

# Database Systems

Nathan Hughes - 1809708

January 18, 2019

## Contents

|          |                                       |           |
|----------|---------------------------------------|-----------|
| <b>1</b> | <b>Database Design</b>                | <b>2</b>  |
| 1.1      | Assumptions . . . . .                 | 2         |
| 1.2      | Normalisation . . . . .               | 2         |
| 1.3      | Entity Relationship Diagram . . . . . | 4         |
| <b>2</b> | <b>Multi-user Databases</b>           | <b>5</b>  |
| 2.1      | ACID Transactions . . . . .           | 5         |
| 2.2      | Concurrency Control . . . . .         | 5         |
| 2.3      | Resource Locking . . . . .            | 6         |
| 2.4      | Database Security . . . . .           | 6         |
| 2.5      | Conclusion . . . . .                  | 6         |
| <b>3</b> | <b>SQL</b>                            | <b>7</b>  |
| 3.1      | Construction of Tables . . . . .      | 7         |
| 3.2      | Audit Tables . . . . .                | 7         |
| 3.3      | Testing . . . . .                     | 8         |
| 3.4      | Transaction Management . . . . .      | 9         |
| 3.5      | Performance Tuning . . . . .          | 11        |
| <b>4</b> | <b>MongoDB Databases</b>              | <b>12</b> |
| 4.1      | NoSQL . . . . .                       | 12        |
| 4.2      | MongoDB . . . . .                     | 12        |
| 4.2.1    | Data . . . . .                        | 12        |
| 4.2.2    | Data Model . . . . .                  | 12        |
| 4.2.3    | Queries . . . . .                     | 13        |
| 4.2.4    | Sharding . . . . .                    | 13        |
| 4.3      | Criticism of Mongo . . . . .          | 13        |
| 4.4      | Conclusion . . . . .                  | 14        |
| 4.5      | Example Code . . . . .                | 14        |

# 1 Database Design

## 1.1 Assumptions

In addition to the information set in the brief, multiple assumptions have been made while designing the database. These assumptions are concerned with the practical use of the database, the roles of employees in the organisation.

- Employees can be either drivers or cleaners, not both. More generally, roles are singular and nobody can possess multiple roles.
- It is assumed that the database is created for the purposes of managing assets, such as buses, routes and employees, rather than being of functional use in creating rotas and timetables.
- It is also assumed that the data that is entered to the database is correct. For example, there is no way that a database can detect if the depot assigned to the route is correct, since the only information given is the route number and the depot.

## 1.2 Normalisation

The types of normal form discussed are

- 1<sup>st</sup>: all entries in the relation contain only one field, and no repeating groups are present,
- 2<sup>nd</sup>: the relation is in first normal form and all non-primary key entries depend on the full primary key, rather than a proper subset of it,
- 3<sup>rd</sup>: the relation is in second normal form and all non-primary key entries are not transitively dependent on the primary key [1].

---

```
Depot(DepotID, DepotName, DepotAddress, Route, BusID, BusType,
      NumberOfSeats, EndOfDayBay, NumberPlate, RoleDescription, EmployeeID,
      RoleID, ScaleNumber, HomeAddress, ContactNumber, DateHired,
      TestDescription, DrivingLicence, ProvisionalLicence, CPCQualified,
      DateCPCQualified, CPCTrainingHours, Salary)
```

---

This table is clearly not in first normal form. This can be seen by considering two employees working at the same depot; the set `DepotID, DepotName, DepotAddress` is a repeating group. Furthermore, many pairs of columns are unrelated, such as `BusID` and `DateCPCQualified`. To refine this, we identify the functional dependencies within the table.

---

```
DepotID->{DepotName, DepotAddress}
BusID->{BusType, NumberOfSeats, NumberPlate}
BusType->{NumberOfSeats}
ScaleNumber->{Salary}
EmployeeID->{RoleDescription, Salary, HomeAddress, ContactNumber,
            DateHired}
(When Role=Driver) EmployeeID->{DrivingLicence, ProvisionalLicence,
                                CPCQualified, DateCPCQualified, CPCTrainingHours}
```

---

Using these, the table can be refined. Tables are added to deal with entities with no functional dependencies, such as `route`. The primary keys are labelled in red, and foreign keys are underlined.

