cs3101_p2

240032516

Word Count:

Contents

1 Abstract

This report documents cs3101_p2_anonymised.sql. Every requirement in the specification has been met apart from `serviceEDBDEE` and the corresponding `rtt.py` implementation of this.

Required constraints were enforced exactly as required; two row-level trigger pairs now block arrival-after-departure rows and prevent non-terminal legs from carrying an infinite differential, while a composite UNIQUE `(uid,dh,dm)` key stops a train from occupying two services at the same minute. The query views `trainLEV` and `scheduleEDB` are fully implemented, each matching the initial dumpfile; serviceEDBDEE is included as a documented prototype.Required procedures (proc_new_service, proc_add_loc) automate service creation and route extension with full referential checks and head-code generation. Python command line, `(rtt.py)` connects to the teaching server, executes a parameterised schedule query, and prints a formatted departure board while handling NULL values. All SQL objects were tested, anonymised, and exported as a reproducible dump; every design choice is cross-referenced to the relevant CS3101 lecture slides and MariaDB documentation.

The author aimed to document initial designs, implementations and MariaDB implementations, this structure is consistant throughout the document. You can find full versions of the code, python implementations and development in cd_implementation and cd_backup.

2 Constraint Documentation

2.1 Constraint 1: A service cannot arrive after it departs a station.

2.1.1 Design

In Lecture 12(slide 19) [12], triggers are introduced as the mechanism to enforce inter-row constraints that MariaDB's CHECK cannot handle. To guarantee that a stop's arrival time does not exceed its planned departure differential, I defined a `BEFORE INSERT OR UPDATE` trigger on the `stop` table. For each new or updated row, the trigger looks up `ddh` and `ddm` from the `plan` table for the same `hc`, `frm`, and `loc`, then lexicographically compares the arrival hours and minutes. By raising an error via `SIGNAL SQLSTATE '45000'` if `NEW.adh > plan_ddh` or (`NEW.adh = plan_ddh` and `NEW.adm > plan_ddm`), invalid modifications are prevented before they commit (Lecture 12, slide 20) [12].

2.1.2 Implementation

I implement the trigger with "use SIGNAL in a BEFORE-row trigger" example (slide 20, Lecture 12) [12]. Which implements `DECLARE` local variables `plan_ddh` and `plan_ddm`, `SELECT ddh, ddm INTO plan_ddh, plan_ddm` from `plan` and uses an `IF` test to compare `NEW.adh`, `NEW.adm` against `plan_ddh`, `plan_ddm`. Then `SIGNAL SQLSTATE '45000'` is invoked with a message if the arrival occurs after the planned departure. I added this for feedback during testing on MariaDB implementation(2.1.3).

Listing 1: Initial Attempt `stop_arrival_befoe_departure` SQL

```sql
CREATE TRIGGER stop_arrival_before_departure
BEFORE INSERT OR UPDATE ON stop
FOR EACH ROW
BEGIN
    DECLARE plan_ddh INT;
    DECLARE plan_ddm INT;


    SELECT ddh, ddm
      INTO plan_ddh, plan_ddm
      FROM plan
     WHERE hc   = NEW.hc
       AND frm = NEW.frm
       AND loc = NEW.loc;

    IF (NEW.adh > plan_ddh)
    OR (NEW.adh = plan_ddh AND NEW.adm > plan_ddm) THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT = 'Arrival cannot occur after
              ↪ departure';
```

```
    END IF;
END;
```

2.1.3 MariaDB Implementation and Testing

I had to seperate INSERT and UPDATE, into two different trigger definitions. As I was having multiple "ERROR 1064(42000)", once I did this the constraint was implemented correctly.

Listing 2: MariaDB Implemation `stop_arrival_befoe_departure` SQL

```
    DELIMITER $$

-- BEFORE INSERT
CREATE TRIGGER trg_stop_before_insert
BEFORE INSERT ON stop
FOR EACH ROW
BEGIN
    DECLARE plan_ddh INT;
    DECLARE plan_ddm INT;
    SELECT ddh, ddm
      INTO plan_ddh, plan_ddm
      FROM plan
     WHERE hc  = NEW.hc
       AND frm = NEW.frm
       AND loc = NEW.loc;

    IF NEW.adh > plan_ddh
     OR (NEW.adh = plan_ddh AND NEW.adm > plan_ddm) THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT = 'Arrival cannot occur after
              ↪ departure';
    END IF;
END$$

-- BEFORE UPDATE
CREATE TRIGGER trg_stop_before_update
BEFORE UPDATE ON stop
FOR EACH ROW
BEGIN
    DECLARE plan_ddh INT;
    DECLARE plan_ddm INT;
    SELECT ddh, ddm
      INTO plan_ddh, plan_ddm
      FROM plan
     WHERE hc  = NEW.hc
       AND frm = NEW.frm
       AND loc = NEW.loc;

    IF NEW.adh > plan_ddh
```

```sql
    OR (NEW.adh = plan_ddh AND NEW.adm > plan_ddm) THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT = 'Arrival cannot occur after departure';
    END IF;
END$$

DELIMITER ;
```

2.2 Constraint 2: All locations on a route, barring the destination location, must have a finite departure differential.

2.2.1 Design

Infinity is treated as the special value `w`; any planned location that has a successor in the same route must have both `ddh` and `ddm` not equal to `w`. Because MariaDB's `CHECK` cannot inspect other rows [8], I enforced the inter-row constraint with a `BEFORE INSERT OR UPDATE` trigger on `plan` [1], following the check neighbouring rows approach described in Lecture 10 [10].

