PR 2

Database Design

Alejandro Pérez Bueno

Jan 16, 2026

Table of Contents

Chapter 1

Data Loading

1.1 Exercise 1a

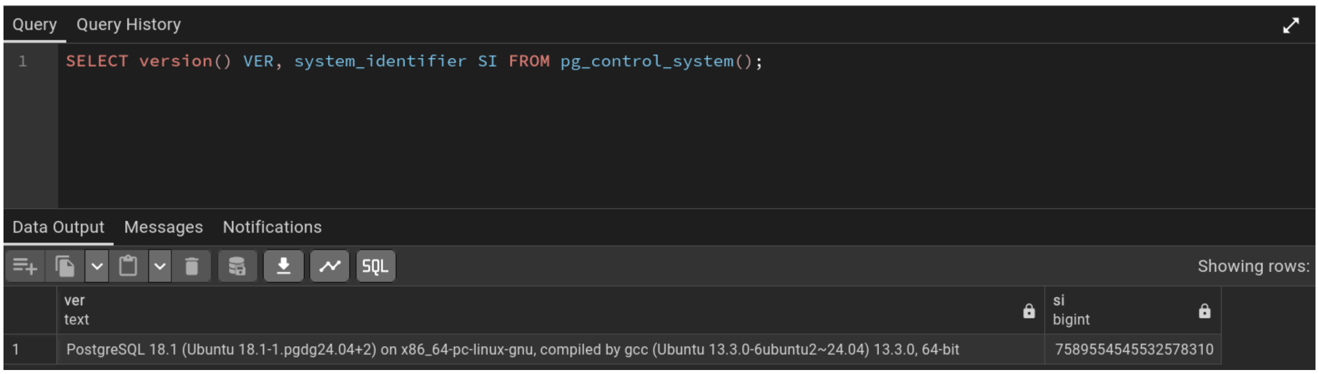


Figure 1.1: Version and System Identifier

1.2 Exercise 1b

Warning

We are asked to create a schema named `S_Q254`, but everywhere else in the subject the schema is actually called `S_Q25`. I created the `S_Q25` schema instead, as otherwise I would get unexpected empty queries and errors.

1.2.1 Missing attribute in Invoice data

The data definition and the UML diagram contains 4 attributes, but the `invoice.csv` file contains 5. Its second column refers to an `incidentID`, which as mentioned in the subject, is what incidents are related to. This field should be a foreign key to the Incident class.

1.2.2 Missing attribute in InvoiceLine data

As before, there is a mismatch between the table of the attributes and what is actually shown in the `invoiceline.csv` file. The first column in it is missing in the table. he diagram shows a `1` to `1..*` relationship between `Invoice` and `InvoiceLine`. It even includes the note "InvoiceLine is weak respect Invoice". This means an `InvoiceLine` cannot exist without an Invoice. Therefore, we must add an `invoiceCode` column as a foreign key.

Furthermore, since the entity is weak, the primary key of `InvoiceLine` should not only be the `Line Number`, but rather a composition of `(invoiceCode, lineNumber)`

1.2.3 SQL table creation

1.2.3.1 Invoice table

```
CREATE TABLE invoice (
    code VARCHAR(50) NOT NULL,
    incident_id VARCHAR(50) NOT NULL,
    invoiceDate DATE NOT NULL,
    amount NUMERIC(12,2) NOT NULL,
    amountWithVAT NUMERIC(12,2) NOT NULL,

    CONSTRAINT pk_invoice PRIMARY KEY (code),
    CONSTRAINT fk_invoice_incident FOREIGN KEY (incident_id)
        REFERENCES incident(code)
        ON UPDATE CASCADE
        ON DELETE RESTRICT
);
```

1.2.3.2 InvoiceLine table

```
CREATE TABLE invoiceline (
    invoice_code VARCHAR(50) NOT NULL,
    lineNumber INTEGER NOT NULL,
    concept VARCHAR(200) NOT NULL,
    amount NUMERIC(12,2) NOT NULL,
    amountWithVAT NUMERIC(12,2) NOT NULL,

    CONSTRAINT pk_invoiceline PRIMARY KEY (invoice_code, lineNumber),
    CONSTRAINT fk_invoiceline_invoice FOREIGN KEY (invoice_code)
        REFERENCES invoice (code)
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
```

1.3 Exercise 1c

Tip

In order to run the `COPY` command from the editor, I first had to copy the CSV files to a location the `postgres` user had access to. Thus, I copied them to `/tmp` and changed their permissions and owner as follows:

```
cp invoice.csv invoiceline.csv /tmp/
chmod 644 /tmp/invoice.csv
chmod 644 /tmp/invoiceline.csv
sudo chown postgres:postgres /tmp/invoice.csv
sudo chown postgres:postgres /tmp/invoiceline.csv
```

1.3.1 Invoice table

```
COPY invoice (code, incident_id, invoiceDate, amount, amountWithVAT)
FROM '/tmp/invoice.csv'
WITH (
    FORMAT CSV,
    DELIMITER ',',
    ENCODING 'UTF8'
);
```

1.3.2 InvoiceLine table

```
COPY invoiceline (invoice_code, lineNumber, concept, amount, amountWithVAT)
FROM '/tmp/invoiceline.csv'
WITH (
    FORMAT CSV,
    DELIMITER ',',
    ENCODING 'UTF8'
);
```

Warning

I initially had issues copying rows to the `invoiceline` table. After some research, I found it had to do with encoding between Linux and Windows. The CSV files were likely created from windows, and when importing them to Linux some special characters appeared in the code, making the invoice codes to mismatch and thus fail with errors similar to the following:

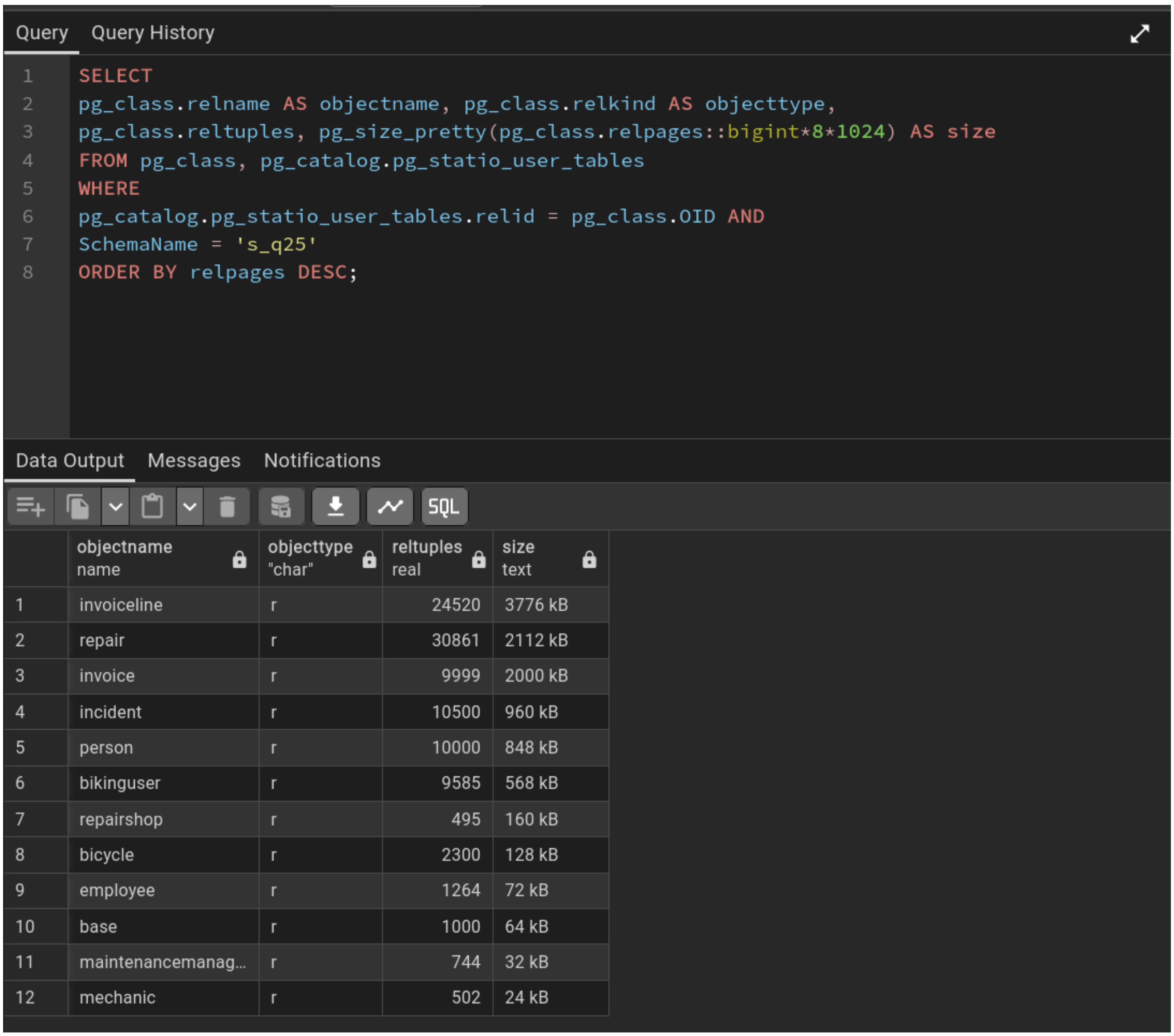
```
ERROR:  insert or update on table "invoiceline" violates foreign key constraint
↪ "fk_invoiceline_invoice"
Key (invoice_code)=(INV0467) is not present in table "invoice".
```

I trimmed special characters to fix the issue with the following commands:

```
UPDATE invoice SET code = TRIM(both ' \r\n\t' from code);
UPDATE invoice SET code = regexp_replace(code, '[^a-zA-Z0-9]', '', 'g');
```

After this all rows were properly copied to the `invoiceline` table.

