# PL/SQL-Based Blockchain Ledger System

**Objective:**

Develop a blockchain ledger system in PL/SQL that ensures **data immutability**, **transaction verification**, and **cryptographic integrity** using **hash chaining** and **digital signatures**.

**Structure**

plsql-blockchain-ledger/

│── scripts/

│   ├── 01\_create\_tables.sql

│   ├── 02\_create\_hash\_function.sql

│   ├── 03\_create\_procedures.sql

│   ├── 04\_create\_triggers.sql

│   ├── 05\_create\_apis.sql

│   ├── 06\_create\_signature\_table.sql

│   ├── 07\_generate\_signature\_function.sql

│   ├── 08\_verify\_signature\_function.sql

│   ├── 09\_create\_blockchain\_status\_function.sql

│── data/

│   ├── genesis\_block.sql

│   ├── sample\_transactions.sql

│── tests/

│   ├── test\_blockchain\_integrity.sql

│── config/

│   ├── db\_connection.sql

│   ├── encryption\_keys.txt

│── deployment/

│   ├── deploy\_blockchain.sql

│   ├── rollback.sql

**Step 1: Create the Blockchain Ledger Table**

Each transaction block contains:

* A **unique transaction ID**.
* **Sender and receiver details**.
* **Amount and timestamp**.
* **Previous block’s hash (hash chaining)**.
* **Current block’s hash**.

01\_create\_tables.sql

CREATE TABLE blockchain\_ledger (

    transaction\_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

    sender VARCHAR2(100) NOT NULL,

    receiver VARCHAR2(100) NOT NULL,

    amount NUMBER NOT NULL,

    transaction\_time TIMESTAMP DEFAULT SYSTIMESTAMP,

    previous\_hash VARCHAR2(64), -- Stores the hash of the previous block

    current\_hash VARCHAR2(64) -- Hash of this transaction

);

**Step 2: Create a Function to Generate SHA-256 Hash**

We use **DBMS\_CRYPTO** to hash transactions for data integrity.

02\_create\_hash\_function.sql

CREATE OR REPLACE FUNCTION generate\_hash(input\_string VARCHAR2) RETURN VARCHAR2 IS

    v\_hash RAW(32);

BEGIN

    v\_hash := DBMS\_CRYPTO.HASH(UTL\_RAW.CAST\_TO\_RAW(input\_string), DBMS\_CRYPTO.HASH\_SH256);

    RETURN RAWTOHEX(v\_hash); -- Convert binary hash to hex format

END generate\_hash;

/

**Explanation:**

* Uses **SHA-256** hashing on input data.
* Converts **binary hash** to **hex format** for readability.
* Used for **hash chaining** between blocks.

**Step 3: Insert a Genesis Block**

The **first block** (Genesis Block) has no previous hash. The process of inserting the **first block (Genesis Block)** into the blockchain ledger. This block is unique because it has no previous hash, signifying the beginning of the blockchain.

genesis\_block.sql

INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

VALUES ('System', 'Genesis', 0, NULL, generate\_hash('Genesis Block'));

COMMIT;

**Insert sample data**sample\_transactions.sql

-- Sample Transactions for Blockchain Ledger

-- Assumes the existence of a generate\_hash function and blockchain\_ledger table

INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

VALUES ('Alice', 'Bob', 50, (SELECT current\_hash FROM blockchain\_ledger WHERE transaction\_id = (SELECT MAX(transaction\_id) FROM blockchain\_ledger)), generate\_hash('Alice|Bob|50'));

INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

VALUES ('Bob', 'Charlie', 30, (SELECT current\_hash FROM blockchain\_ledger WHERE transaction\_id = (SELECT MAX(transaction\_id) FROM blockchain\_ledger)), generate\_hash('Bob|Charlie|30'));

INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

VALUES ('Charlie', 'Dave', 75, (SELECT current\_hash FROM blockchain\_ledger WHERE transaction\_id = (SELECT MAX(transaction\_id) FROM blockchain\_ledger)), generate\_hash('Charlie|Dave|75'));

INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

VALUES ('Dave', 'Eve', 40, (SELECT current\_hash FROM blockchain\_ledger WHERE transaction\_id = (SELECT MAX(transaction\_id) FROM blockchain\_ledger)), generate\_hash('Dave|Eve|40'));

INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

VALUES ('Eve', 'Alice', 20, (SELECT current\_hash FROM blockchain\_ledger WHERE transaction\_id = (SELECT MAX(transaction\_id) FROM blockchain\_ledger)), generate\_hash('Eve|Alice|20'));

COMMIT;

**Explanation of Script:**

* Each transaction references the current\_hash of the **previous block** to maintain the blockchain's cryptographic chain.
* The generate\_hash function is used to create a new hash for each transaction based on the sender, receiver, and amount.
* The SELECT MAX(transaction\_id) ensures that each transaction links to the last inserted block.

