

Table of Contents

Abstract.....	2
Overview of SQL & NoSQL.....	2
SQL.....	2
NoSQL.....	3
Comparison of SQL and NoSQL.....	4
Scalability	4
Performance	4
Cloud Usage.....	4
Data Models	4
Queries and Analytics	5
Security	5
Summary	5
Performance Tuning	5
1. Query for Rental Information	5
2. Query for Bicycle Availability	6
3. Query for Service or Repair History	6
Transaction.....	7
How the Transactions Solve the Problems	7
Problem A: Missing Return Dates in Repair Bookings	7
Problem B: Invalid Bicycle Hire Schedules	7
Why It Works	7
References	8
Appendix 1: ERD Diagram	9
Appendix 2: Problem A – Booking Repair Transaction Code and Execution.....	9
Appendix 3: Problem B – Booking Repair Transaction Code and Execution.....	10

Abstract

This report looks at the differences between SQL and NoSQL databases, helping Pronto Parcels figure out which one works best for their needs. It also dives into ways to speed up Eco Bike Hubs' database by suggesting secondary indexes for their most important queries. Lastly, I talk about how I implemented a couple of transactions to fix issues like double-booking bikes and missing return dates, ensuring things run smoother at Eco Bike Hubs.

Overview of SQL & NoSQL

SQL

The story of SQL (Structured Query Language) begins with the groundbreaking work of E.F. Codd, who introduced the concept of the relational data model in the early 1970s. This model fundamentally changed how data was stored and retrieved by offering a more structured and logical approach. Inspired by Codd's ideas, Donald Chamberlin and Ray Boyce set out to create a language that was both powerful and accessible. Their goal was to design something user-friendly for people who didn't have a strong background in mathematics or programming. Chamberlin stated that they wanted the language to be simple enough that ordinary people could “walk up and use it” with a minimum of training [1].

Their early efforts at IBM led to the development of a language called Sequel, later renamed SQL due to trademark issues. Unlike earlier query languages, which required users to navigate complex networks of pointers, SQL adopted a declarative approach. This meant users could describe what they wanted to retrieve without worrying about how the system would find it. This approach made SQL much easier to use, allowing it to gain traction quickly [1].

By 1986, SQL became a formal standard recognised by ANSI and ISO, and its adoption spread globally. Over time, it evolved to include features like outer joins, recursion, and analytic processing capabilities, making it a versatile tool for managing persistent data [1]. Today, SQL remains a cornerstone of database management, underscoring the significance of Codd's original vision and the innovative work by Boyce and Chamberlin. It's incredible how these early ideas laid the groundwork for systems we still rely on today.

At its core, SQL is a language designed to manage relational databases. A relational database organises data into tables, which consist of rows and columns. Each table represents a different type of data, like *employees* or *pets*, and each column holds specific attributes about the data, such as names, dates, or numbers. SQL allows users to perform key operations, like creating tables, querying data, and updating records. A major strength of SQL is its ability to easily combine multiple tables, enabling users to answer complex questions by linking related data across different sources. For example, a database of pets and veterinarians can be queried to find out which vets can treat a specific pet or which pets need care [2].

SQL's declarative nature allows users to specify what they want to retrieve (e.g., “find all dogs”) without needing to know how to execute the steps to get that data. This makes the language both user-friendly and efficient for querying and managing relational data [2].

An essential feature of SQL databases is their support for **ACID** properties—**Atomicity**, **Consistency**, **Isolation**, and **Durability**. These principles ensure that transactions are processed reliably and maintain the integrity of the data. For example, **Atomicity** ensures that transactions are all-or-nothing, **Consistency** guarantees the database remains in a valid state, **Isolation** allows concurrent transactions to operate

without interfering with each other, and **Durability** ensures that once a transaction is committed, its effects are permanent, even in the event of a system failure [3].

NoSQL

NoSQL databases, introduced in 2009, stand for “Not Only SQL.” While the name might sound like a rejection of traditional relational databases, it’s more about providing alternatives that address the needs of modern applications. Instead of abandoning SQL entirely, NoSQL complements relational systems by offering scalable, flexible, and distributed solutions. These databases are especially valuable in Big Data and Cloud Computing environments, where the limitations of traditional systems become apparent [4].