---

```

Depot(DepotID, DepotName, DepotAddress)
BusType(BusTypeID, BusTypeName, NumberOfSeats)
Route(RouteID, DepotID, RouteNumber)
Bus(BusID, BusTypeID, DepotID, EndOfDayBay, NumberPlate)
Role(RoleID, RoleDescription)
Salary(ScaleNumber, Salary)
Employee(EmployeeID, RoleID, DepotID, ScaleNumber, HomeAddress, ContactNumber,
        DateHired)
Test(TestID, TestDescription)
DriverQualifications(EmployeeID, DrivingLicence, ProvisionalLicence,
                    CPCQualified, DateCPCQualified, CPCTrainingHours)

```

---

These tables are now in first normal form. They are also in second normal form, since each primary key consists of only a single entity. Also, there exist no transitive dependencies between non-primary key entities, since these were listed dealt with immediately after listing the functional relations above, hence they are in third normal form.

To deal with logging information with regards to which drivers are able to drive which routes, for example, junction tables are added.

---

```

Depot(DepotID, DepotName, DepotAddress)
BusType(BusTypeID, BusTypeName, NumberOfSeats)
Route(RouteID, DepotID, RouteNumber)
Bus(BusID, BusTypeID, DepotID, EndOfDayBay, NumberPlate)
Role(RoleID, RoleDescription)
Salary(ScaleNumber, Salary)
Employee(EmployeeID, RoleID, DepotID, ScaleNumber, HomeAddress, ContactNumber,
        DateHired)
Test(TestID, TestDescription)
DriverQualifications(EmployeeID, DrivingLicence, ProvisionalLicence,
                    CPCQualified, DateCPCQualified, CPCTrainingHours)
RouteBusType(RouteID, BusTypeID)
DriverRoute(EmployeeID, RouteID)
DriverTest(EmployeeID, TestID)
DriverBusType(EmployeeID, BusTypeID)
EndOfDayClean(EmployeeID, BusTypeID)

```

---

These junction tables are also in third normal form; they possess no repeated groups, and all data is dependent on the full primary key and not on any non-prime entities.

One might observe that no table is currently recording the exact routes that each bus can take. This can be rectified by selecting *joins* on the existing tables. An example of such a join is provided in the next section.

Below are a list of the check constraints that the entries must meet:

### 1.3 Entity Relationship Diagram

The entity relationship diagram based on the above derivations can be found in figure 1. The relationships between the objects are denoted either by solid lines, implying a strong relationship, and the dotted line implies a weak relationship. A strong relationship is one that the existence of a primary key in a child table is dependent on the parent table. Weak relationships, on the other hand, implies that the existence of entities described by both tables are not dependent on the existence of the other. Additionally, the cardinality and modality of the relationships is denoted using crow's foot notation.