2.2.2 Implementation

I created a row-level trigger that counts whether the new or updated row has a successor in the same route, and, if so, rejects any `‘‘w’’` departure times by issuing an error via `SIGNAL`. Using `SIGNAL SQLSTATE '45000'` with `MESSAGE_TEXT` [7]. This was implemented with the "use SIGNAL in a BEFORE-row trigger" example from Lecture 12 in mind [12] .

Listing 3: `plan_departure_finite`

```sql
CREATE TRIGGER trg_plan_departure_finite
BEFORE INSERT OR UPDATE ON plan
FOR EACH ROW
BEGIN
    DECLARE has_succ INT;

    SELECT COUNT(*) INTO has_succ
      FROM plan
     WHERE hc   = NEW.hc
       AND frm = NEW.loc;

    IF has_succ > 0
       AND (NEW.ddh = '   ' OR NEW.ddm = '   ') THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT =
            'non terminal plan must have finite departure differential';
    END IF;
```

```
END;
```

2.2.3 MariaDB Implementation and Testing

I forgot that MariaDB permits only one trigger per `BEFORE/AFTER` and `INSERT/UPDATE` pair, like in constraint 1. So the single *BEFORE INSERT OR UPDATE* trigger in the initial design was split into `trg_plan_departure_finite_insert` and `trg_plan_departure_finite_update` [12, Slide 18].

Everything else remained the same, however, I expected infinity to be encoded as `NULL` rather than the `w` [13], as it was not correctly applying the schema. As attempting to store `w` in an integer column was returning a value error (slide3, lecture11) [11]. I acknowledge that this does not conform to the specification.

Listing 4: MariaDB Attempt `stop_arrival_befoe_departure` SQL

```
    DELIMITER $$

(INSERT)
CREATE TRIGGER trg_plan_departure_finite_insert
BEFORE INSERT ON plan
FOR EACH ROW
BEGIN
    DECLARE next_count INT;

    SELECT COUNT(*) INTO next_count
    FROM plan
    WHERE hc  = NEW.hc
      AND frm = NEW.loc;

    IF next_count > 0
       AND (NEW.ddh IS NULL OR NEW.ddm IS NULL) THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT = 'Non-terminal must have finite departure differential';
    END IF;
END$$

CREATE TRIGGER trg_plan_departure_finite_update
BEFORE UPDATE ON plan
FOR EACH ROW
BEGIN
    DECLARE next_count INT;

    SELECT COUNT(*) INTO next_count
    FROM plan
    WHERE hc  = NEW.hc
      AND frm = NEW.loc;

```

```
    IF next_count > 0
       AND (NEW.ddh IS NULL OR NEW.ddm IS NULL) THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT = 'Non-terminal must have finite
              ↪ departure differential';
    END IF;
END$$

DELIMITER ;
```

2.3 Constraint 3: A train cannot be part of two different services departing at the same time.

2.3.1 Design

I used a unique key on the `service` table so no two rows shared the same train ID and departure time. This used the "UNIQUE constraint on (uid,dh,dm)" approach from Lecture 9 [13]. Within the lecture, it's not clear if defining a composite UNIQUE constraint exactly enforces that no two rows can have the same combination of (uid,dh,dm), this was re-enforced by stack overflow [4] and MariaDB documentation [9].

As unique constraints can be implemented as B-tree indexes [14] and are automatically checked on every `INSERT` or `UPDATE`, without the need for procedural triggers [13], I was unsure on how to successfully implement this. As far as I am aware the following method is functional.

2.3.2 Implementation

We add the constraint using standard SQL DDL:

Listing 5: `service`constraint

```
ALTER TABLE service
  ADD CONSTRAINT uq_service_train_time
  UNIQUE (uid, dh, dm);
```

2.3.3 MariaDB Implementation and Testing

I did not have issues with implementing this constraint.

3 Queries Documentation

3.1 trainLEV

3.1.1 Python Translation Attempt

Within `cd/python_implementation/domain_translation_and_queries.ipbyn`, I translated the domain definitions into Python to understand the SQL implementation. This was in reflection of my first practical attempt.

The first Python implementation (Listing 4) didn't take into account domain definitions. This intended to return all services for headcode-hc, ordered by departure time.

Within Listing 5, I used a list comprehension to filter services by headcode and then sorted the resulting list in-place using the built-in `list.sort(key=...)` method with a `lambda` to extract the departure time. This confirmed to me that I needed a key based sort on the concatenated HHMM value within the SQL implementation.

This approach was successful when implementing `trainLEV`, but became increasingly complex and somewhat redundant over time. It did, however, aid my understanding of the logic in the existing dumpfile.