**Step 4: Create a Procedure to Add Transactions**

Each new transaction:

* Retrieves the **latest block’s hash**.
* Generates a **new hash** using transaction data.
* Inserts the **new block** into the ledger.

03\_create\_procedures.sql

CREATE OR REPLACE PROCEDURE add\_transaction(

    p\_sender IN VARCHAR2,

    p\_receiver IN VARCHAR2,

    p\_amount IN NUMBER

) AS

    v\_previous\_hash VARCHAR2(64);

    v\_transaction\_string VARCHAR2(4000);

    v\_new\_hash VARCHAR2(64);

BEGIN

    -- Get the last block’s hash

    SELECT current\_hash INTO v\_previous\_hash

    FROM blockchain\_ledger

    ORDER BY transaction\_id DESC FETCH FIRST 1 ROW ONLY;

    -- Create a hash input string

    v\_transaction\_string := p\_sender || p\_receiver || p\_amount || SYSTIMESTAMP || v\_previous\_hash;

    -- Generate a new hash

    v\_new\_hash := generate\_hash(v\_transaction\_string);

    -- Insert the transaction as a new block

    INSERT INTO blockchain\_ledger (sender, receiver, amount, previous\_hash, current\_hash)

    VALUES (p\_sender, p\_receiver, p\_amount, v\_previous\_hash, v\_new\_hash);

    COMMIT;

END add\_transaction;

/

**How it works:**

* Fetches **latest block hash**.
* Creates a **hash input string** using sender, receiver, amount, timestamp, and previous hash.
* Uses generate\_hash to create **new block hash**.
* Inserts **new transaction** into blockchain\_ledger.

**Step 5: Enforce Immutability with a Trigger**

Prevent direct updates or deletions on blockchain data.

04\_create\_triggers.sql

CREATE OR REPLACE TRIGGER enforce\_immutability

BEFORE UPDATE OR DELETE ON blockchain\_ledger

BEGIN

    RAISE\_APPLICATION\_ERROR(-20001, 'Blockchain records are immutable!');

END;

/

**How it works:**

* Prevents **updates or deletions** to maintain **data integrity**.
* Uses RAISE\_APPLICATION\_ERROR to enforce immutability.

**Step 6: Verify Blockchain Integrity**

Check if any block has been **tampered with** by recalculating hashes.

test\_blockchain\_integrity.sql

CREATE OR REPLACE FUNCTION verify\_blockchain RETURN VARCHAR2 IS

    v\_previous\_hash VARCHAR2(64);

    v\_computed\_hash VARCHAR2(64);

    v\_transaction\_string VARCHAR2(4000);

BEGIN

    FOR block IN (SELECT \* FROM blockchain\_ledger ORDER BY transaction\_id) LOOP

        -- Skip Genesis block

        IF block.transaction\_id > 1 THEN

            -- Recompute the hash

            v\_transaction\_string := block.sender || block.receiver || block.amount || block.transaction\_time || block.previous\_hash;

            v\_computed\_hash := generate\_hash(v\_transaction\_string);

            -- Compare stored hash with computed hash

            IF v\_computed\_hash != block.current\_hash THEN

                RETURN 'Blockchain integrity compromised at Transaction ID: ' || block.transaction\_id;

            END IF;

        END IF;

        -- Set previous hash for next iteration

        v\_previous\_hash := block.current\_hash;

    END LOOP;

    RETURN 'Blockchain is valid.';

END verify\_blockchain;

/

**How it works:**

* Iterates through all blocks.
* **Recalculates hash** for each transaction.
* Compares stored hash with newly computed hash.
* **Returns an alert** if tampering is detected.

**Step 7: Enable Digital Signatures for Transaction Verification**

Each transaction is signed using a **private key** and verified using a **public key**.

