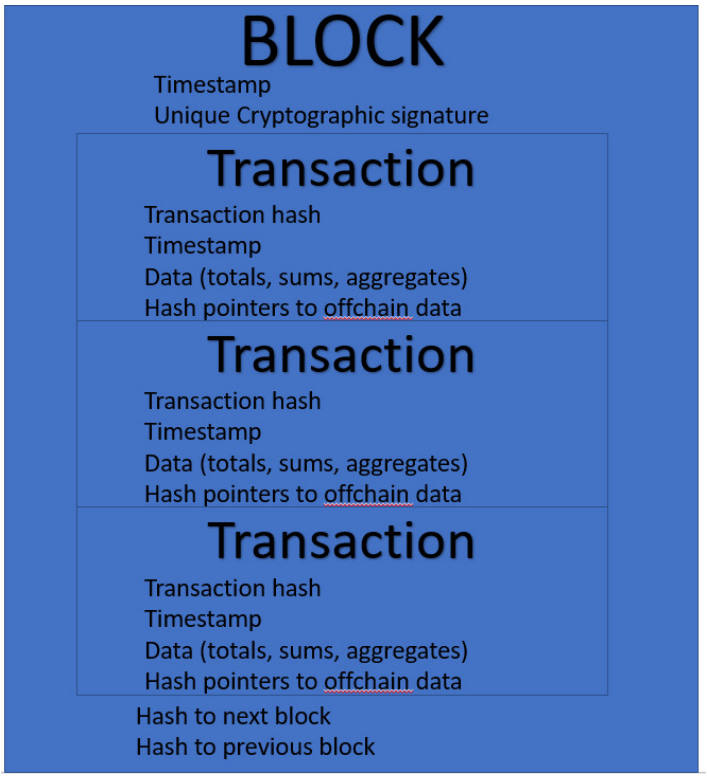


Figure 4: Simplified block structure

Each peer in a blockchain will have a full copy of the blockchain, but, will only be responsible for storing their own off-chain data. This can lead to questions of data security. Each off-chain data object (may be a DOC, BLOB, CLOB, JPEG, or a set of fields in a database record) will have a hash signature stored in the transaction data with the blockchain and that hash will be reverified each time an off-chain data source is referenced.

To validate a single object such as a picture, PDF, or DOC file, it is a matter of calculating the object's hash value and comparing it to the existing hash value. Example 1 shows a simple table for storing these pointers and hash values.

```
CREATE TABLE bfile_table (
  Record_Hash_value RAW(256) NOT NULL,
  exfile_hash RAW(256) NOT NULL,
  Contents Varchar2(2000),
  file_type Varchar2(32),
  F_exfile BFILE);
```

Example 1: Table to hold blockchain reference object pointers

Notice there are two hash values stored in the table in Example 1. One points to the object and the other to the record in the table itself. This way both can be verified from the hash value stored in the transaction block of the blockchain.

3 Slowly changing records and blockchains

Slowly changing records are records whose values may alter over time such as addresses, credit card numbers, and so on. Often it the requirement of the blockchain manager to maintain a running history of these changes over time. In data warehousing, this would be equivalent to a type 2 dimension. It is usually difficult to deal with such slowing changing records because determining the exact value that has changed becomes problematic if there are more than a few fields in the record.

For records with multiple text, date, object, and number fields, the blockchain manager has the choice of a difficult, field-by-field verification of the record or use the field verification method to determine if a field has changed in a record, thus requiring a new record insertion. When a field is altered into a slowly changing record, the old record is marked as invalid by a flag, a date field, or a set of date fields allowing for a bi-temporal record. This requires a new transaction to be inserted into the blockchain, indicating that a source record has been altered and reverified by the certifying agent or agents.

4 Making hash of data

One method for finding out if a record has a changed field is to generate a hash for all the pertinent sections of the record and store this hash with the blockchain record. With retrieval for each record that is to be verified, a hash is calculated for the important fields of the record. This new hash is compared to the stored hash in the blockchain transaction and if the signatures match, the record is used. In case the signature is changed, the new fields are marked as questionable, the old is marked as invalid, and the new is marked as the current record with a new transaction record stored in the block chain showing the change. If the new record matches none of the existing signatures, the record is inserted but no existing records are altered. The records marked as questionable must be reviewed by the certifying authority before they are used. Essentially the program flow would be:

1. Transaction requires off-chain object (either in the DB or in a file system)
2. Hash is generated for the object and hash is stored with transaction
3. Hash is placed in the file manager/database
4. Location pointer and object hash or data hash is stored in database
5. Record is hashed and hash is stored in database and transaction.

