Unified Transactions Reconciliation Platform (UTRP)

SQL Analysis

```sql
-- Creating a small warehouse for development
CREATE OR REPLACE WAREHOUSE UTRP_WH
  WAREHOUSE_SIZE = 'XSMALL'
  AUTO_SUSPEND = 60
  AUTO_RESUME = TRUE;
```

```sql
-- Creating database and schema
CREATE OR REPLACE DATABASE UTRP_DB;
CREATE OR REPLACE SCHEMA UTRP_SCHEMA;
```

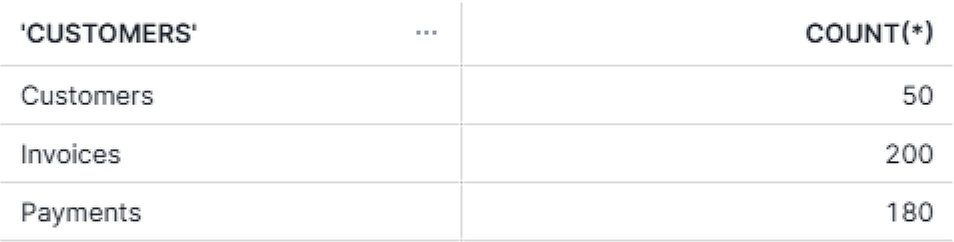
```sql
-- Creating Tables
CREATE OR REPLACE TABLE CUSTOMERS(
ext_customer_key VARCHAR,
name VARCHAR,
email VARCHAR,
created_at DATE
);

CREATE OR REPLACE TABLE INVOICES(
ext_invoice_key VARCHAR,
ext_customer_key VARCHAR,
invoice_date DATE,
amount NUMBER(10, 2),
currency VARCHAR,
status VARCHAR,
source_system VARCHAR,
inserted_at TIMESTAMP_NTZ
);

CREATE OR REPLACE TABLE PAYMENTS(
ext_payment_key VARCHAR,
ext_customer_key VARCHAR,
payment_date DATE,
amount NUMBER(10, 2),
currency VARCHAR,
payment_method VARCHAR,
source_system VARCHAR,
inserted_at TIMESTAMP_NTZ
);
```

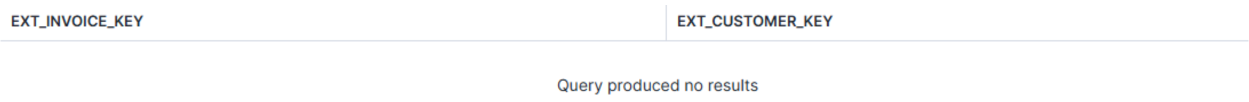
```
-- Data Preview
SELECT * FROM CUSTOMERS;
SELECT * FROM INVOICES;
SELECT * FROM PAYMENTS;
```

```
-- Performing validation of the data
SELECT 'Customers', COUNT(*) FROM CUSTOMERS
UNION ALL
SELECT 'Invoices', COUNT(*) FROM INVOICES
UNION ALL
SELECT 'Payments', COUNT(*) FROM PAYMENTS;
```



'CUSTOMERS' ···	COUNT(*)
Customers	50
Invoices	200
Payments	180

```
-- Checking if there are no matching customer
SELECT
   i.ext_invoice_key, i.ext_customer_key
FROM
   Invoices i
LEFT JOIN
   Customers c ON i.ext_customer_key = c.ext_customer_key
WHERE
   c.ext_customer_key IS NULL;
```

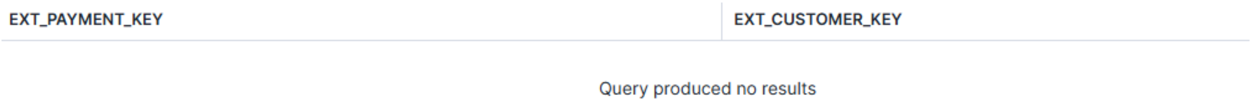


EXT_INVOICE_KEY	EXT_CUSTOMER_KEY

Query produced no results

```
-- Payments without a customer
SELECT
   p.ext_payment_key, p.ext_customer_key
FROM
   Payments p
LEFT JOIN
   Customers c ON p.ext_customer_key = c.ext_customer_key
```