1.4 Exercise 1d



Query Query History

```
SELECT
pg_class.relname AS objectname, pg_class.relkind AS objecttype,
pg_class.reltuples, pg_size_pretty(pg_class.relpages::bigint*8*1024) AS size
FROM pg_class, pg_catalog.pg_statio_user_tables
WHERE
pg_catalog.pg_statio_user_tables.relid = pg_class.OID AND
SchemaName = 's_q25'
ORDER BY relpages DESC;
```

Data Output Messages Notifications

	objectname name	objecttype "char"	reltuples real	size text
1	invoiceline	r	24520	3776 kB
2	repair	r	30861	2112 kB
3	invoice	r	9999	2000 kB
4	incident	r	10500	960 kB
5	person	r	10000	848 kB
6	bikinguser	r	9585	568 kB
7	repairshop	r	495	160 kB
8	bicycle	r	2300	128 kB
9	employee	r	1264	72 kB
10	base	r	1000	64 kB
11	maintenancemanag...	r	744	32 kB
12	mechanic	r	502	24 kB

Figure 1.2: Summary

1.5 Exercise 1e

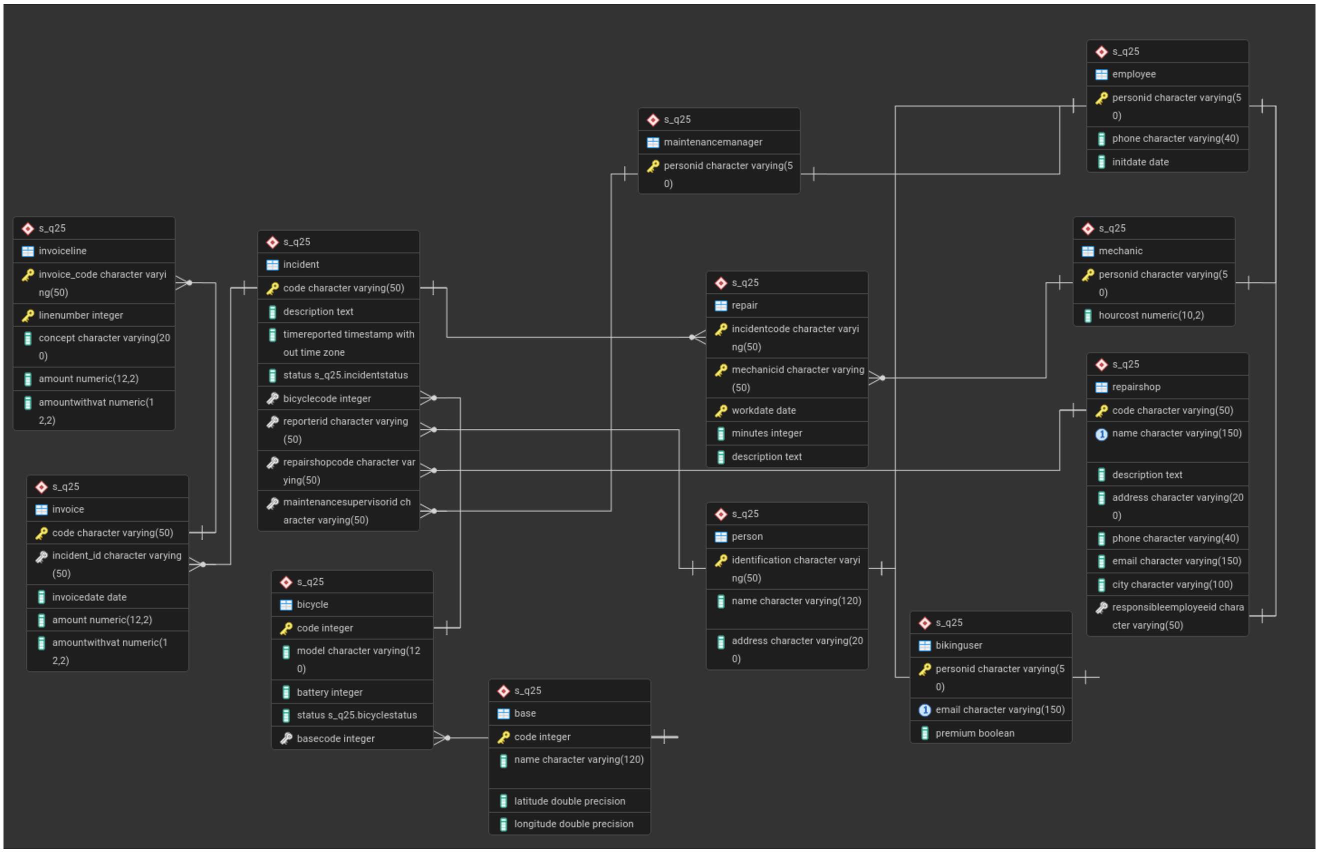


Figure 1.3: ERD Database Diagram

Chapter 2

SQL

2.1 Exercise 2a

2.1.1 Executed SQL command

```sql
SELECT T2.name AS "Name",
       T1.phone AS "Phone"
FROM employee AS T1
JOIN person AS T2 ON T1.personid = T2.identification
WHERE EXTRACT(YEAR FROM T1.initdate) = 2021
  AND EXTRACT(MONTH FROM T1.initdate) = 1;
```

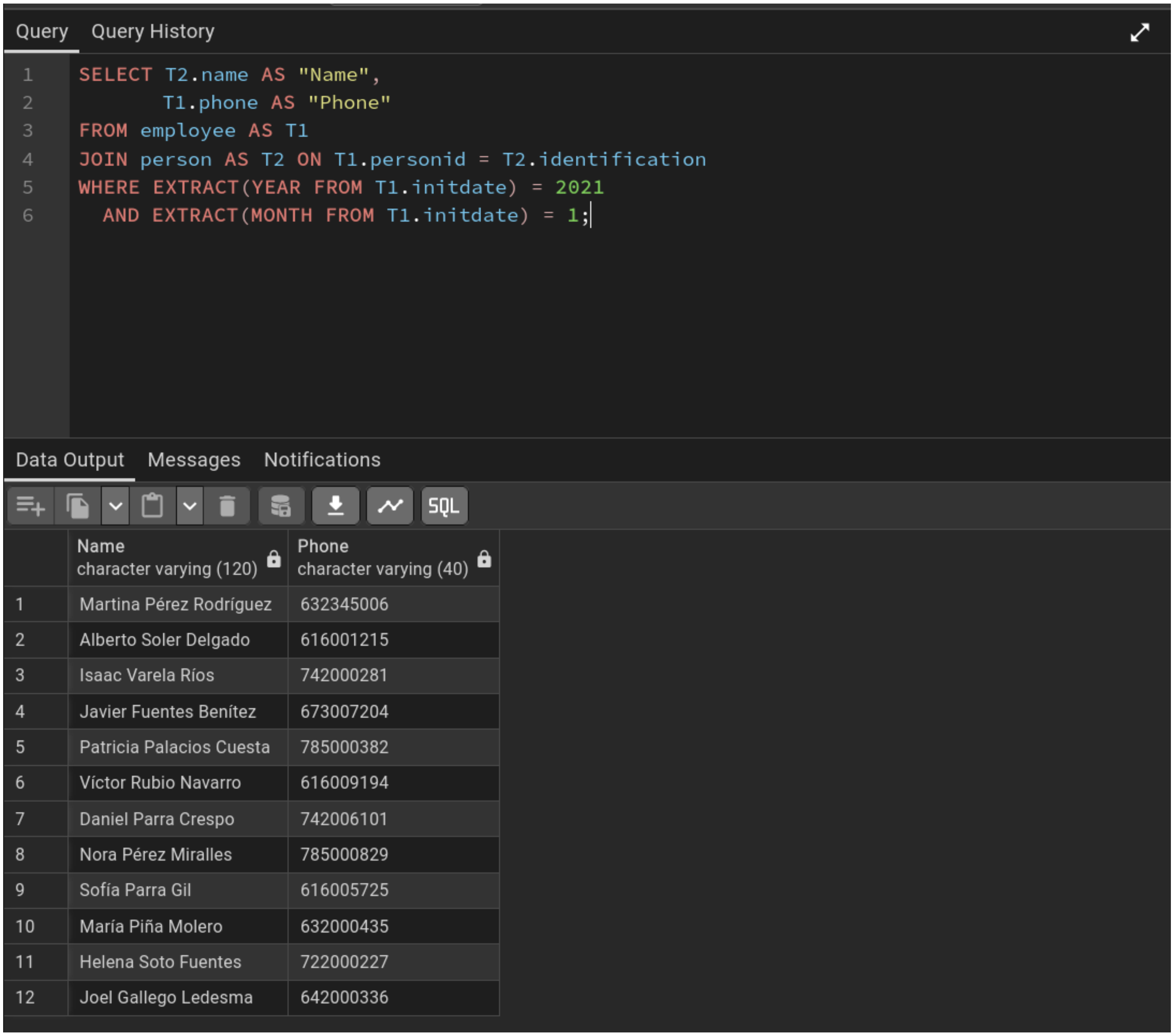
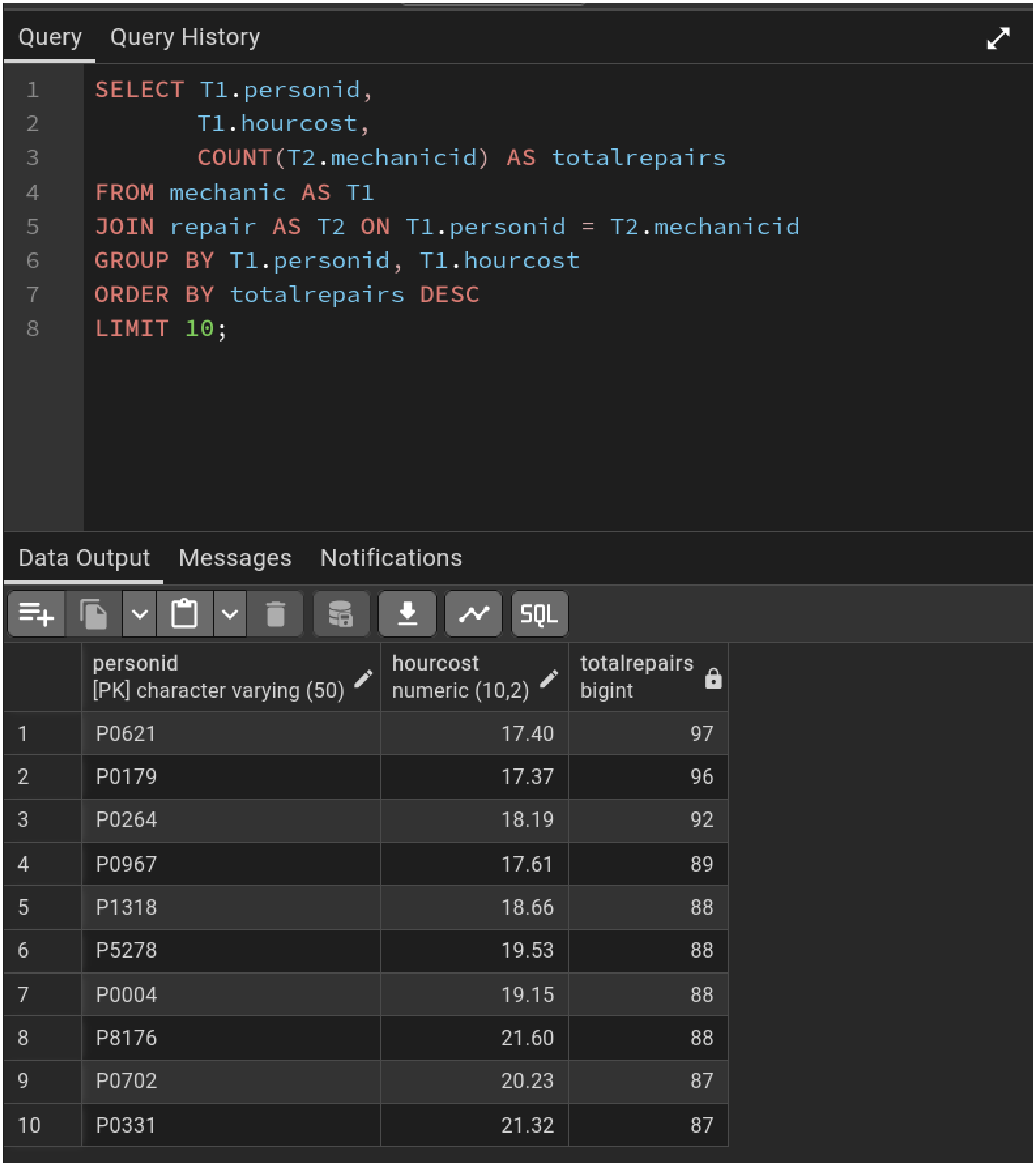


