”Optimizing Logistics Through Data” - Courier Nexus

Chaitali Rajulkumar Thakkar
UB ID: 50557808
MS Engineering Science (Data Science)
thakkar6@buffalo.edu

Pranavi Chintala
UB ID: 50563997
MS Engineering Science (Data Science)
pchintal@buffalo.edu

I. Problem Statement

As the scale-up and operations of the logistics company grow increasingly complex, it becomes progressively difficult to manage or keep track of shipment flow. The tools currently used for tracking packages, senders, receivers, and delivery details—mostly Excel—cannot handle the growth in volume and complexity. This eventually leads to inefficiencies in tracking shipment statuses, managing customer feedback, handling delivery failures, and resolving claims. These shortcomings result in delayed deliveries, poor customer satisfaction, and operational bottlenecks.

Besides that, the basic password protection offered by Excel can be easily circumvented, leaving data vulnerable to accidental exposure or overwriting. Moreover, Excel requires manual backups, increasing the risk of data loss and making recovery from file corruption difficult, especially in fast-paced business environments. This highlights the need for the company to transition to a more secure, automated, and highly scalable database solution to meet its growing logistical needs.

We aim to address these issues by implementing a scalable, data-driven logistics management system. This system will provide real-time shipment tracking, efficient monitoring of delivery attempts, and streamlined claims processing. It will capture detailed package dimensions, weight-based shipping rates, and customer feedback trends while ensuring seamless collaboration across departments. The database solution will enable efficient handling of large datasets, support parallel access for multiple users, and allow the company to generate insightful reports to optimize logistics operations. The goal is to enhance data accuracy, improve decision-making, and drive operational efficiency as the company expands.

II. Target Users

This data-driven platform for logistics optimization is designed to serve four key target user groups.

First, core users are the **Logistics Operations Team** who are in charge of managing and tracking shipments. They will add new shipment details, like package dimensions, sender and receiver information, choosing the appropriate methods of shipment. They will also update the delivery status, monitoring the attempts made to ensure transparency and accuracy in the chain of logistics. For instance, Sarah would access the system, a logistics coordinator, input information about packages, provide a tracking number, and update shipment status as it goes through different stages.

Next, the **Customer Service Representatives** are to use the platform in order to support customers with various inquiries and resolve issues that arise regarding delayed shipments or claims. These representatives are going to use the system to check the status of shipments with a view to informing customers in real-time and initiating claims when necessary. In this way, it smoothes out the process for resolving issues. For instance, John will have no problem pulling shipment details while a customer calls about his or her package to review where his shipment is and its current status, thus giving instant feedback.

The **Warehouse and Delivery Teams** will also interface with the platform to update package statuses, log delivery attempts, and record delivery locations. Members will, in turn, regularly update the system with fresh information regarding the movement of packages. For example, Mike could put it in the system that he couldn’t deliver a package because the recipient wasn’t available; this would keep the logistics team informed so they can reschedule the delivery as soon as possible.

Finally, the **Administrators and IT Team** maintain security, performance, and track the general care of the platform. This group will be in charge of permissions, data security, different types of system performance checks, and periodic backups to ensure the health of the database. For instance, Maria, as a database administrator, will make sure that sensitive information such as customers’ claims is not revealed to unauthorized personnel, but also that the system is in compliance with the established security policies. Such users will ensure smoothness and security for the entire platform, supporting the day-to-day operations of the logistics company.

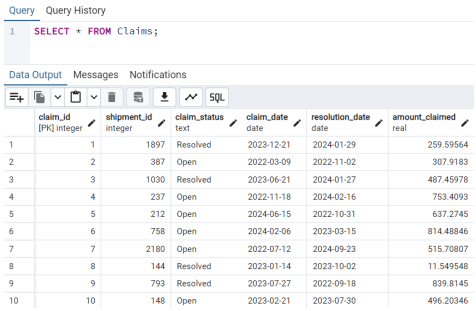
By going back to the specific needs of these varying user groups, **Courier Nexus** will be able to manage logistics efficiently, improve departmental communication, and deliver a seamless experience across all stakeholders touching the shipment lifecycle.

III. Entity - Relationship Diagram

A. Entities

1) ***Claims****:* `claim_id` (Primary Key, INTEGER): Unique identifier for each claim. This cannot be NULL.

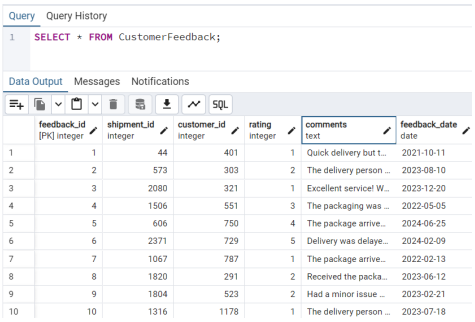
`claim_status` (TEXT): Represents the status of the claim (e.g., ”Pending”, ”Resolved”). Can be NULL.
`claim_date` (DATE): The date when the claim was filed. Can be NULL.
`resolution_date` (DATE): The date when the claim was resolved. Can be NULL.
`amount_claimed` (REAL): The amount claimed in the process. Default value is 0.0.
`shipment_id` (Foreign Key, INTEGER): References the `shipment_id` in the Shipments table to associate a claim with a specific shipment. Cannot be NULL.
Primary Key Justification: The `claim_id` serves as the primary key to ensure that each claim is uniquely identifiable within the system. No duplicate claims are allowed, and every claim needs a distinct ID for traceability.
Foreign Key Action: When a shipment (referenced by `shipment_id`) is deleted, a cascade delete action can be applied, meaning the associated claims would also be deleted.



	claim_id [PK] integer	shipment_id integer	claim_status text	claim_date date	resolution_date date	amount_claimed real
1	1	1897	Resolved	2023-12-21	2024-01-29	259.59564
2	2	387	Open	2022-03-09	2022-11-02	307.9183
3	3	1030	Resolved	2023-06-21	2024-01-27	487.45978
4	4	237	Open	2022-11-18	2024-02-16	753.4093
5	5	212	Open	2024-06-15	2022-10-31	637.2745
6	6	758	Open	2024-02-06	2023-03-15	814.48846
7	7	2180	Open	2022-07-12	2024-09-23	515.70807
8	8	144	Resolved	2023-01-14	2023-10-02	11.549548
9	9	793	Resolved	2023-07-27	2022-09-18	839.8145
10	10	148	Open	2023-02-21	2023-07-30	496.20346

Fig. 1. Table Claims

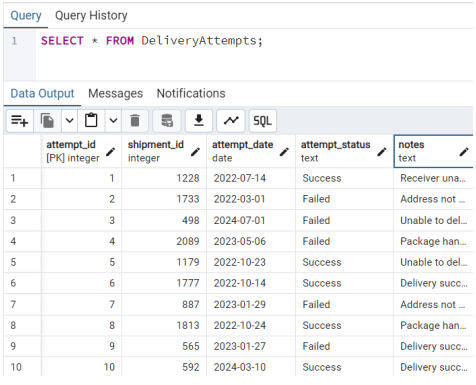
*2) **Customer Feedback** :* `feedback_id` (Primary Key, INTEGER): Unique identifier for customer feedback. Cannot be NULL.
`customer_id` (INTEGER): Represents the customer providing feedback. Can be NULL.
`rating` (INTEGER): The rating given by the customer (e.g., 1–5). Can be NULL.
`comments` (TEXT): Feedback or comments from the customer. Can be NULL.
`feedback_date` (DATE): The date when the feedback was provided. Can be NULL.
`shipment_id` (Foreign Key, INTEGER): References the `shipment_id` in the Shipments table to tie feedback to a specific shipment. Cannot be NULL.
Primary Key Justification: The `feedback_id` uniquely identifies each feedback entry, ensuring no duplicates and allowing feedback to be tracked individually.
Foreign Key Action: On deletion of a referenced shipment, the associated customer feedback can be set to NULL to maintain feedback records without shipment information.



	feedback_id [PK] integer	shipment_id integer	customer_id integer	rating integer	comments text	feedback_date date
1	1	44	401	1	Quick delivery but t...	2021-10-11
2	2	573	303	2	The delivery person ...	2023-08-10
3	3	2080	321	1	Excellent service! W...	2023-12-20
4	4	1506	551	3	The packaging was ...	2022-05-05
5	5	606	750	4	The package arrive...	2024-06-25
6	6	2371	729	5	Delivery was delaye...	2024-02-09
7	7	1067	787	1	The package arrive...	2022-02-13
8	8	1820	291	2	Received the packa...	2023-06-12
9	9	1804	523	2	Had a minor issue ...	2023-02-21
10	10	1316	1178	1	The delivery person ...	2023-07-18

Fig. 2. Table Customer Feedback

*3) **Delivery Attempts** :* `attempt_id` (Primary Key, INTEGER): Unique identifier for each delivery attempt. Cannot be NULL.
`attempt_date` (DATE): The date of the delivery attempt. Can be NULL.
`attempt_status` (TEXT): The result or status of the delivery attempt (e.g., ”Failed”, ”Successful”). Can be NULL.
`notes` (TEXT): Additional information about the attempt. Can be NULL.
`shipment_id` (Foreign Key, INTEGER): References `shipment_id` in the Shipments table. Cannot be NULL.
Primary Key Justification: The `attempt_id` is necessary to uniquely identify each attempt made to deliver a shipment, ensuring that all records are traceable.
Foreign Key Action: On deletion of a shipment, a cascade delete can be applied, removing all associated delivery attempts for that shipment.