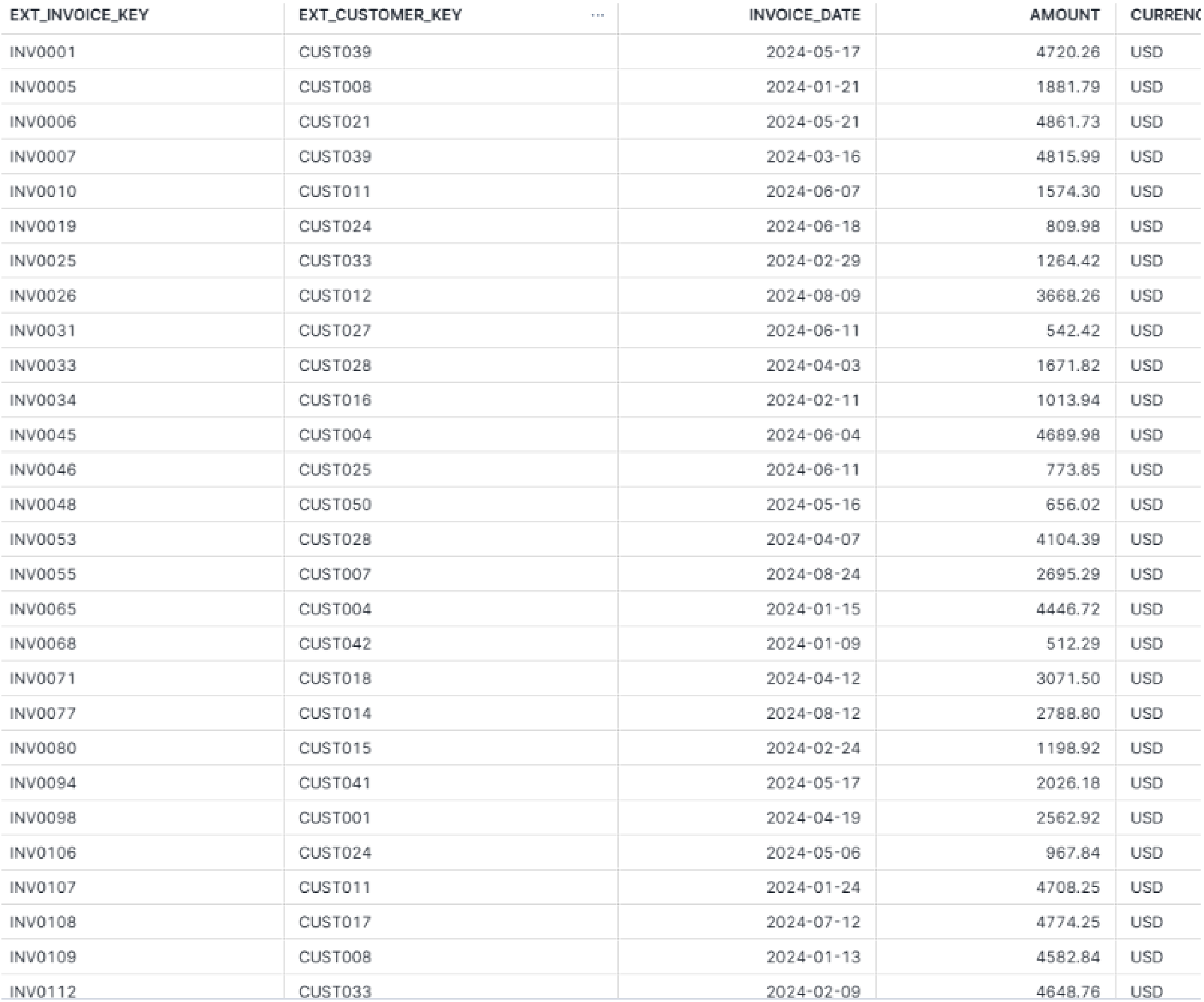
```
WHERE
    c.ext_customer_key IS NULL;
```



EXT_PAYMENT_KEY	EXT_CUSTOMER_KEY

Query produced no results

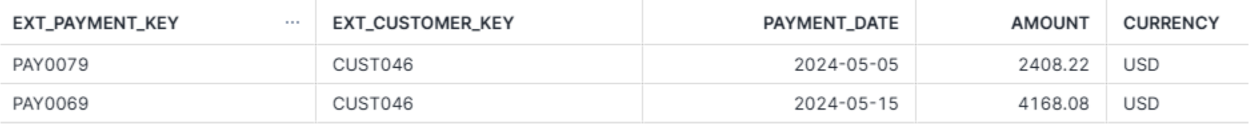
```
-- Invoices with no payments (within 30 days)
SELECT
    i.ext_invoice_key, i.ext_customer_key, i.invoice_date, i.amount, i.currency
FROM
    Invoices i
LEFT JOIN
    Payments p ON i.ext_customer_key = p.ext_customer_key AND ABS(DATEDIFF(day,
i.invoice_date, p.payment_date)) <= 30
WHERE
    p.ext_payment_key IS NULL;
```



EXT_INVOICE_KEY	EXT_CUSTOMER_KEY ···	INVOICE_DATE	AMOUNT	CURRENC
INV0001	CUST039	2024-05-17	4720.26	USD
INV0005	CUST008	2024-01-21	1881.79	USD
INV0006	CUST021	2024-05-21	4861.73	USD
INV0007	CUST039	2024-03-16	4815.99	USD
INV0010	CUST011	2024-06-07	1574.30	USD
INV0019	CUST024	2024-06-18	809.98	USD
INV0025	CUST033	2024-02-29	1264.42	USD
INV0026	CUST012	2024-08-09	3668.26	USD
INV0031	CUST027	2024-06-11	542.42	USD
INV0033	CUST028	2024-04-03	1671.82	USD
INV0034	CUST016	2024-02-11	1013.94	USD
INV0045	CUST004	2024-06-04	4689.98	USD
INV0046	CUST025	2024-06-11	773.85	USD
INV0048	CUST050	2024-05-16	656.02	USD
INV0053	CUST028	2024-04-07	4104.39	USD
INV0055	CUST007	2024-08-24	2695.29	USD
INV0065	CUST004	2024-01-15	4446.72	USD
INV0068	CUST042	2024-01-09	512.29	USD
INV0071	CUST018	2024-04-12	3071.50	USD
INV0077	CUST014	2024-08-12	2788.80	USD
INV0080	CUST015	2024-02-24	1198.92	USD
INV0094	CUST041	2024-05-17	2026.18	USD
INV0098	CUST001	2024-04-19	2562.92	USD
INV0106	CUST024	2024-05-06	967.84	USD
INV0107	CUST011	2024-01-24	4708.25	USD
INV0108	CUST017	2024-07-12	4774.25	USD
INV0109	CUST008	2024-01-13	4582.84	USD
INV0112	CUST033	2024-02-09	4648.76	USD

- List of customers with unpaid invoices.

