

Database Systems - Report

Justin Hendy

January 2025

Abstract

This report explores the differences between SQL and NoSQL databases, will look at database performance tuning through secondary indexes, and discuss the implementation of transactions within the Eco Bike Hubs database. The report will assess the benefits and drawbacks of each database type, the importance of secondary indexing, and the use of transactions to solve specific problems within a database.

Contents

1. SQL vs NoSQL
2. Eco Hubs - Performance Tuning: Secondary Indexes
3. Eco Hubs - Transactions
4. References
5. Appendix A - ERD
6. Appendix B - Query
7. Appendix C - Transaction 1
8. Appendix D - Transaction 2

SQL vs NoSQL

To clarify the debate between SQL and NoSQL database types, it is important first to explore and understand the key attributes and designs that define each database type.

SQL, or Structured Query Language, is used within programming to perform clear or specific tasks. Within database management and creation, SQL is used to complete tasks such as inserting, updating, deleting, and querying data held within a database; it is also the language used to write the document or schema that defines the attributes of the database [1]. A schema defines a database's table structure, data format, relationships or constraints and index structure. A database created and managed with SQL is called an SQL database, and the attributes and behaviours define it as a relational database. Relational databases form relationships between entities or tables, which then, in turn, contain data held within

columns and rows. Using primary and foreign keys allows a relational database to define the relationship between one entity and another, each key being a unique identifier. A schema produced for a relational database is deemed fixed and, therefore, has strict rules around the types of data it accepts; data of this type is called structured data, and relational databases work well with structured or semi-structured data [1]. Areas with strict rules around the use of data and a need for transactional actions and complex queries, such as banking/finance, government institutions, supply chains and retailers, benefit from the use of structured data and, therefore, generally require relational or SQL databases.

NoSQL, or Not only Structured Query Language, does not depend upon a fixed schema and can store and access unstructured data [2]. Unstructured data does not fit into one type of data model, such as ordered collections of mobile phone numbers or postcodes, but instead could be a range of data types, such as audio, video or photos. Having a flexible schema allows the NoSQL database to keep each piece of data in its original format and still have the facilities to store and access it. NoSQL can still action SQL-type queries but no longer holds a relational structure. Where SQL databases use tables with columns and rows to store each piece of structured data, NoSQL databases employ a range of structures depending on the data type [3].

- Document-based databases: index values assigned to each data point
- Key-value stores: similar to a relational database but with just two columns called key and value
- Column-oriented databases: hold data in columns instead of rows
- Graph-based databases: use nodes with links between nodes called relationships

NoSQL databases are helpful in various situations. Like SQL databases, NoSQL databases have uses within transactional situations, but where the transaction data is in various formats, such as police files with photos, audio recordings, and word-processed documents. Other areas where unstructured data storage is necessary include social networking sites, which collect and store user posts, photos, videos, and audio; online streaming platforms for audio and video; and smart home systems that must interact with and store information provided by various sensors or sources. This list is by no means exhaustive.

The applicability of each database will significantly depend on the needs of the system within which the database is to collect and store data. M. Eyada et al. studied the performance of both SQL and NoSQL within the Internet of Things (IoT) data management. IoT encompasses the range of devices and sensors connected to the internet, such as smart home systems. Within the study, the NoSQL performance drop as the workload increased was considerably less than that of the SQL database [1]. Overall, it is apparent that within the world of big data [3], such as the IoT, traditional relational databases suffer from performance issues [1].

In terms of responding to increased data collection requirements, SQL databases require what is known as vertical scaling, the adding of physical computer hardware to allow for increased capacity such as CPU, RAM or hard drives [3] with the expectation that more computers will not be needed, just more processing power within the machine itself. On the other hand, NoSQL requires horizontal scaling due to its distributed nature where additional servers or computers are added to the system [3]. Each has benefits, such as the database structure changes required to accommodate a horizontal scale compared to a vertical scale.