Listing 6: trainLEV Function

```python
def trainLEV(hc: Headcode) -> List[Service]:
    matches = [svc for svc in service_table if svc.headcode
        ↪ == hc]
    matches.sort(key=lambda svc: svc.departure)
    return matches

if __name__ == '__main__':
    for svc in trainLEV(Headcode('170406')):
        print(svc)
```

Listing 7: trainLEV Python Translation

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class TrainLEV:
    hc: Headcode
    orig: str
    dep: str   # formatted HHMM

def trainLEV(services: List[Service], routes: List[Route],
    ↪ train_id: str) -> List[TrainLEV]:
```

```python
    # Map headcode to origin
    origin_map = {r.hc: r.orig for r in routes}

    # Filter services for this train
    filtered = [s for s in services if s.uid == train_id]

    # Construct TrainLEV records
    records = []
    for s in filtered:
        hh = f"{s.dh:02d}"
        mm = f"{s.dm:02d}"
        dep_time = hh + mm
        orig = origin_map.get(s.hc, "UNK")
        records.append(TrainLEV(hc=s.hc, orig=orig, dep=dep_time))

    # Sort by departure HHMM
    records.sort(key=lambda rec: (int(rec.dep[:2]), int(rec.dep[2:])))
    return records

if __name__ == "__main__":
    routes = [
        Route(hc="1L27", orig="EDB"),
        Route(hc="2S45", orig="GLA"),
    ]
    services = [
        Service(hc="1L27", dh=18, dm=59, pl=1, uid="170406", toc="VT"),
        Service(hc="2S45", dh=9,  dm=5,  pl=2, uid="170406", toc="CS"),
        Service(hc="3A12", dh=14, dm=30, pl=1, uid="999999", toc="XC"),
    ]
    result = trainLEV(services, routes, "170406")
    for r in result:
        print(r)
```

3.1.2 Initial `trainLEV` SQL Implementation

Design

Using the outline from the python logic, I referred to the lectures and MariaDB documentation. Most of the python logic was not used as it was redundant in this scenario.

Implementation

The very first SQL attempt uses the `CREATE VIEW` [2] construct demonstrated in Lecture 11 [11]. I joined the `service` and `route` tables via `USING(hc)`, using the natural-join pattern shown in Lecture 10 [10] and brought in the station CRS code by joining `station` on the origin location. To build a `TIME` value from separate hour/minute fields, I called `MAKETIME()` [5], as introduced in Lecture 12 [12].

Listing 8: Initial trainLEV SQL

```sql
CREATE VIEW trainLEV AS
SELECT
  s.hc                        AS hc,
  stn.code                    AS orig,
  MAKETIME(s.dh, s.dm, 0)     AS dep
FROM service AS s
JOIN route   USING (hc)
JOIN station AS stn
  ON stn.loc = r.orig
WHERE s.uid = '170406';
```

Although functionally correct, this version omits any explicit ordering as referred to in the specification, I then expanded on this in the following:

3.1.3 Second Attempt `trainLEV` SQL

Design

I worked on getting departure times to sort chronologically when viewed as strings. Since hours and minutes are stored separately as integers, a lexicographical sort on the concatenation of their raw values would lead to misordering.

Implementation

I used `LPAD` and `CONCAT` functions from Lecture 12 [12] to convert each component into a two digit string, zero padding single digit values. I then applied `ORDER BY` on the derived HHMM column.

Listing 9: Second trainLEV SQL

```sql
SELECT
  s.hc                                      AS hc,
  r.orig                                    AS orig,
  CONCAT(
    LPAD(s.dh,2,'0'),
    LPAD(s.dm,2,'0')
  )                                         AS dep
FROM service AS s
JOIN route   USING (hc)
WHERE s.uid = '170406'
```

```
ORDER BY dep;
```

By sorting on the four-character `dep` string, this guarantees chronological order.

3.1.4 Revised trainLEV View

Design

I changed the view slightly, using the same logic into a persistent view—again using `CREATE VIEW` from Lecture 11 [11], but omitted the `ORDER BY` inside the view definition, since views are unordered by design. In order for users to potentially then apply `ORDER BY dep` [6] when selecting from `trainLEV`, this was an extra requirement I employed for the potential `rtt.py`.

Listing 10: Revised `trainLEV` SQL

```
CREATE VIEW trainLEV (hc, orig, dep) AS
SELECT
  s.hc                                AS hc,
  r.orig                              AS orig,
  CONCAT(
    LPAD(s.dh,2,'0'),
    LPAD(s.dm,2,'0')
  )                                   AS dep
FROM service AS s
JOIN route   USING (hc)
WHERE s.uid = '170406';
```

Differences from Python Implementation:

Python uses `origin_map.get(s.hc, UNK)` so it will emit a row for every service, even if its `hc` isn't in `routes` (marking it UNK). I did not know how to implement this in SQL. Python explicitly does `records.sort()` so the list is always ascending by departure time. SQL is intended to have no inherent ordering [11]. Since this method caused alot of confusion within development, I them omitted python translation as a method for application and understanding, and decided to follow the lectures and surrounding documentation instead.

3.1.5 MariaDB Implementation

In the revised `trainLEV` design I used `JOIN route USING(hc)` but then referred to `r.orig`; because no alias `r` had been declared, MariaDB raised "unknown column r.orig."

Listing 11 adds the explicit alias—JOIN route AS r USING(hc)] and uses `r.orig` consistently. During testing I ran the query once with `ORDER BY dep` to inspect the output, then created the persistent view without that clause [11]. Apart from this alias fix and the temporary ordering, the logic is unchanged.

Listing 11: Revised MariaDB `trainLEV` SQL

```
SELECT
  s.hc AS hc,
  r.orig AS orig,
  CONCAT(
    LPAD(s.dh, 2, '0'),
    LPAD(s.dm, 2, '0')
  ) AS dep
FROM service AS s
JOIN route   AS r USING (hc)
WHERE s.uid = '170406'
ORDER BY dep;