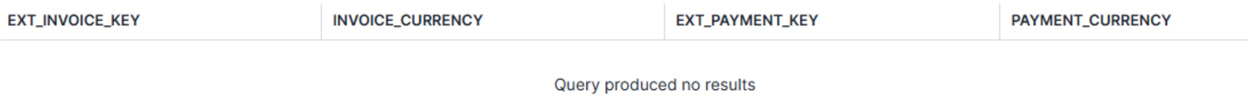
```
-- Payments with no invoices (within 30 days)
SELECT
   p.ext_payment_key, p.ext_customer_key, p.payment_date, p.amount, p.currency
FROM
   Payments p
LEFT JOIN
   Invoices i ON p.ext_customer_key = i.ext_customer_key AND ABS(DATEDIFF(day, i.invoice_date, p.payment_date)) <= 30
WHERE
   i.ext_invoice_key IS NULL;
```



EXT_PAYMENT_KEY	EXT_CUSTOMER_KEY	PAYMENT_DATE	AMOUNT	CURRENCY
PAY0079	CUST046	2024-05-05	2408.22	USD
PAY0069	CUST046	2024-05-15	4168.08	USD

- List of payments that doesn’t seem to be linked with any invoices.

```
-- Currency mismatches
SELECT
   i.ext_invoice_key, i.currency AS Invoice_Currency, p.ext_payment_key, p.currency AS Payment_Currency
FROM
   Invoices i
JOIN
   Payments p ON i.ext_customer_key = p.ext_customer_key AND ABS(DATEDIFF(day, i.invoice_date, p.payment_date)) <= 30
WHERE
   i.currency <> p.currency;
```



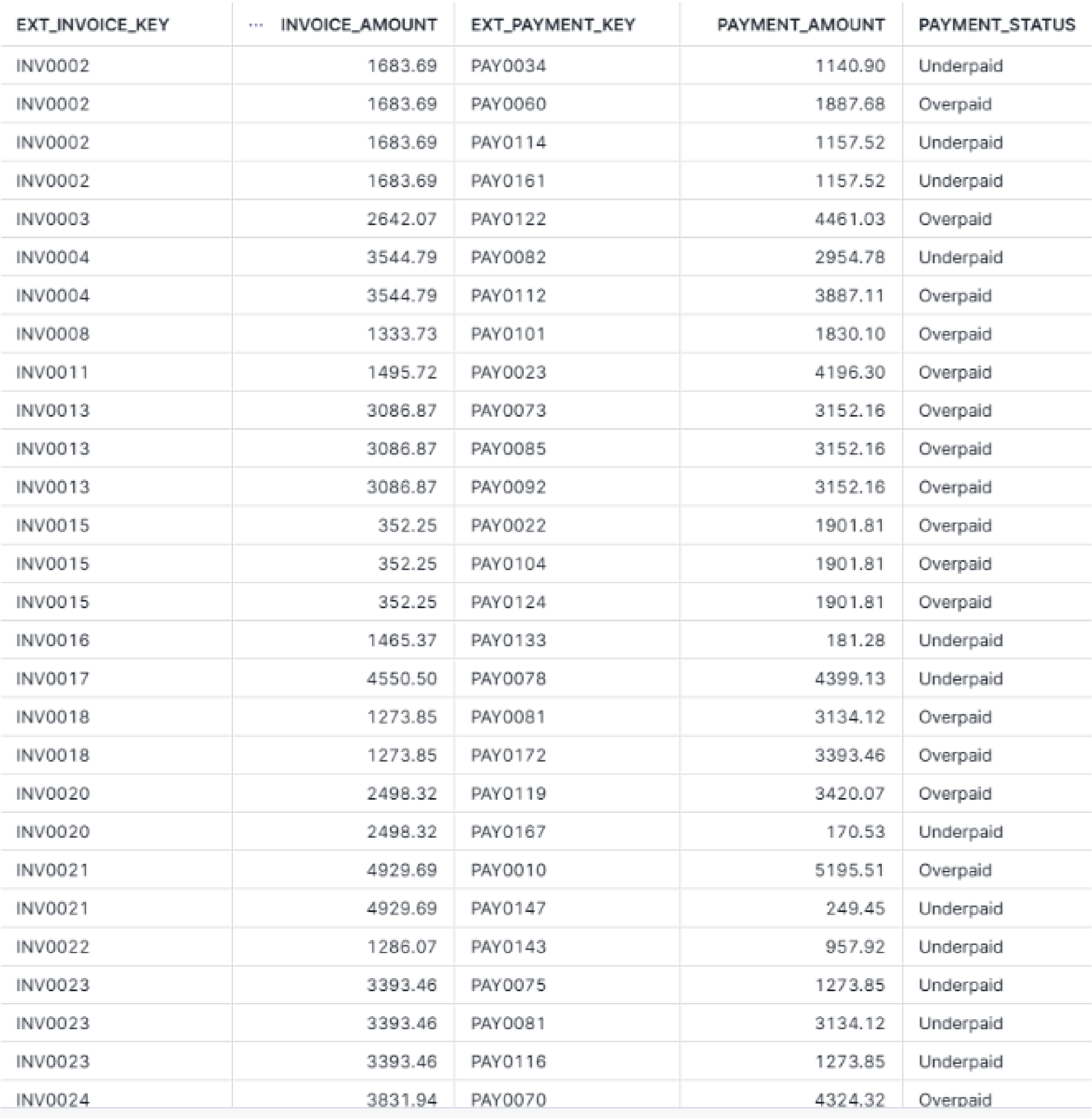
EXT_INVOICE_KEY	INVOICE_CURRENCY	EXT_PAYMENT_KEY	PAYMENT_CURRENCY

Query produced no results

- No mismatch in currencies between invoices and payments tables.

```
-- Overpayment or Underpayment
-- Invoices that have any single payment that doesn't exactly equal the invoice amount
SELECT
   i.ext_invoice_key, i.amount AS Invoice_Amount, p.ext_payment_key, p.amount AS Payment_Amount,
   CASE
      WHEN i.amount > p.amount THEN 'Underpaid'
```

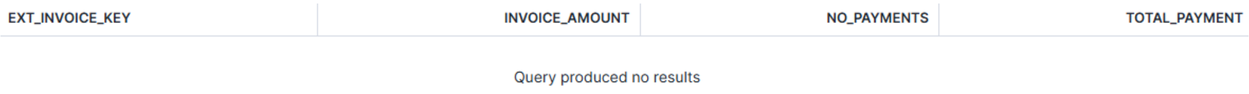
```
        WHEN i.amount < p.amount THEN 'Overpaid'
    END AS payment_status
FROM
    Invoices i
JOIN
    Payments p ON i.ext_customer_key = p.ext_customer_key AND ABS(DATEDIFF(day,
i.invoice_date, p.payment_date)) <= 30
WHERE
    p.amount <> i.amount;
```