The idea is to provide a hash to verify that the object is same, a hash to verify that the location is same, and a hash to verify that the record has not changed. Now, two or more of these could be combined for ease in design.

General Data Protection Regulation (GDPR) specifies that personal information must be erasable. This means that personal information cannot be stored in the blockchain but must be stored in off-chain storage. Example 2 shows a customer table with a hash code inserted. As an alternative, a hash-based, function-based index (example follows) could be used. The hash would also be stored in the blockchain.

```
CREATE TABLE customer (
         custnum    NUMBER(7)     PRIMARY KEY,
         rec_hash   RAW(256)      NOT NULL,
         fname      VARCHAR2(15) NOT NULL,
         lname      VARCHAR2(24) NOT NULL,
         cc_num     NUMBER(16)    ENCRYPT,
         ver_code   NUMBER(4),
         expdate   DATE DEFAULT (sysdate),
         photo      BLOB,
         street_no  NUMBER(6),
         street_nam VARCHAR2(32),
         apt_num    NUMBER(4),
         town       VARCHAR2(32),
         city       VARCHAR2(32),
         zip4       NUMBER(9)
   )
   TABLESPACE customer_tbs
   STORAGE ( INITIAL 50K);
```

Example 2: Table with stored hash

5 Demystifying DBMS_CRYPTO

Calculating a hash can be difficult, unless you know something about what Oracle provides in its various PL/SQL packages. For example, the `DBMS_CRYPTO` contains the `HASH` function. The `HASH` package creates several different hash signatures for RAW, binary large object (BLOB), or character large object (CLOB) data. RAW, CLOB, or BLOB are passed to it either internally or through a procedure call. If the pass-in is from a procedure call from SQL, then the RAW length would be limited to the value of `MAX_STRING_SIZE`, an initialization value. Providing your critical columns are less than `MAX_STRING_SIZE` in concatenated length and you can generate a hash that can be used to compare records. In this scenario, each date or number field should be converted using the appropriate function to a character equivalent.

5.1 An example using DBMS_CRYPTO.HASH

To use the `DBMS_CRYPTO.HASH` function, first make sure your user who owns the schema for the blockchain off-chain schema (for example, user BLCH), has EXECUTE privilege on the DBMS_CRYPTO package:

```
GRANT EXECUTE ON DBMS_CRYPTO TO BLCH;
```

To use a function in `INSERT` statements, it must be deterministic, and, because it generates a hash, you cannot call it with a null value, and therefore, a wrapper is required. This is shown in the following example.

```
CREATE OR REPLACE FUNCTION get_sign (clb clob)
    RETURN RAW
    DETERMINISTIC
    AS
      signature RAW(128);
      clb2 clob;
    BEGIN
      IF clb IS NULL THEN
        clb2:= to_clob('1');
     ELSE
       clb2:=clb;
     END IF;
     Signature:=sys.DBMS_CRYPTO.HASH(clb2,4);
     RETURN signature;
  END;

Note: The 4 in the call to HASH indicates to use a DES256 bit hash
```

Example 3: Creating a deterministic function wrapper around DBMS_CRYPTO.HASH

Note that this is just a bare-bones function, and exception checking should be added.

Notice that you can use a default value of '1' if the input value is null, because the function generates a hash signature and you cannot pass it a null value. So, in the case of a null value, you need to change it to '1'. You can of course use any string value but just be sure to document it!

Note that if your off-chain record contains a pointer to a bfile (externally stored object), you should store a hash for the bfile object that is verified when retrieving that object. Thus, the record containing the pointer is hashed, guaranteeing that no one can repoint the record to a different object and the object can be verified to make sure that no one has replaced it.

After you have a deterministic function you can use it to generate a function-based index, as an INSERT into a table column, or use it in a virtual column. Function-based indexes are especially useful when using existing data stores as a source or target for blockchains. This allows you to regenerate the matching columns without having to update the complete table each time columns are added to the record. Also, there are performance benefits of using a function-based index or an internal column over performing a column-by-column match.
Now using the `DBMS_CRYPTO.HASH` function through the `get_sign` deterministic function is easy. First create a `SELECT` statement that concatenates all the records you want to use to

generate the hash into a single CLOB. Then call the `HASH` function using the CLOB as an input record.