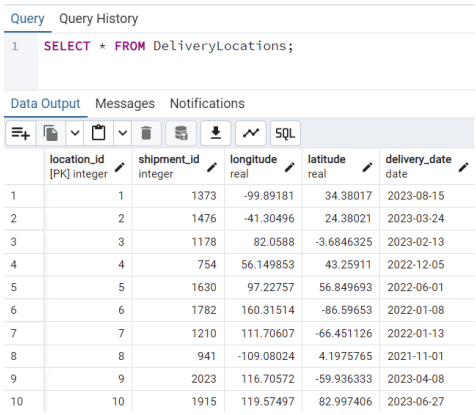


	attempt_id [PK] integer	shipment_id integer	attempt_date date	attempt_status text	notes text
1	1	1228	2022-07-14	Success	Receiver una...
2	2	1733	2022-03-01	Failed	Address not ...
3	3	498	2024-07-01	Failed	Unable to del...
4	4	2089	2023-05-06	Failed	Package han...
5	5	1179	2022-10-23	Success	Unable to del...
6	6	1777	2022-10-14	Success	Delivery succ...
7	7	887	2023-01-29	Failed	Address not ...
8	8	1813	2022-10-24	Success	Package han...
9	9	565	2023-01-27	Failed	Delivery succ...
10	10	592	2024-03-10	Success	Delivery succ...

Fig. 3. Table Delivery Attempts

*4) **Delivery Locations**:* `location_id` (Primary Key, INTEGER): Unique identifier for each delivery location. Cannot be NULL.
`longitude` (REAL): The longitude of the delivery location. Can be NULL.
`latitude` (REAL): The latitude of the delivery location. Can be NULL.
`delivery_date` (DATE): The date of delivery at this location. Can be NULL.

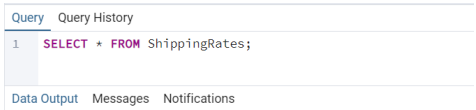
`shipment_id` (Foreign Key, INTEGER): References `shipment_id` in the Shipments table. Cannot be NULL.
Primary Key Justification: The `location_id` is used to ensure that every delivery location is unique, which is crucial for tracking where shipments were delivered.
Foreign Key Action: On deletion of a shipment, all associated delivery locations should also be deleted, enforcing a cascade delete policy.



	location_id [PK] integer	shipment_id integer	longitude real	latitude real	delivery_date date
1	1	1373	-99.89181	34.38017	2023-08-15
2	2	1476	-41.30496	24.38021	2023-03-24
3	3	1178	82.0588	-3.6846325	2023-02-13
4	4	754	56.149853	43.25911	2022-12-05
5	5	1630	97.22757	56.849693	2022-06-01
6	6	1782	160.31514	-86.59653	2022-01-08
7	7	1210	111.70607	-66.451126	2022-01-13
8	8	941	-109.08024	4.1975765	2021-11-01
9	9	2023	116.70572	-59.936333	2023-04-08
10	10	1915	119.57497	82.997406	2023-06-27

Fig. 4. Table Delivery Locations

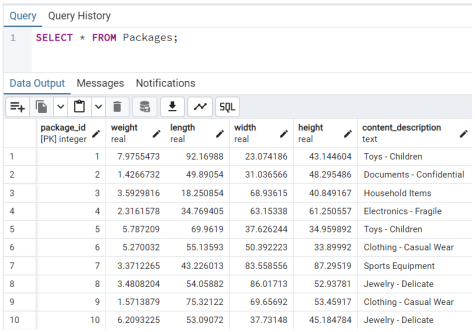
5) ***Shipping Rates*** *:* `rate_id` (Primary Key, INTEGER): Unique identifier for each shipping rate. Cannot be NULL.
`shipping_method` (TEXT): The shipping method used (e.g., "Air", "Ground"). Can be NULL.
`weight_limit` (REAL): The maximum weight allowed for this rate. Default value is 0.0.
`base_price` (REAL): The base price for this shipping rate. Default value is 0.0.
`additional_cost_per_kg` (REAL): Additional cost per kilogram over the weight limit. Default value is 0.0.
Primary Key Justification: The `rate_id` ensures each shipping rate is unique, as different rates may apply based on method and weight.
Foreign Key Action: No foreign keys are present, so no specific action is required for deletions.



	rate_id [PK] integer	shipping_method text	weight_limit real	base_price real	additional_cost_ real
1	1	Sea	9.288893	67.30307	3.0308886
2	2	Ground	18.316689	59.710064	7.6476617
3	3	Air	62.37874	20.341097	6.700099
4	4	Air	52.873016	66.149124	3.5533419
5	5	Sea	39.385883	23.125923	8.530161
6	6	Air	35.220295	31.24818	4.7629867
7	7	Ground	35.649265	33.091633	9.789519
8	8	Ground	10.081452	87.70735	1.4218218
9	9	Air	26.561792	75.10101	5.698608
10	10	Air	95.30269	62.07692	7.74137

Fig. 5. Table Shipping Rates

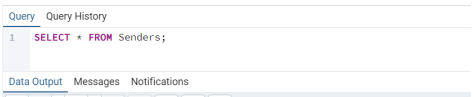
6) ***Packages*** *:* `package_id` (Primary Key, INTEGER): Unique identifier for each package. Cannot be NULL.
`weight` (REAL): Weight of the package. Default value is 0.0.
`length` (REAL): Length of the package. Default value is 0.0.
`width` (REAL): Width of the package. Default value is 0.0.
`height` (REAL): Height of the package. Default value is 0.0.
`content_description` (TEXT): Description of the contents of the package. Can be NULL.
Primary Key Justification: The `package_id` uniquely identifies each physical package, allowing shipments to be tied to specific packages.
Foreign Key Action: No foreign keys are present, so no specific action is required for deletions.



	package_id [PK] integer	weight real	length real	width real	height real	content_description text
1	1	7.9755473	92.16988	23.074186	43.144604	Toys - Children
2	2	1.4266732	49.89054	31.036566	48.295486	Documents - Confidential
3	3	3.5929816	18.250854	68.93615	40.849167	Household Items
4	4	2.3161578	34.769405	63.15338	61.250557	Electronics - Fragile
5	5	5.787209	69.9619	37.626244	34.959892	Toys - Children
6	6	5.270032	55.13593	50.392223	33.89992	Clothing - Casual Wear
7	7	3.3712265	43.226013	83.558556	87.29519	Sports Equipment
8	8	3.4808204	54.05882	86.01713	52.93781	Jewelry - Delicate
9	9	1.5713879	75.32122	69.65692	53.45917	Clothing - Casual Wear
10	10	6.2093225	53.09072	37.73148	45.184784	Jewelry - Delicate

Fig. 6. Table Packages

7) ***Senders*** *:* `sender_id` (Primary Key, INTEGER): Unique identifier for each sender. Cannot be NULL.
`name` (TEXT): Name of the sender. Can be NULL.
`address` (TEXT): Address of the sender. Can be NULL.
`contact` (TEXT): Contact information for the sender. Can be NULL.
Primary Key Justification: The `sender_id` uniquely identifies each sender, allowing packages to be tracked by who sent them.
Foreign Key Action: No foreign keys are present, so no specific action is required for deletions.

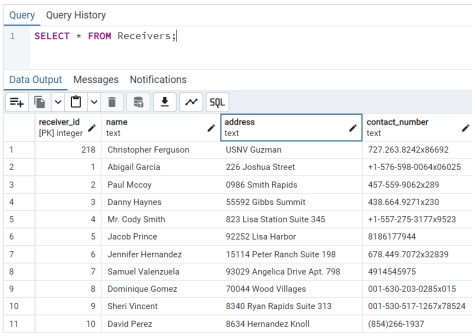


	sender_id [PK] integer	name text	address text	contact_number text
1	1	Ryan Dunn	4055 Monica Key	9372572007
2	2	Aaron Clements	2641 Proctor Trail Apt. 482	(459)757-3054x08142
3	3	Jason Martinez	8047 Welch Tunnel	309-323-8006x367
4	4	Lindsey Hawkins	83902 Smith Island	7456199563
5	5	Kevin Foster	762 Ana Islands	+1-684-247-7873x11655
6	6	Michelle Peters	944 Murray Alley Suite 169	(584)506-6020x3322
7	7	Michael Thomas	448 Castillo Route Apt. 713	836.601.1128
8	8	Stanley White	9221 Gregory Trail Suite 548	919.379.4623x74771
9	9	Natalie Jenkins	999 Compton Knoll	(338)539-6116x6420
10	10	Caleb Ferrell	2474 Kevin Gardens Apt. 305	(467)232-0044

Fig. 7. Table Senders

8) ***Receivers*** *:* `receiver_id` (Primary Key, INTEGER): Unique identifier for each receiver. Cannot be NULL.

`name` (TEXT): Name of the receiver. Can be NULL.
`address` (TEXT): Address of the receiver. Can be NULL.
`contact` (TEXT): Contact information for the receiver. Can be NULL.
Primary Key Justification: The `receiver_id` ensures that each receiver is unique, allowing the system to track who receives which shipment.
Foreign Key Action: No foreign keys are present, so no specific action is required for deletions.