However, vertical scaling will hit a limit in capacity far sooner than horizontal scaling. Away from the physical considerations, NoSQL can produce quick results when dealing with simple queries, but struggle in dealing with complex queries when compared to traditional relational databases [2].

The examples show that each database type can be valuable given certain conditions and data management needs. The decision about which path to follow will significantly depend on the data structure, data types, size, query requirements, data use, and any foreseeable changes to the structure and use of the data.

Eco Hubs - Performance Tuning: Secondary Indexes

Within the Eco Bike Hubs database (ERD is in Appendix A), the following query will be executed frequently and benefit from secondary indexing.

- The number of jobs completed by each mechanic in each store over a given period

Managing the workloads of the mechanics in each store is important for ensuring an equitable distribution of work. Mechanics complete custom builds and service bookings, which can be "repairs" or "full services." The tables required to gather this information include staff, booking, service, and custom build. Key information from the booking table, such as booking type, booking date, closed, and last update, is required to run this query. All three columns are not primary keys and would benefit from secondary indexing. The booking table holds every booking that passes through each of the four shops; this will grow to be very large over time. Without indexing, the query determining the mechanics' workload will trigger a full scan of the booking table to link each key data piece. In order to optimise the query, indexing by booking type, date of booking, whether the booking has been closed, and the last update (which will be the date the booking closed) will allow for a faster, more efficient extraction of the data. The query for this search is in Appendix B.

Eco Hubs - Transactions

The first scenario involves recording dates concerning bicycle service/repair bookings, with the problem being the non-registration of specific return dates requested by customers. Upon further investigation and clarification the issue specifically concerned a scenario where two customers could not request the exact return date. Since the database design allowed multiple mechanics to be attached to each store, the transaction could check the date, staff, and completion status before inserting a booking and service request into the database.

The transaction written to solve this issue is in Appendix C. A DO command around the transaction implements Postgres procedural language, providing the ability to control whether or not a ROLLBACK or COMMIT query executes at the end of the transaction. Before the transaction starts via the BEGIN command, variables issued under a DECLARE command represent a new booking, specifically store ID, service type, staff ID, staff role, bicycle ID and requested return date. After the transaction begins, the service table is locked to prevent any simultaneous queries from being executed. Using the IF EXISTS command, the transaction checks whether the supplied values for the key attributes of staff ID and return date already match existing values within the service table, and if they do, a further check against the service complete status helps determine if the existing service has already

completed; if any of the defined values do not match or if the service has completed, then the booking with an attached service request is inserted, and a COMMIT command executes writing and storing the inserted values to the database. Suppose the two attributes match values in the database, and the complete status is false; a RAISE EXCEPTION command triggers and a description of the issue displays on the screen. The transaction executes a ROLLBACK; the database does not store the record. The printed statement includes helpful information, such as which staff ID and date are causing the issue.

The transaction addresses the problem by ensuring that only permitted dates appear in the database for new service bookings. The example provided in Appendix C, when actioned and given no booking with these details exist, will result in a record insertion, and any second attempt to book a bicycle with the same mechanic staff ID and return date will result in a rollback. Other constraints include checking if the mechanic, booking and bicycle all have the same store ID or stopping a service booking for a currently rented bicycle; either constraint will cause the insertion to fail.

The second scenario concerns hiring rental bicycles during prohibited periods. Further clarification reveals that this issue concerns explicitly hiring a rental bicycle when another customer has already hired it.

The transaction written to solve this issue is in Appendix D. The pattern is similar to the previous example: a DO command contains a transaction, and a range of variables are declared, representing the attempted rental booking. A BEGIN command is issued, starting the transaction, and the table is locked. An IF EXISTS command coupled with an OVERLAP command executes a check against a bicycle's start and return dates in the database, but only where the ID matches the bicycle ID provided as a declared variable. Further checks against the bicycle ID returned status confirm if the rental ended before the scheduled return date. If the bicycle return status is false and the rental dates overlap, a RAISE EXCEPTION command triggers a ROLLBACK, and no record insertion occurs. For further clarity, the RAISE EXCEPTION displays a message detailing that the bicycle is unavailable for rental due to a clash of dates. If there is no active rental for the provided bicycle ID or the dates do not overlap, a COMMIT executes, and the new booking and rental records are inserted and stored in the database.