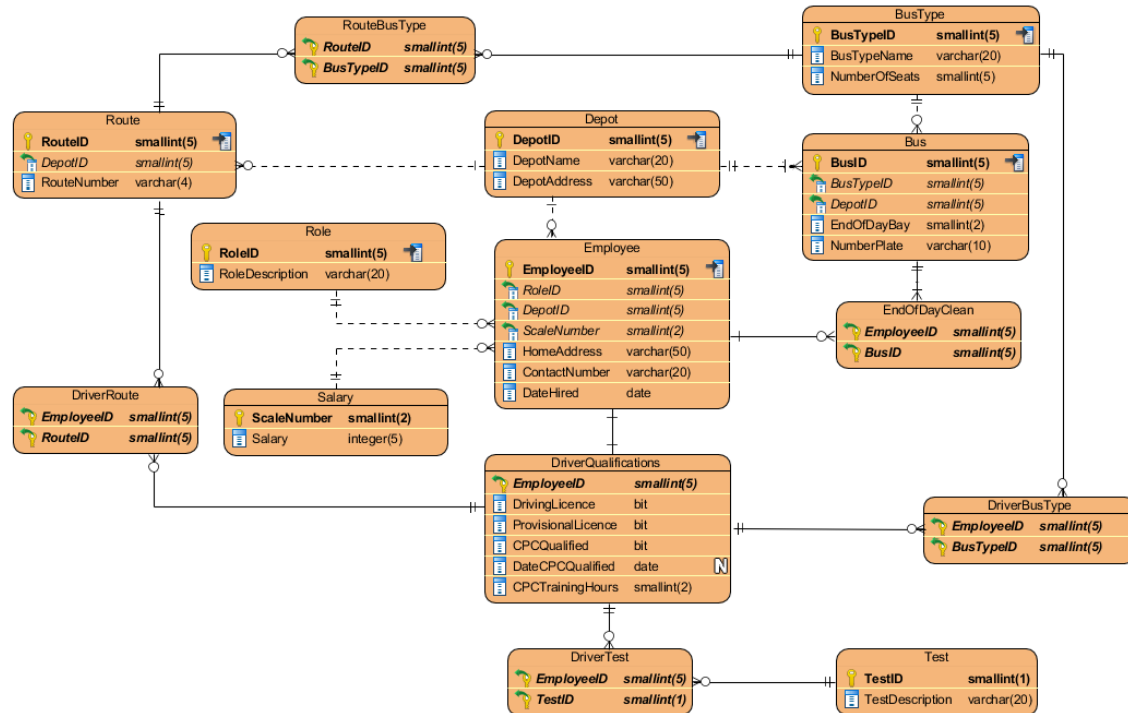


Figure 1: Entity Relationship Diagram based on the tables derived in section 1.2

## 2 Multi-user Databases

Multiple problems arise when a database is constructed to facilitate more than one user. It is possible that users commit changes to the database simultaneously, causing inconsistencies within the system. Additionally, security problems arise if all users are granted the same access to all facilities on offer. Solutions to these problems are available, enabling the use of databases by multiple users.

### 2.1 ACID Transactions

Härder and Reuter introduced the acronym ACID to define transactions that possess favourable properties with regards to the recovery of database information [2].

**Atomicity:** transactions will either be fully committed or not committed at all.

**Consistency:** transactions, once committed, will leave the database in a consistent state; only legal actions are allowed to be committed.

**Isolation:** transactions that are running simultaneously are not allowed to communicate with each other.

**Durability:** once committed, the database system is required to protect the changes from malfunctions occurring after the commit.

Relational database management systems that comply with the ACID property guarantee that their recovery from a system failure is easier; any uncommitted transaction will not have adjusted the database, and the previous commits will be preserved before the failure [2]. Limitations of this definition include the processing of nested and long-lived transactions; for any sub-transactions of a higher level transaction, durability cannot be guaranteed if the latter is not committed before system failure [3].

### 2.2 Concurrency Control

The purpose of concurrency control is to facilitate the simultaneous transactions requested by users while preserving the validity of the data contained within [4]. An example problem given by Cornel, Morris and Rob in their book 'Database Systems' is the Lost Update problem, in which two users try to simultaneously update the same record, resulting in one update being overwritten by the other.

In order to meet the isolation requirement of an ACID transaction, databases can provide levels of *transaction isolation*. The SQL-92 standard defined 4 levels of transaction isolation [5]:

1. Read Uncommitted
2. Read Committed
3. Read Repeatable
4. Serializable

The *Serializable* isolation level is most favoured, and produces results similar to having the transactions being committed in sequence [5]. Serialization errors can occur when multiple users are updating the same entries, for example, in which case a *rollback* is executed to recover a prior state of the database that is consistent.

Again, problems occur when multiple long-lived transactions are present in a database. A paper by Krishnaswamy, Agrawal, Bruno and Abadi suggested relaxing the atomicity condition in order to increase the concurrency of database systems, allowing for such transactions to occur [6]. Notice that relaxing the atomicity condition causes the isolation level to be demoted from serializable.

