# Scalable Database Systems: Assessment Item 1

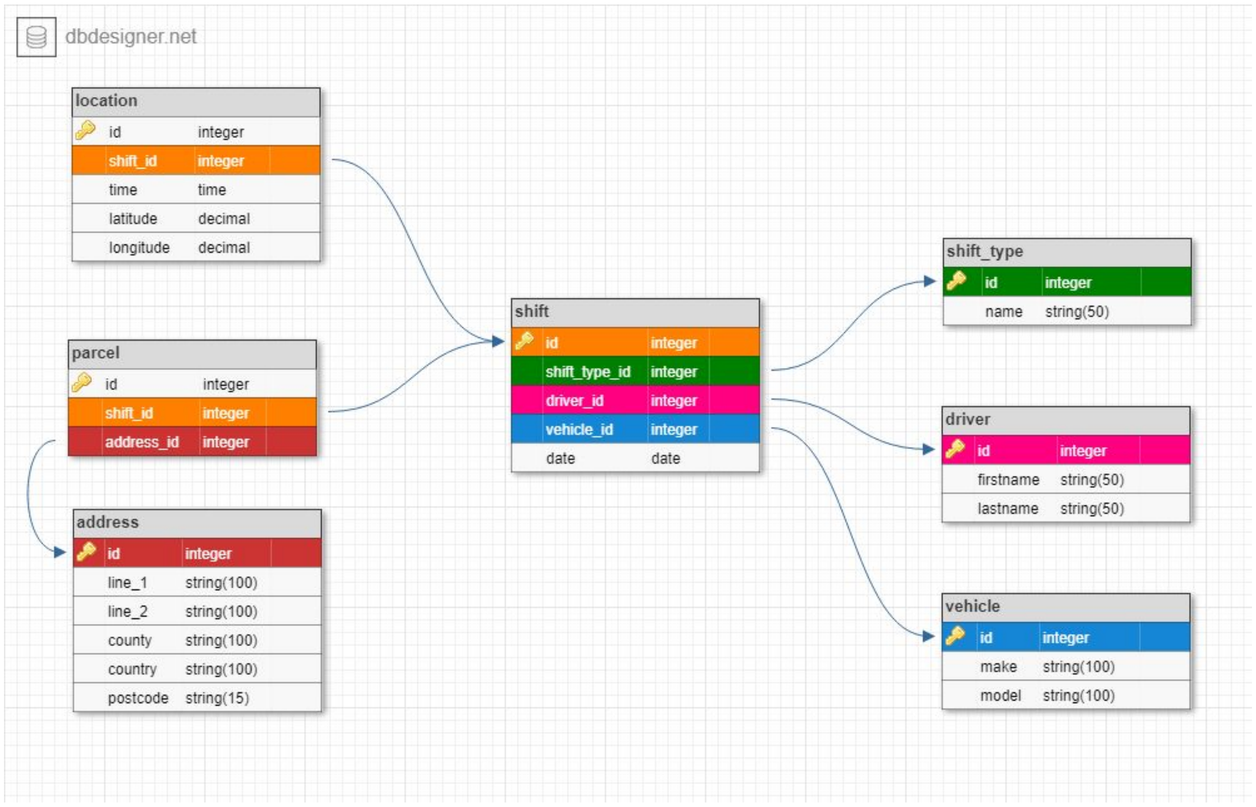
A database schema for a parcel delivery company

## Design

Various forms of abstraction pull the data into separated smaller tables, using foerign keys to link data. The database centres around the *shift* table, as that is where most data relations are defined.

Consideration has been given towards how this database will be manipulated with software. Table subjects are designed in such a way to promote Object-Relational Mapping (ORM), a type of framework allowing the tables to reflect the codebase architecture that is to be built around the database. For example, it would promote the inclusion of a *shift* class, with attributes reflected in the *shift* table. The same goes for the other tables in the database. This design technique makes scaling the database into a usable business solution an intuitive task.

Normalising this database had a large affect, since there were many parts of the business that would create data. Shift attributes must be logged with a high resolution, requiring details of specific hours of the shift. This is implemented with the *location.time* column. Note that there is a *shift.shift_type_id* column which records if the shift was a morning or afternoon shift. Upon casual inspection, it may seem as if there is data redundancy having both *shift.shift_type_id* and *location.time*, since both could be used to denote whether a shift is in the morning or in the afternoon. However, it must also be possible to query the database for drivers who have driven only morning shifts, and the addition of an explicit column allows for easier identification. It also stands to reason that a codebase may be built around this database, and since the business may intend to introduce full-day shifts or night shifts, adding these would be an easy change. As long as the codebase uses *shift.shift_type_id* to denote a morning or afternoon shift, further *shift_type* records can be created and implementing support into the codebase will become far more simple.