Figure 2.1: Screenshot of the result of the SQL command

2.2 Exercise 2b

2.2.1 Executed SQL command

```
SELECT T1.personid,
       T1.hourcost,
       COUNT(T2.mechanicid) AS totalrepairs
FROM mechanic AS T1
JOIN repair AS T2 ON T1.personid = T2.mechanicid
GROUP BY T1.personid, T1.hourcost
ORDER BY totalrepairs DESC
LIMIT 10;
```




```sql
SELECT T1.personid,
       T1.hourcost,
       COUNT(T2.mechanicid) AS totalrepairs
FROM mechanic AS T1
JOIN repair AS T2 ON T1.personid = T2.mechanicid
GROUP BY T1.personid, T1.hourcost
ORDER BY totalrepairs DESC
LIMIT 10;
```

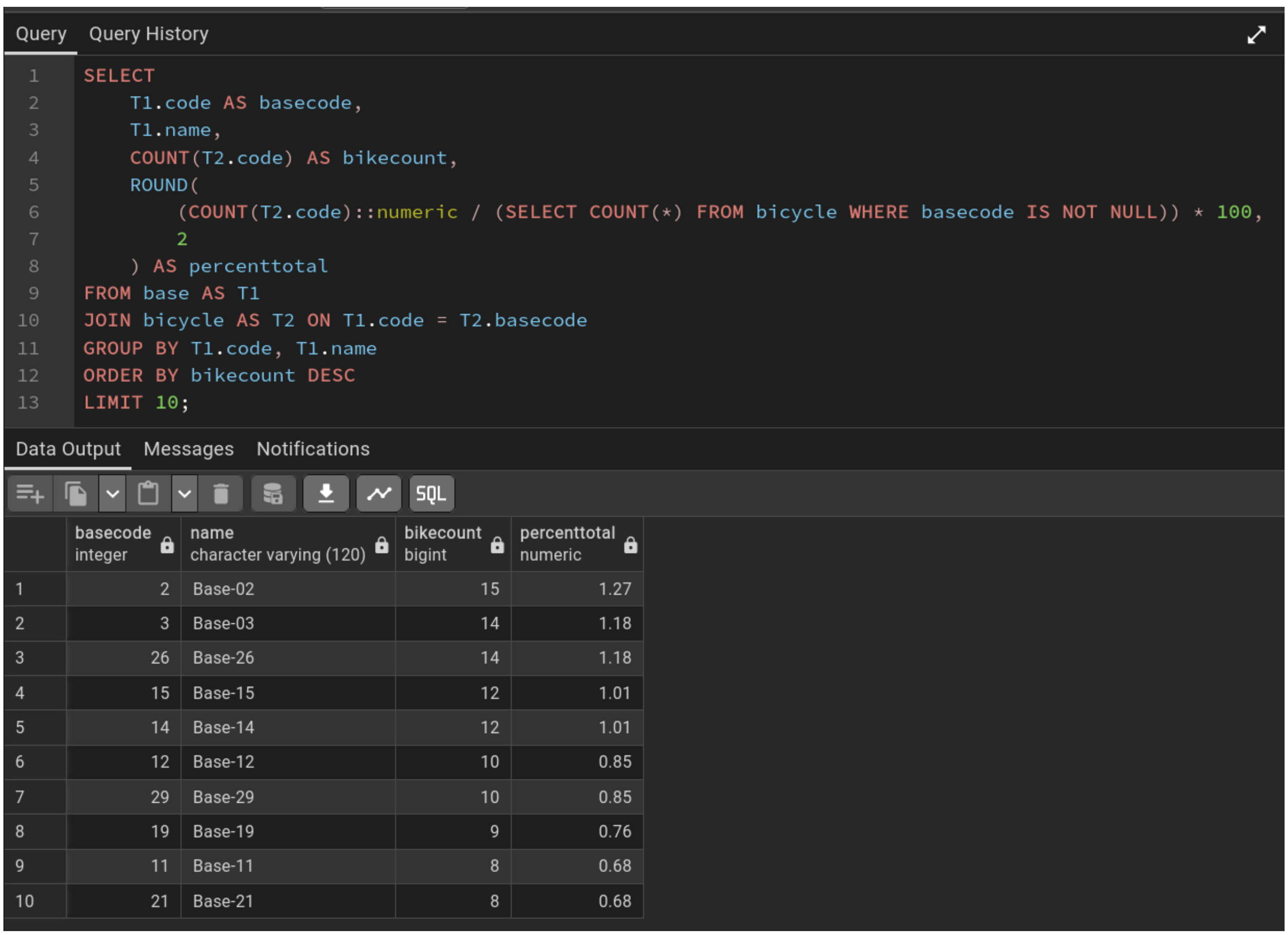
	personid [PK] character varying (50)	hourcost numeric (10,2)	totalrepairs bigint
1	P0621	17.40	97
2	P0179	17.37	96
3	P0264	18.19	92
4	P0967	17.61	89
5	P1318	18.66	88
6	P5278	19.53	88
7	P0004	19.15	88
8	P8176	21.60	88
9	P0702	20.23	87
10	P0331	21.32	87

Figure 2.2: Screenshot of the result of the SQL command

2.3 Exercise 2c

2.3.1 Executed SQL command

```sql
SELECT
    T1.code AS basecode,
    T1.name,
    COUNT(T2.code) AS bikecount,
    ROUND(
        (COUNT(T2.code)::numeric / (SELECT COUNT(*) FROM bicycle WHERE basecode IS NOT NULL)) * 100,
        2
    ) AS percenttotal
FROM base AS T1
JOIN bicycle AS T2 ON T1.code = T2.basecode
GROUP BY T1.code, T1.name
ORDER BY bikecount DESC
LIMIT 10;
```



	basecode integer	name character varying (120)	bikecount bigint	percenttotal numeric
1	2	Base-02	15	1.27
2	3	Base-03	14	1.18
3	26	Base-26	14	1.18
4	15	Base-15	12	1.01
5	14	Base-14	12	1.01
6	12	Base-12	10	0.85
7	29	Base-29	10	0.85
8	19	Base-19	9	0.76
9	11	Base-11	8	0.68
10	21	Base-21	8	0.68

Figure 2.3: Screenshot of the result of the SQL command

2.4 Exercise 2d

2.4.1 Executed SQL command

```
SELECT CAST(EXTRACT(YEAR FROM invoiceDate) AS INTEGER) AS "year",
       CAST(EXTRACT(MONTH FROM invoiceDate) AS INTEGER) AS "month",
       SUM(amountWithVAT) AS "totalAmount"
FROM invoice
WHERE EXTRACT(YEAR FROM invoiceDate) = 2025
GROUP BY EXTRACT(YEAR FROM invoiceDate), EXTRACT(MONTH FROM invoiceDate)
ORDER BY "month";
```

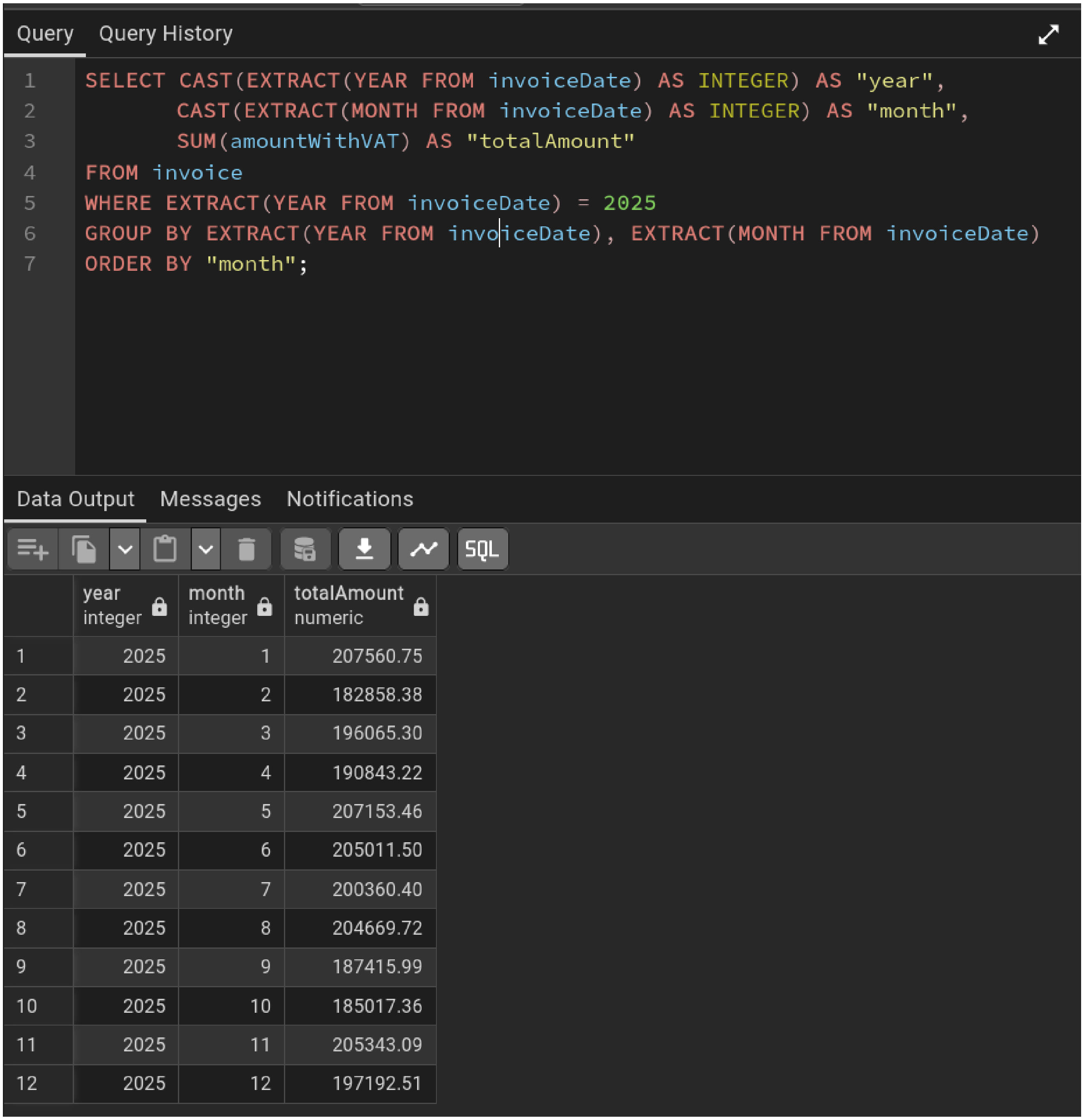


Figure 2.4: Screenshot of the result of the SQL command

Chapter 3

DDL

3.1 Exercise 3a

3.1.1 Default 'pending' value for 'status'

```sql
ALTER TABLE incident
ALTER COLUMN status SET DEFAULT 'Pending';
```

3.1.2 Add new row

```sql
INSERT INTO incident (code, description, timeReported, bicycleCode, reporterId)
VALUES ('I11000', 'Lubrication', NOW(), 41, 'P0136');
```