First, you need to create a demo table TEST. In this example, it has been created from the DBA_OBJECTS view using a simple CTAS as shown in the following example.

```
Table created.
SQL> desc test
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------
 OWNER                                              VARCHAR2(128)
 OBJECT_NAME                                        VARCHAR2(128)
 SUBOBJECT_NAME                                     VARCHAR2(128)
 OBJECT_ID                                          NUMBER
 DATA_OBJECT_ID                                     NUMBER
 OBJECT_TYPE                                        VARCHAR2(23)
 CREATED                                            DATE
 LAST_DDL_TIME                                      DATE
 TIMESTAMP                                          VARCHAR2(19)
 STATUS                                             VARCHAR2(7)
 TEMPORARY                                          VARCHAR2(1)
 GENERATED                                          VARCHAR2(1)
 SECONDARY                                          VARCHAR2(1)
 NAMESPACE                                          NUMBER
 EDITION_NAME                                       VARCHAR2(128)
 SHARING                                            VARCHAR2(18)
 EDITIONABLE                                        VARCHAR2(1)
 ORACLE_MAINTAINED                                  VARCHAR2(1)
 APPLICATION                                        VARCHAR2(1)
 DEFAULT_COLLATION                                  VARCHAR2(100)
 DUPLICATED                                         VARCHAR2(1)
 SHARDED                                            VARCHAR2(1)
 CREATED_APPID                                      NUMBER
 CREATED_VSNID                                      NUMBER
 MODIFIED_APPID                                     NUMBER
 MODIFIED_VSNID                                     NUMBER

SQL> SELECT COUNT(*) FROM test;

  COUNT(*)
----------
     72600

SQL> CREATE TABLE test2 AS SELECT * FROM dba_objects;
SQL> SELECT COUNT(*) FROM test2;

  COUNT(*)
----------
     72601
SQL> DROP test;
SQL> RENAME test 2 to test;
```

Example 4: Creating a populated test table

Note: In order to have TEST in the table, you must create TEST, then recreate it as TEST2, then drop TEST and rename TEST2 to TEST if you want to use a SELECT against TEST in the demo.

5.2 Using a function-based index

Performing a select against individual columns each time to validate the record would be performance prohibitive. In Oracle, you can create a function-based index that does the calculation on each insert into the table. This is shown in the following example.

In the test table, use the function to call the `GET_SIGN` function and then use it to create a function-based index as shown in the following example.

```
SQL> CREATE INDEX test_fbi ON
test(get_sign(owner||object_name||subobject_name||to_char(object_id));

Index created.
```

Example 5: Creating a function-based index

Before using the query rewrite, you must analyze the table and index, and set proper initialization parameters (`query_rewrite_enabled = true` and `query_rewrite_integrity = trusted`).

A query that uses this index and an example of performance is shown in Example 6.

```
SQL> SELECT a.object_type FROM test a
  2  WHERE
  3  get_sign('SYSTEM'||'TEST'||''||to_char(73840)) =
get_sign(a.owner||a.object_name||a.subobject_name||to_char(object_id));


Elapsed: 00:00:00.01

OBJECT_TYPE
-----------
TABLE


Execution Plan
----------------------------------------------------------
Plan hash value: 1589953258

-------------------------------------------------------------------------

| Id  | Operation                           | Name   |Rows | Bytes |Cost (%CPU)| Time  |

-------------------------------------------------------------------------

|   0 | SELECT STATEMENT                    |        |    1|    36 |     3(0)| 00:00:01|

|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| TEST   |    1|    36 |     3(0)| 00:00:01|

|*  2 |   INDEX RANGE SCAN                  |TEST_FBI|    1 |      |     1(0)| 00:00:01|

-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 -
access("SYSTEM"."GET_SIGN"("OWNER"||"OBJECT_NAME"||"SUBOBJECT_NAME||TO_CHAR(73840)")="GET
```

```
_SIGN('SYSTEMTEST73840'))


Statistics
----------------------------------------------------------
         44  recursive calls
          5  db block gets
         11  consistent gets
          0  physical reads
       1052  redo size
        545  bytes sent via SQL*Net to client
        608  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
rows processed
```

