TECHNICAL METHODOLOGY REPORT

Marketplace Analytics

Data Cleaning, Validation Logic, and Query Design

Reporting Period	January 2023 – December 2024
Document Version	1.0
Date Prepared	April 2026
Classification	Internal — Data Engineering
Prepared By	Data & Analytics Team

DISTRIBUTION

Data Engineering · Data Quality · Analytics Leads · External Audit

PREPARED BY	REVIEWED BY	APPROVED BY
_____ *Signature / Date*	_____ *Signature / Date*	_____ *Signature / Dat*

1. Purpose and Scope

This report documents the technical decisions taken to transform the raw marketplace dataset into an analysis-ready form. It is the companion to the Executive Memo (Document 1), the Data Quality Audit Report (Document 3), and the Query Documentation Pack (Document 4).

Audience

Data engineers and analysts who will inherit, extend, or audit this work. The level of detail assumes working knowledge of SQL and PostgreSQL, but every non-obvious decision is annotated.

Scope

Element	Detail
Source dataset	7 raw tables: customers (865 rows), sellers (90), products (280), orders (3,015), order_items (6,426), payments (2,262), reviews (817). 13,755 rows total.
Period of analysis	1 January 2023 to 31 December 2024 (24 months).
Output dataset	7 cleaned tables in a separate cleaned schema, plus an embedded amount_validation_status column in cleaned.orders.
Tooling	PostgreSQL 18, pgAdmin 4 for development. VS Code with the PostgreSQL extension was used for review.
Reproducibility	All transformations are deterministic and contained in a single master cleaning script. Re-running against the original raw tables produces identical output.

Table 1. Project scope summary.

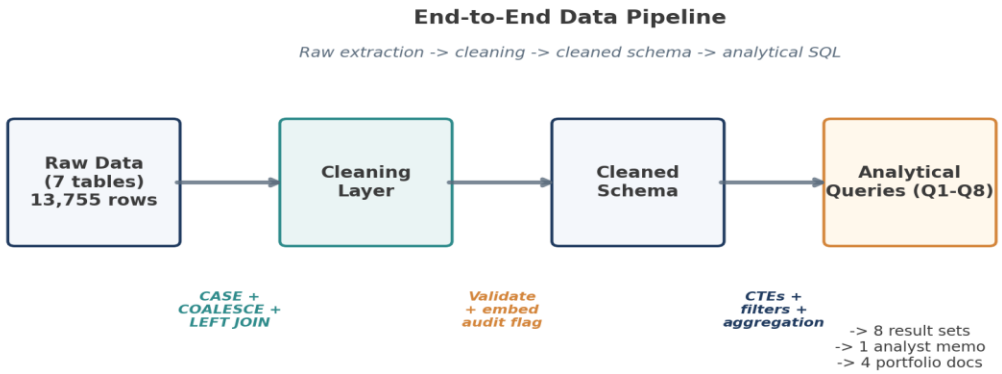

Figure 1. End-to-end pipeline. Raw data flows through a cleaning layer into the cleaned schema, then into the eight analytical queries.

2. Cleaning Philosophy

Three principles governed every cleaning decision.

Principle 1 — Raw tables are immutable

Every transformation produces a new table in the `cleaned` schema[1]. The raw tables are never modified, dropped, or replaced. This means the cleaning logic can be re-run, revised, or rolled back without data loss, and any downstream issue can be traced back to its source by joining cleaned to raw on the primary key.

Principle 2 — Single-pass transformations

Each cleaned table is produced by a single CREATE TABLE AS SELECT statement. Multi-pass UPDATE chains create ordering dependencies and make the script harder to reason about. A single SELECT — even a complex one with CTEs — is easier to validate, test, and modify.

Principle 3 — Audit fields embedded in the data

Where cleaning makes substantive corrections (rather than cosmetic standardisation), a status column is embedded in the cleaned table. An analyst can filter to the corrected records at any time without rerunning the cleaning logic — see the orders table validation_status column in §4.4.

3. Source Schema

The seven raw tables form a star-like schema with orders as the central transactional fact, surrounded by customer, seller, product, and audit dimensions.

[1]Raw tables are preserved in their original form. All transformations produce new tables in a separate schema, ensuring the cleaning process is fully reversible and auditable.