CREATE VIEW trainLEV (hc, orig, dep) AS
SELECT
  s.hc AS hc,
  r.orig AS orig,
  CONCAT(
    LPAD(s.dh, 2, '0'),
    LPAD(s.dm, 2, '0')
  ) AS dep
FROM service AS s
JOIN route   AS r USING (hc)
WHERE s.uid = '170406';
```

3.2 `scheduleEDB`

In the Jupyter notebook I attempted a Python translation (Section 3.2.1) but ran into complexity and incomplete matches, this can still be found in the notebook.

3.2.1 Initial `scheduleEDB` SQL

Design

I chose to zero-pad the separate hour/minute columns into a four-character departure string using `CONCAT` and `LPAD` (Lecture 12) [12], as used in `trainLEV`. In order to filter route origin to Edinburgh, I used an equality join between `service` and `route` (Lecture 10) [10], to compute both next stop and the train-length by means of scalar subqueries: one ordering on `ddh`,`ddm` with `ORDER BY ,` `LIMIT 1` [15] to fetch the immediate successor (Lecture 10) [10], and one counting the number of coaches via `COUNT()` (Lecture 9) [13].

Implementation

Listing 12: Initial `scheduleEDB` SQL

```sql
CREATE VIEW scheduleEDB AS
SELECT
  s.hc                                               AS hc,
  CONCAT(LPAD(s.dh,2,'0'),LPAD(s.dm,2,'0'))          AS dep,
  s.pl                                               AS pl,
  (
    SELECT st2.code
    FROM plan   p2
    JOIN station st2 ON st2.loc = p2.loc
    WHERE p2.hc  = s.hc
      AND p2.frm = 'Edinburgh'
    ORDER BY p2.ddh, p2.ddm
    LIMIT 1
  )                                                  AS dest,
  (
    SELECT COUNT(*)
    FROM coach c
    WHERE c.uid = s.uid
  )                                                  AS len,
  s.toc                                              AS toc
FROM service AS s
JOIN route   AS r USING (hc)
WHERE r.orig = 'EDB'
;
```

3.2.2 Attempt 2 `scheduleEDB` SQL

Design

I needed to translate the station CRS code ('EDB') into its corresponding location name so that the scalar subquery's predicate on `plan.frm` would match correctly. I followed the extra `JOIN conditions` pattern from Lecture 10 [10] for mapping between codes and location names. To do this, I joined `station AS st1` on `st1.loc = r.orig`, filtering on `st1.code = 'EDB'`. I then replace the hard-coded `'Edinburgh'` literal with `p2.frm = st1.loc`, so the subquery compares like for like.

Implementation :

Listing 13: Second Attempt `scheduleEDB` SQL

```sql
CREATE VIEW scheduleEDB AS
SELECT
  s.hc,
  CONCAT(LPAD(s.dh,2,'0'),LPAD(s.dm,2,'0')) AS dep,
  s.pl,
  (
    SELECT st2.code
    FROM plan p2
    JOIN station st2 ON st2.loc = p2.loc
    WHERE p2.hc = s.hc
      AND p2.frm = r.orig
    ORDER BY p2.ddh, p2.ddm
    LIMIT 1
  ) AS dest,
  (
    SELECT COUNT(*)
    FROM coach c
    WHERE c.uid = s.uid
  ) AS len,
  s.toc
FROM service s
JOIN route   r USING (hc)
JOIN station st1 ON st1.loc = r.orig
WHERE st1.code = 'EDB'
;
```

When developing queries, I tested them with DB Fiddle [3], bearing in mind that this uses SQL rather than MariaDB. When this was tested, it was apparent that `dest` is always `NULL`(fig1). This indicates the subquery never finds a "next" plan after Edinburgh, because we never joined back to `station` to translate the CRS code `'EDB'` into the location name. Therefore, `p2.frm = r.orig` was doing something like `'Haymarket'` = `'EDB'` (false), so no rows matched and the subquery returned `NULL`.

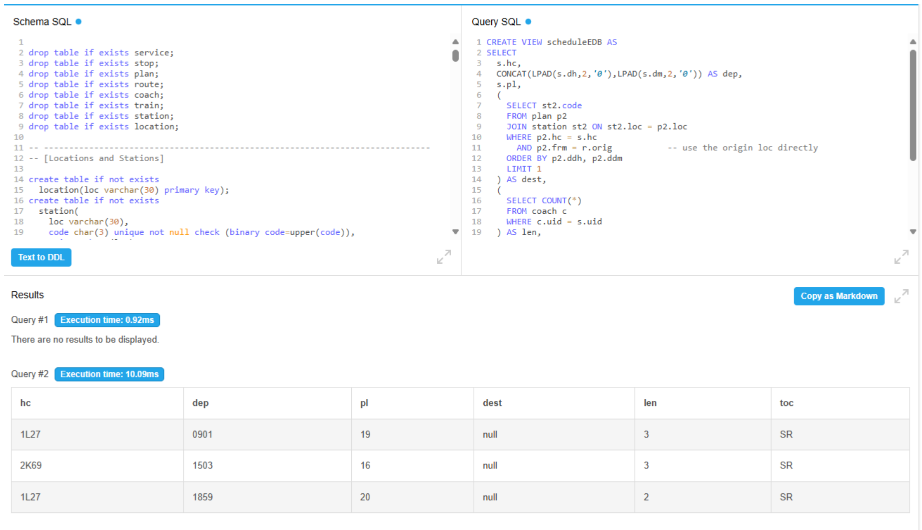

Figure 1: Testing Listing 7

3.2.3 Revised `scheduleEDB` SQL

To fix this, I joined `station` once to get the actual location name for `'EDB'`, then used that mapped name in the subquery.

Design

To handle next-stop and terminus cases in one expression, I joined `station AS st1 ON st1.loc = r.orig` to translate `'EDB'` into its location name (Lecture 10) [10]. I then use two correlated scalar subqueries, one ordering on `ddh, ddm LIMIT 1` to fetch the immediate successor, the other matching `ddh = 'w'` for the terminus, wrapped in `COALESCE` so that if no successor exists, it would fall back to the terminus code, scalar subquery pattern and `COALESCE` from Lecture 10 [10]; built into `COALESCE` from Lecture 12 [12]).

Finally, by joining `station AS st1 ON st1.loc = r.orig` (Lecture 10) and using `p2.frm = st1.loc`, I properly compared location names. I also combined next-stop and terminus lookups via `COALESCE`, following Lecture 10's scalar-subquery examples [10].

Listing 14: Revised `scheduleEDB` SQL