Relational databases—built on fixed schemas, tables, and ACID properties—are powerful but not without challenges. Issues like limited data types, non-optimal join operations, and difficulties maintaining ACID compliance in distributed settings often arise. For tech giants like Google, Amazon, and Facebook, these limitations were deal-breakers as their systems needed to scale to handle massive datasets. Their answer was to create purpose-built solutions like Google’s Bigtable, Amazon’s DynamoDB, and Facebook’s Cassandra, which laid the foundation for the NoSQL movement [4].

One of the driving forces behind NoSQL’s architecture is Eric Brewer’s CAP theorem [5]. It states that in distributed systems, you can only guarantee two out of three properties: Consistency, Availability, and Partition Tolerance. Most NoSQL databases prioritise Partition Tolerance and Availability (PA systems). This ensures uninterrupted operations, even during network partitions. However, this often means sacrificing strict consistency in some scenarios [4].

The primary interests driving NoSQL systems include:

- Providing alternatives to relational databases to address the challenges of managing Big Data.
- Moving away from tabular representation of data and the SQL language, enabling more flexible data modelling approaches.
- Manipulating massive amounts of distributed data, which is increasingly common in modern applications.
- Focusing on performance, including reduced latency, high throughput, extensibility, and availability, particularly in real-time systems.
- Incorporating parallel file systems to enhance data processing efficiency across distributed environments [4].

These drivers address challenges like scaling for e-commerce platforms or enabling real-time analytics for social media services.

NoSQL databases come in four main categories, each suited to specific data needs:

- **Column-Oriented:** Stores data in columns, optimising query performance for analytical tasks.
- **Key-Value Oriented:** Focuses on storing data as key-value pairs, enabling fast lookups and writes.
- **Document-Oriented:** Manages data as semi-structured documents, often in formats like JSON or BSON, allowing flexibility in data representation.
- **Graph-Oriented:** Represents data as nodes and edges, suitable for applications like social networks or recommendation systems [4].

By redefining how we handle data, NoSQL has transformed modern computing, making it possible to meet the demands of today’s interconnected, data-driven world.

Comparison of SQL and NoSQL

For decades, SQL databases dominated data management with their structured approach, ACID compliance, and support for complex queries. However, the rise of unstructured data and scalability demands introduced NoSQL as a more flexible alternative. Unlike SQL, which relies on rigid schemas and centralised servers, NoSQL supports diverse data models, distributed architectures, and cost-effective scaling. This section explores key differences between these systems to help clarify their strengths and trade-offs [\[6\]](#).

Scalability

Relational databases (SQL) scale vertically, meaning they handle more load by upgrading the hardware of a single server—like adding more RAM or getting a faster CPU. However, this can get expensive and complex, especially when data needs to be distributed across multiple servers.

In contrast, NoSQL databases are built for horizontal scalability. Instead of upgrading one server, more servers can be added, spreading the load and making scaling easier and more cost-effective. This flexibility helps NoSQL systems handle the massive data volumes required by large-scale, distributed applications [\[6\]](#).

Performance

Relational databases ensure reliability and data integrity through ACID properties, which are crucial for applications like banking. However, enforcing these can be resource-intensive, causing performance drops with large datasets, especially in real-time or big data scenarios.

In contrast, NoSQL databases follow BASE principles (Basically Available, Soft-state, Eventually Consistent), trading strict consistency for flexibility and speed. This makes them ideal for handling both structured and unstructured data, particularly in applications like social media where rapid updates are more critical than perfect consistency. NoSQL systems also use more affordable hardware, unlike SQL systems that often require costly investments for scalability [\[6\]](#).

Cloud Usage

Relational databases are not ideal for cloud environments due to their reliance on ACID properties and limited scalability, making them unsuitable for handling massive semi-structured or unstructured datasets.

NoSQL databases, on the other hand, excel in cloud applications by offering better scalability, availability, and flexibility. They efficiently handle diverse data types—structured, semi-structured, and unstructured—while maintaining strong performance across distributed cloud infrastructures [\[6\]](#).

Data Models

Relational databases rely on a fixed schema, making changes difficult and costly. Modifying the schema can lead to performance issues or service disruptions, often requiring additional maintenance or adjustments to applications.

NoSQL databases, on the other hand, don't require rigid schemas. This allows for easy changes to the data model without affecting performance, making them adaptable to a wide range of use cases and evolving needs [\[6\]](#).

Queries and Analytics

Relational databases use SQL, a standardised and powerful language, which makes them great for complex queries and business intelligence tasks. They excel at performing joins and advanced analytics with a consistent interface.