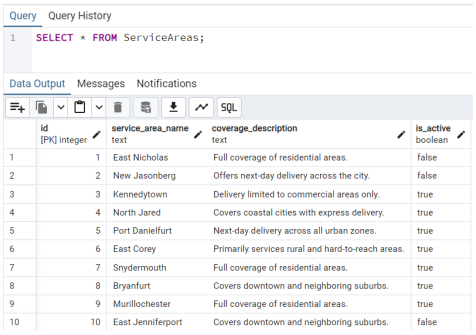
```
SELECT * FROM Receivers;
```

Data Output | Messages | Notifications

	receiver_id [PK] integer	name text	address text	contact_number text
1	218	Christopher Ferguson	USNV Guzman	727.263.8242x86692
2	1	Abigail Garcia	226 Joshua Street	+1-576-598-0064x06025
3	2	Paul Mccoy	0986 Smith Rapids	457-559-9062x289
4	3	Danny Haynes	55592 Gibbs Summit	438.664.9271x230
5	4	Mr. Cody Smith	823 Lisa Station Suite 345	+1-557-275-3177x9523
6	5	Jacob Prince	92252 Lisa Harbor	8186177944
7	6	Jennifer Hernandez	15114 Peter Ranch Suite 198	678.449.7072x32839
8	7	Samuel Valenzuela	93029 Angelica Drive Apt. 798	4914545975
9	8	Dominique Gomez	70044 Wood Villages	001-630-203-0285x015
10	9	Sheri Vincent	8340 Ryan Rapids Suite 313	001-530-517-1267x78524
11	10	David Perez	8634 Hernandez Knoll	(854)266-1937

Fig. 8. Table Receivers

9) ***Service Areas*** *:* `area_id` (Primary Key, INTEGER): Unique identifier for each service area. Cannot be NULL.
`service_area_name` (TEXT): Name of the service area. Can be NULL.
`coverage_description` (TEXT): Description of the area covered by the service. Can be NULL.
`is_active` (BOOLEAN): Indicates whether the service area is active. Default value is TRUE.
Primary Key Justification: The `area_id` uniquely identifies each service area, ensuring distinct coverage zones.
Foreign Key Action: No foreign keys are present, so no specific action is required for deletions.



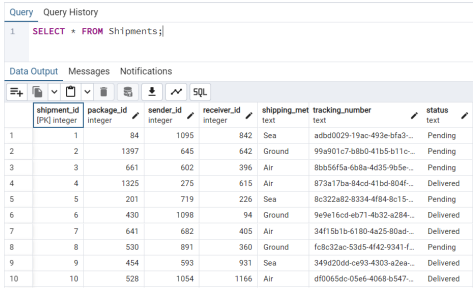
```
SELECT * FROM ServiceAreas;
```

Data Output | Messages | Notifications

	id [PK] integer	service_area_name text	coverage_description text	is_active boolean
1	1	East Nicholas	Full coverage of residential areas.	false
2	2	New Jasonberg	Offers next-day delivery across the city.	false
3	3	Kennedytown	Delivery limited to commercial areas only.	true
4	4	North Jared	Covers coastal cities with express delivery.	true
5	5	Port Danielfurt	Next-day delivery across all urban zones.	true
6	6	East Corey	Primarily services rural and hard-to-reach areas.	true
7	7	Snydermouth	Full coverage of residential areas.	true
8	8	Bryanfurt	Covers downtown and neighboring suburbs.	true
9	9	Murillochester	Full coverage of residential areas.	true
10	10	East Jenniferport	Covers downtown and neighboring suburbs.	false

Fig. 9. Table Service Areas

10) ***Shipments*** *:* `shipment_id` (Primary Key, INTEGER): Unique identifier for each shipment. Cannot be NULL.
`package_id` (Foreign Key, INTEGER): References `package_id` in the Packages table. Cannot be NULL.
`sender_id` (Foreign Key, INTEGER): References `sender_id` in the Senders table. Cannot be NULL.
`receiver_id` (Foreign Key, INTEGER): References `receiver_id` in the Receivers table. Cannot be NULL.
`shipment_method` (TEXT): The method used for shipment. Can be NULL.
`tracking_number` (TEXT): Tracking number for the shipment. Can be NULL.
`status` (TEXT): The status of the shipment (e.g., ”In Transit”, ”Delivered”). Can be NULL.
Primary Key Justification: The `shipment_id` ensures that each shipment is uniquely identified, allowing detailed tracking of every shipment in the system.
Foreign Key Action: For `package_id`, `sender_id`, and `receiver_id`, when the primary key is deleted, all associated shipments should be deleted using cascade delete.



```
SELECT * FROM Shipments;
```

Data Output | Messages | Notifications

	shipment_id [PK] integer	package_id integer	sender_id integer	receiver_id integer	shipping_met text	tracking_number text	status text
1	1	84	1095	842	Sea	adbd0029-19ac-493e-bfa3-...	Pending
2	2	1397	645	642	Ground	99a901c7-b8b0-41b5-b11c-...	Pending
3	3	661	602	396	Air	8bb56f5a-6b8a-4d35-9b5e-...	Pending
4	4	1325	275	615	Air	873a17ba-84cd-41bd-804f-...	Delivered
5	5	201	719	226	Sea	8c322a82-8334-4f84-8c15-...	Pending
6	6	430	1098	94	Ground	9e9e16cd-eb71-4b32-a284-...	Delivered
7	7	641	682	405	Air	34f15b1b-6180-4a25-80ad-...	Delivered
8	8	530	891	360	Ground	fc8c32ac-53d5-4f42-9341-f...	Pending
9	9	454	593	931	Sea	349d20dd-ce93-4303-a2ea-...	Delivered
10	10	528	1054	1166	Air	df0065dc-05e6-4068-b547-...	Delivered

Fig. 10. Table Shipments

B. Relations and Attributes

Table I below gives an overview of some of the important entities that describe the relationships and attributes of the logistics enterprise, including claims, shipments, and customer feedback. It provides full elaboration, featuring specific details for each entity on its main attributes and how those entities relate to one another through foreign keys. The following table shows the structure of this system.

C. Relationships Between Entities

Fig. 11 illustrates the relationship of the logistics system entities. This shows how shipments, packages, senders, and receivers—the major entities in this database—relate to one another. The relationships between different entities are established through foreign keys, which ensure that the flow of data within the system is smooth for tracking and managing purposes. The relations between each entity are given as follows:
1) ***Shipments*** $\rightarrow$ ***Packages****:* There is a one-to-one relationship between Shipments and Packages, meaning that each shipment must have precisely one package, and every package has to be shipped once. This ensures that each package’s information is attached to a specific shipment.
2) ***Shipments*** $\rightarrow$ ***Senders / Receivers****:* Shipments are related to both Senders and Receivers on a many-to-one basis: one sender can send many shipments, and similarly, one receiver can receive many shipments; each shipment is linked with one sender and one receiver.

TABLE I
ENTITY RELATIONS AND ATTRIBUTES

Entity	Attributes
Claims	`claim_id` (primary key), `claim_status`, `claim_date`, `resolution_date`, `amount_claimed`, `shipment_id` (foreign key)
Customer Feedback	`feedback_id` (primary key), `customer_id`, `rating`, `comments`, `feedback_date`, `shipment_id` (foreign key)
Delivery Attempts	`attempt_id` (primary key), `attempt_date`, `attempt_status`, `notes`, `shipment_id` (foreign key)
Delivery Locations	`location_id` (primary key), `longitude`, `latitude`, `delivery_date`, `shipment_id` (foreign key)
Shipping Rates	`rate_id` (primary key), `shipping_method`, `weight_limit`, `base_price`, `additional_cost_per_kg`
Packages	`package_id` (primary key), `weight`, `length`, `width`, `height`, `content_description`
Senders	`sender_id` (primary key), `name`, `address`, `contact`
Receivers	`receiver_id` (primary key), `name`, `address`, `contact`
Service Areas	`area_id` (primary key), `service_area_name`, `coverage_description`, `is_active` (Boolean)
Shipments	`shipment_id` (primary key), `package_id` (foreign key), `sender_id` (foreign key), `receiver_id` (foreign key), `shipment_method`, `tracking_number`, `status`

*3) **Shipments** → **Claims** :* It can be said that the relationship between Shipments and Claims is one-to-one because a shipment can only have one claim, and a claim can only be linked to one shipment. This ensures that a claim is directly related to an individual shipment.

*4) **Shipments** → **Customer Feedback**:* This is a one-to-many relationship; many entries regarding customer feedback are associated with one shipment. A customer can leave multiple pieces of feedback, such as providing updates about experiences or following up after the receipt of delivery.

*5) **Shipments** → **Delivery Attempts**:* The relationship between Shipments and Delivery Attempts is one-to-many because for each shipment, there can be more than one attempt at its delivery. This models real-world logistics where multiple delivery attempts may be required.

*6) **Shipments** → **Delivery Locations**:* This also represents a one-to-many relationship. There might be several delivery locations logged against one shipment, especially if it was moved to another distribution center or if multiple delivery attempts were made.

*7) **Service Areas** → **Shipments**:* While not directly connected in the diagram, Service Areas define the geographical zones in which shipments can be delivered. Shipping Rates depend on the shipment method and the package's weight, helping to estimate costs.