Example 6: Performance with function-based index

Now, drop the index and rerun the query to check performance without it. This is shown in the following example.

```
SQL> DROP INDEX test_fbi;

Index dropped.

SQL> SELECT a.object_type FROM test a
  2    WHERE
  3    get_sign('SYSTEM'||'TEST'||''||TO_CHAR(73840)) =
get_sign(a.owner||a.object_name||a.subobject_name||to_char(a.object_id));

Elapsed: 00:00:04.67

OBJECT_TYPE
-----------
TABLE


Execution Plan
----------------------------------------------------------
Plan hash value: 1357081020

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows| Bytes |Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      | 726 | 31944 |  404   (5)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| TEST | 726 | 31944 |  404   (5)| 00:00:01 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 -
filter("GET_SIGN"("A"."OWNER"||"A"."OBJECT_NAME"||"A"."SUBOBJECT||TO_CHAR(A>OBJEC
T_ID)_
             NAME")="GET_SIGN"('SYSTEMTEST73840'))
Statistics
----------------------------------------------------------
        101  recursive calls
          5  db block gets
```

```
       1466  consistent gets
          0  physical reads
          0  redo size
        545  bytes sent via SQL*Net to client
        608  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Example 7: Performance without the function-based index

As you can see, the simple SELECT statement using the function-based index performs in too small a time interval for Oracle's timing command to measure. The test was run several times, with identical results. The SELECT statement is not able to use the index, and takes a consistent time just over 4.5 seconds with a clear cache.

In situations where the call is repeated many times (such as for high transaction volume tables), the performance improvement should be dramatic.

The pros to the function-based index (FBI) solution is that it does not require any table changes and it is fast. The cons are that it adds a database object and must use query rewrite.

5.3 Using an added column and an index

In other tests, using an added column SIGNATURE, which is a RAW(128) column type, and an index on the column, you can get identical performance to a function-based index and table combination. The performance with just an added hash column is shown in the following example.

```
SQL> alter table test add signature raw(128);

Table altered.

SQL> update test a set
a.signature=get_sign(a.owner||a.object_name||a.subobject_name||to_char(a.object_i
d));

72609 rows updated.

SQL> commit;

Commit complete.

SQL> set autotrace on
SQL> set timing on
SQL> SELECT a.object_type FROM test a
  2    WHERE
  3    get_sign('SYSTEM'||'TEST'||''||TO_CHAR(73840)) = a.signature;

 OBJECT_TYPE
------------
TABLE

Elapsed: 00:00:02.28

Execution Plan
----------------------------------------------------------
```

```
Plan hash value: 1357081020

------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows | Bytes |Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |  726 | 97284 |  397   (4)| 00:00:01|
|*  1 |  TABLE ACCESS FULL| TEST |  726 | 97284 |  397   (4)| 00:00:01|
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("A"."SIGNATURE"="GET_SIGN"('SYSTEMTEST73840'))


Statistics
----------------------------------------------------------
         22  recursive calls
          0  db block gets
       1698  consistent gets
          0  physical reads
          0  redo size
        545  bytes sent via SQL*Net to client
        608  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Example 8: Performance with an internal hash field

Without an index, the use of a special SIGNATURE column performs worse than the table and function-based index combination. However, when you add an index, the performance is the same (in terms of timing and the statistics) as shown in the following example.

```
SQL> create index test_ui on test(signature);

Index created.

Elapsed: 00:00:00.07
SQL> analyze table test compute statistics;

Table analyzed.

Elapsed: 00:00:02.05
SQL> SELECT a.object_type FROM test a
  2    WHERE
  3    get_sign('SYSTEM'||'TEST'||'') = a.signature;

OBJECT_TYPE
-----------
TABLE

Elapsed: 00:00:00.01

Execution Plan
----------------------------------------------------------
Plan hash value: 1530865433

------------------------------------------------------------------------------
```

```
| Id  | Operation                           | Name  | Rows|Bytes|Cost(%CPU| Time|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |       |    1|   36| 3(0)| 00:00:01|
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED|TEST   |    1|   36| 3(0)| 00:00:01|
|*  2 |   INDEX RANGE SCAN                  |TEST_UI|    1|     | 1(0)| 00:00:01|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("A"."SIGNATURE"="GET_SIGN"('SYSTEMTEST'))


Statistics
----------------------------------------------------------
         22  recursive calls
          0  db block gets
          8  consistent gets
          0  physical reads
          0  redo size
        545  bytes sent via SQL*Net to client
        608  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Example 9: Performance with an internal hash field and index

However, if there is a chance that columns will be added or removed, using a column inside the dimension will require the entire dimension to be updated one row at a time. In smaller records this might be acceptable, but in larger records with thousands or millions of rows it might be better to use the table and function-based index solution.

5.4 Using a virtual column and an index

Another method in Oracle is to use a virtual column to specify the `HASH` value and then index that column. This is demonstrated in the following example.