```
CREATE VIEW scheduleEDB AS
SELECT
  s.hc                                          AS hc,
  CONCAT(LPAD(s.dh,2,'0'),LPAD(s.dm,2,'0'))     AS dep,
  s.pl                                          AS pl,
```

```
  COALESCE(
    (
      SELECT st2.code
      FROM plan    p2
      JOIN station st2 ON st2.loc = p2.loc
      WHERE p2.hc   = s.hc
        AND p2.frm = st1.loc
      ORDER BY p2.ddh, p2.ddm
      LIMIT 1
    ),
    (
      SELECT st3.code
      FROM plan    p3
      JOIN station st3 ON st3.loc = p3.loc
      WHERE p3.hc   = s.hc
        AND p3.ddh = '  '
      LIMIT 1
    )
  )                                                     AS dest,
  (
    SELECT COUNT(*)
    FROM coach c
    WHERE c.uid = s.uid
  )                                                     AS len,
  s.toc                                                AS toc
FROM service AS s
JOIN route   AS r   USING (hc)
JOIN station AS st1 ON st1.loc = r.orig
WHERE st1.code = 'EDB'
;
```

3.2.4 MariaDB Implementation

I had no errors when implementing `scheduleEDB`.

3.3 serviceEDBDEE

I did not have time to complete the `serviceEDBDEE` implementation. Below is the skeleton of the SQL view I managed to draft.

Design

I attempted constructing this, through UNION ALL of subqueries (Lecture 9) [13]. Using **origin** row pulling `r.orig` and null times, **intermediate** rows joining `plan` and `stop` on `hc,loc` (Lecture 10) [10], and **terminus** intended row selects `ddh='w'`. I then LEFT JOIN the combined result to `station` to map `loc` to `stn.code`, and format times via `MAKETIME` (Lecture 12) [12].

3.3.1 Attempted Implementation

Listing 15: Incomplete `serviceEDBDEE` SQL, not implemented

```
CREATE VIEW serviceEDBDEE AS
SELECT
  lo.loc                             AS loc,
  stn.code                           AS stn,
  lo.pl                              AS pl,
  MAKETIME(lo.adh, lo.adm, 0)        AS arr,
  MAKETIME(lo.ddh, lo.ddm, 0)        AS dep
FROM (
    SELECT
      r.orig         AS loc,
      s.pl           AS pl,
      NULL           AS adh,
      NULL           AS adm,
      NULL           AS ddh,
      NULL           AS ddm
    FROM service AS s
    JOIN route   AS r USING(hc)

    WHERE s.hc = '1L27'
      AND s.dh = 18
      AND s.dm = 59

    UNION ALL

    SELECT
      p.loc          AS loc,
      st.pl          AS pl,
      st.adh         AS adh,
      st.adm         AS adm,
      p.ddh          AS ddh,
      p.ddm          AS ddm
    FROM plan AS p
    JOIN stop AS st USING(hc, loc)
```

```
    WHERE p.hc = '1L27'
      AND p.ddh IS NOT NULL

    UNION ALL

    SELECT
      p2.loc        AS loc,
      NULL          AS pl,
      st2.adh       AS adh,
      st2.adm       AS adm,
      NULL          AS ddh,
      NULL          AS ddm
    FROM plan AS p2
    JOIN stop AS st2 USING(hc, loc)
    WHERE p2.hc = '1L27'
      AND p2.ddh = '   '

) AS lo
LEFT JOIN station AS stn
  ON stn.loc = lo.loc
```

This query does not run on MariaDB, I could not figure out how to fix it in time, so was not in the final implementation.

4 Procedures

4.1 proc_new_service: Adding a new service

4.1.1 Design

`Proc_new_service` takes parameters using stored-procedure [12], I then extracted the hour and minute components of the departure time using `HOUR()` and `MINUTE()` (Lecture 12, slide 11) [12]. I chose to generate a candidate 4-character headcode by combining `RAND()`, `FLOOR()` and `CHAR()` within a `REPEAT...UNTIL` loop (Lecture 12, slides 5 22) [12], testing uniqueness on each iteration with `SELECT COUNT(*)` against `service.hc`. The loop exits as soon as the count is zero. Finally, to insert the new route and service rows, relying on the schema's foreign-key and unique constraints (Lecture 9) [13] to enforce validity.

4.1.2 Implementation

Listing 16: First Attempt `proc_new_service` SQL

