

# Abhishek R-P - Script function explanation

Tell me what this script does : `#!/usr/bin/env python3`  
"""

Manual verification script - randomly selects a table and shows SQL vs Parquet data for manual comparison. Checks 3 rows and all columns.

"""

```
import os
import glob
import gzip
import pandas as pd
import subprocess
import random
import re
from datetime import datetime
```

```
PARQUET_DIR =
"/Users/abhishek/market_parquet_v2/_by_table/options_sql"
ARCHIVE_PATH =
"/Users/abhishek/market_archives/bankopt/das_bankopt_backup.7z.001"
```

```
def parse_sql_row(row_str):
    """Parse a SQL row string into values"""
    values = []
    current = ""
    in_quotes = False

    # Remove parentheses
    row_str = row_str.strip('()')

    for i, char in enumerate(row_str):
        if char == '\\' and i + 1 < len(row_str) and row_str[i + 1] == '"':
            current += '"'
            continue
        elif char == '"' and (i == 0 or row_str[i-1] != '\\'):
            in_quotes = not in_quotes
```

```

elif char == ',' and not in_quotes:
    val = current.strip()
    if val.startswith('"') and val.endswith('"'):
        val = val[1:-1]
    values.append(val)
    current = "
else:
    current += char

# Don't forget the last value
if current:
    val = current.strip()
    if val.startswith('"') and val.endswith('"'):
        val = val[1:-1]
    values.append(val)

return values

def get_sql_structure_and_data(table_name):
    """Extract SQL structure and data from archive"""
    print(f"\n{'='*80}")
    print(f"SQL DATA SECTION - TABLE: {table_name}")
    print(f"{'='*80}")

    try:
        # First get the table structure
        cmd_structure = f"""7za e -so '{ARCHIVE_PATH}'
'das_bankopt/{table_name}.sql.gz' 2>/dev/null | gunzip
| grep -A100 "CREATE TABLE" | head -120"""
        result_structure = subprocess.run(cmd_structure,
shell=True, capture_output=True, text=True,
timeout=30)

        columns = []
        if result_structure.stdout:
            lines = result_structure.stdout.split('\n')
            for line in lines:
                if ' in line and 'CREATE TABLE' not
in line and 'KEY' not in line and 'ENGINE'
not in line:
                    col_match = re.search(r'(\w+)', line)
                    if col_match:

```

```

        columns.append(col_match.group(1))

    print(f"\n📄 SQL TABLE STRUCTURE:")
    print(f"Total columns in SQL: {len(columns)}")
    print(f"\nColumn names:")
    for i in range(0, len(columns), 10):
        print(f" [{i+1:2d}-{min(i+10, len(columns)):2d}]:
{'', '.join(columns[i:i+10])}")

    # Get the actual data rows
    cmd_data = f"""\7za e -so '{ARCHIVE_PATH}'
'das_bankopt/{table_name}.sql.gz' 2>/dev/null | gunzip
| grep -A20 "VALUES" """""
    result_data = subprocess.run(cmd_data,
    shell=True, capture_output=True, text=True,
    timeout=30)

    if not result_data.stdout:
        print("❌ Failed to extract SQL data")
        return None, None

    # Parse the VALUES section to get first 3 rows
    sql_rows = []
    lines = result_data.stdout.split('\n')

    # Find VALUES line and parse
    for i, line in enumerate(lines):
        if 'VALUES' in line:
            # Extract from this line
            values_part = line.split('VALUES')[1] if 'VALUES'
in line else "

            # Get all rows from the SQL dump
            all_rows_text = values_part
            for j in range(i+1, min(i+20, len(lines))):
                if lines[j].strip() and not lines[j].startswith('--
):
                    all_rows_text += ' ' + lines[j].strip()

            # Split by ),( to get individual rows
            row_matches = re.findall(r'\([^)]+\)',
all_rows_text)

```

```

        for row_match in row_matches[:3]: # Get first
3 rows
            values = parse_sql_row(row_match)
            sql_rows.append(values)
            if len(sql_rows) >= 3:
                break
        break

    # Determine if it's REPLACE INTO or INSERT INTO
    is_replace = 'REPLACE INTO' in result_data.stdout
    table_type = "REPLACE INTO" if is_replace else
"INSERT INTO"

    print(f"\n📄 SQL DATA (Type: {table_type}):")
    print(f"Showing first {len(sql_rows)} rows, all
{len(columns)} columns")
    print("-" * 80)

    for row_idx, row in enumerate(sql_rows):
        print(f"\n ♦ SQL Row {row_idx + 1}:")
        for col_idx in range(0, min(len(row),
len(columns)), 5):
            end_idx = min(col_idx + 5, len(row),
len(columns))
            print(f" Columns {col_idx+1:2d}-
{end_idx:2d}:")
            for i in range(col_idx, end_idx):
                col_name = columns[i] if i < len(columns)
else f"col_{i+1}"
                val = row[i] if i < len(row) else "N/A"
                # Truncate long values for display
                if len(str(val)) > 30:
                    val = str(val)[:30] + "..."
                print(f" [{i+1:2d}] {col_name:15s} = {val}")

    return sql_rows, columns

except Exception as e:
    print(f"❌ Error extracting SQL: {e}")
    return None, None