NoSQL databases don't have a standard query language, which can make them harder to learn. While not as efficient for complex queries and joins, NoSQL excels at handling large, distributed datasets and parallel computations. This makes them ideal for big data tasks but less suited for traditional business intelligence [6].

Security

Relational databases come with strong security features like authentication, encryption, and auditing. However, they are still vulnerable to risks like SQL injection and weak communication protocols.

NoSQL databases focus more on scalability and flexibility, often lacking robust security mechanisms. Encrypting unstructured data is challenging, and sensitive info is often stored in plaintext. External tools are needed for authentication and secure communication. Some NoSQL solutions, like CouchDB, try to address these gaps, but their security measures are often not enough, making them less suitable for high-security applications like healthcare or government systems [6].

Summary

At the end of the day, whether you go with SQL or NoSQL really depends on what you're building. Need rock-solid consistency for financial transactions? SQL's your go-to. Handling a flood of messy, unstructured data in real-time? NoSQL is where it's at.

Performance Tuning

In this section, I will identify three important queries that Eco Bike Hubs may frequently execute and suggest appropriate secondary indexes to improve query performance. Each query is designed to address a key aspect of the business, such as rental bookings, bicycle availability, and service/repair tracking. For a detailed view of the tables and their relationships, please refer to [Appendix 1](#), which includes the Entity-Relationship Diagram (ERD).

1. Query for Rental Information

What it does: This query retrieves all rental bookings (using *booking_rental* table) for a specific customer, filtering by rental dates or return dates.

Columns to index:

- *customer_id*.
- *rental_date*.
- *return_date*.

Reason for indexing:

- *customer_id* will allow for fast lookup of all rental records for a specific customer.
- *rental_date* and *return_date* will help speed up queries when filtering by date ranges, such as finding all rentals in a specific month or year.

2. Query for Bicycle Availability

What it does: This query checks the availability of bicycles in a specific shop (using *company_bicycle* table), filtering by bicycle *status* (e.g., available, in repair, rented) and *category* (e.g., second-hand, brand new).

Columns to index:

- *status*.
- *category*.
- *shop_id*.

Reason for indexing:

- *status* will help quickly identify bicycles that are available for rental (as well as those that are rented, in repair, etc.).
- *category* is important for filtering based on the type of bicycle (e.g., second-hand or new).
- *shop_id* helps locate bicycles within a specific shop, which is important if the query is shop-specific.

3. Query for Service or Repair History

What it does: This query retrieves all service or repair bookings associated with a specific customer, filtering by status (e.g., pending, completed, in progress). Since *customer_id* is stored in the *customer_bicycle* table, we need to join this table to find the associated customer.

Columns to index:

- *bicycle_id* in *booking_service* and *booking_repair*.
- *status* in both *booking_service* and *booking_repair*.
- *customer_id* in *customer_bicycle* for linking to the customer.

Reason for indexing:

- *customer_id* in *customer_bicycle*: Since *customer_id* is not stored in the *booking_repair* or *booking_service* tables, indexing *customer_id* in *customer_bicycle* allows the query to quickly join with the *customer_bicycle* table to find the customer associated with the bicycle. This ultimately helps to quickly find all service or repair records for a given customer.
- *bicycle_id* in *booking_repair* and *booking_service* ensures that queries can efficiently find records for specific bicycles.
- *status* in *booking_repair* and *booking_service* allows filtering based on the progress of repairs or services, such as finding all “pending” or “completed” services.

Transaction

How the Transactions Solve the Problems

Problem A: Missing Return Dates in Repair Bookings

Eco Bike Hubs identified that repair bookings are sometimes created without registering the requested return date, leading to scheduling conflicts or incomplete records. To fix this, I added a return date column to the *booking_repair* table and then wrote a transaction (see [Appendix 2](#)) that checks if the return date is already being used before adding the booking.

Here's how it works:

1. **Lock the table:** The *LOCK TABLE booking_repair IN EXCLUSIVE MODE* stops anyone else from changing the table while we're checking.
2. **Check for conflicts:** The transaction looks for any booking with the same return date. If it finds one, it raises an error and rolls back so nothing is saved.
3. **Add the booking if clear:** If there's no conflict, it adds the new repair booking and commits the changes.

This makes sure no two bookings have the same return date, keeping things clear and organised.