EXT_INVOICE_KEY	··· INVOICE_AMOUNT	EXT_PAYMENT_KEY	PAYMENT_AMOUNT	PAYMENT_STATUS
INV0002	1683.69	PAY0034	1140.90	Underpaid
INV0002	1683.69	PAY0060	1887.68	Overpaid
INV0002	1683.69	PAY0114	1157.52	Underpaid
INV0002	1683.69	PAY0161	1157.52	Underpaid
INV0003	2642.07	PAY0122	4461.03	Overpaid
INV0004	3544.79	PAY0082	2954.78	Underpaid
INV0004	3544.79	PAY0112	3887.11	Overpaid
INV0008	1333.73	PAY0101	1830.10	Overpaid
INV0011	1495.72	PAY0023	4196.30	Overpaid
INV0013	3086.87	PAY0073	3152.16	Overpaid
INV0013	3086.87	PAY0085	3152.16	Overpaid
INV0013	3086.87	PAY0092	3152.16	Overpaid
INV0015	352.25	PAY0022	1901.81	Overpaid
INV0015	352.25	PAY0104	1901.81	Overpaid
INV0015	352.25	PAY0124	1901.81	Overpaid
INV0016	1465.37	PAY0133	181.28	Underpaid
INV0017	4550.50	PAY0078	4399.13	Underpaid
INV0018	1273.85	PAY0081	3134.12	Overpaid
INV0018	1273.85	PAY0172	3393.46	Overpaid
INV0020	2498.32	PAY0119	3420.07	Overpaid
INV0020	2498.32	PAY0167	170.53	Underpaid
INV0021	4929.69	PAY0010	5195.51	Overpaid
INV0021	4929.69	PAY0147	249.45	Underpaid
INV0022	1286.07	PAY0143	957.92	Underpaid
INV0023	3393.46	PAY0075	1273.85	Underpaid
INV0023	3393.46	PAY0081	3134.12	Underpaid
INV0023	3393.46	PAY0116	1273.85	Underpaid
INV0024	3831.94	PAY0070	4324.32	Overpaid

- List of payments which are more or less than the invoice amounts.
- We can also observe that one invoice is linked to multiple payments.

```
-- Partial Payments
-- Invoices where sum of payments within 30 days equals the invoice amount
SELECT
   i.ext_invoice_key, i.amount AS invoice_amount, COUNT(p.ext_payment_key) AS
No_payments, SUM(p.amount) AS total_payment
FROM
   Invoices i
JOIN
   Payments p ON i.ext_customer_key = p.ext_customer_key AND ABS(DATEDIFF(day,
i.invoice_date, p.payment_date)) BETWEEN 0 AND 30
GROUP BY
   i.ext_invoice_key, i.amount
HAVING
   SUM(p.amount) = i.amount AND COUNT(p.ext_payment_key) > 1;
```



EXT_INVOICE_KEY	INVOICE_AMOUNT	NO_PAYMENTS	TOTAL_PAYMENT

Query produced no results

- No partially settled payments.

```
-- Canonical tables are created as they represent single source of true data.
-- They hold clean, de-duplicated, and standardized data that is ready for analysis.
-- In our project, the raw CSV data might be messy or may contain duplicate data
-- In order to proceed with better analysis, raw data is merged into canonical tables
(dim_customer, fact_invoice, fact_payment)
-- so that reconciliation rules can be applied consistently.
-- This step ensures Power BI reports are built only on trusted data,
-- avoiding duplicates, errors, and mismatches that exist in raw files.

CREATE OR REPLACE TABLE UTRP_SCHEMA.dim_customer (
  customer_id       NUMBER AUTOINCREMENT,  -- surrogate key
  ext_customer_key  VARCHAR,
  name             VARCHAR,
  email            VARCHAR,
  created_at       TIMESTAMP_NTZ,
  inserted_at      TIMESTAMP_NTZ,
  updated_at       TIMESTAMP_NTZ,
  PRIMARY KEY (customer_id)
);
```

```
CREATE OR REPLACE TABLE UTRP_SCHEMA.fact_invoice (
  invoice_id       NUMBER AUTOINCREMENT,
  ext_invoice_key   VARCHAR,
  ext_customer_key  VARCHAR,
  customer_id      NUMBER,
  invoice_date     DATE,
  amount           NUMBER(10,2),
  currency         VARCHAR,
  status           VARCHAR,
  source_system     VARCHAR,
  inserted_at      TIMESTAMP_NTZ,
  updated_at       TIMESTAMP_NTZ,
  PRIMARY KEY (invoice_id)
);

CREATE OR REPLACE TABLE UTRP_SCHEMA.fact_payment (
  payment_id       NUMBER AUTOINCREMENT,
  ext_payment_key   VARCHAR,
  ext_customer_key  VARCHAR,
  customer_id      NUMBER,
  payment_date     DATE,
  amount           NUMBER(10,2),
  currency         VARCHAR,
  payment_method    VARCHAR,
  source_system     VARCHAR,
  inserted_at      TIMESTAMP_NTZ,
  updated_at       TIMESTAMP_NTZ,
  PRIMARY KEY (payment_id)
);

CREATE OR REPLACE TABLE UTRP_SCHEMA.audit_merge_log (
  run_id           NUMBER AUTOINCREMENT,
  object_type      VARCHAR,
  rows_processed    NUMBER,
  rows_inserted     NUMBER,
  rows_updated      NUMBER,
  run_by           VARCHAR,
  run_at           TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP(),
  status           VARCHAR,
  message          VARCHAR
);
```

