

D. Simulator Project Report

Isaac Yang
Yuxiang Lin
Shanruo Xu

December 2023

Abstract

The Detective Simulation encapsulates the quintessence of detective work, striving to both educate and captivate players while evaluating their analytical and critical thinking proficiencies. This game plunges players into the intricate responsibilities of a chief of police, charged with the formidable task of deciphering a series of murder cases. The ensuing report delineates the intricacies of the database schema integral to the implementation of this simulator, providing a comprehensive elucidation of the complex and important queries that underpin the simulation. Subsequently, a demonstration will be expounded upon, accompanied by a comparative analysis vis-à-vis existing applications and this project.

1 Introduction

The Detective Simulation encapsulates the essence of detective work, aiming to educate and entertain players while testing their analytical and critical thinking abilities. This game immerses players into the challenging role of a chief of police entrusted with the mission of unraveling a series of mysterious murder cases. In this simulation, a murderer is selected and, based on their specific attributes such as occupation, gender, and more, they undertake a distinct *modus operandi*. Throughout each round of the game, players are equipped with vital information, including police records on all inhabitants, details of murder cases, and accounts from eyewitnesses. They are also presented with the web of relationships between victims, potentially employing queries to examine inhabitant details. Armed with this wealth of data, players must strategically deploy police resources such as lockdown and arrest to apprehend the suspected culprit. The simulation concludes when the player successfully captures the murderer or when they reach a point where the murderer remains elusive after a set number of rounds. The Detective Simulation replicates the tools and techniques used by real investigators, including working with relationship webs of suspects and utilizing a database to access essential residence information. The game leverages a Relational model with SQL to effectively manage the substantial volume of data involved. While real-time aspects are not a focus,

the game’s turn-based nature and relational mechanics make SQL databases a fitting choice.

2 Motivation

The simulation, by replicating the intricacies of murder cases, provides a valuable training ground for aspiring detectives. It allows them to hone their investigative techniques, navigate complex relationships within a case, and develop proficiency in utilizing databases and relational models—a crucial aspect of modern detective work. The scarcity of real murder cases for training emphasizes the importance of simulated environments, where detectives can practice and refine their skills in a risk-free yet realistic setting. The Detective Simulation serves as an indispensable tool, bridging the gap between theoretical knowledge and practical application, ultimately preparing detectives for the challenges they may encounter in the unpredictable world of criminal investigations.

3 Database Schema

The presented Database Schema encompasses relations crucial for the effective implementation of the simulation. Foundational elements like *inhabitant* and *building* delineate the environment within which the simulation unfolds. These relations define the backdrop against which the intricate web of murder cases and detective work transpires. Concurrently, relations such as *status* assume a pivotal role in capturing the dynamic state of the simulation. Specifically, the *status* relation encapsulates vital information regarding the simulation’s current state, including details on when it is slated to conclude. This temporal aspect proves integral for the seamless management of simulation progress, facilitating the functionalities of saving and loading specific simulation instances. Presented herein is the initial version of the relational model, succeeded by the subsequent Entity-Relationship (ER) model, and culminating in the final, refined iteration of the relational model.

Some design choices regarding the attribute types should be clarified as they are related to the design of the SQLite database management system that we are using. SQLite is designed to be dynamically typed, which means that an attribute in a row could deviate from the datatype specified for that column.[1] This, however, is not reflected by our project as we strictly followed the datatype of each column. Nevertheless, there is another aspect of this dynamically-typed design, which is that there is no way to specify a specific type, but merely denote one of the five storage classes: **NULL**, **INTEGER**, **REAL**, **TEXT**, and **BLOB**. This, for example, means that the storage size of an **INTEGER** is automatically decided and not manually specified. Also, as SQLite lacks a dedicated **BOOLEAN** type, **INTEGER** constrained to be 0 or 1 is used in its place.