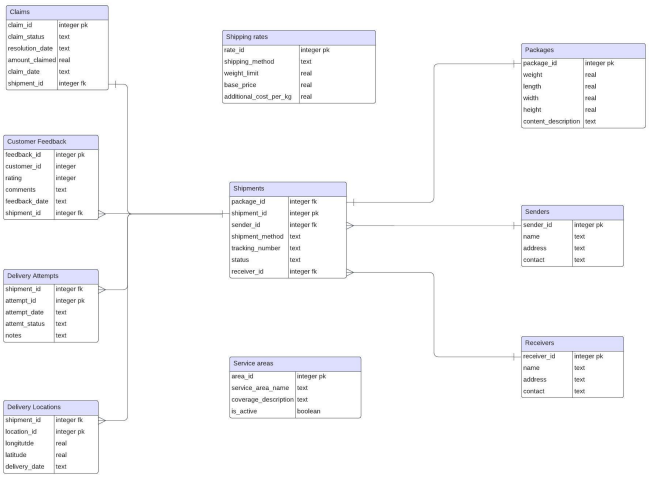


Fig. 11. Entity Relationship Diagram for Logistics Enterprise

IV. DATABASE GENERATION

In the project, Python scripts were written to create a mock dataset representative of an imaginary domain. The idea behind creating this dataset was to develop a large but manageable database for SQL-based applications. The dataset needs to involve queries, trend analysis, and updates in the specified domain. As the data is synthetic, we have the flexibility to shape and modify it so that the database efficiently meets its intended purpose.

A. Generating the table Packages

```
import sqlite3
from faker import Faker
import random

# Initialize Faker
fake = Faker()

# Connect to SQLite database (or create it)
conn = sqlite3.connect('courier_services5.db')
cursor = conn.cursor()

# Create Tables
cursor.execute('''CREATE TABLE IF NOT
EXISTS Packages (
    package_id INTEGER PRIMARY KEY,
    weight REAL,
    length REAL,
    width REAL,
    height REAL,
    content_description TEXT
)''')
```

```
# Commit table creation
conn.commit()

# Helper functions to generate random data
def create_packages(n):
    for _ in range(n):
        cursor.execute('''INSERT INTO Packages
        (weight, length, width, height,
        content_description)
             VALUES (?, ?, ?, ?, ?)''',
            (random.uniform(1, 10),
            random.uniform(10, 100),
            random.uniform(10, 100),
            random.uniform(10, 100),
            fake.text(max_nb_chars=50)))
    conn.commit()

# Generate Data for a larger set
num_packages = 1500

create_packages(num_packages)

# Close the connection
conn.close()
```

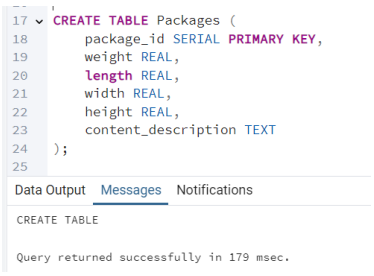


Fig. 12. Creating the table Packages

B. Generating the table Shipments

```
import sqlite3
from faker import Faker
import random

# Initialize Faker
fake = Faker()

# Connect to SQLite database
conn = sqlite3.connect('courier_services5.db')
cursor = conn.cursor()

# Create Tables
cursor.execute('''CREATE TABLE IF NOT
EXISTS Shipments (
    shipment_id INTEGER PRIMARY KEY,
    package_id INTEGER,
    sender_id INTEGER,
    receiver_id INTEGER,
    shipping_method TEXT,
    tracking_number TEXT,
    status TEXT,
    FOREIGN KEY (package_id) REFERENCES
    Packages(package_id),
    FOREIGN KEY (sender_id) REFERENCES
    Senders(sender_id),
    FOREIGN KEY (receiver_id) REFERENCES
    Receivers(receiver_id)
)''')

# Helper functions to generate random data
def create_shipments(n):
    for _ in range(n):
        cursor.execute('''INSERT INTO Shipments
        (package_id, sender_id, receiver_id,
        shipping_method, tracking_number, status)
            VALUES (?, ?, ?, ?, ?, ?)''',
            (random.randint(1, num_packages),
            random.randint(1, num_senders),
            random.randint(1, num_receivers),
            random.choice(['Air', 'Ground',
            'Sea'])
            ,fake.uuid4(),
            random.choice(['Pending',
            'Delivered'])))
    conn.commit()

# Generate Data for a larger set
num_shipments = 2500

create_shipments(num_shipments)

# Close the connection
conn.close()
```

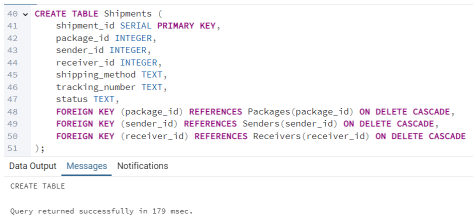


Fig. 13. Creating the table Shipments

As generated above, we have created the remaining tables in the database.

V. Query Execution

Various SQL queries on a logistics database may be designed to retrieve and analyze key information related to shipments, claims, customer feedback, and delivery attempts. SQL can serve potentially powerful capabilities given through commands such as 'GROUP BY', 'JOIN', sub-queries, and

aggregate functions. These queries enable the system to answer essential questions such as, but not limited to, how many shipments are in each status, which shipments have claims, and so on. Following are four different kinds of SQL queries which use GROUP BY, sub-queries, JOIN, and aggregate functions on the Logistics Dataset:

A. Using GROUP BY to Get the Count of Shipments by Status

```
/*This query counts how many shipments are in each status
(e.g., "Pending", "Shipped", "Delivered").*/
SELECT status, COUNT(*) AS shipment_count
FROM Shipments
GROUP BY status;
```

Fig. 14. Using Group by function

Below is the output to the query in Fig. 14:

	status text	shipment_count bigint
1	Delivered	1212
2	Pending	1288

Fig. 15. Output of the Group by function

B. Using a Sub-query to Find Shipments with Claims

```
/*This query retrieves details of all shipments
that have had claims filed against them using a sub-query to filter shipments.*/
SELECT shipment_id, package_id, sender_id, receiver_id, status
FROM Shipments
WHERE shipment_id IN (
    SELECT shipment_id FROM Claims
);
```

Fig. 16. Using Sub-query

Below is the output to the above query in Fig. 16:

	shipment_id [PK] integer	package_id integer	sender_id integer	receiver_id integer	status text
1	12	460	632	267	Pending
2	19	1476	157	734	Pending
3	21	654	932	1082	Pending
4	22	941	280	428	Delivered
5	33	413	660	763	Delivered
6	37	110	97	889	Delivered
7	38	1024	483	19	Delivered
8	42	1412	589	330	Pending
9	43	675	519	995	Pending
10	57	1471	7	763	Delivered

Total rows: 459 of 459 Query complete 00:00:00.057 Ln 9735, Col 3

Fig. 17. Output of the Sub-query

C. Using JOIN to Get Detailed Information About Shipments

```
/*This query uses an inner JOIN to get a detailed view of the shipments,
including the sender's name and receiver's name.*/
SELECT s.shipment_id, p.package_id, s.status, se.name AS sender_name, r.name AS receiver_name
FROM Shipments s
JOIN Packages p ON s.package_id = p.package_id
JOIN Senders se ON s.sender_id = se.sender_id
JOIN Receivers r ON s.receiver_id = r.receiver_id;
```

Fig. 18. Using Join function

Below is the output to the above query in Fig. 18:

	shipment_id integer	package_id integer	status text	sender_name text	receiver_name text
1	1	84	Pending	Joseph Pennington	Shane Williams
2	2	1397	Pending	Christopher Nash Jr.	Kimberly Reynolds
3	3	661	Pending	Brian Alvarez	Scott Booth
4	4	1325	Delivered	Michael Leon	Michael Smith
5	5	201	Pending	Brianna Cooper	Robert Soto
6	6	430	Delivered	Omar Donovan	Brandon Schwartz
7	7	641	Delivered	Ashlee Hamilton	Krista Jackson
8	8	530	Pending	Crystal Blake	Nicole Bailey
9	9	454	Delivered	Heather Mcgee	Vanessa Campbell
10	10	528	Delivered	Seth Rivera	Mr. Corey Smith

Total rows: 1000 of 2500 Query complete 00:00:00.047 Ln 9745, Col 51

Fig. 19. Output of the Join function

D. Using Aggregate Functions with a Sub-query to get Total Claims to specified Status

```
/*This query calculates the total value of all claims for shipments that have a specific status
(e.g., "Delivered") using an aggregate function and sub-query.*/
SELECT SUM(c.amount_claimed) AS total_claimed
FROM Claims c
WHERE c.shipment_id IN (
    SELECT shipment_id FROM Shipments WHERE status = 'Delivered'
);
```

Fig. 20. Using Aggregate function on Sub-query

Below is the output to the above query in Fig. 20:

	total_claimed real
1	124772.97

Fig. 21. Output of the Aggregate function on Sub-query

VI. Decompositions

The process of decomposition is critical to ensure that all relations comply with Boyce-Codd Normal Form (BCNF). After evaluating the functional dependencies and applying the Chase Test, it was determined that the given schema already satisfies BCNF. No further decomposition was necessary, as every determinant in the functional dependencies is a candidate key.