- We created canonical tables to make sure we always have clean and reliable data for reconciliation. Instead of building reports directly on raw data which can be messy and duplicated.
- So, we first standardize and store it in these canonical tables so that Power BI dashboards and business users only see the trusted version.

```
/*
===========================================================
Procedure: sp_merge_invoices
Purpose: Merge (upsert) invoices from staging into canonical fact_invoice table.

Why MERGE?
- Ensures invoices are always clean and up-to-date.
- If invoice is new then insert it.
- If invoice already exists but changed then update it.
- If invoice already exists and unchanged then do nothing.
- This avoids duplicates and stale data.
Why in a stored procedure?
- Automates the full process (count → merge → log → mark as processed).
- Guarantees idempotency (safe to rerun, no duplicates).
- Creates an audit trail (rows processed, inserted, updated).
Why important for this project?
- Canonical tables (fact_invoice, fact_payment, dim_customer) are the trusted single source of truth.
- Reconciliation logic depends on invoices being accurate.
- Finance teams can rely on this clean layer for Power BI dashboards.
===========================================================
*/
CREATE OR REPLACE PROCEDURE UTRP_SCHEMA.sp_merge_invoices(run_by STRING)
RETURNS VARIANT
LANGUAGE SQL
EXECUTE AS CALLER
AS
$$
DECLARE
  run_ts          TIMESTAMP_NTZ := CURRENT_TIMESTAMP();
  rows_processed   NUMBER := 0;
  rows_inserted    NUMBER := 0;
  rows_updated     NUMBER := 0;
BEGIN
  -- Count rows to process
  SET rows_processed = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.invoices WHERE processed = FALSE
  );
  -- Merge invoices into canonical fact_invoice
  MERGE INTO UTRP_SCHEMA.fact_invoice AS target
  USING (
    SELECT s.*, c.customer_id
    FROM UTRP_SCHEMA.invoices s
    LEFT JOIN UTRP_SCHEMA.dim_customer c
      ON s.ext_customer_key = c.ext_customer_key
    WHERE s.processed = FALSE
  ) AS src
  ON target.ext_invoice_key = src.ext_invoice_key
     AND target.source_system = src.source_system
  WHEN MATCHED AND (
       COALESCE(target.amount,0) <> COALESCE(src.amount,0)
    OR COALESCE(target.status,'') <> COALESCE(src.status,'')
    OR target.invoice_date <> src.invoice_date
  )
  THEN UPDATE SET
    amount     = src.amount,
    status     = src.status,
    invoice_date = src.invoice_date,
    updated_at = :run_ts
  WHEN NOT MATCHED THEN
    INSERT (ext_invoice_key, ext_customer_key, customer_id, invoice_date, amount, currency,
status, source_system, inserted_at, updated_at)
    VALUES (src.ext_invoice_key, src.ext_customer_key, src.customer_id, src.invoice_date,
src.amount, src.currency, src.status, src.source_system, :run_ts, :run_ts);

  -- Compute row counts
  SET rows_inserted = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.fact_invoice WHERE inserted_at = :run_ts
  );
  SET rows_updated = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.fact_invoice WHERE updated_at = :run_ts AND
inserted_at <> :run_ts
  );

  -- Log run
  INSERT INTO UTRP_SCHEMA.audit_merge_log(object_type, rows_processed, rows_inserted,
rows_updated, run_by, run_at, status)
  VALUES('fact_invoice', :rows_processed, :rows_inserted, :rows_updated, :run_by, :run_ts,
'SUCCESS');

  -- Mark invoices as processed
  UPDATE UTRP_SCHEMA.invoices
     SET processed = TRUE
   WHERE processed = FALSE;

  RETURN OBJECT_CONSTRUCT('status','SUCCESS',
                          'processed', :rows_processed,
                          'inserted', :rows_inserted,
                          'updated', :rows_updated);

EXCEPTION
  WHEN STATEMENT_ERROR OR EXPRESSION_ERROR THEN
    INSERT INTO UTRP_SCHEMA.audit_merge_log(object_type, rows_processed, rows_inserted,
rows_updated, run_by, run_at, status, message)
    VALUES('fact_invoice', :rows_processed, 0, 0, :run_by, :run_ts, 'FAILED', :sqlerrm);

    RETURN OBJECT_CONSTRUCT(
      'status','FAILED',
      'error_message', :sqlerrm,
      'sqlstate', :sqlstate
    );
END;
$$;


/*
=============================================================
Procedure: sp_merge_customers
Purpose  : Merge (upsert) customers from staging into
           canonical dim_customer table.

Why MERGE?
- Ensures customer records are clean and consistent.
- If customer is new then insert it.
- If customer already exists but details changed then update it.
- If customer already exists and unchanged then do nothing.
- This avoids duplicates and keeps master data reliable.

Why in a stored procedure?
- Automates the end-to-end process (count → merge → log → mark processed).
- Guarantees idempotency (safe to rerun, no duplicates).
- Creates an audit log (rows processed, inserted, updated).

Why important for this project?
- Customer is the key dimension linking invoices and payments.
- Canonical customer table ensures reconciliation always maps
  invoices and payments to the right entity.
- Provides a single source of truth for Power BI and downstream reporting.
============================================================
*/
CREATE OR REPLACE PROCEDURE UTRP_SCHEMA.sp_merge_customers(run_by STRING)
RETURNS VARIANT
LANGUAGE SQL
EXECUTE AS CALLER
AS
$$
DECLARE
  run_ts           TIMESTAMP_NTZ := CURRENT_TIMESTAMP();
  rows_processed   NUMBER := 0;
  rows_inserted    NUMBER := 0;
  rows_updated     NUMBER := 0;
BEGIN
  -- Count rows to process (assignment context: no colon on left; SQL inside needs : if using var)
  SET rows_processed = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.customers WHERE processed = FALSE
  );

  -- MERGE: note all scripting variables used in SQL are prefixed with :
  MERGE INTO UTRP_SCHEMA.dim_customer AS target
  USING (
    SELECT *
    FROM UTRP_SCHEMA.customers