3.1 The First Version

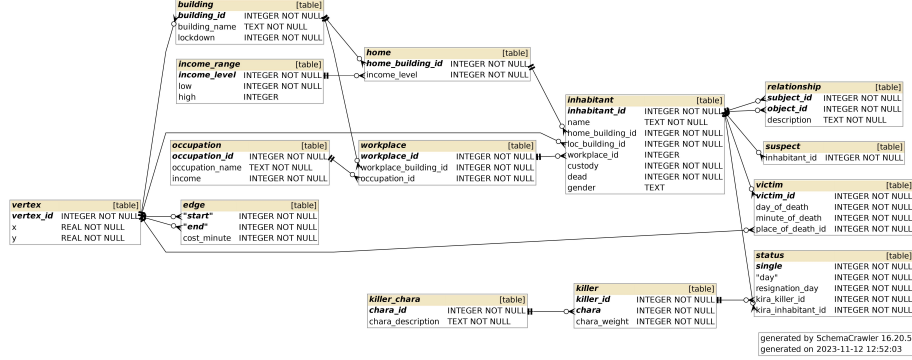


Figure 1: Database Schema First Version

The initial iteration of the Database Schema, displayed as Figure 1, was formulated during the second week, a period when our acquaintance with traditional relational modeling was still developing. The following section provide a concise overview of each relation, elucidating their intended purpose and highlighting key attributes that contribute to the structural foundation of the database. This exploration aims to offer a clear understanding of the schema’s components and their roles within the broader context of the simulation.

3.1.1 *inhabitant*

The linchpin of the schema, the *inhabitant* relation, serves as the cornerstone around which most other relations orbit. This table encapsulates vital characteristics defining the inhabitants in the simulation, encompassing both potential victims and perpetrators, the killers.

- *inhabitant_id*: A primary key crucial for distinguishing each inhabitant uniquely.
- *name*: The first and last name of the inhabitant, providing individual identity.
- *home_building_id*: Denotes the building where the inhabitant resides or commences each day.
- *loc_building_id*: Tracks the current location of the inhabitant as they move throughout the simulation.
- *workplace_id*: Indicates the occupation and the building where the inhabitant works.
- *custody*: A boolean value signaling whether the player/user has designated this inhabitant as a suspect.

- *dead*: A boolean indicator of the inhabitant's vital status, specifying whether they are deceased.
- *gender*: Specifies the gender of the inhabitant, adding a demographic dimension to their profile.

3.1.2 *relationship*

This table specifies the relationships between two inhabitants.

- *subject_id*: Subject of the relation.
- *object_id*: Object of the relation.
- *description*: The description of what the relationship is, i.g. if the description is Friend, this means object in this relationship is the friend of the subject.

3.1.3 *suspect*

This is a list of inhabitants the user has marked to be a suspect of the murders.

- *inhabitant_id*: the id of the inhabitant.

3.1.4 *victim*

This table specifies conditions of the deaths.

- *victim_id*: ID of the victim.
- *day_of_death*: The day when the victim died.
- *min_of_death*: The minute when the victim died.
- *place_of_death_id*: The vertex where the victim died.

3.1.5 *killer*

This table specifies the characteristics that a killer holds.

- *killer_id*: Primary key distinguishing each killer.
- *chara*: Multi-valued attribute indicating the characteristics that a killer holds, references *killer_chara* table to display the details of the characteristic.
- *weight*: Multi-valued attribute indicating the weight of the characteristic, i.e. the higher the weight for an attribute, the more priority the killer gives it.

3.1.6 *killer_chara*

This table provides description for every available killer characteristic.

- *chara_id*: Primary key distinguishing each characteristic.
- *chara_description*: The discription of the characteristic, i.g. a characteristic may be 'low income,' this means the killer holding this characteristic has a tendency to target low income populations (depends on the weight from *killer* table).

3.1.7 *vertex*

This table identifies the locations in which inhabitants move in the simulation.

- *vertex_id*: Primary key identifying each vertex in the simulation.
- *x*: X coordinate of the vertex.
- *y*: Y coordinate of the vertex.