This transaction addresses the problem by ensuring rentals cannot coincide for a single bicycle. In the example provided, assuming that the bicycle ID used has not been booked for a rental, running the transaction once will issue a COMMIT and store the booking; running the same transaction will ROLLBACK as clearly this bicycle is now booked for the given period. As with the previous example, additional check constraints and triggers stop bicycles from being rented or returned on a Sunday, and rentals can only start or complete between 8 am and 8 pm.

References

- [1] M. M. Eyada, W. Saber, M. M. El Genidy, and F. Amer, “Performance evaluation of iot data management using mongodb versus mysql databases in different cloud environments,” *IEEE Access*, vol. 8, pp. 110 656–110 668, 2020.
- [2] Y. Li and S. Manoharan, “A performance comparison of sql and nosql databases,” in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, Aug. 2013, pp. 15–19.
- [3] T. Capris, P. Melo, N. M. Garcia, I. M. Pires, and E. Zdravevski, “Comparison of sql and nosql databases with different workloads: Mongodb vs mysql evaluation,” in *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*. IEEE, Oct. 2022, pp. 214–218.

Appendix A - ERD

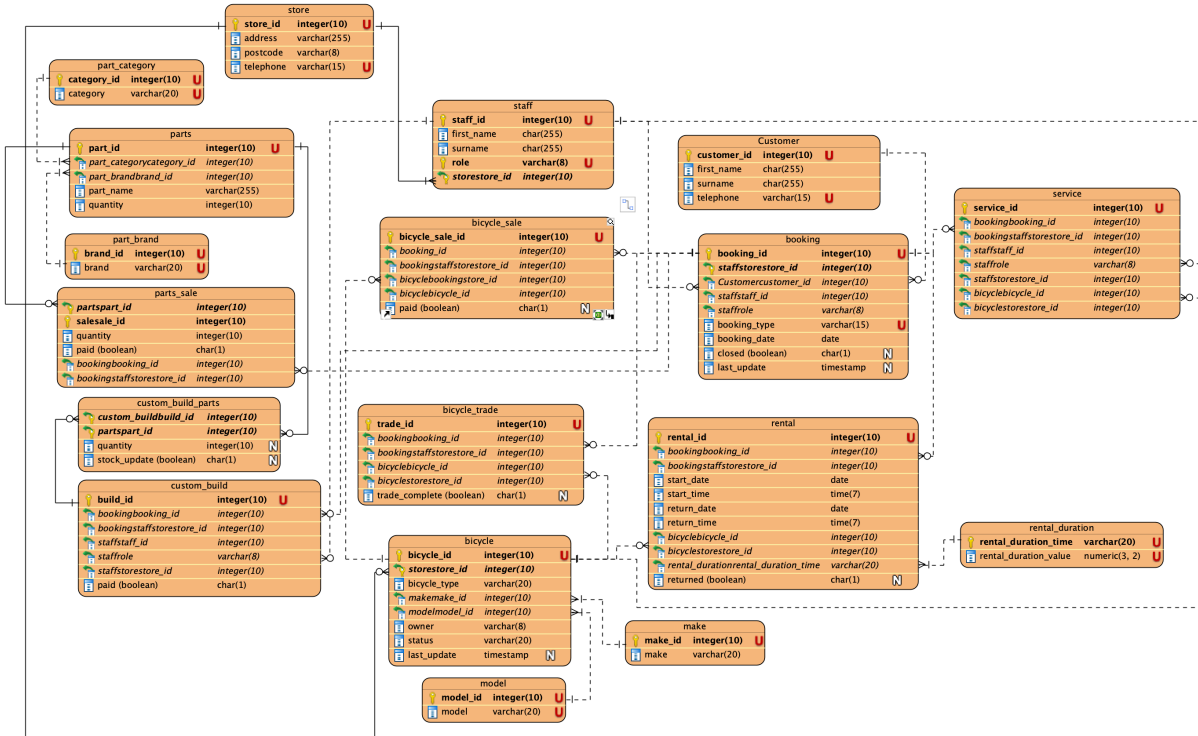


Figure 1: Eco Bike Hubs ERD

Appendix B - Query

```
EXPLAIN SELECT st.first_name, COUNT(*) AS job_count
FROM staff st
FULL OUTER JOIN service s ON st.staff_id = s.staff_id
FULL OUTER JOIN custom_build cb ON st.staff_id = cb.staff_id
FULL OUTER JOIN booking b ON cb.booking_id = b.booking_id OR s.booking_id = b.booking_id
WHERE b.closed = false
AND b.last_update <= '2025-01-31'
AND b.booking_date >= '2025-01-01'
AND (b.booking_type = 'custom build' OR b.booking_type = 'service')
GROUP BY st.first_name
```

Figure 2: Query

Appendix C - Transaction 1

```
1
2
3
4
5
6 -- a. Sometimes bicycle repair bookings are recorded but the details
7 -- of the specific return date(s) as requested by customers are not registered
   in the system.
8
9 -- NB
10 -- this requirement has been interpreted to mean that if a particular date has
   already been
11 -- booked for a service then the transaction should rollback
12 -- as there are multiple stores and multiple mechanics at each store this has
   been expanded
13 -- so that it will only rollback if the mechanic at particular store still has
   an active
14 -- service job, if that job has been closed then the new booking can still be
   added
15
16 -- STEPS
17 -- 1. execute script so that a new booking and service of type repair are