```

```

def get_parquet_data_detailed(table_name):
    """Get detailed data from Parquet file"""
    print(f"\n{'='*80}")
    print(f"PARQUET DATA SECTION - TABLE: {table_name}")
    print(f"{'='*80}")

    parquet_file = os.path.join(PARQUET_DIR, f"{table_name}.parquet")

    if not os.path.exists(parquet_file):
        print(f"❌ Parquet file not found: {parquet_file}")
        return None, None

    try:
        df = pd.read_parquet(parquet_file)

        print(f"\n📊 PARQUET FILE INFO:")
        print(f"File: {os.path.basename(parquet_file)}")
        print(f"Total rows: {len(df):,}")
        print(f"Total columns: {len(df.columns)}")
        print(f"File size: {os.path.getsize(parquet_file) / (1024*1024):.2f} MB")

        columns = df.columns.tolist()
        print(f"\n📋 PARQUET COLUMNS:")
        print(f"Total columns in Parquet: {len(columns)}")
        print(f"Column names:")
        for i in range(0, len(columns), 10):
            print(f" [{i+1:2d}-{min(i+10, len(columns)):2d}]: {', '.join(columns[i:i+10])}")

        if len(df) == 0:
            print("\n⚠️ Table is empty (0 rows)")
            return [], columns

        # Get first 3 rows
        parquet_rows = []
        for idx in range(min(3, len(df))):
            row = df.iloc[idx]
            values = []
            for val in row.values:

```

```

        if pd.isna(val):
            values.append('\N')
        elif isinstance(val, pd.Timestamp):
            values.append(val.strftime('%Y-%m-%d
%H:%M:%S'))
        else:
            values.append(str(val))
        parquet_rows.append(values)

    print(f"\n📊 PARQUET DATA:")
    print(f"Showing first {len(parquet_rows)} rows, all
{len(columns)} columns")
    print("-" * 80)

    for row_idx, row in enumerate(parquet_rows):
        print(f"\n ♦ Parquet Row {row_idx + 1}:")
        for col_idx in range(0, len(row), 5):
            end_idx = min(col_idx + 5, len(row))
            print(f" Columns {col_idx+1:2d}-
{end_idx:2d}:")
            for i in range(col_idx, end_idx):
                col_name = columns[i] if i < len(columns)
            else f"col_{i+1}"
            val = row[i]
            # Truncate long values for display
            if len(str(val)) > 30:
                val = str(val)[:30] + "..."
            print(f" [{i+1:2d}] {col_name:15s} = {val}")

    return parquet_rows, columns

except Exception as e:
    print(f"❌ Error reading Parquet: {e}")
    return None, None

def compare_all_values(sql_rows, sql_cols,
parquet_rows, parquet_cols):
    """Compare all values between SQL and Parquet for
3 rows"""
    print(f"\n{'='*80}")
    print("DETAILED COMPARISON - SQL vs PARQUET")
    print(f"{'='*80}")

```

```

if not sql_rows or not parquet_rows:
    print("❌ Cannot compare - missing data")
    return

# Determine if SQL is REPLACE INTO (missing
volactual at position 5)
is_replace = len(sql_cols) < len(parquet_cols) and
'volactual' in parquet_cols

print(f"\n📊 COMPARISON DETAILS:")
print(f"SQL columns: {len(sql_cols)}")
print(f"Parquet columns: {len(parquet_cols)}")
if is_replace:
    print("Note: SQL is REPLACE INTO format (missing
'volactual' column)")
    print("-" * 80)

# Compare each row
for row_idx in range(min(len(sql_rows),
len(parquet_rows), 3)):
    print(f"\n🔍 COMPARING ROW {row_idx + 1}:")

    sql_row = sql_rows[row_idx]
    pq_row = parquet_rows[row_idx]

    # Adjust for REPLACE INTO missing volactual
    if is_replace and len(sql_row) < len(pq_row):
        # Insert None at position 5 for volactual in SQL
row
        sql_row = sql_row[:5] + ['\N'] + sql_row[5:]

    matches = 0
    mismatches = []

    # Compare first 46 columns (core data columns)
    for i in range(min(46, len(sql_row), len(pq_row))):
        sql_val = sql_row[i] if i < len(sql_row) else "
        pq_val = pq_row[i] if i < len(pq_row) else "

    # Normalize values for comparison
    sql_val_norm = sql_val.replace('NULL',