## 2.3 Resource Locking

Locking is a broad term, described by Gray in his book 'Transaction Processing' as any procedure that requests an object in the database and holds it until the transaction is complete [7]. The granularity of the lock is a measure of how much data the lock covers; the coarsest lock will reserve the whole database until the transaction has completed. Effectively, the higher granularity there is to a lock, the more concurrent transactions can occur, increasing the usability of the database by multiple users. Since locking prevents other users from interacting with a transaction in process, locking is a means of achieving isolation.

## 2.4 Database Security

To deal with the levels of access that users are granted when using a database, Ferraiolo and Kuhn devised *Role-Based Access Controls* [8]. They noted that, unlike previous models for access control such as Mandatory Access Controls and Discretionary Access Controls, the policy-neutral approach of RBAC allows organisations to tailor user-access to the roles of the users interacting with the database. SQL-99 introduced full support for role-based access control [9].

## 2.5 Conclusion

The problems faced when implementing multi user databases are complex but not entirely insurmountable. Ensuring that transactions meet the definition of an ACID transaction will help protect against data anomalies and ease recovery, but for larger systems will create problems if many long-term transactions are present. For the bus company scenario given, the database is meant for only occasional use, so implementing ACID transactions is advisable.

To protect against errors caused by simultaneous transactions, it is recommended that each transaction is serializable, and users of the database are prepared to rollback to previous consistent versions of the database if a problem were to arise. Resource locks should be implemented to increase the concurrency of the database, depending on how high the volume of use is.

Finally, role-based access controls should be implemented so no employee is given any more access to modifying the database than is needed. The IBM 2017 Cyber Security Intelligence Index found that 58% of attacks on companies were carried out by insiders, so protecting against the possibility of malicious insider attacks will ensure confidence that the database will maintain uptime and not be compromised [10].

## 3 SQL

### 3.1 Construction of Tables

The tables are created using lines 1 to 104 in the accompanying SQL code. The sample data used is found in lines 106 to 403.

To construct additional tables, such as the list of all buses that can drive routes according to the restrictions provided in the `RouteBusType` table, joins can be used. This example is provided below, and is also found at lines 485-490 in the accompanying SQL file.

---

```
SELECT BusRoute.busID, BusRoute.routeID FROM
      (SELECT * FROM bus INNER JOIN route
        ON bus.depotID = route.depotID) AS BusRoute
      INNER JOIN
        routebustype
ON (routebustype.routeID = BusRoute.routeID AND routebustype.
    bustypeID = BusRoute.busTypeID);
```

---

### 3.2 Audit Tables

Audit tables for changes in salary and changes of depot were created. The audit table for salary is found at 405-413. The trigger function to log the change can be written as

---

```
CREATE OR REPLACE FUNCTION LogSalaryChange()
      RETURNS trigger AS
$BODY$
BEGIN
      IF New.scaleNumber <> OLD.scaleNumber THEN
          INSERT INTO SalaryAudit(EmployeeID, EmployeeName,
            OldScaleNumber, NewScaleNumber, DateChanged)
            VALUES
              (OLD.EmployeeID, OLD.EmployeeName, OLD.scaleNumber,
                NEW.ScaleNumber, now());
      END IF;
      RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
```

---

The associated trigger is written at 432-436. Another trigger has been created to log the movement of employees between depots. All parts can be found at lines 438-466.

To enforce business rules concerning the salary of employees, the `LogSalaryChange` function can be adjusted to restrict the change of salary. The assumption that cleaners cannot get a scale number greater than 7 is used here for demonstration purposes. This can be found at 415-430.

### 3.3 Testing

Tests on audit tables, check constraints and triggers were conducted.

```
1 INSERT INTO Salary (scaleNumber, salary) VALUES
2 (12,-1);
```

[Data Output](#) [Explain](#) [Messages](#) [Notifications](#) [Query History](#)

ERROR: new row for relation "salary" violates check constraint "salary\_positive"  
DETAIL: Failing row contains (12, -1).  
SQL state: 23514

Figure 2: Test that check constraint on salary is functional.

```
1 UPDATE Employee
2 SET ScaleNumber = 1
3 WHERE EmployeeID = 2;
```

[Data Output](#) [Explain](#) [Messages](#) [Notifications](#)

UPDATE 1

Query returned successfully in 169 msec.