3.1.3 Query table to show status as 'pending' by default

```sql
SELECT code, description, status, timeReported, bicycleCode, reporterId
FROM incident
WHERE code = 'I11000';
```

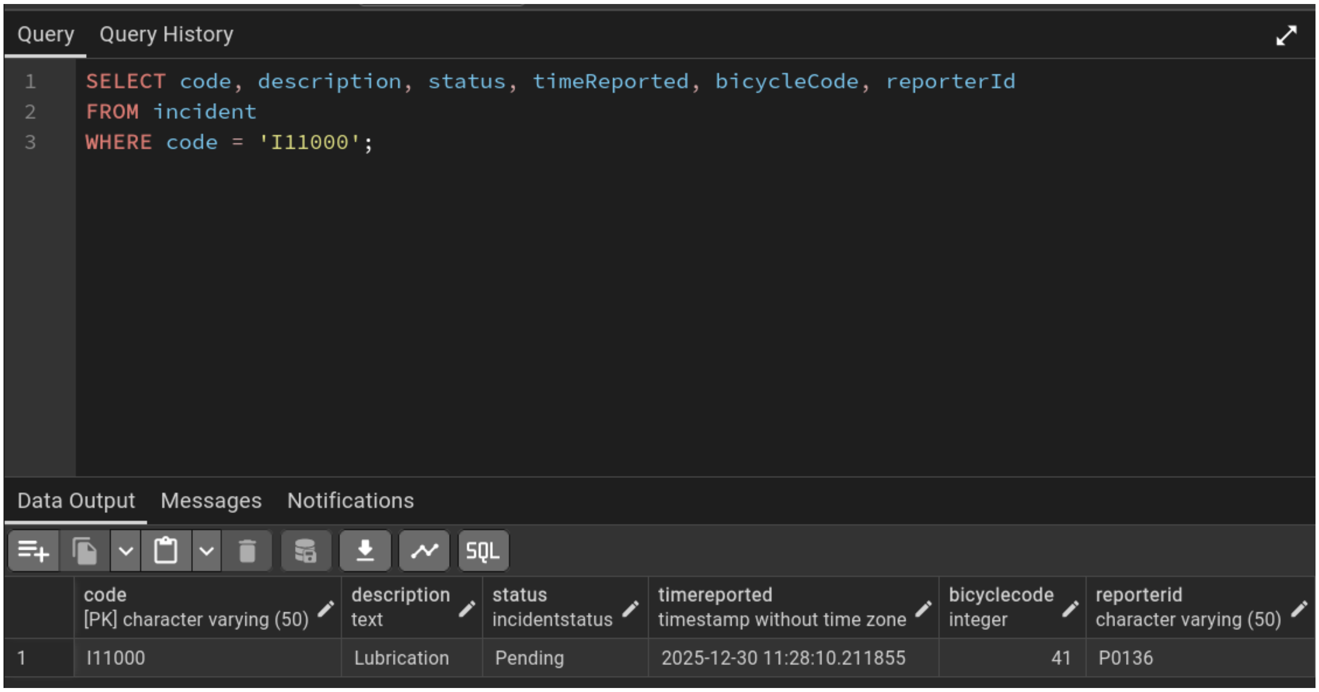


Figure 3.1: Query result

3.2 Exercise 3b

3.2.1 Make 'battery' for 'bicycle' to use 2 decimals instead of int

```
ALTER TABLE bicycle
ALTER COLUMN battery TYPE NUMERIC(5,2);
```

3.2.2 Add new row with the 'battery' as a float

> ⚠ Warning
>
> The subject says the code should be 301, but that one already exists. I created one with the code **2**301, since that is the next available code. It was probably a typo.

```
INSERT INTO bicycle (code, model, battery, status, basecode)
VALUES (2301, 'Model-01', 75.50, 'Available', 4);
```

3.2.3 Query table to show 'battery' as float

```
SELECT code, model, battery, status, basecode
FROM bicycle
WHERE code = 2301;
```