3.1.8 *edge*

This table represents the weighted and directed connections, or, roads between vertices.

- *start*: The vertex forming the beginning of the edge.
- *end*: The vertex specifying the end of the edge.
- *cost_minute*: An indication of distance/how long it takes to travel through this edge.

3.1.9 *building*

This table lists the important vertices in the simulation, buildings are vertices that act as destinations and sources in the path that an inhabitant takes every day.

- *building_id*: Primary key distinguishing each building, this key also references the *vertex_id* in the *vertex* relation, i.e., buildings are a subset of vertices.
- *building_name*: The name of the building.
- *lockdown*: indication of whether the user have imposed a lockdown on the building, a building under lockdown means it no longer can be a part of a path nor can it become the designation of all inhabitants.

3.1.10 *occupation*

This table specifies the available occupations that inhabitants can have.

- *occupation_id*: Primary key distinguishing each occupation.
- *occupation_name*: The name of the occupation.
- *income*: The income of the inhabitant who has the occupation, there is an implicit reference to the *income_range* table which shall be explained later.

3.1.11 *workplace*

This table identifies the workplace of an inhabitant, that is, the building where an inhabitant goes to work. A workplace building can host up to three different occupations

- *workplace_id*: Primary key distinguishing each workplace.
- *workplace_building_id*: The vertex/location of the workplace.
- *occupation_id*: One of the occupations in the workplace.

3.1.12 *income_range*

This table specifies an inhabitant's wealth status based on the income provided by the Occupation table.

- *income_level*: The level of income, every inhabitant is categorized to 'low income,' 'medium income,' or 'high income.'
- *low*: Attribute specifying the low range (inclusive) of the income level.
- *high*: Attribute specifying the high range (inclusive) of the income level, using null to denote positive infinity.

3.1.13 *home*

This table specifies the home of the inhabitants, each home is a building in the vertex map.

- *home_building_id*: Primary key distinguishing each home. This is also a reference to the *building* relation.
- *income_level*: Each inhabitant is assigned to a home based on their income level, which in turn is based on their occupation.

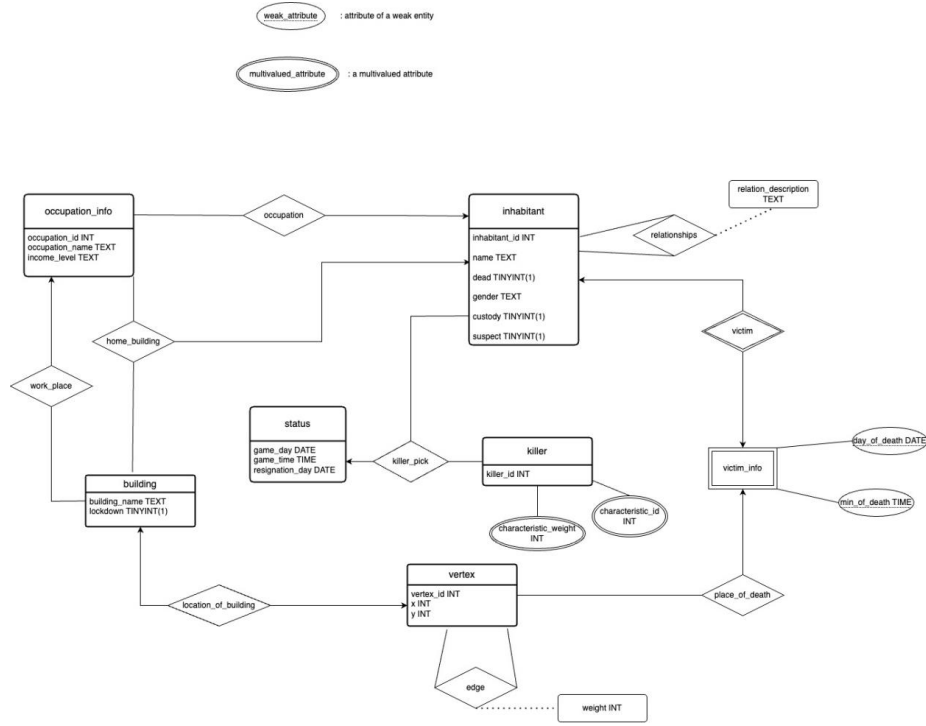


Figure 2: ER model in the fifth week

3.1.14 *status*

This table specifies the progress of the simulation, information that needs to be kept for the purpose of loading and saving a simulation instance.