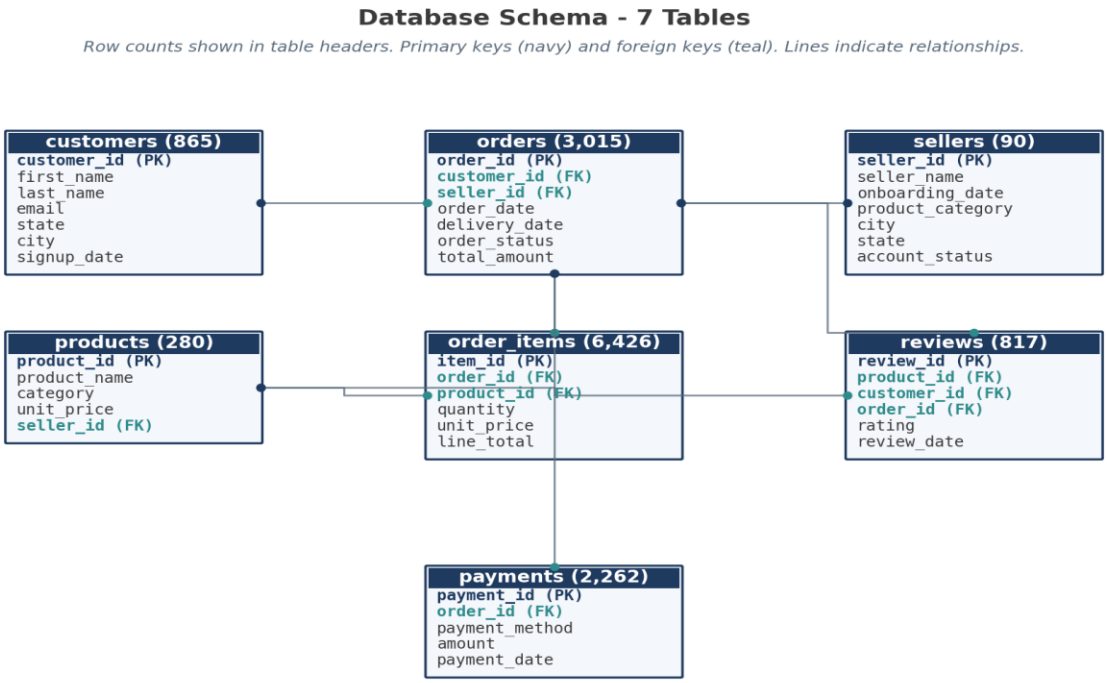

Figure 2. Source schema with row counts. Primary keys shown in navy, foreign keys in teal.

Row volumes

Table	Rows	Notes
customers	865	Sign-up dates span the full period; 16 NULL emails retained.
sellers	90	All active. Distributed across 5 cities.
products	280	40 products per category × 7 categories. 4 with NULL prices.
orders	3,015	All statuses. 1,505 delivered; 1,510 in non-final states.
order_items	6,426	Average ~2.1 items per order.
reviews	817	5 ratings out of valid range; 15 products lack any review.
payments	2,262	1:1 with delivered orders.

Table 2. Source table row counts and notable characteristics.

4. Per-table Cleaning Logic

4.1 sellers — City standardisation

Problem

Manual data entry produced 23 distinct spellings across the 5 cities in customers and sellers tables. Variants observed include 'Port-Harcourt', 'Port Harcourt', 'PortHarcourt', 'PORT HARCOURT', 'Lago s' (internal space), trailing whitespace, and case variations across all five hub cities.

Cleaning rule

Two preprocessing passes followed by canonical mapping, with INITCAP as the safety net for any unmatched value.

```
CREATE TABLE cleaned.sellers AS
SELECT
    seller_id,
    seller_name,
    onboarding_date,
    product_category,
    CASE
        -- 1. Port Harcourt: catches spaces, dashes, and internal typos
        WHEN LOWER(REPLACE(REPLACE(city, '-', ''), ' ', ''))
             LIKE '%portharcourt%' THEN 'Port Harcourt'

        -- 2. Lagos: implicitly fixes 'Lago s' by stripping spaces first
        WHEN LOWER(REPLACE(city, ' ', '')) = 'lagos' THEN 'Lagos'

        -- 3. Other major hubs
        WHEN LOWER(REPLACE(city, ' ', '')) = 'abuja'  THEN 'Abuja'
        WHEN LOWER(REPLACE(city, ' ', '')) = 'ibadan' THEN 'Ibadan'
        WHEN LOWER(REPLACE(city, ' ', '')) = 'kano'   THEN 'Kano'

        -- 4. Default fallback: trim and capitalise
        ELSE INITCAP(TRIM(city))
    END AS city,
    state,
    account_status
FROM sellers;
```

Listing 1. cleaned.sellers — city standardisation logic.