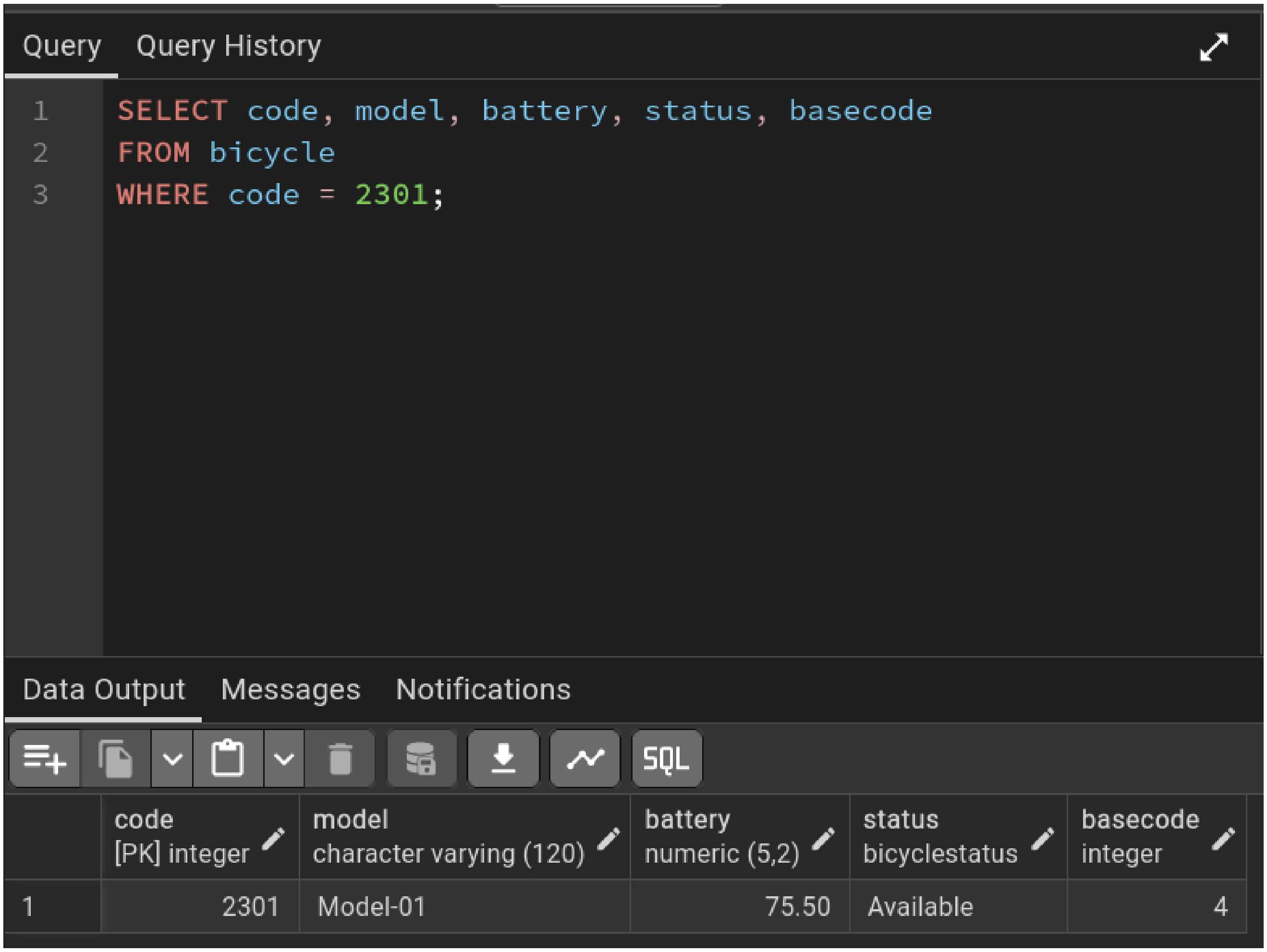


Figure 3.2: Query result

3.3 Exercise 3c

⚠ Warning

What is requested here was already done as part of Section 1.2.3.
However, this would be the way to insert the constraint:

```
ALTER TABLE invoiceline
ADD CONSTRAINT fk_invoiceline_invoice
    FOREIGN KEY (invoice_code)
    REFERENCES invoice (code)
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

3.4 Exercise 3d

Adding the `UNIQUE` constraint to a column prevents duplicates in its table.

3.4.1 Advantages

1. Reliable data rules: the database itself enforces uniqueness instead of relying on the app to do it. That guards against race conditions when two requests try to insert the same email or username at the same time and ensures duplicates are always rejected.
2. Faster lookups on the constrained column: most relational databases implement UNIQUE with a unique index. Tasks like finding a user by email can use that index instead of scanning the whole table, so reads on that field are much quicker.

3.4.2 Disadvantages

1. Extra cost on writes: Every `INSERT` or `UPDATE` that affects the unique column must check and maintain the index. That extra work could add latency, which can be a problem in some systems.
2. Less flexibility if requirements change: declaring a value unique could be a rigid rule. If the business later allows duplicates dropping or changing the constraint on a large table can require index rebuilds, which may affect the runtime of the application relying on the DB.

Chapter 4

Indexes and Views

4.1 Exercise 4a

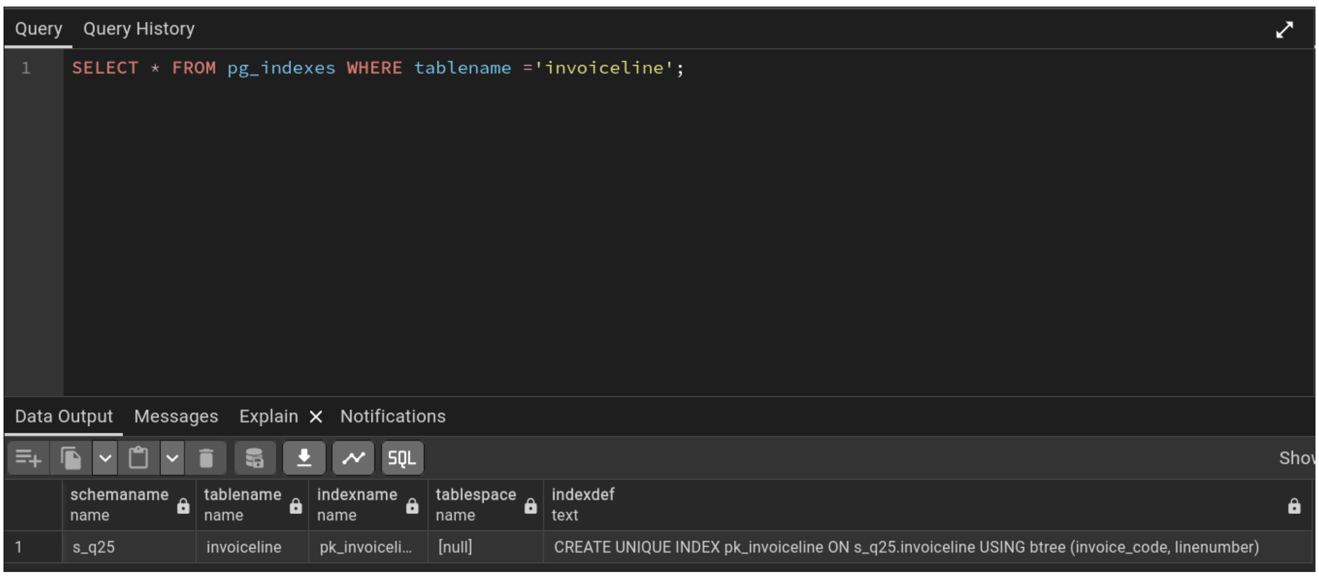


Figure 4.1: Query result

In `pg_indexes`, the `indexdef` column stores the exact `CREATE INDEX` statement used to build each index.

The command is

```
CREATE UNIQUE INDEX pk_invoiceline ON invoiceline USING btree (invoice_code, linenumber)
```

From this we can tell:

- It is UNIQUE, so the pair (invoice_code, linenumber) can't repeat. In practice, this is the index backing the table's primary key
- The index name is `pk_invoiceline`.

- It uses the `btree` (Balanced Tree) method, PostgreSQL's default, which works well for equality lookups and range queries.
- The key is composite: `invoice_code` + `linenumber`. A line number may repeat across invoices, but not within the same invoice, showing that an invoiceline is identified by its parent invoice together with the line number.

4.2 Exercise 4b

4.2.1 Execution plan

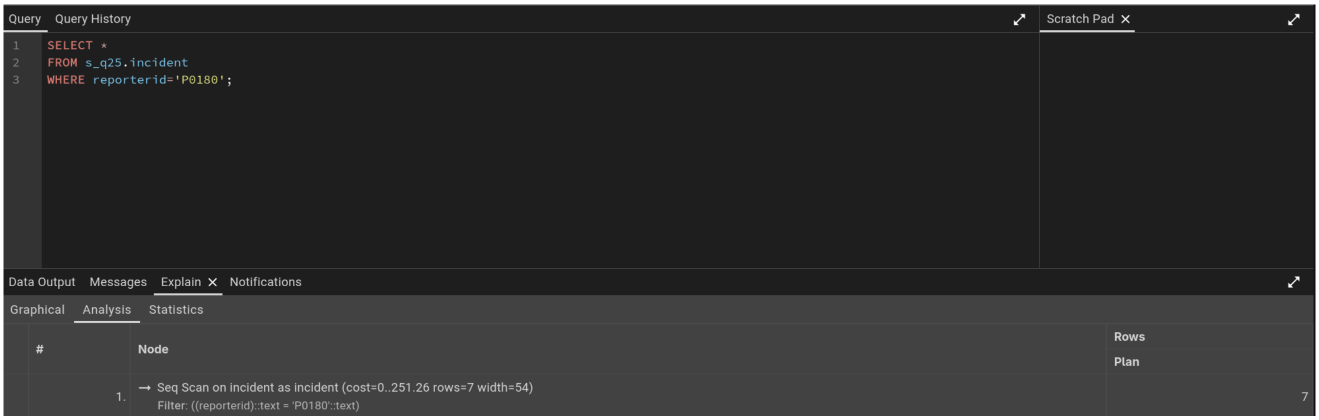


Figure 4.2: Query execution plan

4.2.2 Index creation

```sql
CREATE INDEX idx_incident_reporterid ON incident (reporterid);
```

> Tip
>
> Re-run analyze:
>
> ```sql
> ANALYZE incident;
> ```

4.2.3 New execution plan

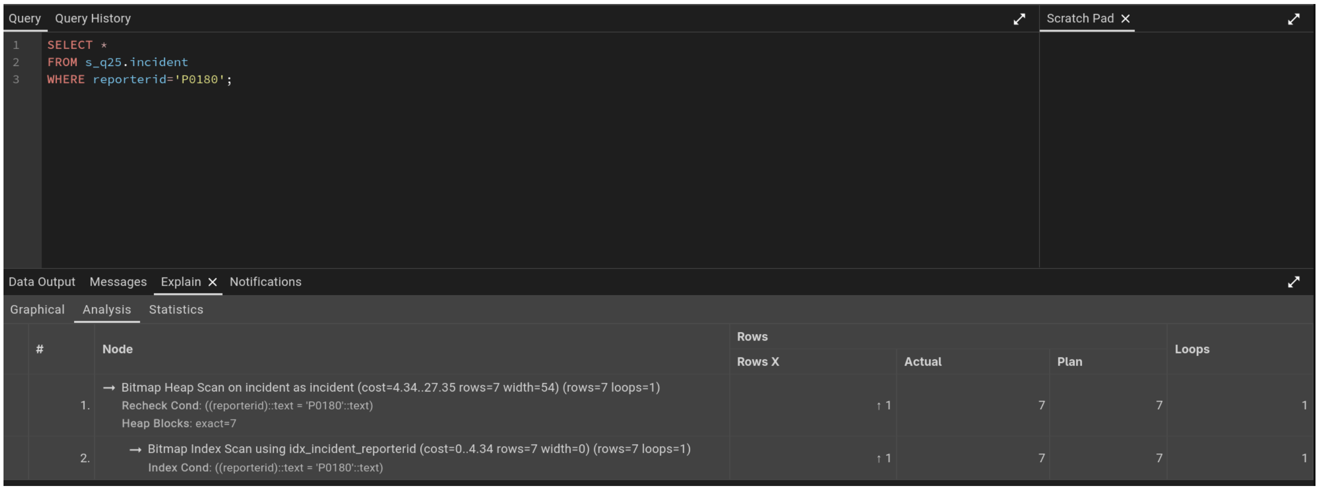


Figure 4.3: Query execution plan after index

4.2.4 Performance analysis

Creating the index clearly sped up the target query. Before, the planner chose a sequential scan with an estimated total cost of 251.26 because every row had to be inspected. Afterward, it used a bitmap index scan with an estimated total cost of 27.35, which is roughly a **90% reduction**.

Regarding the writes, the index in question is on the incident table (`idx_incident_reporterid` on `incident`). Index maintenance happens only when rows in that same table change. Inserts, updates, or deletes on the `Property` table do not touch this index and therefore suffer no overhead from it.

Therefore, keeping this index is the right decision to make. It meaningfully improves read performance for the query on incident, and it has no negative effect on write activity in `Property`.

4.3 Question 4c

4.3.1 Execution plan

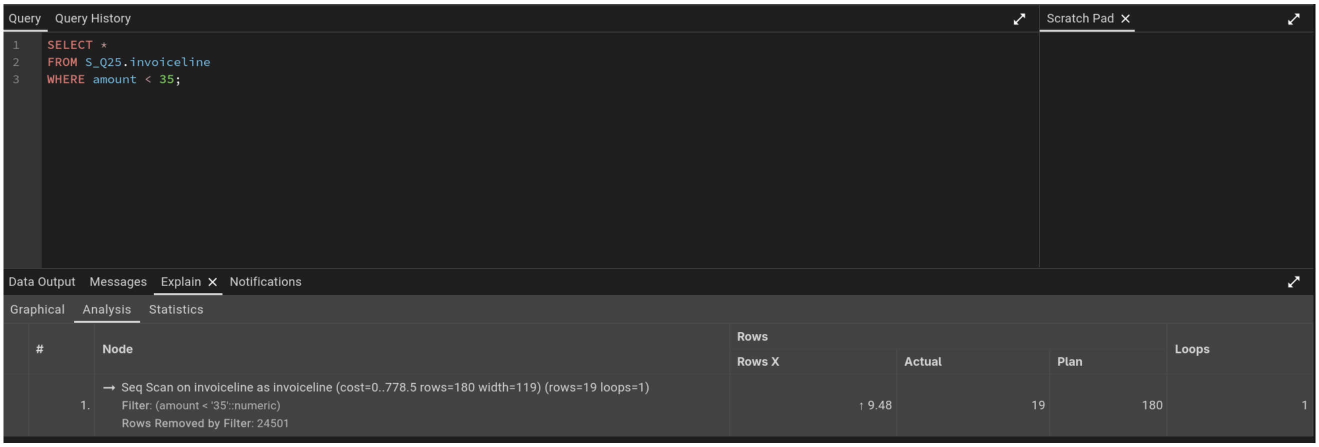


Figure 4.4: Query execution plan

4.3.2 Index choice

We should use a B-tree index. It's standard in PostgreSQL and shines for range and comparison operators:

- Range lookups: `amount < 35` lets the engine seek to the key near 35 and then scan in order, which is precisely how a B-tree is laid out.
- Numeric column: `amount` is numeric and B-trees are built for ordered types (numbers, timestamps, text) and operators like `<`, `<=`, `=`, `>=`, `>`.
- Discarded indexes:
  - Hash: supports equality only, so it is of no use for ranges.
  - GiST/GIN: aimed at things such as full-text, arrays, or geometry, not plain numeric comparisons.

4.3.3 Index creation

```
CREATE INDEX idx_invoiceline_amount
ON invoiceline USING btree (amount);