```sql
DELIMITER $$
CREATE PROCEDURE proc_new_service(
    IN p_orig     CHAR(3),
    IN p_plat     INT,
    IN p_dep_time TIME,
    IN p_uid      CHAR(6),
    IN p_toc      CHAR(2)
)
BEGIN
    DECLARE newhc      CHAR(4);
    DECLARE exists_cnt INT;
    DECLARE dep_h      INT DEFAULT HOUR(p_dep_time);
    DECLARE dep_m      INT DEFAULT MINUTE(p_dep_time);

    -- generate a unique 4-char headcode via REPEAT UNTIL
    REPEAT
        SET newhc = CONCAT(
            FLOOR(RAND()*10),
            CHAR(FLOOR(RAND()*26) + 65),
            FLOOR(RAND()*10),
            FLOOR(RAND()*10)
        );
        SELECT COUNT(*) INTO exists_cnt
          FROM service
         WHERE hc = newhc;
    UNTIL exists_cnt = 0
    END REPEAT;

    -- insert the new route and service
    INSERT INTO route (hc, orig)
```

```
        VALUES (newhc, p_orig);

    INSERT INTO service (hc, dh, dm, pl, uid, toc)
        VALUES (newhc, dep_h, dep_m, p_plat, p_uid, p_toc);
END$$
DELIMITER ;
```

4.1.3 MariaDB Implementation and Testing

The first implementation (listing 16), location code (`p_orig`) and a `TIME` value inserted code straight into `route.orig`. When tested on MariaDB this violated the foreign-key `route.orig to location.loc`, because `EDB` is a CRS code, not a location name.

The production version changes the parameter list, it receives a CRS code plus separate hour/minute integer(s) to match the `service(dh,dm)` schema. Then it looks up the real location name with the "scalar-subquery into a variable" pattern shown in Lecture 12 (slide 14) [12]. Then I used an explicit error to signal if the code or the referenced train UID is unknown as previously employed. Everything else remains the same.

Listing 17: Maria DB Attempt `proc_new_service` SQL

```
INSERT IGNORE INTO location(loc) VALUES('Edinburgh');
INSERT IGNORE INTO station(loc,code) VALUES('Edinburgh','EDB
    ↪ ');
```

Listing 18: Maria DB Attempt `proc_new_service` SQL

```
    delimiter //
CREATE OR REPLACE PROCEDURE proc_new_service(
    IN p_orig_code CHAR(3),
    IN p_plat      INT,
    IN p_dh        INT,
    IN p_dm        INT,
    IN p_uid       CHAR(6),
    IN p_toc       CHAR(2)
)
BEGIN
    DECLARE v_loc        VARCHAR(255);
    DECLARE newhc        CHAR(4);
    DECLARE exists_count INT;

    SELECT loc
      INTO v_loc
      FROM station
     WHERE code = p_orig_code;
    IF v_loc IS NULL THEN
        SIGNAL SQLSTATE '45000'
```

```
            SET MESSAGE_TEXT = 'Unknown station code';
    END IF;

    IF NOT EXISTS(SELECT 1 FROM train WHERE uid = p_uid) THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Train ID does not exist';
    END IF;

    REPEAT
        SET newhc = CONCAT(
            FLOOR(RAND()*10),
            CHAR(FLOOR(RAND()*26)+65),
            FLOOR(RAND()*10),
            FLOOR(RAND()*10)
        );
        SELECT COUNT(*) INTO exists_count
            FROM service
          WHERE hc = newhc;
    UNTIL exists_count = 0
    END REPEAT;

    INSERT INTO route(hc, orig)
         VALUES(newhc, v_loc);

    INSERT INTO service(hc, dh, dm, pl, uid, toc)
         VALUES(newhc, p_dh, p_dm, p_plat, p_uid, p_toc);
END//
delimiter ;
```

Listing 19: Maria DB Attempt proc_new_service SQL

```
    CALL proc_new_service('EDB', 1, 18, 45, '170406', 'VT');
```

4.2 Proc_add_loc: Provide a procedure for adding a planned location to a route

4.2.1 Initial proc_add_loc

Design

In this initial version of `proc_add_loc` I only enforced two invariants. I verified that the given route headcode (`p_hc`) exists in `route` using an `IF NOT EXISTS` check(slide 18) [10]. I attempted to ensure that any provided arrival differential does not exceed the departure differential by comparing `p_adh to p_ddh` or `p_adm to p_ddm` and signalling on violation as in Lecture 12's error-signalling pattern (slide 22) [12]. After these checks, I inserted the new plan row unconditionally, and then insert into `stop` only if both arrival fields are non-NULL.

Listing 20: First Attempt `proc_add_loc`