Why this works

The two `REPLACE` calls strip dashes and spaces before comparison, so 'Port-Harcourt', 'Port Harcourt', and 'PortHarcourt' all collapse to 'portharcourt' before the LIKE pattern fires. The Lagos rule strips spaces before equality, catching 'Lago s'. The fallback `INITCAP(TRIM(city))`

ensures any city not in the canonical list still emerges in clean title-case form[2], preserving long-tail cities for future analysis without manual enumeration.

Result

5 clean cities — Lagos (294 customers, 32 sellers), Abuja (181/19), Kano (135/14), Port Harcourt (130/15), Ibadan (125/10). The same logic is applied to both the customers and sellers tables to keep names consistent across joins.

4.2 products — Category standardisation

Problem

28 distinct category variants for 7 intended categories. Issues included casing (ELECTRONICS / electronics), typos (Electronis, Fashon), separator inconsistency ('and' vs '&'), truncated forms (BEAUTY, SPORTS, FOOD, BOOKS), and sub-categories used as categories ('Men', 'Women').

Cleaning rule

CASE with LIKE patterns for broad matching, plus explicit IN lists for typos and sub-category folding. 'Men' and 'Women' are folded into 'Fashion' since they are apparel sub-types without a separate reporting line.

```
CREATE TABLE cleaned.products AS
SELECT
    product_id,
    product_name,
    CASE
        WHEN LOWER(category) LIKE '%electr%' THEN 'Electronics'

        WHEN LOWER(category) IN ('fashon','fashion','men','women')
             OR LOWER(category) LIKE '%fash%' THEN 'Fashion'

        WHEN LOWER(category) LIKE '%beauty%'      THEN 'Beauty & Personal Care'
        WHEN LOWER(category) LIKE '%home%'        THEN 'Home & Garden'
        WHEN LOWER(category) LIKE '%sport%'       THEN 'Sports & Fitness'
        WHEN LOWER(category) LIKE '%food%'        THEN 'Food & Beverages'
        WHEN LOWER(category) LIKE '%book%'        THEN 'Books & Stationery'

        ELSE INITCAP(TRIM(category))
    END AS category,
    unit_price,
    seller_id
FROM products;
```

Listing 2. cleaned.products — category standardisation.

[2]PostgreSQL's INITCAP function capitalises the first letter of each word and lowercases the rest, producing consistent title-case output for any unmatched values.

Why ordering matters

The Fashion branch must come AFTER the Electronics check but BEFORE the generic '%home%' / '%sport%' branches, because 'Men' and 'Women' are explicit IN-list entries that need to match before any pattern-based fallback fires. The fall-through order is: (1) explicit single-word matches, (2) LIKE patterns, (3) INITCAP fallback.

> **PITFALL CAUGHT DURING VERIFICATION**
> An earlier draft used %home%garden% (a more specific pattern) and the master script only handled Electronics and Fashion explicitly. Combined with the INITCAP fallback, this produced 14 categories instead of 7 — variants like 'Home And Garden', 'Sports', and 'Food' fell through to the title-case branch as separate categories. The fix was to broaden the LIKE patterns and let the INITCAP fallback only catch genuinely unknown values.

Result

7 categories, 40 products each: Electronics, Fashion, Beauty & Personal Care, Home & Garden, Sports & Fitness, Food & Beverages, Books & Stationery. Same logic applied to sellers.product_category.

4.3 order_items — Bottom-up imputation

Problem

Two systematic data quality issues: missing unit_price values (recoverable from the master products table for most cases), and missing or incorrect line_total values (recoverable arithmetically when unit_price and quantity are known).