Figure 3: Updating Salary of employee

|    |   |                     |   |   |   |
|----|---|---------------------|---|---|---|
| 20 | 2 | Joshua C. Gallagher | 2 | 1 | 1 |
|----|---|---------------------|---|---|---|

Figure 4: Update as recorded in the `Employee` table.

|   |   |   |                     |   |   |                            |
|---|---|---|---------------------|---|---|----------------------------|
| 1 | 1 | 2 | Joshua C. Gallagher | 1 | 2 | 2019-01-17 14:33:41.657357 |
|---|---|---|---------------------|---|---|----------------------------|

Figure 5: Updating as recorded in the `SalaryAudit` table.

```

1  update employee
2  set scalenumber = 1
3  where employeeid = 1;
4  --select * from salaryaudit;

```

| Data Output                                                                                                                                         | Explain | Messages | Notifications | Query History |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|---------|----------|---------------|---------------|
| ERROR: Montana S. Kerr scale number cannot be lower than than 5<br>CONTEXT: PL/pgSQL function logsalarychange() line 7 at RAISE<br>SQL state: P0001 |         |          |               |               |

Figure 6: Unsuccessful update due to restriction on salary for bus drivers.

### 3.4 Transaction Management

Suppose one user of the database wishes to add a new vehicle to the **bus** table and count the number of buses count the number of vehicles the company possesses, and another user wishes to the same process for a different bus at the same time.

Using the serializable isolation level, the database will attempt to produce results similar to having both transactions committed in some order. In this case, it will fail to do so, since no ordering can be found, and a rollback is automatically executed. The example code used in figures 7 and 8 can be found at 468-483.

postgres on postgres@PostgreSQL 11

```

1  LEVEL SERIALIZABLE;
2  DepotID, EndOfDayBay, NumberPlate)
3  VALUES (3,2,14,'XC45WQE');
4  SELECT COUNT(*) FROM BUS;

```

Data Output Explain Messages Notifications Query History

| count | begin |
|-------|-------|
| 1     | 41    |

postgres on postgres@PostgreSQL 11

```

1  BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2  INSERT INTO bus (busTypeID, DepotID, EndOfDayBay, NumberPlate)
3  VALUES (2,1,7,'RE12VMN');
4  SELECT COUNT(*) FROM BUS;
5  COMMIT;

```

Data Output Explain Messages Notifications Query History

COMMIT

Query returned successfully in 256 msec.

Figure 7: The code above is executed, left panel first. The second panel contains a commit message.

```

postgres on postgres@PostgreSQL 11
1 COMMIT;

Data Output Explain Messages Notifications Query History
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:   The transaction might succeed if retried.
SQL state: 40001

```

Figure 8: Unsuccessful update due to restriction on salary for bus drivers. This automatically executes a rollback on the content of the first window.

A *rollback* can be used in the event that an incorrect update is made to the database. In figure 9, an attempt was made to change the salary of **ScaleNumber** 2 to 12,500.

```

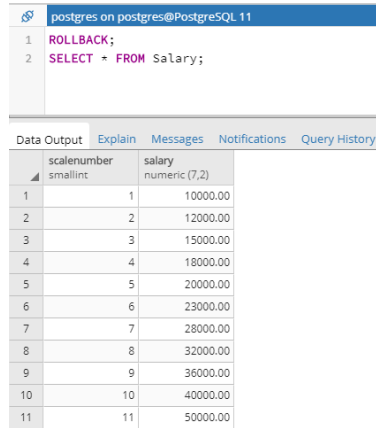
postgres on postgres@PostgreSQL 11
1 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2 UPDATE Salary
3 SET salary = 12500;
4 SELECT * FROM Salary;

```

|    | scalenum<br>smallint | salary<br>numeric (7,2) |
|----|----------------------|-------------------------|
| 1  | 1                    | 12500.00                |
| 2  | 2                    | 12500.00                |
| 3  | 3                    | 12500.00                |
| 4  | 4                    | 12500.00                |
| 5  | 5                    | 12500.00                |
| 6  | 6                    | 12500.00                |
| 7  | 7                    | 12500.00                |
| 8  | 8                    | 12500.00                |
| 9  | 9                    | 12500.00                |
| 10 | 10                   | 12500.00                |
| 11 | 11                   | 12500.00                |

Figure 9: An attempt to change the salary of the second tier goes horribly wrong and changes all salaries to 12,500.