```
SQL> alter table test add hash RAW(2000) GENERATED ALWAYS AS
(get_sign(owner||object_name||subobject_name||to_char(object_id))) VIRTUAL;
Table altered.

Elapsed: 00:00:00.01

SQL> create index test_ind on test(hash);

Index created.

Elapsed: 00:00:02.99

SQL> ANALYZE TABLE test COMPUTE STATISTICS
SQL>  SELECT a.object_type FROM test a
  2    WHERE
  3*   get_sign('SYSTEM'||'TEST'||''||to_char(73840)) = hash;

OBJECT_TYPE
-----------------------
TABLE
```

```
Elapsed: 00:00:00.01

Execution Plan
----------------------------------------------------------
Plan hash value: 1744442303

-------------------------------------------------------------------------------------------
| Id  | Operation                           | Name    |Rows |Bytes|Cost (%CPU)| Time |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |         |72609|9785K|  1 (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| TEST    |72609|9785K|  1 (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | TEST_IND|    1|     |  1 (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("HASH"="GET_SIGN"('SYSTEMTEST73840'))

Statistics
----------------------------------------------------------
         42  recursive calls
          0  db block gets
         10  consistent gets
          0  physical reads
          0  redo size
        548  bytes sent via SQL*Net to client
        607  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Example 10: Using a virtual column and an index

Performance without the index is slightly worse than a full table scan because of additional sort operations as shown in the following example.

```
SQL> SELECT a.object_type FROM test a
  2    WHERE
  3*   get_sign('SYSTEM'||'TEST'||''||to_char(73850)) = a.hash
SQL> /

OBJECT_TYPE
-----------------------
TABLE

Elapsed: 00:00:04.60

Execution Plan
----------------------------------------------------------
Plan hash value: 1357081020

-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   726 |    97K|   404   (5)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| TEST |   726 |    97K|   404   (5)| 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("A"."HASH"="GET_SIGN"('SYSTEMTEST73850'))
```

```
Statistics
----------------------------------------------------------
         53  recursive calls
          0  db block gets
       1709  consistent gets
          0  physical reads
          0  redo size
        548  bytes sent via SQL*Net to client
        607  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Example 11: Virtual column without index

As can be seen, the performance for a function-based index, an internal normal column and index, and a virtual column with index are identical. However, the virtual column is automatically calculated and updated. So, it may be a more efficient option on par with the function-based index.

6 Summary

This paper described the method to use `DBMS_CRYPTO` to generate a hash signature for data that is then used to verify that data in future. It also discussed how a blockchain and its associated data is similar to a star schema with its fact and dimension tables, and explained how they are both plagued by slowly changing data.

Using hash signatures, verification of BLOB, CLOB and internal table data can be made easier and more secure.

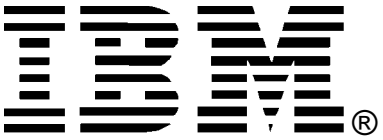
In terms of performance, overall a normal column, populated at row creation, with an index, performed the best due to:

- Fewer recursive calls
- Fewer consistent gets
- No redo generation

In a stable environment with no changes to table structure, a normal column and index might be best. However, it requires many changes to the applications to implement.

The virtual column could provide the benefits of the FBI with slightly better performance but the overall difference between the two would make FBI a better choice because it does not require changes to the existing table.

In an environment where changes to existing tables are forbidden, FBI is a good choice and gives better performance even though it does result in some additional redo generation. For existing applications that requires least programming changes, the FBI would be a good fit.



IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PAPER “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes may be made periodically to the information herein; these changes may be incorporated in subsequent versions of the paper. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this paper at any time without notice.

Any references in this document to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
4205 South Miami Boulevard
Research Triangle Park, NC 27709 U.S.A.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.