Cleaning rule

LEFT JOIN to products as a fallback for unit_price, then COALESCE for line_total using the recovered price. Fatally flawed records — those with NULL on order_id, product_id, or quantity — are dropped because they cannot meaningfully participate in any aggregation.

```
CREATE TABLE cleaned.order_items AS
SELECT
    oi.item_id,
    oi.order_id,
    oi.product_id,
    oi.quantity,

    -- Impute missing unit_price from the master products table
    COALESCE(oi.unit_price, p.unit_price) AS unit_price,

    -- Recalculate line_total if missing, using the fallback price
    COALESCE(
        oi.line_total,
        (oi.quantity * COALESCE(oi.unit_price, p.unit_price))
    ) AS line_total
```

```
FROM order_items oi
LEFT JOIN products p
    ON oi.product_id = p.product_id

-- Drop fatally flawed records (cannot recover order/product/quantity)
WHERE oi.order_id   IS NOT NULL
  AND oi.product_id IS NOT NULL
  AND oi.quantity   IS NOT NULL;
```

Listing 3. cleaned.order_items — imputation and filtering.

WHY BOTTOM-UP

The bottom-up approach is deliberate: order_items is the ground-truth source for transactional revenue. Cleaning order_items first means the orders table can validate against a reliable foundation. Doing this in reverse would mean trusting potentially incorrect order totals to validate the line items.

4.4 orders — Validation against line items

Problem

Two distinct integrity issues: 131 orders with NULL *total_amount* but valid line items, and 124 orders where the recorded total diverged materially from the sum of its line items.

Cleaning rule

Calculate the ground-truth total from cleaned order_items, then apply a three-way decision:

- If total_amount is NULL → impute from line item sum
- If |total_amount − calculated_total| > ₦10 → replace with calculated_total
- Otherwise → keep the original total_amount

An *amount_validation_status* column is embedded in the output, recording which of the four branches was taken for each order[3].

```
CREATE TABLE cleaned.orders AS
WITH item_sums AS (
    -- Step 1: Calculate ground-truth total from validated items
    SELECT
```

[3]The ₦10 threshold reflects acceptable rounding tolerance. Differences below this floor are treated as floating-point artifacts; differences above it indicate either data corruption or legitimate adjustments (refunds, manual discounts) that the available data cannot distinguish between.

```
        order_id,
        SUM(line_total) AS calculated_total
    FROM cleaned.order_items
    GROUP BY order_id
)
SELECT
    o.order_id,
    o.customer_id,
    o.seller_id,
    o.order_date,
    o.delivery_date,
    o.order_status,

    -- Step 2: Impute or correct total_amount
    CASE
        WHEN o.total_amount IS NULL
             THEN i.calculated_total
        WHEN i.calculated_total IS NOT NULL
             AND ABS(o.total_amount - i.calculated_total) > 10
             THEN i.calculated_total
        ELSE o.total_amount
    END AS total_amount,

    -- Step 3: Embedded audit flag
    CASE
        WHEN o.total_amount IS NULL AND i.calculated_total IS NOT NULL
             THEN 'Imputed from Items'
        WHEN i.calculated_total IS NULL
             THEN 'Warning: No Line Items Found'
        WHEN ABS(o.total_amount - i.calculated_total) > 10
             THEN 'Corrected: > 10 Discrepancy'
        ELSE 'Valid'
    END AS amount_validation_status

FROM orders o
LEFT JOIN item_sums i ON o.order_id = i.order_id;
```

Listing 4. cleaned.orders — bottom-up validation logic.

Validation status distribution

amount_validation_status	Count	Interpretation
Valid	2,760	Source total matched line item sum within ₦10.
Imputed from Items	131	Source total was NULL; replaced with line item sum.
Corrected: > ₦10 Discrepancy	124	Source total diverged from line item sum by more than ₦10.

amount_validation_status	Count	Interpretation
Warning: No Line Items Found	0	Sentinel value — no orders triggered this branch.
Total	**3,015**	**All cleaned orders accounted for.**

Table 3. Distribution of orders by validation status.

4.5 reviews — Range validation

Problem

5 reviews had ratings outside the valid 1–5 range: three with rating = -1 (likely entry error), one with rating = 0, and one with rating = 7. Treating these as valid would skew average rating calculations.

Cleaning rule

```
CREATE TABLE cleaned.reviews AS
SELECT
    review_id,
    product_id,
    customer_id,
    order_id,
    CASE
        WHEN rating BETWEEN 1 AND 5 THEN rating
        ELSE NULL
    END AS rating,
    review_date
FROM reviews;
```

Listing 5. cleaned.reviews — range validation.