This compliance was verified for all relations, including packages, recipients, senders, service areas, deliveries, claims, customer feedback, delivery attempts, and delivery locations. The following section details the functional dependencies used in this analysis.

VII. Functional Dependencies

Each table was evaluated to ensure that all determinants are a candidate key, thus satisfying the BCNF criteria. The following is the list of Functional Dependencies (FDs) identified for all tables:

package_id → {weight, length, width, height, content_description}
receiver_id → {name, address, contact_number}
sender_id → {name, address, contact_number}
area_id → {service_area_name, coverage_description, is_active}
shipment_id → {package_id, sender_id, receiver_id, shipping_method, tracking_number, status}
tracking_number → {shipment_id, package_id, sender_id, receiver_id, shipping_method, status}
claim_id → {shipment_id, claim_status, claim_date, resolution_date, amount_claimed}
feedback_id → {shipment_id, customer_id, rating, comments, feedback_date}
attempt_id → {shipment_id, attempt_date, attempt_status, notes}
location_id → {shipment_id, longitude, latitude, delivery_date}

Table	Determinants	Dependents
Packages	package_id	weight, length, width, height, content_description
Receivers	receiver_id	name, address, contact_number
Senders	sender_id	name, address, contact_number
ServiceAreas	area_id	service_area_name, coverage_description, is_active
Shipments	shipment_id	package_id, sender_id, receiver_id, shipping_method, tracking_number, status
	tracking_number	shipment_id, package_id, sender_id, receiver_id, shipping_method, status
Claims	claim_id	shipment_id, claim_status, claim_date, resolution_date, amount_claimed
CustomerFeedback	feedback_id	shipment_id, customer_id, rating, comments, feedback_date
DeliveryAttempts	attempt_id	shipment_id, attempt_date, attempt_status, notes
DeliveryLocations	location_id	shipment_id, longitude, latitude, delivery_date

TABLE II
FUNCTIONAL DEPENDENCIES

Each of these dependencies was analyzed for compliance with BCNF, ensuring that there are no partial or transitive dependencies. In every relation, the determinant is a superkey, making all tables BCNF-compliant. Below is the analysis of each table in terms of its superkeys:

A. Packages

The **Packages** table contains the attributes {package_id, weight, length, width, height, content_description}. The functional dependency package_id → {weight, length, width, height, content_description} shows that package_id is a superkey, as it uniquely identifies all attributes in the table. Thus, the table satisfies BCNF.

B. Receivers

The **Receivers** table contains the attributes {receiver_id, name, address, contact_number}. The functional dependency receiver_id → {name, address, contact_number} demonstrates that receiver_id is a superkey, as it uniquely identifies all attributes in the table. Therefore, this table is BCNF-compliant.

C. Senders

The **Senders** table contains the attributes {sender_id, name, address, contact_number}. The functional dependency sender_id → {name, address, contact_number} ensures that sender_id is a superkey. This guarantees that the table is in BCNF.

D. Service Areas

The **Service Areas** table contains the attributes {area_id, service_area_name, coverage_description, is_active}. The functional dependency area_id → {service_area_name, coverage_description, is_active} ensures that area_id is a superkey. As a result, the table satisfies BCNF.

E. Shipments

The **Shipments** table contains the attributes {shipment_id, package_id, sender_id, receiver_id, shipping_method, tracking_number, status}.
The functional dependency shipment_id → {package_id, sender_id, receiver_id, shipping_method, tracking_number, st confirms that shipment_id is a superkey.
Additionally, the functional dependency tracking_number → {shipment_id, package_id, sender_id, receiver_id, shipping_method, status} confirms that tracking_number is also a superkey.
Since all dependencies involve superkeys, the table satisfies BCNF.

F. Claims

The **Claims** table contains the attributes {claim_id, shipment_id, claim_status, claim_date, resolution_date, amount_claimed}. The functional dependency claim_id → {shipment_id, claim_status, claim_date, resolution_date, amount_claimed} demonstrates that claim_id is a superkey. Thus, this table satisfies BCNF.

G. Customer Feedback

The **Customer Feedback** table contains the attributes {feedback_id, shipment_id, customer_id, rating, comments, feedback_date}.
The functional dependency feedback_id → {shipment_id, customer_id, rating, comments, feedback_date} confirms that feedback_id is a superkey.
This makes the table compliant with BCNF.

H. Delivery Attempts

The **Delivery Attempts** table contains the attributes {attempt_id, shipment_id, attempt_date, attempt_status, notes}. The functional dependency attempt_id → {shipment_id, attempt_date, attempt_status, notes} confirms that attempt_id is a superkey. Therefore, this table satisfies BCNF.

I. Delivery Locations

The **Delivery Locations** table contains the attributes {location_id, shipment_id, longitude, latitude, delivery_date}. The functional dependency location_id → {shipment_id, longitude, latitude, delivery_date} ensures that location_id is a superkey. As such, this table satisfies BCNF.

VIII. Chase Test

The Chase test was applied to verify whether the decomposition of each table ensures lossless joins and preserves functional dependencies. Below is the detailed analysis for all 10 tables.

A. Packages

The **Packages** table has the schema {package_id, weight, length, width, height,content_description} and the functional dependency

$$\text{package\_id} \rightarrow \begin{cases} \text{weight, length, width,} \\ \text{height, content\_description} \end{cases}$$

Starting with:

$$t_1[\text{package\_id}] = a,\ t_1[\text{weight}] = b,$$
$$t_1[\text{length}] = c,\ t_1[\text{width}] = d,$$
$$t_1[\text{height}] = e,\ t_1[\text{content\_description}] = f.$$

Applying the functional dependency unifies b, c, d, e, f with a, resulting in one row.
Lossless Join: Yes.
Dependency Preservation: Yes.

B. Receivers

The **Receivers** table has the schema {receiver_id, name, address, contact_number} and the functional dependency
receiver_id → {name, address, contact_number}. Starting with

$$t_2[\text{receiver\_id}] = a,\ t_2[\text{name}] = b,\ t_2[\text{address}] = c,$$
$$t_2[\text{contact\_number}] = d.$$

Applying the functional dependency unifies b, c, d with a, resulting in one row.
Lossless Join: Yes. **Dependency Preservation**: Yes.

C. Senders

The **Senders** table has the schema {sender_id, name, address, contact_number} and the functional dependency
sender_id → {name, address, contact_number}. Starting with

$$t_3[\text{sender\_id}] = a,\ t_3[\text{name}] = b,\ t_3[\text{address}] = c,$$
$$t_3[\text{contact\_number}] = d.$$

Applying the functional dependency unifies b, c, d with a, resulting in one row.
Lossless Join: Yes. **Dependency Preservation**: Yes.

D. Service Areas

The **Service Areas** table has the schema {area_id, service_area_name, coverage_description, is_active} and the functional dependency area_id → {service_area_name, coverage_description, is_active}. Starting with

$$t_4[\text{area\_id}] = a,\ t_4[\text{service\_area\_name}] = b,\ t_4[\text{coverage\_description}] =$$
$$t_4[\text{is\_active}] = d.$$

Applying the functional dependency unifies b, c, d with a, resulting in one row.
Lossless Join: Yes. **Dependency Preservation**: Yes.

E. Shipments

The **Shipments** table has the schema {shipment_id, package_id, sender_id, receiver_id, shipping_method, track and the functional dependencies:

shipment_id → {package_id, sender_id, receiver_id, shipping_method, tra

tracking_number → {shipment_id, package_id, sender_id, receiver_id, sh

Starting with

$$t_5[\text{shipment\_id}] = a,\ t_5[\text{package\_id}] = b,\ t_5[\text{sender\_id}] = c,$$
$$t_5[\text{receiver\_id}] = d,\ t_5[\text{shipping\_method}] = e,\ t_5[\text{tracking\_number}] = f,$$
$$t_5[\text{status}] = g.$$

Applying FD1 unifies b, c, d, e, f, g with a, and FD2 unifies a, b, c, d, e, g with f.
Lossless Join: Yes. **Dependency Preservation**: Yes.

IX. Differences Between 3NF and BCNF

Third Normal Form (3NF) and Boyce-Codd Normal Form (BCNF) are both designed to eliminate redundancy and dependency anomalies in relational databases. A relation is in 3NF if it is already in 2NF and every non-prime attribute depends only on a superkey, or the dependency is trivial (i.e., the determinant is part of a candidate key). This means that 3NF allows functional dependencies where the determinant is not necessarily a superkey, as long as the dependency is trivial or the determinant is a candidate key.

Table	Lossless Join	Dependency Preservation
Packages	Yes	Yes
Receivers	Yes	Yes
Senders	Yes	Yes
Service Areas	Yes	Yes
Shipments	Yes	Yes
Claims	Yes	Yes
Customer Feedback	Yes	Yes
Delivery Attempts	Yes	Yes
Delivery Locations	Yes	Yes
Shipping Rates	Yes	Yes

TABLE III
ANALYSIS OF LOSSLESS JOIN AND DEPENDENCY PRESERVATION FOR ALL TABLES

In contrast, BCNF is stricter than 3NF. A relation is in BCNF if, for every functional dependency $X \rightarrow Y$, X is a superkey. This implies that no functional dependency can exist where the determinant is not a superkey. Thus, while 3NF relaxes its criteria to allow dependencies involving candidate keys, BCNF eliminates any such exceptions.