In figure 10, a rollback is called to rectify the problem caused by the previous procedure. This takes the database back to its prior state, undoing all changes made after the call of the transaction. Note that this is not possible without explicitly stating that a transaction has begun. Implicitly each script will initiate a transaction and will commit automatically if the change was successful.



The screenshot shows a PostgreSQL terminal window with the title 'postgres on postgres@PostgreSQL 11'. The command history shows two lines: '1 ROLLBACK;' and '2 SELECT \* FROM Salary;'. Below the command history, there is a tabbed interface with 'Data Output' selected. The 'Data Output' tab displays a table with two columns: 'scalenum' (smallint) and 'salary' (numeric(7,2)). The table contains 11 rows of data, numbered 1 through 11 in the first column, with corresponding salary values in the second column.

|    | scalenum<br>smallint | salary<br>numeric(7,2) |
|----|----------------------|------------------------|
| 1  | 1                    | 10000.00               |
| 2  | 2                    | 12000.00               |
| 3  | 3                    | 15000.00               |
| 4  | 4                    | 18000.00               |
| 5  | 5                    | 20000.00               |
| 6  | 6                    | 23000.00               |
| 7  | 7                    | 28000.00               |
| 8  | 8                    | 32000.00               |
| 9  | 9                    | 36000.00               |
| 10 | 10                   | 40000.00               |
| 11 | 11                   | 50000.00               |

Figure 10: Since the user chose to input **BEGIN TRANSACTION** at the start, and did not commit, the mistake can be rectified simply by calling **ROLLBACK**.

### 3.5 Performance Tuning

PostgreSQL allows users to modify the amount server resources consumed by the database. For example, shared memory buffers and memory allocated to sort operations hash tables and maintenance [11].

In terms of the data used, indexes are preferred over data types such as strings when used as primary keys. This increases the speed of queries using conditions such as **JOIN** or **WHERE**; comparing indexes is likely faster than comparing a composition of Number plates and countries, for example [12].

The structure of queries also have an impact on the performance of the database. For example, correlated subqueries, which have a query dependent on the information fed by the parent query, increases the time to completion of the whole query, since it is processed row-by-row [13]. Using **JOIN** is a potential workaround to this.

## 4 MongoDB Databases

### 4.1 NoSQL

NoSQL refers to a database that stores, manipulates and retrieves data using a non-relational database model. NoSQL databases attempt to solve a prominent problem of relational databases; processing large volumes of transactions [14].

Multiple forms of NoSQL database exist, using different structures and designs. Yen gave a classification of these databases, including types such as Key-Valued Caches, Tuple Stores, Object Databases and Document Stores [15].

### 4.2 MongoDB

#### 4.2.1 Data

Records in MongoDB are stored as *documents*, a structure that is a composition of value and field pairs [16]. These should be compared to rows in a relational database. An example of a record might be

---

```
{
    BusType: "Minibus",
    DepotLocation: "Swansea",
    NumberOfSeats: 12,
    ValidRoutes: [1,3,4,6]
}
```

---

These records are stored in *collections*, which are analogous to the concept of a table found in relational databases [17]. Creating a collection is done by calling `db.createCollection(<name>, <specifications>)`, where the specifications are a document of properties of the collection, such as size, index options and validation levels.

The records stored in the collections are allowed to have a variety of different data types, such as integers, strings, arrays and other records. Additionally, records stored in the same collection are not required to have the same data types as other records [17]. This is unlike SQL relational databases, which won't allow fields in tables to be strings or tables, and require all rows in a table to have the same fields. The assumption made for the database in PostgreSQL that employees could not have multiple roles can now be easily addressed using MongoDB.