- *single*: Specifying that there is can be only one tuple in this relation. It is constrained so that it must have the default value of 0 and that it is also the primary key.
- *day*: How many “days” have passed in this simulation.
- *resignation_day*: When the user will “resign” in the simulation. When the *resignation_day* equals the day attribute, this is when the simulation ends, signalling a failure to identify the killer.
- *kira-killer_id*: Selects a killer in the *killer* table.
- *kira-inhabitant_id*: Matches the killer with an identity in *inhabitant*.

3.2 The ER Model

Figure 2 illustrates the Entity-Relationship (ER) model crafted during the fifth week of the course. During the refinement process, redundant attributes were

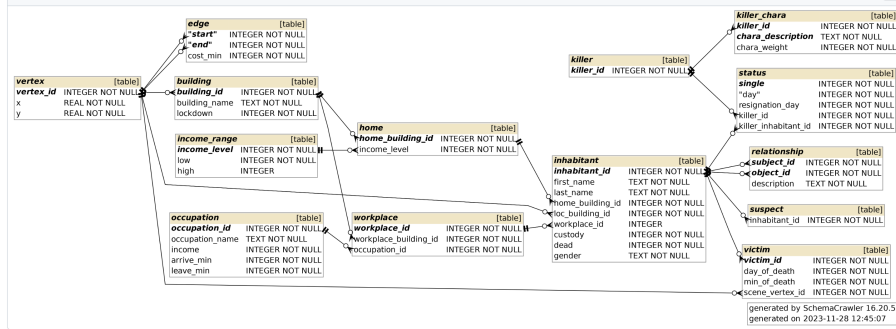


Figure 3: ER model in the fifth week

systematically removed, streamlining the model for enhanced clarity and efficiency. Noteworthy adjustments were made to certain attributes, such as the *killer* table, where a redesign rendered its attributes multi-valued. This modification aligns with conventional ER model design principles. While a relational model have been converted from this ER model, it was ultimately not used as the final database design. The final schema, however, did incorporate some changes made in the ER model, the following section highlights some of these changes.

3.3 The Final Version

The Final Database Schema is presented in Figure 3. This changes made to form this version of the schema reflected several issues that were not considered in the initial schema.

3.3.1 Key Changes

inhabitant Separation of *name* to atomic components first and last names. This change was made to make build relationships between inhabitants via the *relationship* table. One challenge faced in the generation of test dataset was the formation of resonable relationships between inhabitants. With the separation into first and last name, inhabitants with the same last name can be easily placed in *relationship* to be “relatives”.

killer and killer_chara *killer* and *killer_chara* is entirely redesigned. Originally the *killer_id* and *chara* attribute formed the primary key of table *killer*, this eventually proved to be an issue in the implementation of several queries including Victim Selection; this is because many tables needed to form a foreign key reference with the *killer_id* attribute in the *killer* table. This problem was address by having the *killer_id* isolated to form the killer table on its own,

killer_chara is then moved to *killer_chara* to form an “id-characteristic” pair that uniquely identifies a characteristic of a killer.

occupation Two attributes, *arrive_min* and *leave_min* were added to the *occupation* table in order to add uniqueness to an inhabitant’s trajectory in a day.