    WHERE processed = FALSE
  ) AS src
  ON target.ext_customer_key = src.ext_customer_key
  WHEN MATCHED AND (
       COALESCE(target.name,'') <> COALESCE(src.name,'')
    OR COALESCE(target.email,'') <> COALESCE(src.email,'')
  )
  THEN UPDATE SET
    name       = src.name,
    email      = src.email,
    updated_at = :run_ts
  WHEN NOT MATCHED THEN
    INSERT (ext_customer_key, name, email, created_at, inserted_at, updated_at)
    VALUES (src.ext_customer_key, src.name, src.email, src.created_at, :run_ts, :run_ts);

  -- Compute row counts: use :run_ts inside the subqueries (SQL context)
  SET rows_inserted = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.dim_customer WHERE inserted_at = :run_ts
  );
  SET rows_updated = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.dim_customer WHERE updated_at = :run_ts AND
inserted_at <> :run_ts
  );

  -- Log run: use :vars inside the INSERT
  INSERT INTO UTRP_SCHEMA.audit_merge_log(
    object_type, rows_processed, rows_inserted, rows_updated, run_by, run_at, status
  )
  VALUES('dim_customer', :rows_processed, :rows_inserted, :rows_updated, :run_by, :run_ts,
'SUCCESS');

  -- Mark staging rows processed
  UPDATE UTRP_SCHEMA.customers
     SET processed = TRUE
   WHERE processed = FALSE;

  -- Return a JSON-like summary (use :vars)
  RETURN OBJECT_CONSTRUCT(
    'status','SUCCESS',
    'processed', :rows_processed,
    'inserted', :rows_inserted,
    'updated', :rows_updated
  );

-- Catch-all handler for Snowflake scripting
EXCEPTION
  WHEN STATEMENT_ERROR OR EXPRESSION_ERROR THEN
    INSERT INTO UTRP_SCHEMA.audit_merge_log(
      object_type, rows_processed, rows_inserted, rows_updated, run_by, run_at, status,
message
    )
    VALUES('dim_customer', :rows_processed, 0, 0, :run_by, :run_ts, 'FAILED', :sqlerrm);

    RETURN OBJECT_CONSTRUCT(
      'status','FAILED',
      'error_message', :sqlerrm,
      'sqlstate', :sqlstate
    );
END;
$$;
```

```sql
/*
==========================================================
Procedure: sp_merge_payments
Purpose: Merge (upsert) payments from staging into
         canonical fact_payment table.

Why MERGE?
- Ensures payments are stored once and always up-to-date.
- If payment is new then insert it.
- If payment already exists but details changed then update it.
- If payment already exists and unchanged then do nothing.
- This prevents duplicates and ensures reliable reconciliation.

Why in a stored procedure?
- Automates the process end-to-end (count → merge → log → mark processed).
- Idempotent design means you can rerun safely without duplication.
- Audit logging ensures every run is tracked for compliance.

Why important for this project?
- Payments are the core fact table to reconcile against invoices.
- Clean, canonical payments data allows accurate matching rules
  (exact match, partial, over/under, currency mismatch).
- Provides the foundation for exception reporting in Power BI.
==========================================================
*/
CREATE OR REPLACE PROCEDURE UTRP_SCHEMA.sp_merge_payments(run_by STRING)
RETURNS VARIANT
LANGUAGE SQL
EXECUTE AS CALLER
AS
$$
DECLARE
  run_ts          TIMESTAMP_NTZ := CURRENT_TIMESTAMP();
  rows_processed  NUMBER := 0;
  rows_inserted   NUMBER := 0;
  rows_updated    NUMBER := 0;
BEGIN
  -- Count rows to process
  SET rows_processed = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.payments WHERE processed = FALSE
  );

  -- Merge payments into canonical fact_payment
  MERGE INTO UTRP_SCHEMA.fact_payment AS target
  USING (
    SELECT s.*, c.customer_id
    FROM UTRP_SCHEMA.payments s
    LEFT JOIN UTRP_SCHEMA.dim_customer c
      ON s.ext_customer_key = c.ext_customer_key
    WHERE s.processed = FALSE
  ) AS src
  ON target.ext_payment_key = src.ext_payment_key
     AND target.source_system = src.source_system
  WHEN MATCHED AND (
       COALESCE(target.amount,0) <> COALESCE(src.amount,0)
    OR COALESCE(target.payment_method,'') <> COALESCE(src.payment_method,'')
    OR target.payment_date <> src.payment_date
  )
  THEN UPDATE SET
    amount         = src.amount,
    payment_method = src.payment_method,
    payment_date   = src.payment_date,
    updated_at     = :run_ts
  WHEN NOT MATCHED THEN
    INSERT (ext_payment_key, ext_customer_key, customer_id, payment_date, amount,
currency, payment_method, source_system, inserted_at, updated_at)
    VALUES (src.ext_payment_key, src.ext_customer_key, src.customer_id, src.payment_date,
src.amount, src.currency, src.payment_method, src.source_system, :run_ts, :run_ts);

  -- Compute row counts
  SET rows_inserted = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.fact_payment WHERE inserted_at = :run_ts
  );
  SET rows_updated = (
    SELECT COUNT(*) FROM UTRP_SCHEMA.fact_payment WHERE updated_at = :run_ts AND
inserted_at <> :run_ts
  );

  -- Log run
  INSERT INTO UTRP_SCHEMA.audit_merge_log(object_type, rows_processed, rows_inserted,
rows_updated, run_by, run_at, status)
  VALUES('fact_payment', :rows_processed, :rows_inserted, :rows_updated, :run_by, :run_ts,
'SUCCESS');

  -- Mark payments as processed
  UPDATE UTRP_SCHEMA.payments
     SET processed = TRUE
```

```
   WHERE processed = FALSE;

  RETURN OBJECT_CONSTRUCT('status','SUCCESS',
                         'processed', :rows_processed,
                         'inserted', :rows_inserted,
                         'updated', :rows_updated);

EXCEPTION
  WHEN STATEMENT_ERROR OR EXPRESSION_ERROR THEN
    INSERT INTO UTRP_SCHEMA.audit_merge_log(object_type, rows_processed, rows_inserted,
rows_updated, run_by, run_at, status, message)
    VALUES('fact_payment', :rows_processed, 0, 0, :run_by, :run_ts, 'FAILED', :sqlerrm);

    RETURN OBJECT_CONSTRUCT(
      'status','FAILED',
      'error_message', :sqlerrm,
      'sqlstate', :sqlstate
    );
END;
$$;
```