ANALYZE invoiceline;
```

4.3.4 New execution plan

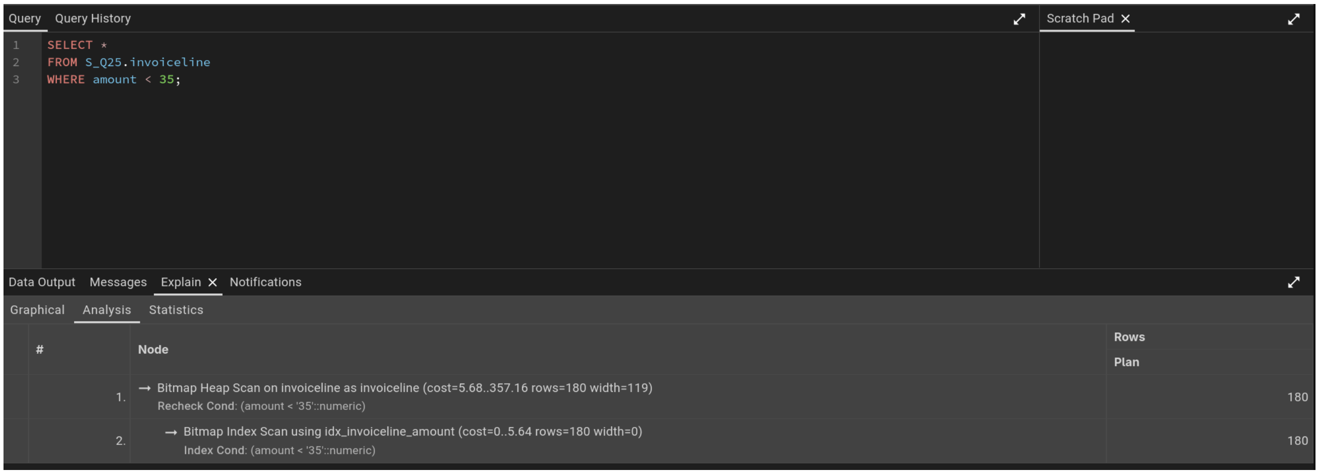


Figure 4.5: Query execution plan after index

4.3.5 Performance analysis

4.3.5.1 Before indexing (sequential scan)

- Estimated cost: 778.5
- The planner chose a full table scan, checking every row for amount < 35.
- Filtered out: 24,501 rows to keep 19 actual matches.

4.3.5.2 After indexing

- New estimated cost: 357.16
- The B-tree index was used to find the matching row pointers first (Bitmap Index Scan) and then only those pages were fetched.

4.3.5.3 Verdict

My recommendation is therefore to keep the B-tree the index. Range filters like `amount < 35` are a great fit for this index type, the reduction is well over 50% and the small overhead on inserts is a fair trade for much faster reads on this query pattern.

4.4 Question 4d

4.4.1 View SQL create code

```
CREATE OR REPLACE VIEW VW_IncidentsDetail AS
SELECT
    i.code AS "Incident Code",
    i.timereported AS "Reported Date/Time",
    i.description AS "Description",
```

```sql
    i.repairshopcode AS "Assigned Workshop",
    i.maintenancesupervisorid AS "Assigned Supervisor",
    b.model AS "Bicycle Model",
    i.status AS "Status"
FROM
    incident i
JOIN
    bicycle b ON i.bicyclecode = b.code;
```

4.4.2 View SQL query

```sql
SELECT
    "Incident Code",
    "Reported Date/Time",
    "Bicycle Model"
FROM
    VW_IncidentsDetail
WHERE
    "Status" = 'Resolved'
ORDER BY
    "Reported Date/Time" DESC
LIMIT 5;
```

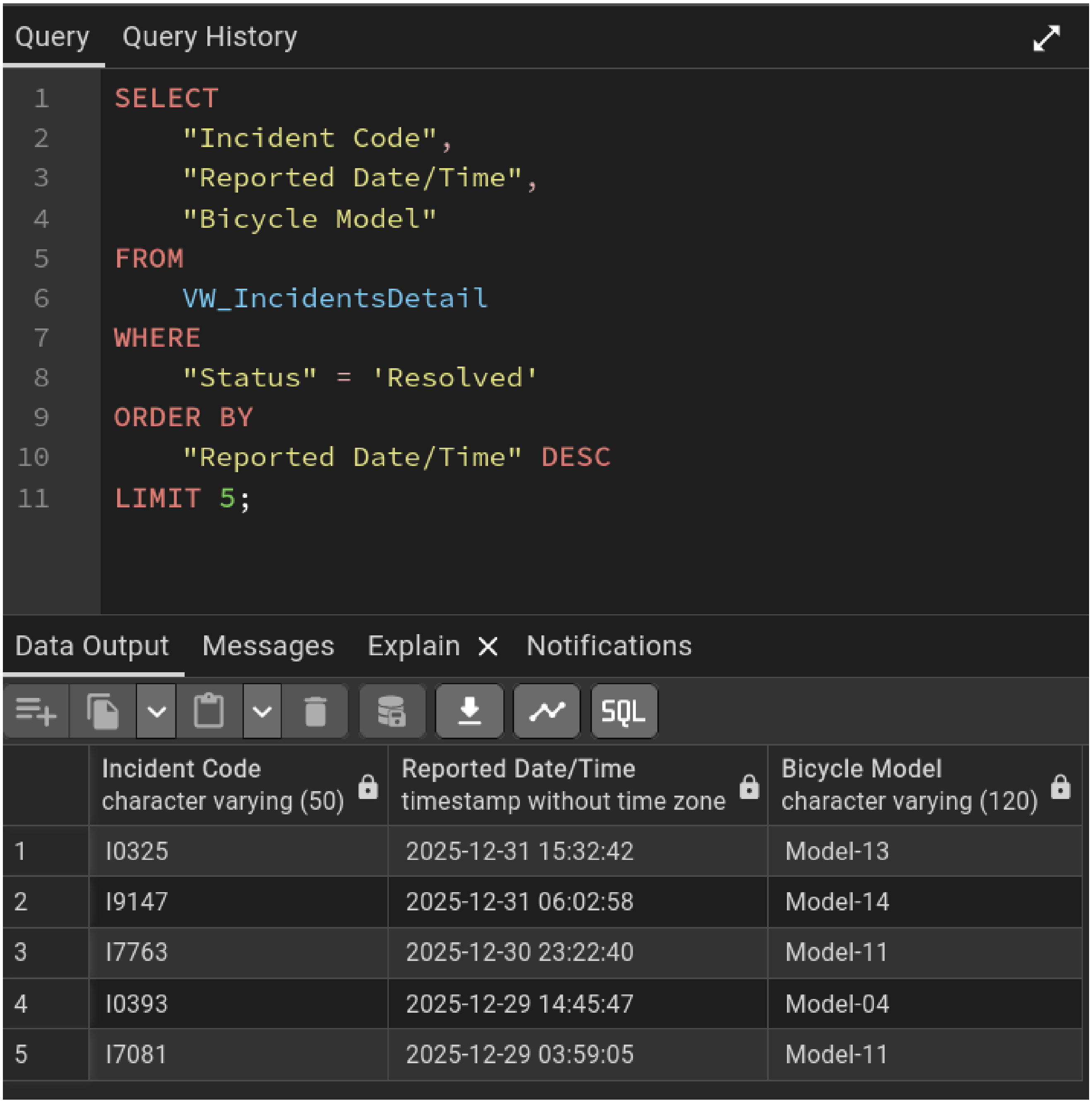


Figure 4.6: View SQL query Screenshot

4.5 Exercise 4e

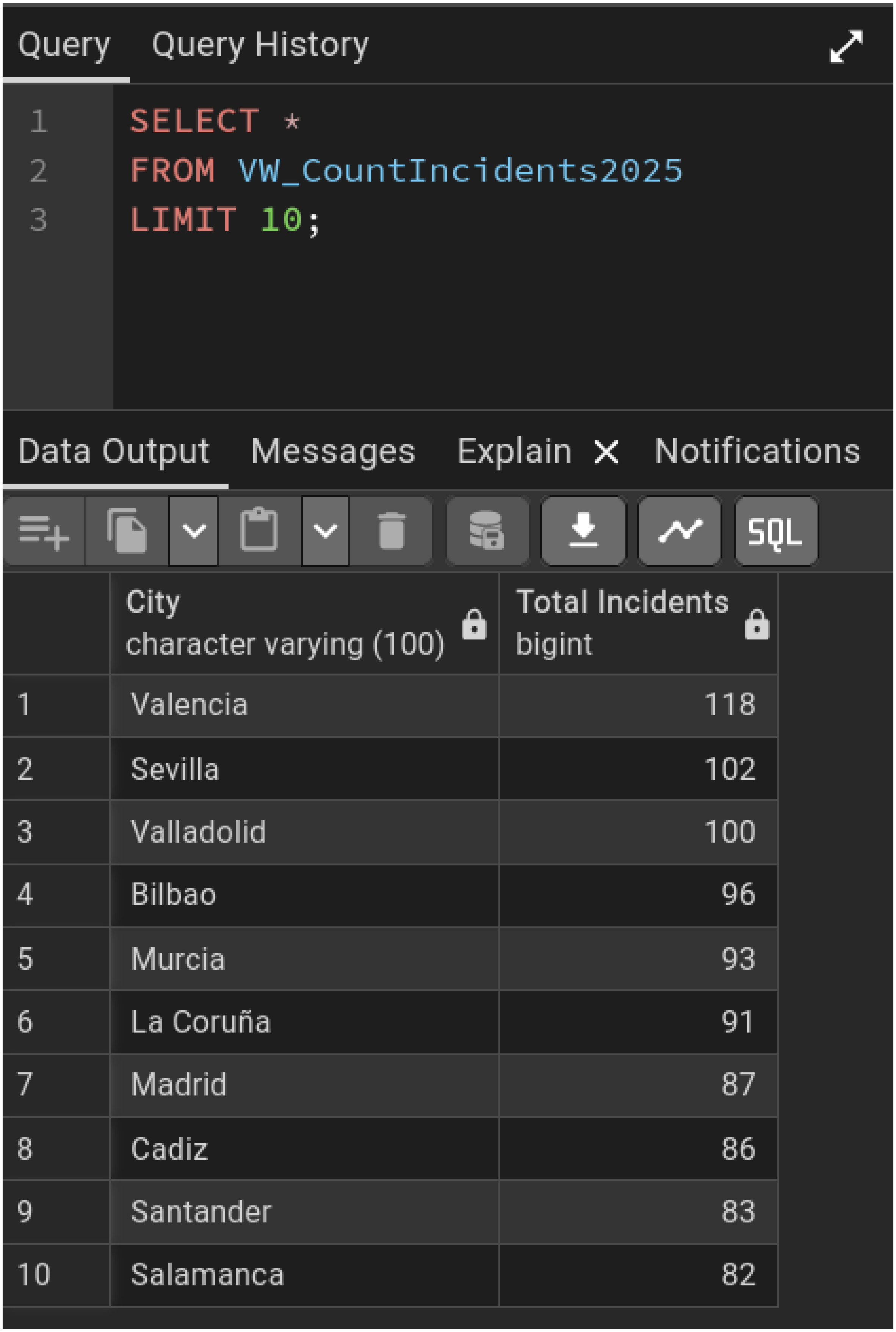
4.5.1 View SQL create code

```
CREATE OR REPLACE VIEW VW_CountIncidents2025 AS
SELECT
    rs.city AS "City",
```

```
    COUNT(*) AS "Total Incidents"
FROM
    VW_IncidentsDetail v
JOIN
    repairshop rs ON v."Assigned Workshop" = rs.code
WHERE
    EXTRACT(YEAR FROM v."Reported Date/Time") = 2025
GROUP BY
    rs.city
ORDER BY
    "Total Incidents" DESC,
    "City" ASC;
```

4.5.2 Query to view 10 first records

```
SELECT *
FROM VW_CountIncidents2025
LIMIT 10;
```




```sql
SELECT *
FROM VW_CountIncidents2025
LIMIT 10;
```

	City character varying (100)	Total Incidents bigint
1	Valencia	118
2	Sevilla	102
3	Valladolid	100
4	Bilbao	96
5	Murcia	93
6	La Coruña	91
7	Madrid	87
8	Cadiz	86
9	Santander	83
10	Salamanca	82

Figure 4.7: View SQL query Screenshot

4.5.3 Materialized View

> **Warning**
>
> We were asked to create the Materialized View of the view `VW_CountIncidentsByCity2025`, but it is not referenced before this statement. Therefore, will use the previously created `VW_CountIncidents2025` view.