X. EVALUATION OF RELATIONS FOR 3NF

Each relation in the schema was evaluated to determine whether it could remain in 3NF instead of BCNF. Below is the detailed evaluation for each relation:
The **Packages** table has the functional dependency package_id $\rightarrow$ {weight, length, width, height, content_description}. Since package_id is a superkey, this table satisfies both BCNF and 3NF. Similarly, the **Receivers** table has the functional dependency receiver_id $\rightarrow$ {name, address, contact_number}, where receiver_id is a superkey. Therefore, this table is also in both BCNF and 3NF.
The **Senders** table has the functional dependency sender_id $\rightarrow$ {name, address, contact_number}, with sender_id as a superkey. The **ServiceAreas** table has the dependency area_id $\rightarrow$ {service_area_name, coverage_description, is_active}, where area_id is a superkey. In both cases, these tables satisfy BCNF and 3NF without any differences.

A. Shipments

The **Shipments** table is defined by two functional dependencies:

$$\text{shipment\_id} \rightarrow \left\{ \begin{array}{l} \text{package\_id, sender\_id, receiver\_id,} \\ \text{shipping\_method, tracking\_number, status} \end{array} \right.$$

and

$$\text{tracking\_number} \rightarrow \left\{ \begin{array}{l} \text{shipment\_id, package\_id, sender\_id,} \\ \text{receiver\_id, shipping\_method, status} \end{array} \right.$$

Both shipment_id and tracking_number are candidate keys, ensuring no non-prime attributes depend on another non-prime attribute. Thus, this table satisfies both BCNF and 3NF.

B. Claims

The **Claims** table has the dependency:

$$\text{claim\_id} \rightarrow \left\{ \begin{array}{l} \text{shipment\_id, claim\_status, claim\_date,} \\ \text{resolution\_date, amount\_claimed} \end{array} \right.$$

with claim_id as a superkey.

C. Customer Feedback

The **CustomerFeedback** table has the dependency:

$$\text{feedback\_id} \rightarrow \left\{ \begin{array}{l} \text{shipment\_id, customer\_id, rating, comments,} \\ \text{feedback\_date} \end{array} \right.$$

where feedback_id is a superkey. In both cases, the relations meet the stricter criteria of BCNF, so there is no difference between 3NF and BCNF compliance.

D. Delivery Attempts and Locations

The **DeliveryAttempts** table has the functional dependency:

$$\text{attempt\_id} \rightarrow \left\{ \text{shipment\_id, attempt\_date, attempt\_status, notes} \right.$$

with attempt_id as a superkey. Similarly, the **DeliveryLocations** table has:

$$\text{location\_id} \rightarrow \left\{ \text{shipment\_id, longitude, latitude, delivery\_date} \right.$$

with location_id as a superkey. Both tables satisfy BCNF as well as 3NF.

XI. SUMMARY AND JUSTIFICATION FOR BCNF RETENTION

In summary, all the relations in this schema already satisfy BCNF, which is a stricter normal form than 3NF. No non-prime attributes depend on other non-prime attributes, meaning there is no advantage to relaxing the normalization criteria to 3NF. BCNF eliminates the possibility of redundancy and dependency anomalies by enforcing stricter rules. Since all relations comply with BCNF, there is no need to relax the schema to 3NF. Keeping the relations in BCNF ensures better data integrity and consistency in the database.

XII. REVISED ENTITY - RELATIONSHIP DIAGRAM

A. Entities

The entities in the schema remain unchanged as they have satisfied BCNF requirements, ensuring the least decomposition necessary. The schema is already normalized to eliminate partial and transitive dependencies, making it robust and efficient. The primary entities are Packages, Receivers, Senders, Service Areas, Shipments, Claims, Customer Feedback, Delivery Attempts, and Delivery Locations.

B. Relations and Attributes

The relations and their attributes also remain consistent with the previous schema, as the design meets BCNF requirements. Each table is normalized with its respective primary keys and attributes. These relations are mentioned in Table I.

C. Relationship Between Entities

The relationships remain unchanged, with Shipments linking Packages, Senders, and Receivers. Claims are associated with Shipments, while Customer Feedback connects to Shipments and optionally to Senders. Delivery Attempts and Delivery Locations also reference Shipments. These relationships are enforced through foreign key constraints, ensuring data consistency and referential integrity.

D. Constraints

1) Primary Keys: Every entity includes a primary key to uniquely identify records within the schema. Examples of primary keys include package_id in the Packages table, shipment_id in the Shipments table, and claim_id in the Claims table. These primary keys ensure that each record is uniquely identifiable and prevent duplication within the respective tables.

2) Foreign Keys: Relationships between entities are enforced using foreign key constraints. These constraints ensure referential integrity by linking related tables. For instance, package_id in the Shipments table references the primary key in the Packages table, while shipment_id in the Claims table references the primary key in the Shipments table. These relationships allow for efficient data retrieval and ensure consistency across the schema.

3) Unique Constraints: Attributes that require uniqueness, such as tracking_number in the Shipments table, are defined with unique constraints. This ensures that no duplicate values are entered for critical fields, preventing data inconsistencies and maintaining the integrity of the schema.

4) Not Null Constraints: Essential fields such as package_id, receiver_id, and sender_id are defined as NOT NULL. This ensures that these attributes cannot have missing values, which is critical for maintaining the integrity of relationships and ensuring the completeness of data within the schema.

5) Check Constraints: Logical constraints are implemented to ensure data validity. For example, rating in the Customer Feedback table must have a value between 1 and 5, amount_claimed in the Claims table must be greater than zero, and base_price in the Shipping Rates table must be a positive value. These constraints enforce rules on the values allowed within specific fields, preventing invalid or out-of-range data from being entered.

6) Default Values: Certain attributes are assigned default values to ensure that records are initialized with meaningful data. For instance, the status field in the Shipments table defaults to ”Pending,” providing a predefined state for new records. This simplifies data entry and ensures consistency.

7) Cascade Actions: Cascading deletes are implemented to maintain referential integrity. For example, if a parent record such as a Shipment is deleted, all related records in dependent tables like Claims, Delivery Attempts, and Delivery Locations are automatically removed. This ensures that the database remains free of orphaned records and maintains consistency across all relationships.

E. Redundancy Analysis

Since all tables comply with BCNF, the schema minimizes redundancy. Functional dependencies ensure that no non-prime attribute depends on anything other than a superkey. With proper normalization, data anomalies are avoided, and the design is optimized for both storage efficiency and data consistency.

XIII. Challenges and Optimization Techniques for Managing Large Datasets

A. Problems Encountered

Working with larger datasets presented several challenges that impacted performance and usability. One of the primary issues was the increase in query execution time due to the sheer size of the tables and the complexity of the queries. For example, retrieving data from the **Shipments** table, which contains relationships with **Packages**, **Senders**, and **Receivers**, became significantly slower as the number of rows increased. This was especially true for operations involving joins and filters on non-indexed attributes, such as searching by tracking_number or filtering shipments by status and receiver_id.

Another problem was memory utilization during data retrieval. Full table scans for frequently queried fields led to increased resource consumption, slowing down other database operations. These performance issues made it clear that optimization was required to handle the growing volume of data efficiently.

B. Adopting Indexing Concepts

To address these issues, indexing was implemented on frequently queried fields and keys to optimize database performance. For instance, a unique index was created on tracking_number in the **Shipments** table, as this field was often used to retrieve specific shipment details. Similarly, non-clustered indexes were added to attributes such as receiver_id and status in the **Shipments** table, which were commonly used in filter conditions.

In addition, composite indexes were introduced for multi-column queries. For example, a composite index on shipment_id, package_id in the **Shipments** table improved the performance of queries involving both fields in join operations or WHERE clauses.

C. Outcome and Lessons Learned

Implementing indexing significantly reduced query execution time and optimized memory usage. Queries that previously took several seconds to execute were now completed in milliseconds. The adoption of indexing also minimized the need for full table scans, reducing resource utilization and improving overall system responsiveness.

However, indexing introduced trade-offs, such as increased storage requirements for maintaining the indexes and slightly slower write operations due to the need to update the indexes. These were deemed acceptable compared to the performance gains in read-heavy operations. The experience

highlighted the importance of analyzing query patterns and selecting appropriate indexes tailored to the workload. It also emphasized the need for periodic index maintenance, such as reindexing and monitoring unused indexes, to ensure sustained performance.

XIV. SQL Operations and Query Execution Results

A. Insertion Queries

For the Insert Query - I, we inserted a row in the Packages table as seen in Fig . 22.

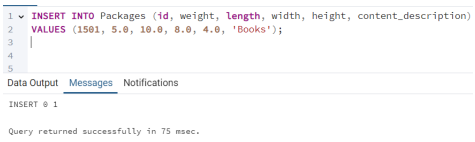


Fig. 22. Insert Query - I

For the Insert Query - II, we inserted a row in the Senders table as seen in Fig . 23.

Fig. 23. Insert Query - II

B. Update Queries

For the first update query, we updated the content description of the package id '125' to 'Electronics' from 'Books - Hardcover'.