A python script was created to generate data for some tables, as manually creating data would be extremely time-consuming. Assume *lat_data* and *long_data* are both arrays of example figures. The script can be found below:

```
for i in range(0,4):
    hour = str(i + 8)
    if (i + 8 < 10):
        hour = "0" + hour
    print("INSERT INTO `location` (`shift_id`, `time`, `latitude`, `longitude`)
VALUES (1, '" + hour + ":30:00', " + str(random.choice(lat_data)) + ", " +
str(random.choice(long_data)) + ");")
```

This script can quickly generate data for an entire shift. To change which shift it generates, the `*shift_id*` column in the values list would need to be changed. This made filling in the *location* table quicker and I was better able to use my time for other aspects of the project, although it is worth noting that this data is unrealistic for the travelling time of the company vehicles. However, for the purposes of demonstrating the database, it will suffice.

The destination address could be denoted as columns in the parcel table, however, using a separate address table allows repeat deliveries to the same address, minimising duplicate information.

## Queries

Below, numerous example queries show how the database can be used. Note that the database is not necessarily full, and unexpectedly empty sets of data may be observed. This is especially true in relation to the *parcel* table, where only one day of data has been created.

To find the location of any vehicle and it's driver at a given hour, during the working day:

```
SELECT l.`latitude`, l.`longitude`
FROM `location` l
JOIN `shift` s ON l.`shift_id` = s.`id`
WHERE s.`driver_id` = 4
AND l.`time` = "11:30:00"
AND s.`date` = "2021-01-25";
```

This will return:

| \`latitude\` | \`latitude\` |
|---|---|
| 50.55 | -0.98 |

Here, the query connects the *shift* and *location* table. This is done with a *join* statement, which connects two rows from different tables, usually using "=". In the query above, we are using *location.shift_id* to link the *shift* table, allowing access to more information about the shift. For example, it would then be possible to link to the *driver* table using *shift.driver_id* so that information from the *driver* table would be visible.

The records themselves are filtered by the *where* clause. Filtering limits the set of results by removing a subset with each clause. In the query above, only records for a specific time on a specific day relating to a single driver will be shown.

To find the number of parcels delivered by any specific driver during a day's work:

```
SELECT COUNT(p.`id`) AS `parcel_count`
FROM `parcel` p
JOIN `shift` s ON s.`id` = p.`shift_id`
JOIN `driver` d ON d.`id` = s.`driver_id`
WHERE d.`id` = 3
AND s.`date` = "2021-01-25";
```

*COUNT()* is useful when viewing how many records would be returned by the *select* query. If the *p.`id`* column was selected instead of counted, the query would show all IDs that match the clauses. Wrapping *p.`id`* in *COUNT()* hides the individual records and instead shows the amount of rows returned.

This will return:

| `parcel_count` |
|---|
| 2 |

To retrieve a list of all drivers:

```
SELECT id, firstname, lastname
FROM `driver`;
```

Instead of detailing the specific columns, the * character could be used to return all columns. This was not used because, in future, more columns may be added to this table. Since only these columns are wanted and there may be back-end code relying on having three columns specifically, the choice was made to be more explicit.

This will return:

| id | firstname | lastname |
|---|---|---|
| 1 | Jim | Henshaw |
| 2 | Toby | Best |
| 3 | Ben | Jameson |
| 4 | Madilyn | Perrot |
| 5 | Toby | Clarke |
| 6 | Mike | Hemsworth |
| 7 | Penny | Simonsson |
| 8 | Josephine | Desroches |
| 9 | Eric | Revie |
| 10 | Maude | Van Buren |

To retrieve a list of drivers who have driven only during morning shifts:

```
SELECT DISTINCT d.`id`, firstname, lastname
FROM `driver` d
JOIN `shift` s ON s.`driver_id` = d.`id`
JOIN `shift_type` st ON st.`id` = s.`shift_type_id`
WHERE st.`name` = "morning"
AND d.`id` NOT IN (
	SELECT DISTINCT d.`id`
	FROM `driver` d
	JOIN `shift` s ON s.`driver_id` = d.`id`
	JOIN `shift_type` st ON st.`id` = s.`shift_type_id`
	WHERE st.`name` = "afternoon"
);
```

The embedded *select* statement is being used as a subquery. This allows us to further filter the results by using *not in*. The subquery retrieves a list of all drivers who have worked an afternoon shift and the outer query retrieves a list of all drivers who have worked a morning shift. By subtracting all records from the outer query that exist in the inner subquery, all drivers who have worked a morning shift but have never worked an afternoon shift are listed. The keyword *distinct* groups records with a matching attribute, *d.`id`* in this case. If all records were selected, many duplicate IDs would show due to the type of join and many shifts having the same *driver_id*.

This will return:

| id | firstname | lastname |
|---|---|---|
| 2 | Toby | Best |
| 6 | Mike | Hemsworth |

## Procedures

*GetAllDrivers()* is a simple procedure that can be used to retrieve every driver in the *driver* table. It will display their ID, firstname, and lastname, and is invoked using the *call* keyword in MySQL, like so: `CALL GetAllDrivers();`

GetLatLongFromLocationId() takes a *location_id* and will return the *latitude* and *longitude* of that record in the *location* table. This stored procedure is more complex because it requires both in and out parameters. To use it, write: `CALL GetLatLongFromLocationId(X, @lat, @long);` where X is *location.id*. The return values are stored in "*@lat"* and "*@long"*, which can then be selected by a *select* statement for display.