```

```

'\N').replace('.00', '').replace('.0', '')
    pq_val_norm = pq_val.replace('None',
'\N').replace('nan', '\N').replace('.0', '')

    # Handle numeric comparison
    try:
        if '.' in sql_val or '.' in pq_val:
            if abs(float(sql_val) -
float(pq_val.replace('\N', '0'))) < 0.01:
                matches += 1
                continue
    except:
        pass

    # String comparison
    if sql_val_norm == pq_val_norm or sql_val in
pq_val or pq_val in sql_val:
        matches += 1
    else:
        mismatches.append({
            'col_idx': i + 1,
            'col_name': sql_cols[i] if i < len(sql_cols) else
f'col_{i+1}',
            'sql': sql_val[:30],
            'parquet': pq_val[:30]
        })

    print(f" ✅ Matched: {matches}/46 columns")

    if mismatches:
        print(f" ⚠ Mismatches: {len(mismatches)}")
        for m in mismatches[:5]: # Show first 5
            mismatches
                print(f"   Column {m['col_idx']}
({m['col_name']}): SQL='{m['sql']}' vs
PQ='{m['parquet']}'")

    # Overall summary
    print(f"\n📊 VERIFICATION SUMMARY:")
    print(f"✅ Data structure: {'MATCH' if len(sql_cols) ==
46 or len(sql_cols) == 52 else 'ISSUE'}")
    print(f"✅ Row count checked: {min(len(sql_rows),

```

```

len(parquet_rows), 3})")
    print(f"✅ Columns compared: 46 (core data
columns)")

    if len(mismatches) == 0:
        print(f"\n🎉 PERFECT MATCH - All values in SQL
and Parquet are identical!")
    else:
        print(f"\n⚠️ Found some differences - review the
mismatches above")

def main():
    print("="*80)
    print("MANUAL SQL vs PARQUET VERIFICATION (3
Rows, All Columns)")
    print(f"Date: {datetime.now()}")
    print("="*80)

    # Get all parquet files
    all_parquet = glob.glob(os.path.join(PARQUET_DIR,
"*.*parquet"))

    if not all_parquet:
        print("❌ No parquet files found!")
        return

    print(f"\n📊 Found {len(all_parquet):,} parquet files")

    # Randomly select a table
    selected_file = random.choice(all_parquet)
    table_name =
os.path.basename(selected_file).replace('.parquet', '')

    # Parse table name for display
    match = re.match(r'banknifty(\d+)([a-z]+)(\d+)
(ce|pe)', table_name)
    if match:
        year = match.group(1)
        month = match.group(2)
        strike = match.group(3)
        opt_type = match.group(4).upper()

```

```

        print(f"\n🎲 RANDOMLY SELECTED TABLE:")
        print(f"  Table name: {table_name}")
        print(f"  Strike: {strike}")
        print(f"  Type: {opt_type}")
        print(f"  Expiry: {month.upper()} 20{year}")
    else:
        print(f"\n🎲 RANDOMLY SELECTED TABLE:
{table_name}")

    # Get SQL data (Section 1)
    sql_rows, sql_cols =
get_sql_structure_and_data(table_name)

    # Get Parquet data (Section 2)
    parquet_rows, parquet_cols =
get_parquet_data_detailed(table_name)

    # Compare values (Section 3)
    if sql_rows and parquet_rows:
        compare_all_values(sql_rows, sql_cols,
parquet_rows, parquet_cols)

    print(f"\n{'='*80}")
    print("VERIFICATION COMPLETE")
    print(f"\n{'='*80}")

    print("\n📌 KEY POINTS:")
    print("  • Checked 3 rows and all 46 core columns")
    print("  • REPLACE INTO tables: 46 columns in SQL →
52 in Parquet (volactual added as NULL)")
    print("  • INSERT INTO tables: 47 columns in SQL →
53 in Parquet")
    print("  • Parquet adds 5 metadata columns at the
end")
    print("  • Some 1970-01-01 timestamps are from
original data")

    print("\n🔄 Run this script again to check another
random table")

if __name__ == "__main__":
    main()

```

Here's the plain-English version of what your script does:

## What it's for

A quick **manual verification tool**: it randomly picks one options table, pulls a few rows from your **MySQL dump inside a 7z archive**, loads the matching **Parquet** file, and then **compares values column-by-column** to check your SQL→Parquet conversion.