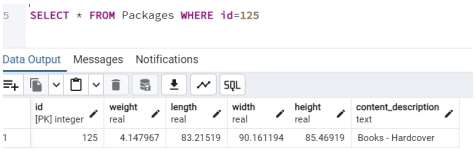


Fig. 24. Update Query - I (Pre update)

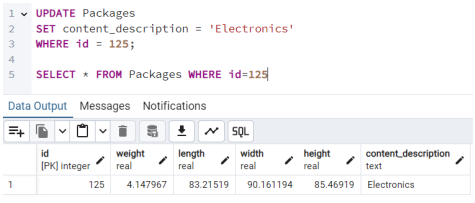


Fig. 25. Update Query - I (Post update)

For the second update query, we increased the base price present in the table 'ShippingRates' by 10% for all the rows where the given weight limit was more than the average weight limit of all the shipping methods.

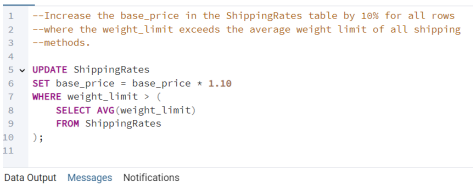


Fig. 26. Update Query - II

C. Select Queries

In query 1, we retrieve detailed information about shipments, including sender and receiver details, where the shipment status is not "Delivered."

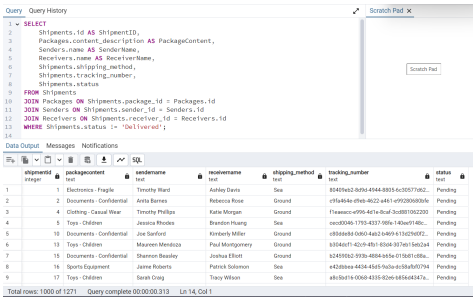


Fig. 27. Query - I

In query 2, we get the total number of shipments delivered and pending, grouped by shipping method.

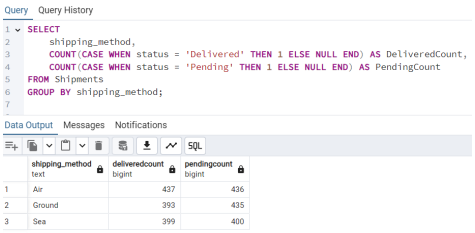


Fig. 28. Query - II

In query 3, we get the top 5 highest-rated feedback comments for shipments, including sender names and feedback details.

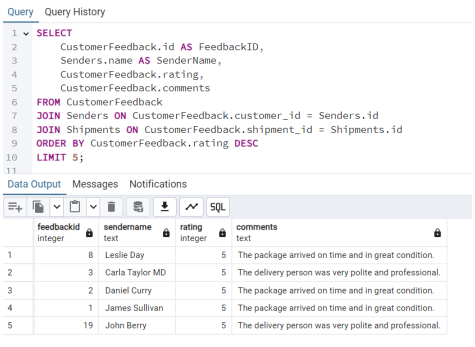
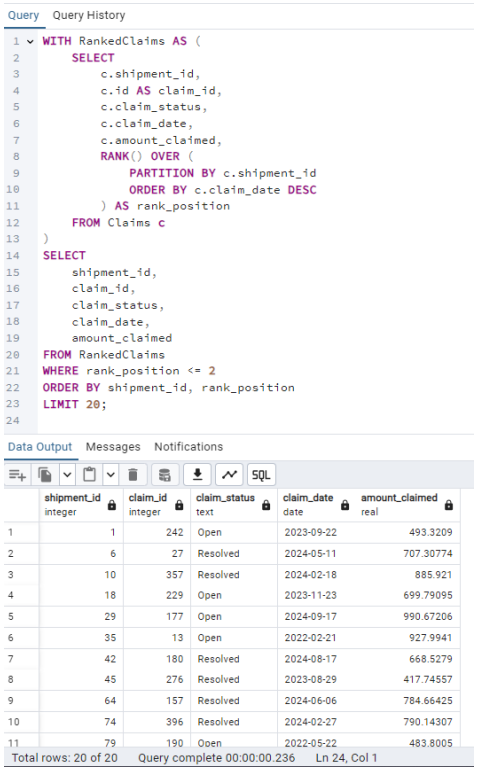


Fig. 29. Query - III

In query 4, we find the two most recent claims filed for each shipment and rank them by claim date.



```
WITH RankedClaims AS (
    SELECT
        c.shipment_id,
        c.id AS claim_id,
        c.claim_status,
        c.claim_date,
        c.amount_claimed,
        RANK() OVER (
            PARTITION BY c.shipment_id
            ORDER BY c.claim_date DESC
        ) AS rank_position
    FROM Claims c
)
SELECT
    shipment_id,
    claim_id,
    claim_status,
    claim_date,
    amount_claimed
FROM RankedClaims
WHERE rank_position <= 2
ORDER BY shipment_id, rank_position
LIMIT 20;
```

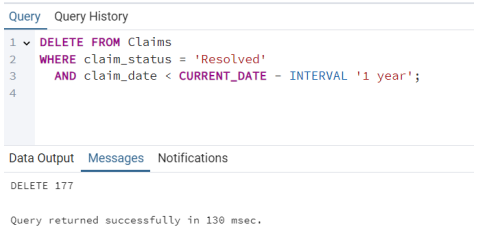
	shipment_id integer	claim_id integer	claim_status text	claim_date date	amount_claimed real
1	1	242	Open	2023-09-22	493.3209
2	6	27	Resolved	2024-05-11	707.30774
3	10	357	Resolved	2024-02-18	885.921
4	18	229	Open	2023-11-23	699.79095
5	29	177	Open	2024-09-17	990.67206
6	35	13	Open	2022-02-21	927.9941
7	42	180	Resolved	2024-08-17	668.5279
8	45	276	Resolved	2023-08-29	417.74557
9	64	157	Resolved	2024-06-06	784.66425
10	74	396	Resolved	2024-02-27	790.14307
11	79	190	Open	2022-05-22	483.8005

Total rows: 20 of 20 Query complete 00:00:00.236 Ln 24, Col 1

Fig. 30. Query - IV

D. Deletion Queries

In the first deletion query, we removed entries from the Claims table for claims that are older than one year and have been resolved.



```
DELETE FROM Claims
WHERE claim_status = 'Resolved'
  AND claim_date < CURRENT_DATE - INTERVAL '1 year';
```

DELETE 177

Query returned successfully in 130 msec.

Fig. 31. Deletion Query - I

In the second deletion query, we delete claims where the claimed amount exceeds the shipment's base price.

```
DELETE FROM Claims
WHERE shipment_id IN (
    SELECT Shipments.id
    FROM Shipments
    JOIN ShippingRates ON Shipments.shipping_method = ShippingRates.shipping_method
    WHERE Claims.amount_claimed > ShippingRates.base_price
);
```

DELETE 318

Query returned successfully in 615 msec.

Fig. 32. Deletion Query - II

XV. Query Execution Analysis

- Targeted Indexing: We created indexes on key columns that are often used for joins, filters, and aggregations. For example, we added indexes to sender_id, receiver_id, and shipment_id in the Shipments table, and claim_status and feedback_date in the Claims and CustomerFeedback tables.
- Faster Queries: By adding these indexes, we helped PostgreSQL skip the slow process of scanning the entire table. Instead, the database can now find the right data much faster, cutting down on execution time.
- Smoother Joins: Indexes on foreign key columns made the joins between tables like Shipments, Claims, and CustomerFeedback quicker and more efficient, which is key when pulling together data from different sources.
- Better Filtering and Aggregations: When filtering data or running aggregation queries (like counting or averaging), indexes on columns like claim_status and feedback_date sped things up by quickly narrowing down the results.
- Less Strain on the System: Overall, by reducing the time it takes to run queries, we reduced the load on the system, which means it can handle more data and respond faster, even as things scale up.

A. Query 1

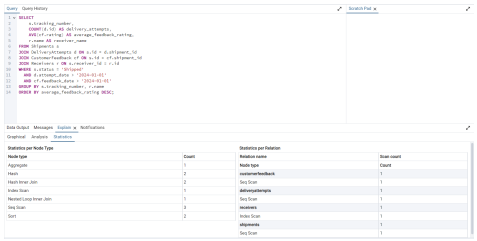


Fig. 33. Query - I (Pre Improvement)

```
SELECT
    s.tracking_number,
    COUNT(d.id) AS delivery_attempts,
    AVG(cf.rating) AS average_feedback_rating,
    r.name AS receiver_name
FROM
    Shipments s
JOIN
    DeliveryAttempts d ON s.id = d.shipment_id
JOIN
    CustomerFeedback cf ON s.id = cf.shipment_id
JOIN
    Receivers r ON s.receiver_id = r.id
WHERE
    s.status = 'Shipped'
    AND d.attempt_date > '2024-01-01'
    AND cf.feedback_date > '2024-01-01'
GROUP BY
    s.tracking_number, r.name
ORDER BY
    average_feedback_rating DESC;
```