   generated
18 -- on first execution this will commit as no record of this type exists
19
20 -- 2. execute script again and a rollback will occur as this record has
   already been created
21
22
23 DO $$
24 -- allows for procedural language postgresql to be used, such as the if
   statement below
25
26
27 DECLARE
28 -- declare the details of the new service booking so that they can be checked
   as part of the if statement
29 new_store_id INT = 3;
30 new_service_type VARCHAR(255) = 'repair';
31 new_staff_id INT = 14;
32 new_staff_role VARCHAR(255) = 'mechanic';
33 new_bicycle_id INT = 26;
34 new_return_date DATE = '2025-02-10';
35
36
37 BEGIN
38 -- begin transaction
39
40 LOCK service;
41 -- locks the table inside the transaction,
42 -- LOCK defaults to IN EXCLUSIVE MODE and
43 -- prevents anyone from accessing this record
44 -- during the transaction
45
46 IF EXISTS (
```

```

47  -- checks if any record in the service table matches the new details
48  -- specifically checking the mechanic (staff_id)
49  -- the return date and whether the job has been completed
50  SELECT *
51  FROM service
52  WHERE staff_id = new_staff_id
53         AND return_date = new_return_date
54         AND complete = FALSE
55  ) THEN
56  RAISE EXCEPTION 'a service with staff id % on % is already booked',
new_staff_id, new_return_date;
57  -- if all three checks match rollback occurs and the statement above is
output
58  END IF;
59
60  INSERT INTO booking (staff_id, store_id, staff_role, customer_id,
booking_type)
61  VALUES (12,3,'sales',26,'service');
62  -- if any one of the above attributes does not match then insert the new
booking
63  -- with the given values (not declared at top as not needed for checking)
64
65  INSERT INTO service (
66  -- insert declared values into the attached service booking
67      booking_id,
68      store_id,
69      service_type,
70      staff_id,
71      staff_role,
72      bicycle_id,
73      return_date
74  )
75  VALUES (
76      (SELECT booking_id FROM booking ORDER BY booking_id DESC LIMIT 1),
77      -- sorts the booking table so that the most recent is at the top and
limits the return to one,
78      -- uses this id for the service booking
79      new_store_id,
80      new_service_type,
81      new_staff_id,
82      new_staff_role,
83      new_bicycle_id,
84      new_return_date
85      -- calls intially declared variables
86  );
87  COMMIT;
88  -- commits changes to the database
89  END;
90  -- releases table lock and ends transaction
91  $$ LANGUAGE plpgsql