Out-of-range values are coerced to NULL rather than excluded — this preserves the review row (which carries useful metadata about review_date and product coverage) while ensuring the rating itself does not contaminate aggregations. Downstream queries filter on `rating IS NOT NULL` to exclude the coerced records.

4.6 payments — Pass-through

The payments table required no transformation. All payment_id, order_id, payment_method, amount, and payment_date values were valid and complete. The cleaned schema preserves the table for consistency and to keep the audit chain in a single place.

4.7 customers — Duplicate verification

Customers were not transformed, but a duplicate-detection check was run during cleaning to confirm no customer appeared twice with different *customer_id* values. The same check was applied to sellers (grouped by name, city, state) and orders (grouped by customer_id, seller_id, order_date, total_amount). All three returned zero rows.

```
-- Check for duplicate users with same identifying details
SELECT first_name, last_name, email, signup_date, COUNT(*)
FROM customers
GROUP BY first_name, last_name, email, signup_date
HAVING COUNT(*) > 1;
-- Result: 0 rows
```

Listing 6. Duplicate detection query (customers; same pattern applied to sellers and orders).

5. Query Design Decisions

Eight analytical queries (Q1–Q8) operate on the cleaned schema. Several design decisions apply across all of them and are worth documenting once.

5.1 Filtering to delivered orders for revenue

All revenue-based queries (Q2, Q4, Q5, Q7, Q8) filter to delivered orders only, on two grounds:

- Revenue is realised at delivery, not at order placement.
- Cancelled, processing, returned, and shipped-but-undelivered orders inflate apparent demand without contributing actual revenue.

The exception is Q1 (state conversion), which uses any order to measure first-purchase *intent* — a customer who placed and cancelled still expressed intent.

5.2 Date filtering with literal bounds

Date ranges use explicit literal bounds rather than EXTRACT-based predicates:

```
-- Used throughout: literal-bound date filtering
WHERE order_date >= '2024-01-01'
  AND order_date <= '2024-12-31'

-- NOT used: EXTRACT-based filtering
WHERE EXTRACT(YEAR FROM order_date) = 2024
```

Listing 7. Date filtering pattern.

The literal-bound form is sargable: PostgreSQL can use a B-tree index on order_date directly. The EXTRACT-based form requires a function evaluation per row, defeating the index. For 3,000 rows the performance difference is immaterial, but the literal form is the production-ready pattern.

5.3 NULL handling

Three NULL-handling rules apply throughout the query suite.

Rule	Application
Filter NULL revenue values	WHERE total_amount IS NOT NULL — applied in Q2, Q4, Q5, Q8.
Filter NULL ratings (post-cleaning coercion)	WHERE rating IS NOT NULL — applied in Q3, Q7, Q8.
Filter NULL delivery dates for fulfilment	WHERE delivery_date IS NOT NULL — applied in Q3 alongside the Delivered status filter.

Table 4. Standard NULL-handling rules.

5.4 CTE-first composition

Every query in this suite uses Common Table Expressions for staging. This is more verbose than nested subqueries but easier to read, debug, and modify. A reviewer can run any CTE in isolation to see exactly what its output looks like before it joins the next stage.

5.5 Window functions for ranking

Q6 uses ROW_NUMBER() OVER (PARTITION BY state ORDER BY transaction_count DESC) to identify the most popular payment method per state. This is preferred over a self-join correlated subquery — it runs in a single pass and the intent is more legible.

5.6 Percentage calculations

Percentage calculations multiply by **`100.0`** (with the decimal point), not `100`. The decimal forces floating-point arithmetic; otherwise PostgreSQL's integer division would truncate fractional values to zero[4].

[4]All percentage calculations multiply by 100.0 (not 100) to force floating-point division in PostgreSQL, avoiding integer truncation.

6. Validation Framework

The cleaning script is accompanied by validation queries that confirm the post-cleaning data meets defined contracts. These run after the cleaning script and produce exception lists rather than pass/fail flags.