Interaction with *GetLocationOfDriver()* is done in much the same way, but a *time* and *date* value needs to be passed. An example use may look like this: `CALL GetLocationOfDriver(2, '09:30:00', '2021-01-25', @lat, @long);` This will save the return into *@lat* and *@long*. This stored procedure contains a nested stored procedure, meaning that it contains a call to another stored procedure. In this case, it calls *GetLatLongFromLocationId()* to retrieve the *latitude* and *longitude* values once it has found a *location_id* value. This is a simple example of a nested procedure, but works well as a demonstration of the concept. As the database scales and complexity rises, it is easy to see how nested stored procedures could make program flow easier to write and comprehend.

## Security

Data breaches are a large threat to databases, especially those containing publicly identifiable information. Before storing identifiable information, question why the information is needed and if it is not necessary to store it, then it should not be stored. This is why extra information is not stored about employees since it would be the job of Human Resources to store sensitive information about them, not the job of a production server. Their information should be stored offline on a separate server. The information about parcel destinations is purposefully kept vague since there is no need to store more than an address. No phone numbers or email addresses are stored, since this is more information that could be leaked which is not needed in the first place.

If more information was required to be stored about the recipients or customers, a pseudonymous configuration should be used. To implement this, the main production database would have no further information about an individual other than an ID. Addresses, phone numbers, payment information, and full names would all be on a separate database that would have tigher access access restrictions. Every time identifiable information needs to be retrieved through an ID, a higher-security system would allow a reply of the required identifiable information. For example, custom rules may exclusively allow connections from a subset of IPs following authentication with a hardware key or similar secondary authentication. This separates the database into one anonymised dataset and another highly-secured identifiable dataset, and since the attack surface of the secondary server is smaller, it allows a far higher degree of security.

User accounts should be optimised for security by removing all permissions that are unnecessary. It is also a good idea to have separate user accounts for read and write permissions. This helps defend against destructive SQL injection attacks, since if an attacker appends a data manipulation query onto a select query, the appended data manipulation query will not have permissions to execute.  However, responsibility mainly falls on the codebase-to-database interface to secure against SQL injection. Popular methods include data manipulation using only stored procedures and locking down data manipulation permissions, or using ORM frameworks to parse queries that better conform to security standards. Deleting the default user account is also a security measure often overlooked.

Encryption should be used on both user passwords and the database as a whole. Passwords should be salted and hashed using an up-to-date algorithm, along with appropriate restrictions on password strength, forcing inclusion of letters, numbers, special characters, and a minimum character count. The database decryption keys should be stored offline in a separate location. Server backups must also be stored away from decryption keys, since they may still contain sensitive information.

Lastly, server software must be kept up to date on security patches. This includes everything from frameworks used in the codebase, the codebase language itself, the database version, webserver, and operating system. Hardware must also be taken into consideration, given recent CPU vulnerability discoveries such as Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2018).

## Scalability

For large production environments, more resources can be allocated to the server. This will instantly increase performance, but there is a risk of overprovisioning resources. Financial cost is also a factor, since cloud servers will charge for extra hardware resources and local servers will require purchasing the parts to extend capabilities (Redgate, 2019).

Alternatively, the database can be regularly copied to secondary servers. Write and update queries will take place on the primary server and read queries can take place on the secondary servers. This will decrease the read and write time, since less operations per server will be taking place. This comes with a financial cost, however, since multiple servers will be running at once.

Sharding the database separates the dataset into multiple parts, copying the schema into each database. To apply this to the database schema in this report, the *shift* table would be separated based on the *shift.id* column, and when a maximum limit is reached, a new database would be initialised and used as a production database. This keeps data volume low, increasing performance. There are also more advanced methods of separating the data into multiple production databases, such as hashing and categorisation (Redgate, 2019).

## References

1. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M. (2018) Meltdown: Reading Kernel Memory from User Space. In: 27th USENIX Security Symposium, Baltimore, MD, USA, 15-17 August. Baltimore, USA: USENIX. Available from https://meltdownattack.com/meltdown.pdf [accessed 4 February 2021].
2. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Preacher, T., Schwarz, M., Yarom, Y. (2018) Spectre Attacks: Exploiting Speculative Execution. In: 40th IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20-22 May. San Francisco, USA: IEEE. Available from https://spectreattack.com/spectre.pdf [accessed 4 February 2021].
3. Redgate (2019) *Designing Highly Scalable Database Architectures.* Cambridge: Redgate. Available from https://www.red-gate.com/simple-talk/cloud/cloud-data/designing-highly-scalable-database-architectures/ [accessed 4 February 2021]