Data Output Messages Explain Notifications

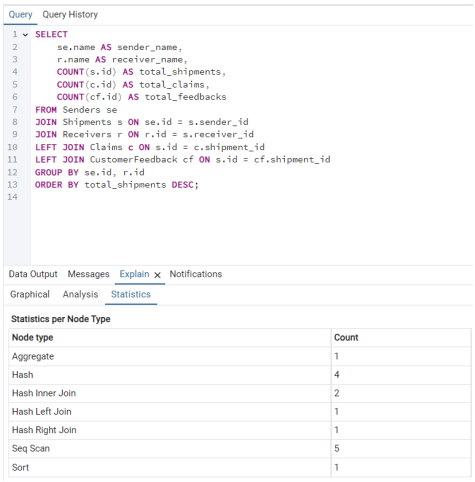
Graphical Analysis Statistics

Statistics per Node Type

Node type	Count
Aggregate	1
Index Scan	4
Nested Loop Inner Join	3
Sort	2

Fig. 34. Query - I (Post Improvement)

B. Query 2



```
SELECT
    se.name AS sender_name,
    r.name AS receiver_name,
    COUNT(s.id) AS total_shipments,
    COUNT(c.id) AS total_claims,
    COUNT(cf.id) AS total_feedbacks
FROM Senders se
JOIN Shipments s ON se.id = s.sender_id
JOIN Receivers r ON r.id = s.receiver_id
LEFT JOIN Claims c ON s.id = c.shipment_id
LEFT JOIN CustomerFeedback cf ON s.id = cf.shipment_id
GROUP BY se.id, r.id
ORDER BY total_shipments DESC;
```

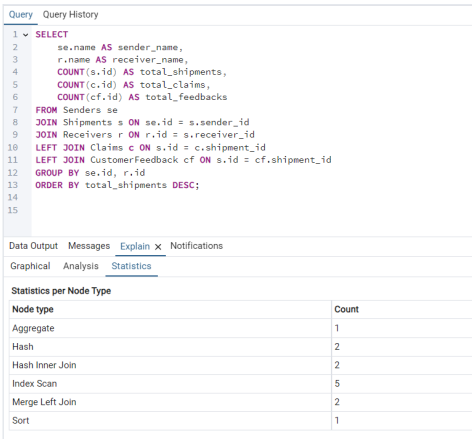
Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	4
Hash Inner Join	2
Hash Left Join	1
Hash Right Join	1
Seq Scan	5
Sort	1

Fig. 35. Query - II (Pre Improvement)



```
SELECT
    se.name AS sender_name,
    r.name AS receiver_name,
    COUNT(s.id) AS total_shipments,
    COUNT(c.id) AS total_claims,
    COUNT(cf.id) AS total_feedbacks
FROM Senders se
JOIN Shipments s ON se.id = s.sender_id
JOIN Receivers r ON r.id = s.receiver_id
LEFT JOIN Claims c ON s.id = c.shipment_id
LEFT JOIN CustomerFeedback cf ON s.id = cf.shipment_id
GROUP BY se.id, r.id
ORDER BY total_shipments DESC;
```

Data Output | Messages | Explain | Notifications

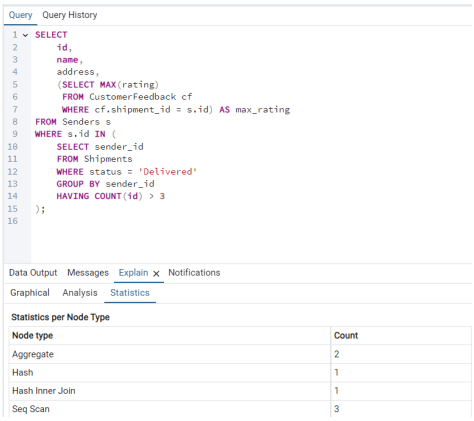
Graphical | Analysis | Statistics

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	2
Hash Inner Join	2
Index Scan	5
Merge Left Join	2
Sort	1

Fig. 36. Query - II (Post Improvement)

C. Query 3



```
SELECT
    id,
    name,
    address,
    (SELECT MAX(rating)
     FROM CustomerFeedback cf
     WHERE cf.shipment_id = s.id) AS max_rating
FROM Senders s
WHERE s.id IN (
    SELECT sender_id
    FROM Shipments
    WHERE status = 'Delivered'
    GROUP BY sender_id
    HAVING COUNT(id) > 3
);
```

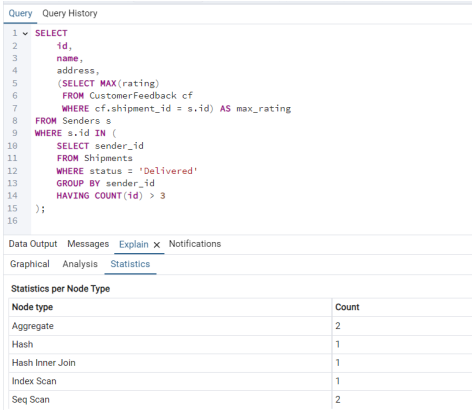
Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

Statistics per Node Type

Node type	Count
Aggregate	2
Hash	1
Hash Inner Join	1
Seq Scan	3

Fig. 37. Deletion Query - III (Pre Improvement



```
SELECT
    id,
    name,
    address,
    (SELECT MAX(rating)
     FROM CustomerFeedback cf
     WHERE cf.shipment_id = s.id) AS max_rating
FROM Senders s
WHERE s.id IN (
    SELECT sender_id
    FROM Shipments
    WHERE status = 'Delivered'
    GROUP BY sender_id
    HAVING COUNT(id) > 3
);
```

Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

Statistics per Node Type

Node type	Count
Aggregate	2
Hash	1
Hash Inner Join	1
Index Scan	1
Seq Scan	2

Fig. 38. Deletion Query - III (Post Improvement

XVI. Bonus Task: Web Application for Database Query Visualization

Below are a few screenshots of the running website required for the bonus task.

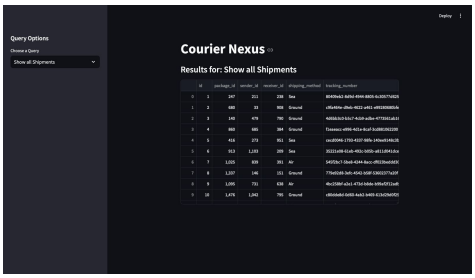


Fig. 39.

Fig. 40.

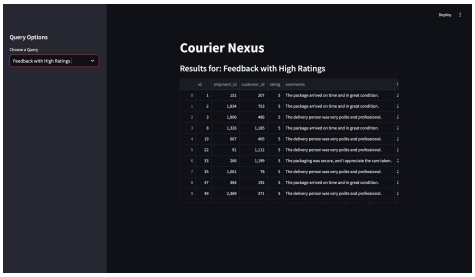


Fig. 41.

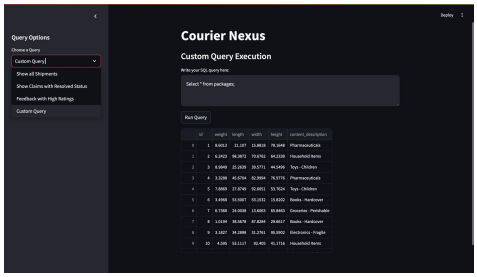

Deploy
Query Options
Choose a Query
Custom Query
Show all Shipments
Show Claims with Resolved Status
Feedback with High Ratings
Custom Query
Courier Nexus
Custom Query Execution
Write your SQL query here:
Select * from packages;
Run Query
id weight length width height content_description
1 8.6013 22.187 13.8818 78.2498 Pharmaceuticals
2 6.2423 86.3871 73.6761 64.2338 Household Items
3 8.5949 21.2628 39.5771 44.5456 Toys - Children
4 3.3288 45.0764 82.3994 76.9776 Pharmaceuticals
5 7.6603 27.8745 92.6051 53.7024 Toys - Children
6 5.4968 53.5087 93.1531 15.8202 Books - Hardcover
7 6.7588 34.0038 13.6063 65.8403 Groceries - Perishable
8 1.0194 86.9878 87.8284 29.8017 Books - Hardcover
9 5.1927 34.2898 31.2761 86.9902 Electronics - Fragile
10 4.595 53.1117 92.401 41.1716 Household Items

Fig. 42.

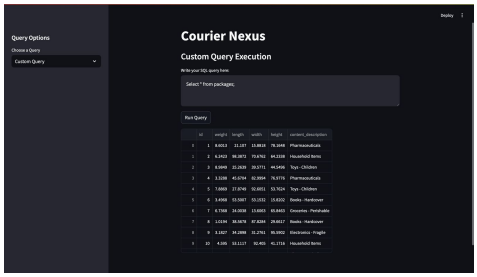

Deploy
Query Options
Choose a Query
Custom Query
Courier Nexus
Custom Query Execution
Write your SQL query here:
Select * from packages;
Run Query
id weight length width height content_description
1 8.6013 22.187 13.8818 78.2498 Pharmaceuticals
2 6.2423 86.3871 73.6761 64.2338 Household Items
3 8.5949 21.2628 39.5771 44.5456 Toys - Children
4 3.3288 45.0764 82.3994 76.9776 Pharmaceuticals
5 7.6603 27.8745 92.6051 53.7024 Toys - Children
6 5.4968 53.5087 93.1531 15.8202 Books - Hardcover
7 6.7588 34.0038 13.6063 65.8403 Groceries - Perishable
8 1.0194 86.9878 87.8284 29.8017 Books - Hardcover
9 5.1927 34.2898 31.2761 86.9902 Electronics - Fragile
10 4.595 53.1117 92.401 41.1716 Household Items

Fig. 43.