```
CREATE MATERIALIZED VIEW V_VW_CountIncidents2025 AS
SELECT *
FROM VW_CountIncidents2025;
```

4.5.4 Performance analysis

4.5.4.1 Standard view (VW_CountIncidents2025)

- Estimated cost: 379.69
- 16-step plan:
  - Sequential scans of incident (step 11), bicycle (step 14), and repairshop (step 16)
  - Several hash joins to connect those tables (steps 7 and 9)
  - Sorting and a GroupAggregate (step 3) to count incidents by city
- A view is just a stored query, so every SELECT reruns all joins, filters, and aggregations over the base tables.

4.5.4.2 Materialized view (V_VW_CountIncidents2025)

- Estimated cost: 13.20
- One sequential scan of the physical table v_vw_countincidents2025
- The materialized view stores the query's result on disk; reading it doesn't touch incident, bicycle, or repairshop or recompute joins/counts.

All in all, querying the materialized view is almost 30x cheaper because the expensive work—joining and counting—was done beforehand and only needs to be redone when you refresh it.

4.5.5 Advantages and Disadvantages

Given that 2025 is still underway, here's a concise take on using a materialized view for this data.

4.5.5.1 Advantages

- The plan cost drops quite significantly, so year-to-date dashboards open quickly without redoing heavy joins.
- The analytical reads shift to the view, so inserts/updates on incident aren't competing with reporting queries.

4.5.5.2 Disadvantages

- A materialized view is a snapshot. Anything added after the last refresh won't appear, so counts drift until you refresh.
- You need a refresh schedule. Refreshing too often eats into performance; refreshing rarely makes the numbers too old to trust.

4.5.6 Updating view

Materialized views do not update automatically. One must regularly refresh it with the following:

```
REFRESH MATERIALIZED VIEW V_VW_CountIncidents2025;
```

Chapter 5

Model review and optimization

5.1 Question 5a

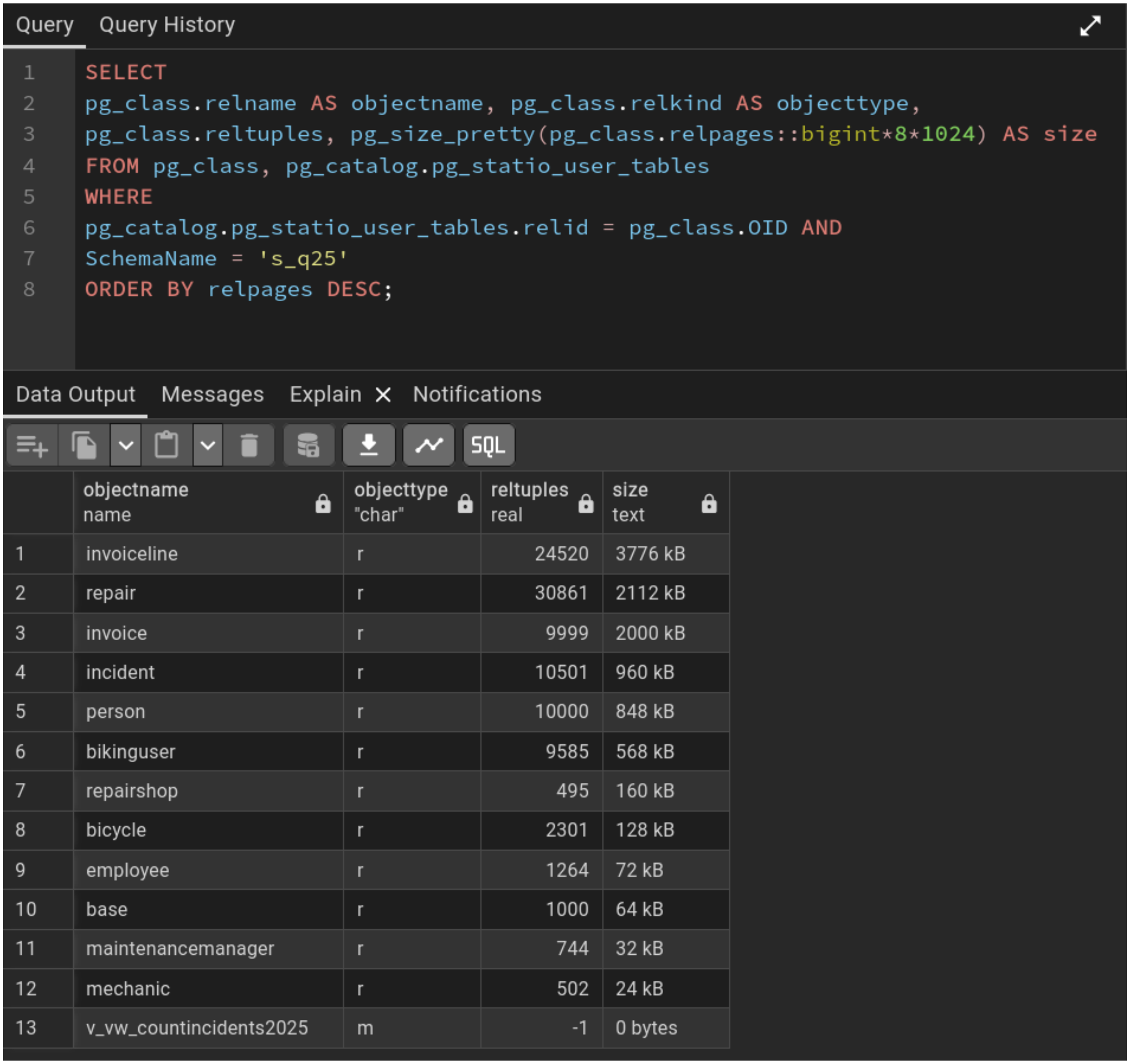
5.1.1 hourCost SQL query

```
SELECT
    'hourCost' AS column_name,
    'Mechanic' AS table_name,
    COUNT(*) AS number_of_rows,
    COUNT(DISTINCT hourcost) AS distinct_values,
    AVG(pg_column_size(hourcost)) AS average_occupation
FROM mechanic;
```

5.1.2 Space occupied by tables

Query run:

```
SELECT
pg_class.relname AS objectname, pg_class.relkind AS objecttype,
pg_class.reltuples, pg_size_pretty(pg_class.relpages::bigint*8*1024) AS size
FROM pg_class, pg_catalog.pg_statio_user_tables
WHERE
pg_catalog.pg_statio_user_tables.relid = pg_class.OID AND
SchemaName = 's_q25'
```




```sql
SELECT
pg_class.relname AS objectname, pg_class.relkind AS objecttype,
pg_class.reltuples, pg_size_pretty(pg_class.relpages::bigint*8*1024) AS size
FROM pg_class, pg_catalog.pg_statio_user_tables
WHERE
pg_catalog.pg_statio_user_tables.relid = pg_class.OID AND
SchemaName = 's_q25'
ORDER BY relpages DESC;
```

	objectname name	objecttype "char"	reltuples real	size text
1	invoiceline	r	24520	3776 kB
2	repair	r	30861	2112 kB
3	invoice	r	9999	2000 kB
4	incident	r	10501	960 kB
5	person	r	10000	848 kB
6	bikinguser	r	9585	568 kB
7	repairshop	r	495	160 kB
8	bicycle	r	2301	128 kB
9	employee	r	1264	72 kB
10	base	r	1000	64 kB
11	maintenancemanager	r	744	32 kB
12	mechanic	r	502	24 kB
13	v_vw_countincidents2025	m	-1	0 bytes

Figure 5.1: Query result

5.2 Exercise 5b

The most appropriate column to move to a separate table is **`concept` from the `InvoiceLine` table** due to the following:

- Very low cardinality: 11 unique values across 24,520 rows, so the same strings repeat thousands of times.
- The strings are fairly large (about 95 bytes on average). If those 11 concepts live in a small lookup table (`ConceptID`, `Description`), `InvoiceLine` would keep only a 4-byte foreign key.

How the others compare:

- `hourCost` (Mechanic): 342 distinct out of 502 rows. High variety, little duplication—normalizing won't help much.

- `description` (Incident): 20 distinct in 10,501 rows, but the text is short (~15 bytes), so the payoff is limited.
- `description` (RepairShop): Only 3 distinct and fairly long (~92 bytes), so it's a reasonable candidate, but with just 495 rows the total gain is small next to InvoiceLine.

5.3 Exercise 5c

```
CREATE TABLE invoice_concept (
    id SERIAL PRIMARY KEY,
    description VARCHAR(200) NOT NULL,