6.1 Row count contracts

Cleaned table	Expected vs raw	Tolerance
cleaned.sellers	= raw.sellers	Exact match
cleaned.products	= raw.products	Exact match
cleaned.order_items	≤ raw.order_items	Up to 97 dropped (unrecoverable price)
cleaned.orders	= raw.orders	Exact match
cleaned.reviews	= raw.reviews	Exact match (5 ratings coerced to NULL)
cleaned.payments	= raw.payments	Exact match
cleaned.customers	= raw.customers	Exact match

Table 5. Row count contracts.

6.2 Domain integrity checks

```
-- Rating range
SELECT COUNT(*) FROM cleaned.reviews
WHERE rating IS NOT NULL AND (rating < 1 OR rating > 5);
-- Expected: 0

-- Total amount validation
SELECT amount_validation_status, COUNT(*)
FROM cleaned.orders
GROUP BY amount_validation_status;
-- Expected: 'Valid' + 'Imputed from Items' + 'Corrected: > 10 Discrepancy'
--           with no 'Warning: No Line Items Found'

-- Category enumeration
SELECT DISTINCT category FROM cleaned.products;
-- Expected: 7 canonical values

-- Foreign key referential integrity
SELECT COUNT(*) FROM cleaned.order_items oi
LEFT JOIN cleaned.orders o ON oi.order_id = o.order_id
WHERE o.order_id IS NULL;
-- Expected: 0
```

Listing 8. Domain integrity validation queries.

6.3 Spot-check sampling

After bulk validation, ten randomly-sampled orders from each validation_status bucket are joined back to their raw counterparts and inspected manually. This catches subtle issues that pass type and range checks but reflect cleaning logic errors — for example, an imputed price that is plausible in isolation but inconsistent with the product's other order_items.

7. Reproducibility

7.1 Execution order

Cleaning steps must run in this order due to dependencies between tables:

#	Step	Why this order
1	cleaned.sellers	No dependencies.
2	cleaned.products	No dependencies.
3	cleaned.order_items	Joins products to recover unit_price.
4	cleaned.orders	Joins cleaned.order_items to compute calculated_total.
5	cleaned.reviews	No dependencies, but conventionally placed after orders.
6	cleaned.payments	Pass-through.
7	Validation suite	After all cleaned tables exist.

Table 6. Cleaning execution sequence.

7.2 Idempotency

The script uses `CREATE TABLE` (without IF NOT EXISTS) by design. Re-running against an existing cleaned schema raises an error rather than silently appending. To re-run, drop the cleaned schema first:

```
DROP SCHEMA cleaned CASCADE;
CREATE SCHEMA cleaned;
```

Listing 9. Schema reset prior to re-running the cleaning script.

7.3 Submission dump

For grading purposes, the cleaned schema is replicated into the public schema of a fresh database and exported with pg_dump in plain SQL format. This produces a self-contained dump file with CREATE TABLE statements and full row data, with no dependency on the custom cleaned schema.

```
# Generate the submission dump
pg_dump -U postgres -d marketplace_cleaned_submit \
        -F p --no-owner --no-privileges \
        -f cleaned_dump.sql

# Verify the dump
grep "CREATE TABLE" cleaned_dump.sql | wc -l   # expect 7
```

```
grep "cleaned\." cleaned_dump.sql | wc -l     # expect 0
```

Listing 10. Reproducible dump generation and verification.

8. Open Questions and Future Work

Three areas were deliberately left for future iterations.

8.1 City fuzzy matching

The current city standardisation handles spaces, dashes, and case variation but not arbitrary misspellings. A Levenshtein-distance threshold against the canonical list would extend coverage. Deferred: marginal coverage gain at the cost of false-positive risk.

8.2 Recovery of NULL prices

Four products have NULL unit_price in both products and order_items. If the source system retains historical pricing data (e.g. in a separate price-history table), a one-time backfill could recover these records. Deferred: requires access to additional source systems.

8.3 Rating semantics for coerced NULLs

Five reviews had out-of-range ratings, coerced to NULL. The original values (-1, 0, 7) may carry semantic meaning — for example, -1 could indicate an unfilled mandatory field rather than a negative rating. Deferred: requires conversation with the upstream team.

Document Control

Version	Date	Change Summary	Author
0.1	2026-03-08	Initial cleaning script and notes	Data Engineering
0.5	2026-03-22	Added query design section	Data Engineering
0.9	2026-04-02	Validation framework added; reviewed by Audit	Data Engineering
1.0	2026-04-15	Approved as final reference document	Data Engineering

Table 7. Revision history.