#### 4.2.2 Data Model

An important distinction to make between NoSQL databases such as MongoDB and relational databases such as PostgreSQL is the lack of a normalisation constraint. As was stated above, records in collections need not possess the same fields and multiple values can be stored in each field, even other records, meaning they have already broken first normal form. Even though normalisation need not occur in a MongoDB database, informal relationships can be defined between collections, meaning the time-saving property of normalisation is somewhat salvaged [18].

### 4.2.3 Queries

Queries in MongoDB are executed in a similar fashion to that of SQL databases, as summarised in their 'Query Documents' documentation page [19]. Commands such as `db.collection.find()` are similar to the command `SELECT * FROM <table>`. Conditions can be entered inside the braces to add constraints to the search, such as `{bustype: "minibus"}` to select all records with bus type minibus. These can be extended by using AND/OR operations and inequalities.

### 4.2.4 Sharding

A MongoDB database can distribute its data among multiple systems and machines using a process called *sharding*, documented by MongoDB Inc. in their documentation [20]. The object being considered is the *sharded cluster*, which consists of subsets of the data, called shards, query routers called mongos and config servers that store properties of the sharded cluster, such as configuration settings.

To partition the shard, a *shard key* is used, which is a set containing immutable fields that are present in the documents of the target [20]. The key is chosen when the sharding process is started. The resulting partitions are called *chunks*, containing exclusive bounds to their range that is dependent on the shard key.

Research has uncovered many favourable properties of databases with the ability to shard. In particular, a paper by Bagui and Ngyuen showed that sharding allows databases to have a high up-time with a large capacity to scale [21].

Research into the comparative security of NoSQL databases found that MongoDB was more secure than competitors like CouchDB, Redis and Cassandra [22]. According to the authors the authors of the paper, MongoDB achieved relatively high ratings in security due to

1. support of SSL with X.509 authentication certificates,
2. custom roles for role based access control,
3. data encryption at the transport level, again using SSL,
4. basic level of data auditing.

On the other hand, MongoDB does not provide what the authors consider to be the 'highest' level of security. For example, two-factor authentication is not provided, data-at-rest is not encrypted and database configuration security is supported only at a low level by the database administrator.

## 4.3 Criticism of Mongo

A study by Györödi, Györödi, Pecherle and Olah found that MongoDB is preferred to relational database alternatives such as MySQL with respect to the execution time of inputting, updating, querying and deleting information [23].

One disadvantage to MongoDB is that not all join operations are not permitted on groups of tables. Rather, the command `$lookup` is used, which matches fields with the same value, essentially creating a left outer join, but is unavailable to be used to query sharded collections [24].

Experts in the field, such as Mohan, criticise the movement towards mass implementation of NoSQL databases, likening it to previous hyped-up movements such as OODBMS and XQuery [14]. One criticism is that too much responsibility is being given to database administrators and application

writers since statements are not guaranteed to meet the atomicity criteria of ACID. Furthermore, he claims that companies utilise the scaling feature without consideration of the concurrency of the database, failing to provide adequate granularity to the locks on the database, which was discussed in section 2.3.

## 4.4 Conclusion

MongoDB, like many NoSQL databases, possesses properties favourable to many companies operating today, such as easy of scalability, potential denormalisation of data. While security risks are present, it outperforms many other NoSQL databases in this area. The potential to shard the data between many computers and the very quick operating time compared to relational databases makes MongoDB a competitive product in the market.

## 4.5 Example Code

An example of a database for the scenario was created using MongoDB. Two collections were created under the database name 'wHoleBusCompany'. A few records were added to the collections called 'buses' and 'depots'. The sample code can be found in the accompanying MongoDB file.

```
> use wHoleBusCompany
switched to db wHoleBusCompany
> db.buses.find()
{ "_id" : ObjectId("5c41fc616b8510591faed928") }
{ "_id" : ObjectId("5c41fdc96b8510591faed929"), "bustype" : "minibus", "depot" : "Swansea", "numberplate" : "R892VWVN" }
{ "_id" : ObjectId("5c41fdc96b8510591faed92a"), "bustype" : "double decker", "depot" : "Cardiff", "numberplate" : "WE34WEJ" }
> db.buses.update( { bustype: "minibus" }, { $set: { numberplate: "WE12AAS" } }, { upsert: true } )
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 1,
  "nModified" : 0,
  "_id" : ObjectId("5c421ea73e9953bfc4946d85")
})
>
```

Figure 11: A search for all records in the `buses` collection is conducted and a recorded is subsequently updated.