**First, create a table to store digital signatures:**

06\_create\_signature\_table.sql

CREATE TABLE transaction\_signatures (

    transaction\_id NUMBER REFERENCES blockchain\_ledger(transaction\_id),

    user\_signature VARCHAR2(4000),

    CONSTRAINT pk\_signature PRIMARY KEY (transaction\_id)

);

**Generate a digital signature for transactions:**

07\_generate\_signature\_function.sql

CREATE OR REPLACE FUNCTION generate\_signature(p\_transaction\_id NUMBER, p\_private\_key VARCHAR2) RETURN VARCHAR2 IS

BEGIN

    RETURN DBMS\_CRYPTO.ENCRYPT(p\_transaction\_id || p\_private\_key, DBMS\_CRYPTO.ENCRYPT\_AES128, UTL\_RAW.CAST\_TO\_RAW(p\_private\_key));

END generate\_signature;

/

**Verify the digital signature before processing transactions:**

08\_verify\_signature\_function.sql

CREATE OR REPLACE FUNCTION verify\_signature(p\_transaction\_id NUMBER, p\_user\_signature VARCHAR2, p\_public\_key VARCHAR2) RETURN VARCHAR2 IS

    v\_decrypted\_value VARCHAR2(4000);

BEGIN

    v\_decrypted\_value := DBMS\_CRYPTO.DECRYPT(UTL\_RAW.CAST\_TO\_RAW(p\_user\_signature), DBMS\_CRYPTO.ENCRYPT\_AES128, UTL\_RAW.CAST\_TO\_RAW(p\_public\_key));

    IF v\_decrypted\_value = p\_transaction\_id || p\_public\_key THEN

        RETURN 'Signature Valid';

    ELSE

        RETURN 'Signature Invalid';

    END IF;

END verify\_signature;

/

**Step 8: Create APIs to Query Blockchain Data**

Allow users to fetch **transaction history** and **blockchain status**.

**Get transaction details:**

05\_create\_apis.sql

CREATE OR REPLACE FUNCTION get\_transaction\_details(p\_transaction\_id NUMBER) RETURN CLOB IS

    v\_details CLOB;

BEGIN

    SELECT JSON\_OBJECT(

        'Transaction ID' VALUE transaction\_id,

        'Sender' VALUE sender,

        'Receiver' VALUE receiver,

        'Amount' VALUE amount,

        'Time' VALUE transaction\_time,

        'Hash' VALUE current\_hash

    ) INTO v\_details

    FROM blockchain\_ledger

    WHERE transaction\_id = p\_transaction\_id;

    RETURN v\_details;

END get\_transaction\_details;

/

**Get blockchain status:**

09\_create\_blockchain\_status\_function.sql

CREATE OR REPLACE FUNCTION get\_blockchain\_status RETURN VARCHAR2 IS

BEGIN

    RETURN verify\_blockchain;

END get\_blockchain\_status;

/

**Step 9: Configure Database Connections & Encryption Keys**

Secure database access and store encryption keys for signature verification.

db\_connection.sql

-- Database Connection Configuration

DEFINE DB\_USER = 'blockchain\_admin';

DEFINE DB\_PASSWORD = 'securepassword';

DEFINE DB\_HOST = 'localhost';

DEFINE DB\_PORT = 1521;

DEFINE DB\_SERVICE\_NAME = 'XE';

encryption\_keys.txt

# Encryption Keys (Store securely and restrict access)

PRIVATE\_KEY=1234567890abcdef1234567890abcdef

PUBLIC\_KEY=abcdef1234567890abcdef1234567890

**Step 10: Deployment & Rollback Scripts *(Filename: deployment/ & deployment/)***

Deploy the blockchain system with a single script.

deploy\_blockchain.sql

-- deploy\_blockchain.sql

START scripts/01\_create\_tables.sql;

START scripts/02\_create\_functions.sql;

START scripts/03\_create\_procedures.sql;

START scripts/04\_create\_triggers.sql;

START scripts/05\_create\_apis.sql;

START data/genesis\_block.sql;

START data/sample\_transactions.sql;

COMMIT;

rollback.sql

-- rollback.sql

DROP TABLE transaction\_signatures;

DROP TABLE blockchain\_ledger;

DROP FUNCTION generate\_hash;

DROP FUNCTION verify\_blockchain;

DROP PROCEDURE add\_transaction;

DROP TRIGGER enforce\_immutability;

COMMIT;

**Final Outcome**

**Fully functional blockchain ledger in PL/SQL.**  
**Data integrity maintained using SHA-256 hashing.**  
**Transactions are immutable (no updates/deletes).**  
**Signature-based verification ensures authenticity.**  
**APIs for querying blockchain data.**