Problem B: Invalid Bicycle Hire Schedules

This problem is about customers booking bikes for times when they're already rented out. To handle this, I created a transaction that checks if the bike is free before allowing the booking. The transaction for preventing rental booking conflicts is detailed in [Appendix 3](#).

Here's what it does:

1. **Lock the table:** The *LOCK TABLE booking_rental IN EXCLUSIVE MODE* makes sure no one else can mess with the table while we're checking.
2. **Look for overlaps:** It checks if the requested rental time overlaps with another booking for the same bike. I used the *OVERLAPS* operator to make this easier.
3. **Raise an error if there's a conflict:** If the bike is already booked, the transaction rolls back so no changes are saved.
4. **Add the booking if it's available:** If there's no conflict, the booking is added and committed.

This ensures bikes can't be double-booked, so schedules stay accurate.

Why It Works

By using transactions with locks, conflict checks, and rollback/commit behaviour, these solutions make sure:

- Repair bookings always have proper return dates without duplicates.
- Rental bookings don't overlap, so every bike is only rented out once at a time.

This keeps the system running smoothly and avoids the problems Eco Bike Hubs was having.

References

- [1] D. D. Chamberlin, "Early History of SQL," in *IEEE Annals of the History of Computing*, vol. 34, no. 4, pp. 78-82, Oct.-Dec. 2012, doi: 10.1109/MAHC.2012.61.
- [2] T. McGlynn and M. Santisteban, "Introduction to SQL," Goddard Space Flight Center, 2007.
- [3] S. Palanisamy and P. SuvithaVani, "A survey on RDBMS and NoSQL Databases MySQL vs MongoDB," *2020 International Conference on Computer Communication and Informatics (ICCCI)*, Coimbatore, India, 2020, pp. 1-7, doi: 10.1109/ICCCI48352.2020.9104047.
- [4] G. Belalem, H. Matallah and K. Bouamrane, "Evaluation of NoSQL Databases: MongoDB, Cassandra, HBase, Redis, Couchbase, OrientDB," *International Journal of Software Science and Computational Intelligence*, vol. 12, (4), pp. 71-91, 2020. Available: <https://www.proquest.com/scholarly-journals/evaluation-nosql-databases-mongodb-cassandra/docview/2932389113/se-2>. DOI: <https://doi.org/10.4018/IJSSCI.2020100105>.
- [5] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51-59, Jun. 2002, doi: 10.1145/564585.564601.
- [6] A. Oussous, F.-Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "NoSQL databases for big data," *International Journal of Big Data Intelligence*, vol. 4, no. 3, pp. 171-185, 2017, doi: 10.1504/IJBDI.2017.085537.
- [7] "DO statement," PostgreSQL Documentation. Accessed: Jan. 03, 2025. [Online]. Available: <https://www.postgresql.org/docs/current/sql-do.html>
- [8] "PL/pgSQL block structure," Neon Documentation. Accessed: Jan. 04, 2025. [Online]. Available: <https://neon.tech/postgresql/postgresql-plpgsql/plpgsql-block-structure>
- [9] "Random Data Generator and API Mocking Tool," Mockaroo. Accessed: Jan. 16, 2025. [Online]. Available: <https://www.mockaroo.com/>
- [10] "What is Entity Relationship Diagram (ERD)?," Visual Paradigm. Accessed: Jan. 16, 2025. [Online]. Available: <https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/>