## References

- [1] W. Kent, “A simple guide to five normal forms in relational database theory,” *Commun. ACM*, vol. 26, pp. 120–125, 02 1983.
- [2] T. Härder and A. Reuter, “Principles of transaction-oriented database recovery,” vol. 15, no. 4, pp. 287–317, December 1983.
- [3] J. Gray, “Readings in database systems,” M. Stonebraker, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, ch. The Transaction Concept: Virtues and Limitations, pp. 140–150.
- [4] C. Cornel, S. Morris, and P. Rob, *Database Systems*, 9th ed. Boston, MA, USA: Course Technology CENCAGE Learning, 2010.
- [5] ISO/IEC 9075, “SQL-92,” Tech. Rep.
- [6] “Relative serializability: An approach for relaxing the atomicity of transactions,” *Journal of Computer and System Sciences*, vol. 55, no. 2, pp. 344 – 354, 1997.
- [7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.
- [8] D. Ferraiolo and R. Kuhn, “Role-based access control,” *15th National Computer Security Conference*, pp. 554–563, 1992.
- [9] ANSI/ISO/IEC, “SQL-929,” Tech. Rep.
- [10] IBM. (2017, March) Ibm x-force threat intelligence index 2017.
- [11] PostgreSQL Documentation. Resource allocation. Accessed: 18-01-2019. [Online]. Available: <https://www.postgresql.org/docs/9.1/runtime-config-resource.html>
- [12] M. Cain and K. Milligan, “Ibm db2 for i indexing methods and strategies,” Tech. Rep., July 2011.
- [13] R. Koch. Sql database performance tuning for developers. Accessed: 18-01-2019. [Online]. Available: <https://www.toptal.com/sql-server/sql-database-tuning-for-developers>
- [14] C. Mohan, “History repeats itself: Sensible and nonsensical aspects of the nosql hoopla,” *EDBT/ICDT 2013 Joint Conference, Genoa, Italy*, March 2013.
- [15] S. Yen. (2009, November) Nosql is a horseless carriage. Accessed: 18-01-2019. [Online]. Available: <http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf>
- [16] MongoDB. Introduction to mongodb. Accessed: 18-01-2019. [Online]. Available: <https://docs.mongodb.com/manual/introduction/>
- [17] ——. Introduction to mongodb. Accessed: 18-01-2019. [Online]. Available: <https://docs.mongodb.com/manual/core/databases-and-collections/>
- [18] A. Kanade, A. Gopal, and S. Kanade, “A study of normalization and embedding in mongodb,” February 2014, pp. 416–421.

- [19] MongoDB. Introduction to mongodb. Accessed: 180-01-2019. [Online]. Available: <https://docs.mongodb.com/manual/tutorial/query-documents/index.html>
- [20] ——. Introduction to mongodb. Accessed: 180-01-2019. [Online]. Available: <https://docs.mongodb.com/manual/sharding/>
- [21] S. Bagui and L. Tang Nguyen, “Database sharding:: To provide fault tolerance and scalability of big data on the cloud,” *International Journal of Cloud Applications and Computing*, vol. 5, pp. 36–52, 04 2015.
- [22] A. Zahid, R. Masood, and A. Shibli, “Security of sharded nosql databases: A comparative analysis,” 06 2014, pp. 1–8.
- [23] C. Györödi, R. Györödi, G. Pecherle, and A. Olah, “A comparative study: Mongodb vs. mysql,” June 2015.
- [24] MongoDB. Introduction to mongodb. Accessed: 180-01-2019. [Online]. Available: <https://docs.mongodb.com/manual/reference/sql-comparison/>