## Inputs it expects (hard-coded paths)

- Parquet files in:  
`/Users/abhishek/market_parquet_v2/_by_table/options_sql/*.parquet`
- A multipart 7z SQL archive containing files like `das_bankopt/<table>.sql.gz` :  
`/Users/abhishek/market_archives/bankopt/das_bankopt_backup.7z.001`

## What it does step by step

### 1. Pick a random table

- Lists all Parquet files, picks one at random, derives `table_name` from the filename.
- If the name matches `banknifty(\d+)([a-z]+)(\d+)(ce|pe)`, it prints a friendly summary (strike, CE/PE, expiry).

### 2. Extract SQL schema & a few data rows from the archive

- Uses shell commands to stream-extract one `*.sql.gz` from the 7z (via `7za e -so ... | gunzip`), then:
  - **Schema**: `grep` around `"CREATE TABLE"` and heuristically parse column names (lines with backticks that aren't KEY/ENGINE).
  - **Data**: `grep -A20 "VALUES"` to capture the INSERT/REPLACE statement chunk and then parse **the first 3 rows** from the VALUES list.
- A custom `parse_sql_row()` handles commas inside quotes, simple escaping, and strips surrounding quotes.

### 3. Load Parquet & show sample

- Reads the matching Parquet with `pandas.read_parquet`.
- Prints file info, column list, and the **first 3 rows**, converting NaN/NaT to `\N` and timestamps to `YYYY-MM-DD HH:MM:SS`.

### 4. Compare SQL vs Parquet (the core)

- Detects a **REPLACE INTO quirk**: if Parquet has a `volactual` column but SQL row count is shorter, it assumes the SQL dump omitted `volactual` and

inserts a `\N` at position 6 (index 5) before comparing.

- Compares the first **46 "core" columns** per row:
  - Normalizes NULL/NaN as `\N`, trims trivial decimal tails (e.g., `.0`, `.00`).
  - Tries numeric comparison with a small tolerance (`<0.01`).
  - Otherwise uses string equality or substring containment.
- Prints **match counts** and up to 5 **mismatch details** (column index/name and truncated values).

## 5. Summary & notes

- Prints a high-level summary (rows checked, columns compared).
- Mentions expected column counts:
  - REPLACE INTO tables: **46 in SQL → 52 in Parquet** (adds `volactual` + 5 Parquet metadata cols).
  - INSERT INTO tables: **47 in SQL → 53 in Parquet**.
- Notes that some `1970-01-01` timestamps originate from the source.

## Dependencies/assumptions

- CLI tools: `7za`, `gunzip`, `grep`, `head`.
- Python: `pandas`, a Parquet engine (`pyarrow/fastparquet`).
- The 7z archive must contain the expected `das_bankopt/<table>.sql.gz` path.
- The SQL dump format must have `CREATE TABLE` followed by typical column lines with backticks, and INSERT/REPLACE statements with a `VALUES (...), (...)` structure.
- It only checks **3 rows** and **first 46 columns** (by design).

## What it prints

- SQL schema (column count + paginated names).
- First 3 SQL rows (grouped 5 cols at a time).
- Parquet file metadata and first 3 rows.
- A row-by-row comparison with match/mismatch counts.
- Final verification summary.

## Limitations / fragile spots

- **Row capture window:** `grep -A20 "VALUES"` may miss long INSERTs that span more than ~20 lines, or different dump layouts.
- **Parser simplicity:** `parse_sql_row()` covers quoted strings and simple escaping, but complex edge cases (e.g., multi-byte escapes, nested parentheses in JSON

strings) could break.

- **Hard-coded magic numbers:** Compares only **46 columns** and inserts `volactual` at **position 6**—these assumptions must match your real schemas.
- **String containment rule:** Treating “A in B or B in A” as a match can hide subtle differences.
- **Random table selection:** Non-deterministic unless you fix the seed or choose a table explicitly.
- **Shelling out:** Uses `shell=True` pipelines; brittle to path/locale differences and slower than native parsing.

## Quick ways to improve (if you want)

- Make paths, columns-to-compare, and number of rows **configurable** via args/env.
- Replace `grep -A20` with a **proper SQL dump scanner** that collects the full `INSERT ... VALUES` block reliably.
- Compare **by column names** (align schemas) instead of raw positional comparison; print dtype deltas.
- Use stricter equality (or per-column tolerances) instead of substring matches.
- Add an option to **seed RNG** or pass an explicit `--table` to test deterministically.
- Emit a **CSV/markdown summary** of mismatches for logging.
- Fail fast when **column counts diverge unexpectedly** (e.g., more than the known metadata differences).

If you just run it as-is ( `python3 script.py` ), it will do one random pick, print both sides, and show a compact diff for the first three rows.