    CONSTRAINT uq_invoice_concept_description UNIQUE (description)
);
```

Added the `uq_invoice_concept_description` constraint to ensure descriptions are unique so normalization works properly.

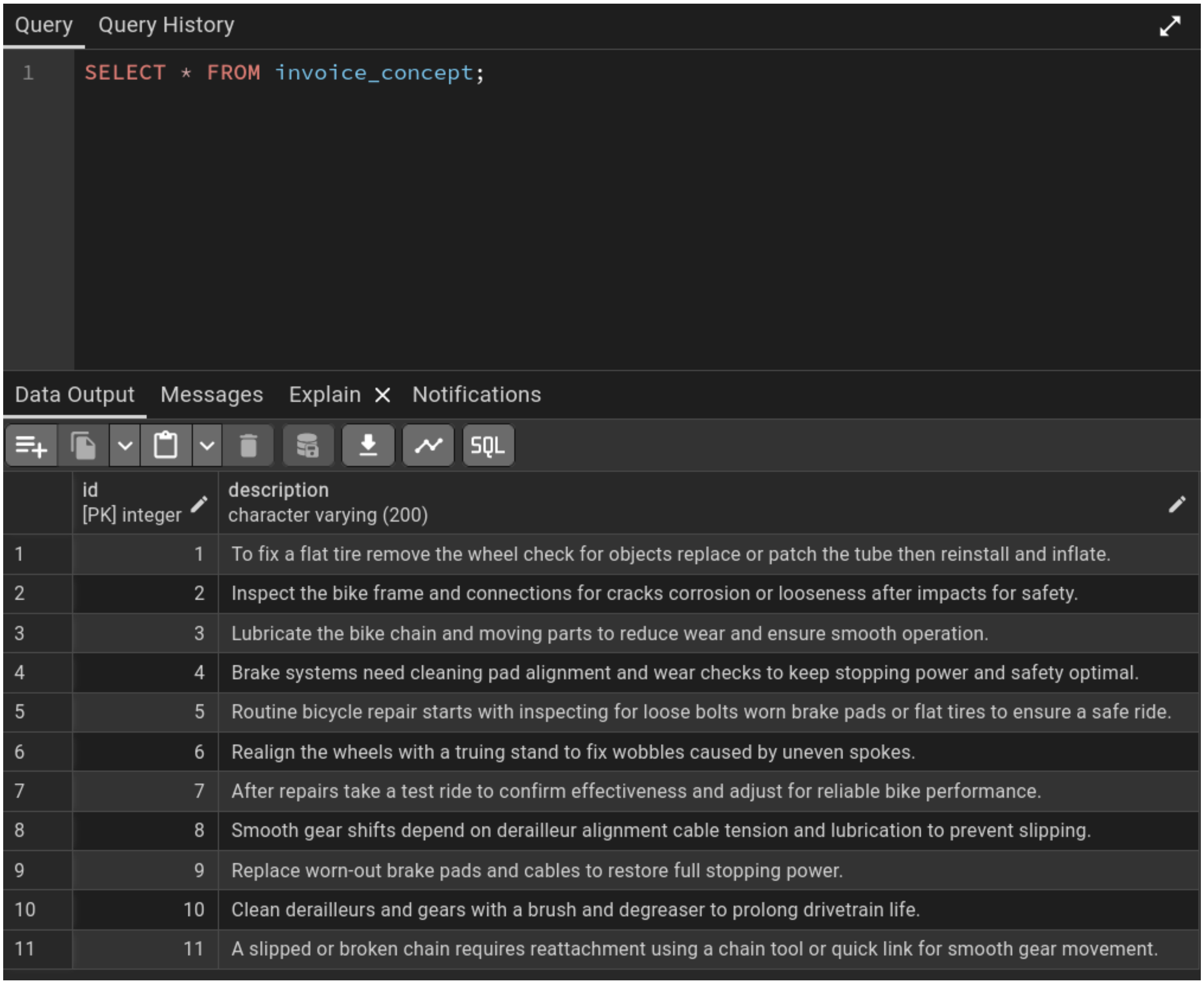
5.4 Exercise 5d

5.4.1 Populate the table

```
INSERT INTO invoice_concept (description)
SELECT DISTINCT concept
FROM invoiceline;
```

5.4.2 Query the table

```
SELECT * FROM invoice_concept;
```



	id [PK] integer	description character varying (200)
1	1	To fix a flat tire remove the wheel check for objects replace or patch the tube then reinstall and inflate.
2	2	Inspect the bike frame and connections for cracks corrosion or looseness after impacts for safety.
3	3	Lubricate the bike chain and moving parts to reduce wear and ensure smooth operation.
4	4	Brake systems need cleaning pad alignment and wear checks to keep stopping power and safety optimal.
5	5	Routine bicycle repair starts with inspecting for loose bolts worn brake pads or flat tires to ensure a safe ride.
6	6	Realign the wheels with a truing stand to fix wobbles caused by uneven spokes.
7	7	After repairs take a test ride to confirm effectiveness and adjust for reliable bike performance.
8	8	Smooth gear shifts depend on derailleur alignment cable tension and lubrication to prevent slipping.
9	9	Replace worn-out brake pads and cables to restore full stopping power.
10	10	Clean derailleurs and gears with a brush and degreaser to prolong drivetrain life.
11	11	A slipped or broken chain requires reattachment using a chain tool or quick link for smooth gear movement.

Figure 5.2: Query result

5.5 Exercise 5e

- Add a new column to store the identifier

```
ALTER TABLE invoiceline
ADD COLUMN concept_id INTEGER;
```

- Update the values of the new column

```
UPDATE invoiceline il
SET concept_id = ic.id
FROM invoice_concept ic
WHERE il.concept = ic.description;
```

- Delete the original column

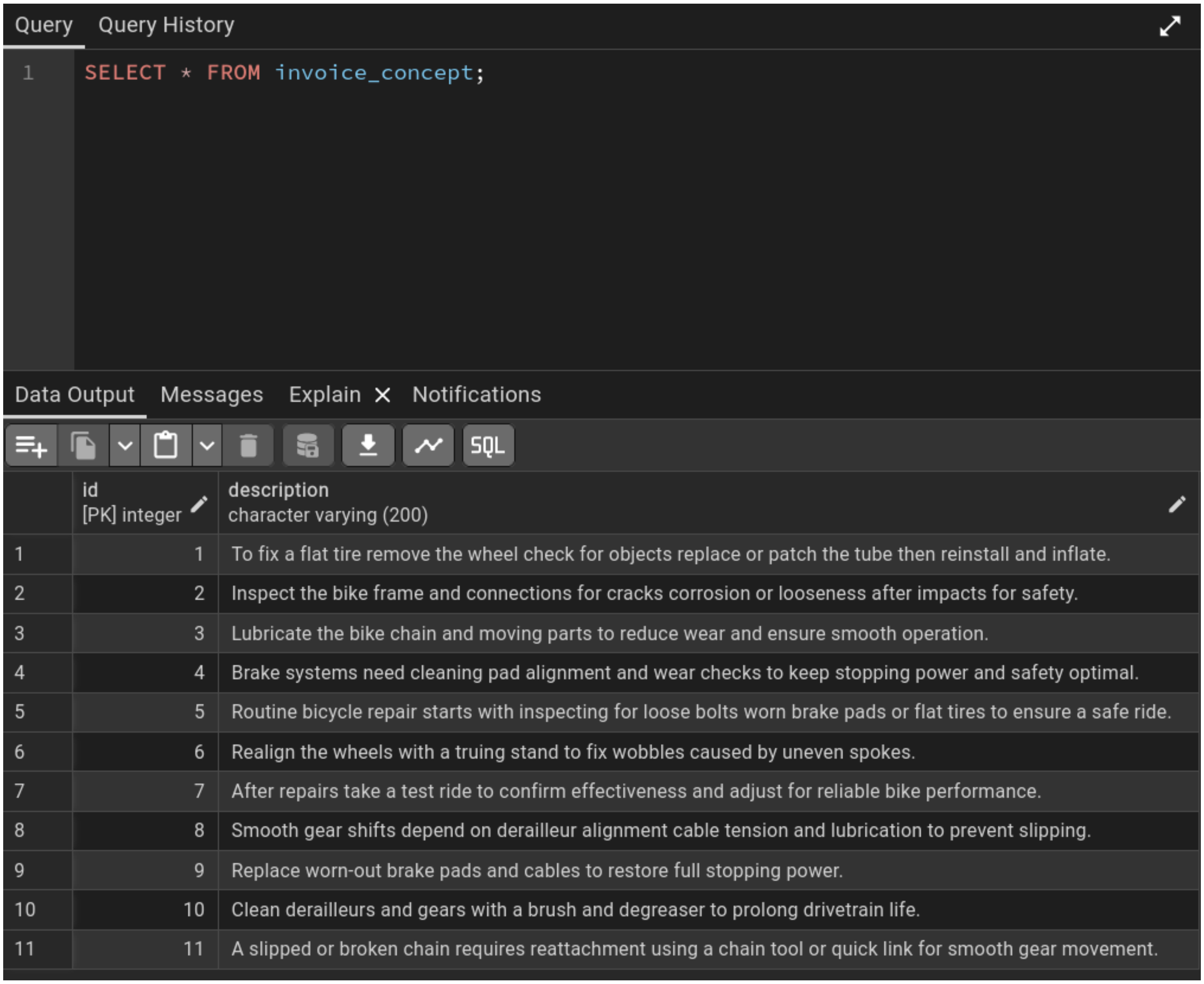
```
ALTER TABLE invoiceline
DROP COLUMN concept;
```

- Define the foreign key constraint

```
ALTER TABLE invoiceline
ADD CONSTRAINT fk_invoiceline_concept
FOREIGN KEY (concept_id)
REFERENCES invoice_concept(id);
```

- Verification: Show the combined values (First 10 records)

```
SELECT
    il.invoice_code,
    il.linenumber,
    ic.description AS concept,
    il.amount,
    il.amountwithvat
FROM
    invoiceline il
JOIN
    invoice_concept ic ON il.concept_id = ic.id
LIMIT 10;
```




```sql
SELECT * FROM invoice_concept;
```

	id [PK] integer	description character varying (200)
1	1	To fix a flat tire remove the wheel check for objects replace or patch the tube then reinstall and inflate.
2	2	Inspect the bike frame and connections for cracks corrosion or looseness after impacts for safety.
3	3	Lubricate the bike chain and moving parts to reduce wear and ensure smooth operation.
4	4	Brake systems need cleaning pad alignment and wear checks to keep stopping power and safety optimal.
5	5	Routine bicycle repair starts with inspecting for loose bolts worn brake pads or flat tires to ensure a safe ride.
6	6	Realign the wheels with a truing stand to fix wobbles caused by uneven spokes.
7	7	After repairs take a test ride to confirm effectiveness and adjust for reliable bike performance.
8	8	Smooth gear shifts depend on derailleur alignment cable tension and lubrication to prevent slipping.
9	9	Replace worn-out brake pads and cables to restore full stopping power.
10	10	Clean derailleurs and gears with a brush and degreaser to prolong drivetrain life.
11	11	A slipped or broken chain requires reattachment using a chain tool or quick link for smooth gear movement.

Figure 5.3: Query result

5.5.1 Final comparison

Here's how the table changed.

- Space used by `invoiceline`
  - Before normalization:
    - The concept text averaged about 95 bytes per row.
    - With roughly 24,520 rows, that single column was ~2.3 MB on its own (24,520 × 95 bytes).
    - Once you include other columns and storage overhead, the table would land around 3–4 MB.
  - After normalization:
    - `invoiceline` now shows about 1,448 kB (~1.4 MB).
    - `concept_id` is an int (4 bytes), so every row swapped ~95 bytes of text for 4 bytes.
- Net effect: a reduction over 50%.

5.5.1.1 Trade-offs

- Pros
  - Much smaller footprint, which saves disk and helps the table fit in memory caches.
  - Single place to maintain each concept
  - Quicker scans on `invoiceline` (sums, date filters) because there are fewer pages to read.
- Cons
  - Queries are less direct: to see the concept text you now join with `invoice_concept`.
  - Reconstructing the full view costs a join, which is heavier than reading a single wide row.