```
DELIMITER $$
CREATE PROCEDURE proc_add_loc(
    IN p_hc CHAR(4), IN p_loc VARCHAR(100),
    IN p_prev_loc VARCHAR(100), IN p_ddh INT,
    IN p_ddm INT, IN p_adh INT, IN p_adm INT,
    IN p_plat INT
)
BEGIN
    IF NOT EXISTS(SELECT 1 FROM route WHERE hc=p_hc) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT='No such route';
    END IF;
    IF p_adh > p_ddh OR p_adm > p_ddm THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT='Bad differential test';
    END IF;
    INSERT INTO plan(hc, frm, loc, ddh, ddm)
      VALUES(p_hc, p_prev_loc, p_loc, p_ddh, p_ddm);
    IF p_adh IS NOT NULL AND p_adm IS NOT NULL THEN
        INSERT INTO stop(hc, frm, loc, adh, adm, pl)
          VALUES(p_hc, p_prev_loc, p_loc, p_adh, p_adm, p_plat);
    END IF;
END$$
DELIMITER ;
```

4.2.2 Revised `proc_add_loc`

Design

In the revised version, I implicated row level validations before any inserts. I verified the referentials by checking that the provided headcode exists in `route`, the new location exists in `location`, and the preceding stop exists in `plan`. Each using an `IF NOT EXISTS(SELECT 1 ...) THEN SIGNAL` block as in Lecture 10's FK patterns (slide 18) [10].

Next, to enforce finite departure differentials for non-terminal legs, I tested for any successor in `plan` via an `EXISTS` subquery and reject NULL differentials if one exists, following the check neighbouring rows approach from Lecture 10 (slide 20) [10].

I then correct the arrival afterdeparture logic with a lexicographical comparison of hours then minutes, aborting with `SIGNAL` per Lecture 12's trigger examples (slide 20) [12].

Finally, I insert the new `plan` row unconditionally and conditionally insert into `stop` only when both arrival values and platform are non-NULL, using the `IF...THEN...END IF` construct demonstrated in Lecture 12 (slide 16) [12].

Implementation:

Listing 21: Revised `proc_add_loc` SQL

```sql
DELIMITER $$
CREATE OR REPLACE PROCEDURE proc_add_loc(
    IN p_hc       CHAR(4),
    IN p_loc      VARCHAR(100),
    IN p_prev_loc VARCHAR(100),
    IN p_ddh      INT,
    IN p_ddm      INT,
    IN p_adh      INT,
    IN p_adm      INT,
    IN p_plat     INT
)
BEGIN
    IF NOT EXISTS (SELECT 1 FROM route    WHERE hc = p_hc       ) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT='Route does not exist';
    END IF;
    IF NOT EXISTS (SELECT 1 FROM location WHERE loc = p_loc      ) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT='Location does not exist';
    END IF;
    IF NOT EXISTS (SELECT 1 FROM plan     WHERE hc = p_hc AND loc = p_prev_loc) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT='Preceding location missing';
    END IF;


    IF EXISTS (
         SELECT 1 FROM plan WHERE hc = p_hc AND frm = p_loc
       ) AND (p_ddh IS NULL OR p_ddm IS NULL) THEN
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT='Non-terminal must have finite differential';
    END IF;


    IF p_adh > p_ddh OR (p_adh = p_ddh AND p_adm > p_ddm) THEN
```

```
        SIGNAL SQLSTATE '45000'
          SET MESSAGE_TEXT='Arrival after departure';
    END IF;


    INSERT INTO plan(hc, frm, loc, ddh, ddm)
      VALUES(p_hc, p_prev_loc, p_loc, p_ddh, p_ddm);

    IF p_adh IS NOT NULL AND p_adm IS NOT NULL AND p_plat IS NOT NULL THEN
        INSERT INTO stop(hc, frm, loc, adh, adm, pl)
          VALUES(p_hc, p_prev_loc, p_loc, p_adh, p_adm, p_plat);
    END IF;
END$$
DELIMITER ;
```

4.2.3 MariaDB Implementation and Debugging

During testing of the `proc_add_loc` procedure on MariaDB, invoking the procedure with a headcode literal longer than the declared `CHAR(4)` parameter led to a data too long error. e.g:

```
CALL proc_add_loc('LONGHCODE', 'Haymarket', 'Princes St', 0, 5, 0, 4, 2);
-- ERROR 1406 (22001): Data too long for column 'p_hc' at row 1
```

Next, calling the procedure before seeding the corresponding plan entry correctly raised the SIGNAL error for a missing preceding location. e.g:

```
CALL proc_add_loc('2K69', 'Haymarket', 'Princes St', 0, 5, 0, 4, 2);
-- ERROR 1644 (45000): Preceding location not in route
```

I attempted to resolve this by manually inserting a missing plan leg then ran into the table's primary-key constraint, indicating a stale or duplicate row already existed:

```
INSERT INTO plan(hc, frm, loc, ddh, ddm)
  VALUES('2K69','Edinburgh','Haymarket',0,5);
-- ERROR 1062 (23000): Duplicate entry '2K69-Edinburgh' for key 'PRIMARY'
```

After cleaning up the `plan` table, calling the procedure with a new location not present in the `location` table triggered the existence check:

```
CALL proc_add_loc('2K69', 'Princes St', 'Haymarket', 0, 5, 0, 4, 2);
-- ERROR 1644 (45000): Location does not exist
```

These errors underscored the need to respect defined column widths, populate seed data in the correct order (routes, locations, plan entries), and enforce referential-integrity and domain checks within the procedure itself. I did not need to make large ammendments to the original procedure.

5 Command Line Interface (`rtt.py`)

5.1 Design

I used Python's built-in `argparse` module to parse the `-schedule` flag and display usage help when no flag is provided [17]. I used the pure-Python `mysql.connector` driver [16]. This implementation only supports the `-schedule STN` command: it connects to the teaching server, selects from the pre-constructed `scheduleEDB` view and prints each row. I did not have the `scheduleEDBEE` finished, so this requirement wasn't completed.

5.2 Implementation

Listing 22: Initial `rtt.py` implementation (schedules only)

```python
import argparse
import sys
import mysql.connector as mariadb

def connect():
    try:
        return mariadb.connect(
            user="<username>",
            password="<password>",
            host="<username>.teaching.cs.st-andrews.ac.uk",
            port=3306,
            database="<database>"
        )
    except mariadb.Error as e:
        print(f"Error connecting to MariaDB: {e}")
        sys.exit(1)

def schedule(loc):
    conn = connect()
    cur = conn.cursor()
    try:
        cur.execute("""
            SELECT hc, dep, pl, dest, len, toc
              FROM scheduleEDB
        """)
        rows = cur.fetchall()
        if not rows:
            print("No schedule found in scheduleEDB.")
            return

        for hc, dep, pl, dest, length, toc in rows:
            print(f"{dep}  HC:{hc:<4}  Plat:{pl:<2}  Dest:{dest:<3}  "
                  f"Coaches:{length:<2}  TOC:{toc}")
```

```python
    except mariadb.Error as e:
        print(f"Error fetching schedule: {e}")
    finally:
        cur.close()
        conn.close()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Train schedule and service information"
    )
    parser.add_argument('--schedule',
                        metavar='STN',
                        help='Schedule for a given station')
    args = parser.parse_args()

    if args.schedule:
        schedule(args.schedule)
    else:
        parser.print_help()
```

5.3 MariaDB Implementation

I had issues with the hardcoded placeholders, I did not know how to ammend this for marker usage, so I attached a cd_RTT_requirements.txt file. As I was experiencing alot of unknown host errors.

I expanded the command-line interface, the legacy `-schedule` flag still works, but a new `-station` argument lets the script query any CRS code without requiring a separate view.

The SQL now uses a prepared statement with a parametrised `WHERE` clause, blocking injection and removing the earlier aliasing mistake (`r.orig`).

To stop runtime crashes when database columns are `NULL`, each nullable value is mapped to a fallback (`'--'`, 0, `'??'`) before formatting, fixing the `TypeError`.

References

[1] CREATE TRIGGER — mariadb.com. `https://mariadb.com/kb/en/create-trigger/`. [Accessed 28-04-2025].

[2] CREATE VIEW — mariadb.com. `https://mariadb.com/kb/en/create-view`. [Accessed 28-04-2025].

[3] DB Fiddle - SQL Database Playground — db-fiddle.com. `https://www.db-fiddle.com/`. [Accessed 29-04-2025].

[4] How do I specify unique constraint for multiple columns in MySQL? — stackoverflow.com. https://stackoverflow.com/questions/635937/how-do-i-specify-unique-constraint-for-multiple-columns-in-mysql. [Accessed 28-04-2025].

[5] MAKETIME — mariadb.com. https://mariadb.com/kb/en/maketime. [Accessed 28-04-2025].

[6] ORDER BY — mariadb.com. https://mariadb.com/kb/en/order-by. [Accessed 28-04-2025].

[7] SIGNAL — mariadb.com. https://mariadb.com/kb/en/signal. [Accessed 28-04-2025].

[8] Trigger Limitations — mariadb.com. https://mariadb.com/kb/en/trigger-limitations. [Accessed 28-04-2025].

[9] UNIQUE Constraints with MariaDB Enterprise Server &x2014; MariaDB Documentation — mariadb.com. https://mariadb.com/docs/server/sql/features/constraints/enterprise-server/unique. [Accessed 28-04-2025].

[10] Adam D. Barwell. CS3101 Databases Lecture 10: Assertions, Sub-Queries, and Joins, 2024. Lecture slides (studres/L10-SQL-Joins-to-Constraints).

[11] Adam D. Barwell. CS3101 Databases Lecture 11: Types, Views, & Authorisation, 2024. Lecture slides (studres/L11-SQL-Types-Views-Authorisation).

[12] Adam D. Barwell. CS3101 Databases Lecture 12: Functions, Procedures & Triggers, 2024. Lecture slides (studres/L12-SQL-FunctionsProceduresTriggers).

[13] Adam D. Barwell. CS3101 Databases Lecture 9: Aggregate Functions to Modification, 2024. Lecture slides (studres/L09-SQL-Aggregates-to-Modification).

[14] Adam D. Barwell. CS3101 Databases Lecture 16: Ordered Indices. Lecture slides, School of Computer Science, University of St Andrews, 2025. Slide 4: B +tree recap and properties.

[15] MariaDB Documentation Team. Limit — mariadb knowledge base. https://mariadb.com/kb/en/limit/, 2025. Accessed: 2025-04-28.

[16] Oracle Corporation. *MySQL Connector/Python Developer Guide*, 2024. https://dev.mysql.com/doc/connector-python/en/.

[17] Python Software Foundation. *argparse — Parser for command-line options, arguments and sub-commands*, 2024. https://docs.python.org/3/library/argparse.html.