Appendix 1: ERD Diagram

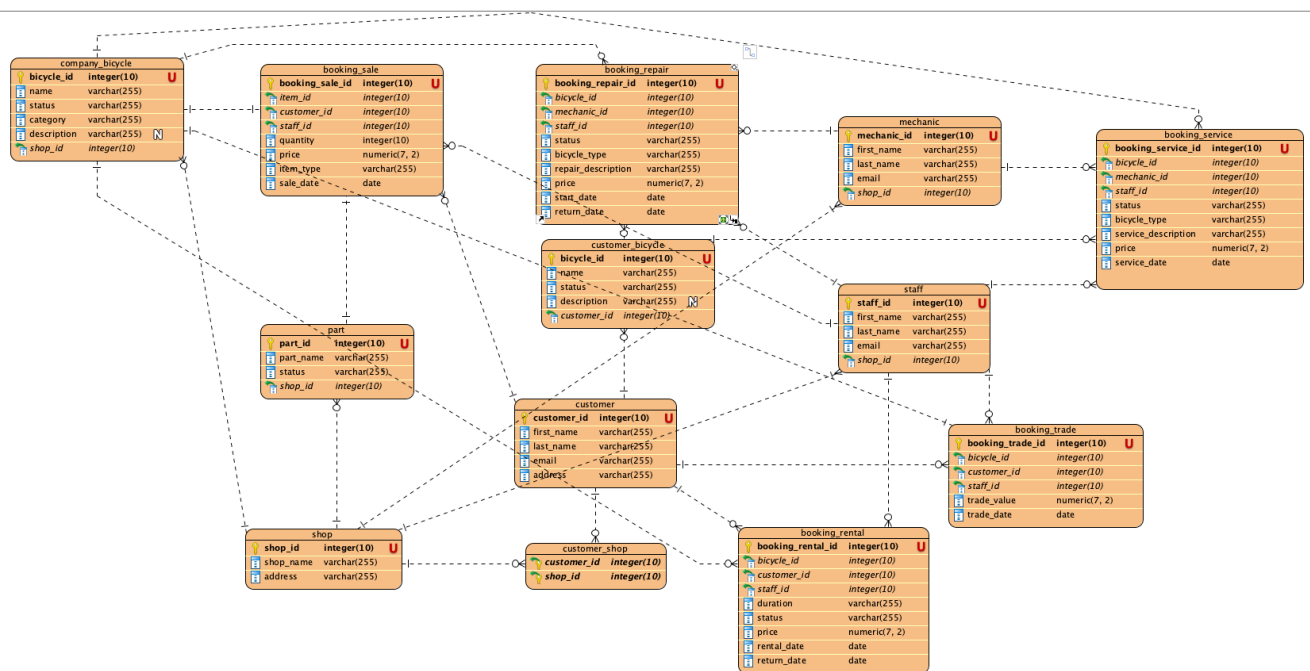


Figure 1: Screenshot of ERD Diagram

Appendix 2: Problem A – Booking Repair Transaction Code and Execution

```
DO $$
BEGIN
    -- Lock the booking_repair table to ensure consistency during the check
    LOCK TABLE booking_repair IN EXCLUSIVE MODE;

    -- Check if the return date already exists
    IF EXISTS (
        SELECT 1
        FROM booking_repair
        WHERE DATE(return_date) = '2025-01-15'
    ) THEN
        -- Conflict detected, raise an exception (transaction will automatically rollback)
        RAISE EXCEPTION 'The return date is already booked for another repair. Only one bike can be returned per
    ELSE
        -- No conflict, insert the new repair booking
        INSERT INTO booking_repair (bicycle_id, mechanic_id, staff_id, status, bicycle_type, repair_description,
        (8, 4, 4, 'pending', 'customer', 'Replace tire', 15.00, '2025-01-13 09:00:00', '2025-01-15 09:00:00');
        -- Commit the transaction if everything succeeds
        COMMIT;
    END IF;
END
$$ LANGUAGE plpgsql;
```

Figure 2: Screenshot of Booking Repair Transaction Code

Appendix 3: Problem B – Booking Repair Transaction Code and Execution

```
-- -- Booking - preventing rental time
DO $$
BEGIN
    -- Lock the booking_rental table to prevent concurrent modifications
    LOCK TABLE booking_rental IN EXCLUSIVE MODE;

    -- Check if the proposed rental period overlaps with an existing booking for the same bicycle
    IF EXISTS (
        SELECT 1
        FROM booking_rental
        WHERE bicycle_id = 1 -- Same bicycle
        AND (
            ('2025-01-21 09:00:00'::TIMESTAMP, '2025-01-21 17:00:00'::TIMESTAMP) OVERLAPS (rental_date, return_date)
        )
    ) THEN
        -- If conflict is detected, raise an exception (transaction will automatically rollback)
        RAISE EXCEPTION 'The bicycle is already booked for the selected time period.';
    ELSE
        -- If no conflict, insert the new rental record
        INSERT INTO booking_rental (
            bicycle_id, customer_id, staff_id, duration, status, price, rental_date, return_date
        )
        VALUES (
            1, 1, 1, 'full day', 'reserved', 50.00, '2025-01-21 09:00:00', '2025-01-21 17:00:00'
        );

        -- Commit the transaction if everything succeeds
        COMMIT;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Figure 3: Screenshot of Booking Rental Transaction Code