What does the query actually do?

- Count rows to process - Check how many new/unprocessed invoices are waiting in the staging table.
- Match staging to canonical:
  - If invoice already exists but with changes then update it.
  - If invoice doesn't exist then insert it.
  - If invoice exists and nothing changed then leave it.
- Log results - Insert a row into audit_merge_log with how many rows were processed, inserted, or updated.
- Mark staging rows as processed - So they won't be picked up again in the next run.
- Return a summary - Success/failure + row counts.

RECONCILIATION ENGINE

-- Reconciliation Engine Design

```
CREATE OR REPLACE TABLE UTRP_SCHEMA.recon_invoice_payment (
    recon_id        STRING DEFAULT UUID_STRING(),
    ext_invoice_key STRING,
    customer_id     STRING,
    invoice_amount  NUMBER,
    payment_total   NUMBER,
    difference      NUMBER,
    match_status    STRING,   -- e.g. 'MATCHED', 'PARTIAL', 'OVERPAID', 'UNDERPAID', 'UNPAID'
    run_at          TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP()
);

CREATE OR REPLACE TABLE UTRP_SCHEMA.recon_exceptions (
    exception_id    STRING DEFAULT UUID_STRING(),
    ext_invoice_key STRING,
    customer_id     STRING,
    issue_type      STRING,   -- e.g. 'NO_MATCH', 'PARTIAL', 'OVERPAYMENT',
'MISSING_CUSTOMER'
    issue_details   STRING,
    suggested_action STRING,
    resolved        BOOLEAN DEFAULT FALSE,
    run_at          TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP()
);
```

-- Reconciliation Engine

```
CREATE OR REPLACE PROCEDURE UTRP_SCHEMA.sp_reconcile_invoices_payments(run_by
STRING)
RETURNS VARIANT
LANGUAGE SQL
EXECUTE AS CALLER
AS
$$
DECLARE
  run_id STRING DEFAULT UUID_STRING();   -- unique identifier for this run
  run_ts TIMESTAMP_NTZ := CURRENT_TIMESTAMP();
  rows_processed NUMBER := 0;
  rows_matched NUMBER := 0;
  rows_exceptions NUMBER := 0;
BEGIN
  -- Insert reconciliation results
  INSERT INTO UTRP_SCHEMA.recon_invoice_payment
  (
    ext_invoice_key, customer_id, invoice_amount, payment_total, difference, match_status,
run_at, run_id
  )
  SELECT
      fi.ext_invoice_key,
      fi.customer_id,
      fi.amount AS invoice_amount,
      COALESCE(SUM(fp.amount),0) AS payment_total,
      fi.amount - COALESCE(SUM(fp.amount),0) AS difference,
      CASE
      WHEN COALESCE(SUM(fp.amount),0) = fi.amount THEN 'MATCHED'
      WHEN COALESCE(SUM(fp.amount),0) = 0 THEN 'UNPAID'
      WHEN COALESCE(SUM(fp.amount),0) < fi.amount THEN 'PARTIAL'
      WHEN COALESCE(SUM(fp.amount),0) > fi.amount THEN 'OVERPAID'
      ELSE 'UNKNOWN'
    END AS match_status,
    :run_ts,
    :run_id
  FROM UTRP_SCHEMA.fact_invoice fi
  LEFT JOIN UTRP_SCHEMA.fact_payment fp
       ON fi.customer_id = fp.customer_id
  GROUP BY fi.ext_invoice_key, fi.customer_id, fi.amount;

  -- Count reconciled rows for this run
  SELECT COUNT(*) INTO :rows_processed
  FROM UTRP_SCHEMA.recon_invoice_payment
  WHERE run_id = :run_id;

  SELECT COUNT(*) INTO :rows_matched
  FROM UTRP_SCHEMA.recon_invoice_payment
  WHERE run_id = :run_id
    AND match_status = 'MATCHED';

  -- Insert exceptions for non-matches
  INSERT INTO UTRP_SCHEMA.recon_exceptions
  (
    ext_invoice_key, customer_id, issue_type, issue_details, suggested_action, run_at, run_id
  )
  SELECT
    r.ext_invoice_key,
    r.customer_id,
    CASE r.match_status
      WHEN 'UNPAID' THEN 'NO_PAYMENT'
      WHEN 'PARTIAL' THEN 'PARTIAL_PAYMENT'
      WHEN 'OVERPAID' THEN 'OVERPAYMENT'
      ELSE 'UNKNOWN'
    END AS issue_type,
    'Invoice amount = ' || r.invoice_amount || ', Payments = ' || r.payment_total || ',
Difference = ' || r.difference AS issue_details,
    CASE r.match_status
      WHEN 'UNPAID' THEN 'Investigate missing payment or follow up with customer'
      WHEN 'PARTIAL' THEN 'Review outstanding balance and initiate collection'
      WHEN 'OVERPAID' THEN 'Check for duplicate/refund required'
      ELSE 'Manual review needed'
```

```
   END AS suggested_action,
   :run_ts,
   :run_id
 FROM UTRP_SCHEMA.recon_invoice_payment r
 WHERE r.run_id = :run_id
  AND r.match_status <> 'MATCHED';


 -- Count exceptions
 SELECT COUNT(*) INTO :rows_exceptions
 FROM UTRP_SCHEMA.recon_exceptions
 WHERE run_id = :run_id;

 -- Return proper JSON summary
 RETURN OBJECT_CONSTRUCT(
  'status','SUCCESS',
  'processed', :rows_processed,
  'matched', :rows_matched,
  'exceptions', :rows_exceptions,
  'run_id', :run_id,
  'run_by', :run_by,
  'run_at', :run_ts
 );

EXCEPTION
 WHEN STATEMENT_ERROR OR EXPRESSION_ERROR THEN
  RETURN OBJECT_CONSTRUCT(
   'status','FAILED',
   'error_message', :sqlerrm,
   'sqlstate', :sqlstate,
   'run_by', :run_by,
   'run_at', :run_ts,
   'run_id', :run_id
  );
END;
$$;
```