```

Appendix D - Transaction 2

```
1  -- b. Customers are able to book bicycle hire times that are not registered
2  as possible schedules available.
3
4  -- NB
5  -- this requirement has been interpreted to mean that if a particular bike's
6  rental period
7  -- overlaps with a new booking then that bike cannot be rented
8  -- there are already other check constraints and triggers in place for when
9  a customer bike or
10 -- a bike in service cannot be rented
11
12 -- STEPS
13 -- 1. execute script so that a new booking and a week long rental are
14 generated
15 -- on first execution this will commit as no record of this type exists
16
17 -- 2. execute script again and a rollback will occur as this record has
18 already been created
19 -- you can experiment by amending the start date so that it starts within
20 the original week long booking
21 -- and it will still rollback
22
23 DO $$
24 -- allows for procedural language postgresql to be used, such as the if
25 statement below
26
27 DECLARE
28 -- declare the details of the new rental booking so that they can be checked
29 as part of the if statement
30 new_store_id INT = 2;
31 new_bicycle_id INT = 18;
32 new_duration VARCHAR(20) = 'one week';
33 new_start_date DATE = '2025-02-15';
34 new_start_time TIME(0) = '11:00';
35 new_return_date DATE = (new_start_date + new_start_time) + INTERVAL '1 day
36 ' * (
37     SELECT rental_duration_value
38     FROM rental_duration
39     WHERE rental_duration_time = new_duration); -- calculates the end date
40 using start date and duration
41 new_return_time TIME(0) = (new_start_time) + INTERVAL '1 day' * (
42     SELECT rental_duration_value
43     FROM rental_duration
44     WHERE rental_duration_time = new_duration); -- calculates the end time
45 using start date and duration
46
47 BEGIN
48 -- begin transaction
49
50 LOCK rental;
51 -- locks the table inside the transaction,
52 -- LOCK defaults to IN EXCLUSIVE MODE and
```

```

45  -- prevents anyone from accessing this record
46  -- during the transaction
47
48  IF EXISTS (
49  -- checks if any record in the rental table matches the new details
50  -- specifically checking the start and end dates of the rental
51  -- and whether the rental has already been returned
52      SELECT *
53      FROM rental
54      WHERE bicycle_id = new_bicycle_id
55            AND ((start_date + start_time)::timestamp, (return_date + return_time)
56               ::timestamp) OVERLAPS ((new_start_date + new_start_time)::timestamp, (
57               new_return_date + new_return_time)::timestamp)
58            AND returned = FALSE
59  ) THEN
60      RAISE EXCEPTION 'bicycle with id % is booked as a rental for part or all
61      of the selected dates', new_bicycle_id;
62      -- if all three checks match rollback occurs and the statement above is
63      output
64  END IF;
65
66  INSERT INTO booking (staff_id, store_id, staff_role, customer_id,
67  booking_type)
68  VALUES (6,new_store_id,'sales',26,'rental');
69  -- if any one of the above attributes does not match then insert the new
70  booking
71  -- with the given values (not declared at top as not needed for checking)
72
73  INSERT INTO rental (
74  -- insert declared values into the attached service booking
75  booking_id,
76  store_id,
77  bicycle_id,
78  duration,
79  start_date,
80  start_time
81  )
82  VALUES (
83  (SELECT booking_id FROM booking ORDER BY booking_id DESC LIMIT 1),
84  -- sorts the booking table so that the most recent is at the top and
85  limits the return to one,
86  -- uses this id for the rental booking
87  new_store_id,
88  new_bicycle_id,
89  new_duration,
90  new_start_date,
91  new_start_time
92  );
93  COMMIT;
94  -- commits changes to the database
95  END;
96  -- releases table lock and ends transaction
97  $$ LANGUAGE plpgsql

```