-- Calling Stored Procedure

```
CALL UTRP_SCHEMA.sp_reconcile_invoices_payments('shijin');
```

-- Adding columns in both the recon tables

```
ALTER TABLE UTRP_SCHEMA.recon_invoice_payment
 ADD COLUMN run_id STRING;
```

```
ALTER TABLE UTRP_SCHEMA.recon_exceptions
  ADD COLUMN run_id STRING;

ALTER TABLE UTRP_SCHEMA.fact_payment
ADD COLUMN invoice_id INT;
```

Summary

- In most real-world finance and IT systems, invoices and payments don't always line up perfectly. Some invoices are fully paid, some are unpaid, some are partially paid, and sometimes customers even overpay.
- Manually tracking these mismatches is time-consuming and error-prone for finance teams.
- To solve this, we built a Reconciliation Engine that automatically compares invoices against payments and classifies the results.

How it works

1. **Join Invoices and Payments**

- We linked invoices from fact_invoice with payments from fact_payment based on customer_id.
- For each invoice, we calculated the total payments made by that customer.

2. **Calculate Differences**

- We compared invoice amount vs total payments.
- The difference tells us whether the invoice is fully settled or has issues.

3. **Classify Match Status**

- If payments = invoice amount → MATCHED
- If no payments → UNPAID
- If payments < invoice amount → PARTIAL
- If payments > invoice amount → OVERPAID

4. **Store Results**

- All reconciliation results are stored in recon_invoice_payment.
- Any problematic cases (UNPAID, PARTIAL, OVERPAID) are logged into recon_exceptions with:
- Issue Type (e.g., No Payment, Partial, Overpayment)

- Issue Details (amounts and differences)
- Suggested Action (e.g., “Follow up with customer”, “Check for duplicate/refund”)

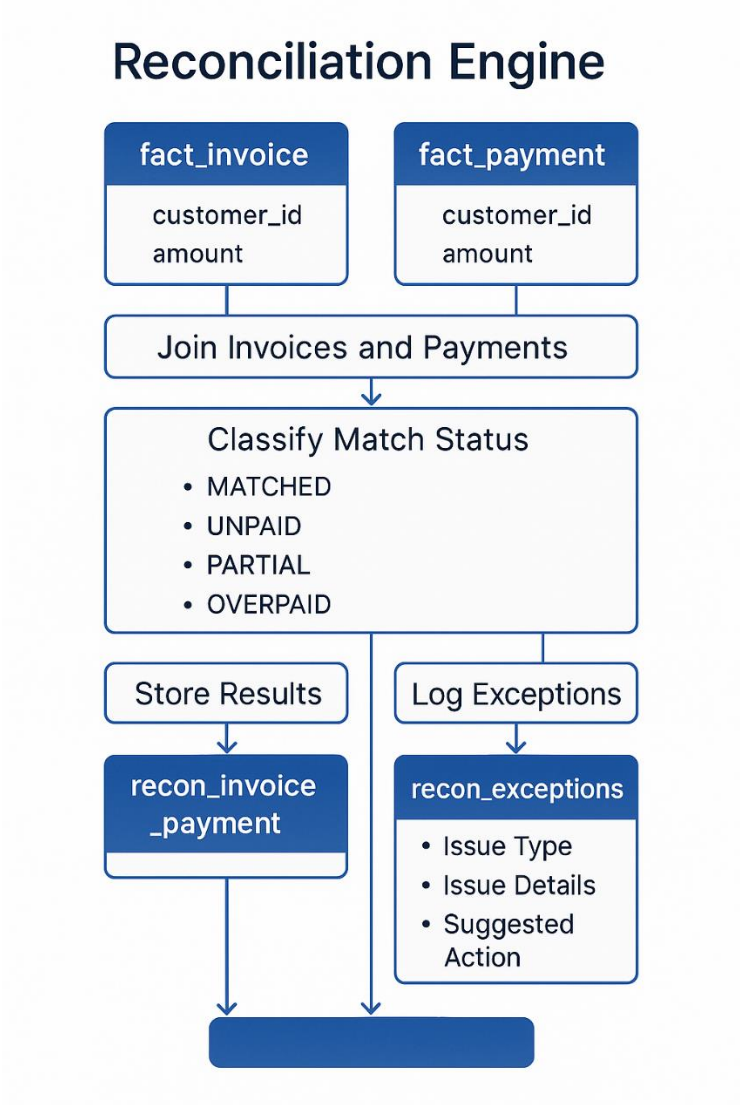
5. **Run Tracking with run_id**

- Each reconciliation run is tagged with a unique run_id.
- This makes it easy to audit and analyze multiple runs.

Why these matters in the industry?

- Finance teams don’t need to manually compare invoice and payment records.
- The engine automatically highlights mismatches and gives clear next steps.
- Exceptions are captured in a structured way, ready for reporting in Power BI.
- Improves accuracy, saves time, and reduces financial risk.

With this, we automated the invoice vs payment matching process. Instead of relying on manual checks, our reconciliation engine now classifies every invoice into Matched, Unpaid, Partial, or Overpaid, and logs exceptions for review.


Reconciliation Engine
fact_invoice
customer_id
amount
fact_payment
customer_id
amount
Join Invoices and Payments
Classify Match Status
• MATCHED
• UNPAID
• PARTIAL
• OVERPAID
Store Results
Log Exceptions
recon_invoice
_payment
recon_exceptions
• Issue Type
• Issue Details
• Suggested Action