4 Important Queries and Implementations

4.1 Path Finding and Generation

The path generation query serves a pivotal role in the simulation by generating a temporary table that captures the movements of all inhabitants during a round. Executed at the initiation of each new round, this query stands as one of the most crucial operations in the simulation. Its significance lies in introducing randomness to the actions of both inhabitants and the killer. The killer’s modus operandi heavily depends on the paths taken by both the inhabitants and themselves each day. The intricacies of the killing mechanism hinge on the precise intersection of paths, as a killer can only perpetrate a murder when an inhabitant’s and the killer’s trajectories coincide at the same period.

Path generation is divided to two steps: calculation of the shortest path using Dijkstra’s all pairs shortest path and generating a viable path for the inhabitant using information from the shortest path. Both steps are implemented recursively using triggers due to that stored function/stored procedure is not supported by SQLite. Since SQLite is executed in the same process as the application, there would be no performance benefits in using stored function/stored procedure compared to a client-server setup where the stored code could be run in the server side as opposed to the client side.

4.1.1 All Pairs Shortest Path Algorithm Via Dijkstra’s Algorithm

The *dist* table as shown in Listing 1 stores the shortest distance from *src* to *dst* in *d*. The *visited* attribute is a boolean variable that is set to true when the *dst* vertex is added to the set of visited vertices starting from *src* as described in Dijkstra’s algorithm.[2] A visited set is maintained for each *src*, which indicates that the shortest distance from *src* to these vertices are known.

```

CREATE TEMP TABLE dist(
    src      INTEGER NOT NULL,
    dst      INTEGER NOT NULL,
    d        INTEGER,
    visited  INTEGER NOT NULL CHECK(visited IN (0, 1)),
    PRIMARY KEY(src, dst)

```

Listing 1: The *dist* table.

Dijkstra’s algorithm is implemented as a recursive trigger.[3]. Initially, the distance between all pairs of vertices are null as shown in Listing 2. The distance between each vertex and itself is then set to 0 and each source vertex is added to the visited set starting from itself. By updating the *visited* attribute of each source vertex, the trigger is fired.

```

-- Initialize the table with NULL distances.
INSERT INTO dist
    SELECT a.vertex_id, b.vertex_id, NULL, FALSE
    FROM vertex AS a, vertex AS b;

-- Set the distance from each vertex to itself to be 0 and
-- initiate the recursive trigger.
UPDATE dist
    SET d = 0, visited = TRUE

```

Listing 2: The initial insertion of null distance pairs and updating distances from each vertex to themselves to 0.

During each execution of the trigger shown in Listing 3, the known shortest distances and the visited set starting from *NEW.src* are updated. Firstly, the sums of the distance between *NEW.src* and *NEW.dst* plus the edge cost from *NEW.dst* to the adjacent vertices are used to update the shortest distances to these adjacent vertices. Then, the unvisited vertex that is the closest to *NEW.src* is added to the visited set. This then fires the trigger recursively.

```

-- Called when a vertex is added to the "visited" set.
CREATE TEMP TRIGGER update_dist AFTER UPDATE OF visited ON dist
BEGIN
    -- Update the shortest distance to the neighbor of this
    ↪ vertex.
    UPDATE dist
        SET d = nd
        FROM (SELECT end AS nv, (NEW.d + cost_min) AS nd
              FROM edge
              WHERE start = NEW.dst)
    WHERE src = NEW.src AND dst = nv AND visited = FALSE AND
    ↪ (d IS NULL OR d > nd);

    -- Add the closest vertex that has not been visited into
    ↪ the "visited" set.
    UPDATE dist
        SET visited = TRUE
    WHERE src = NEW.src AND visited = FALSE
        AND d = (SELECT MIN(d) FROM dist WHERE visited =
    ↪ FALSE);

```

Listing 3: Dijkstra’s algorithm implemented using a recursive trigger.

4.1.2 Path Generation

A inhabitant’s daily path could be described by location-time pairs, indicating that they are at a specific vertex for a specific period time of their day. Their path is constrained by their daily routine, such as when they get up, when they must be at work, when they could leave work, and when they must be at home. The *src_dst* table shown in Listing 4 describes such constraints, specifying that the inhabitant with *inhabitant_id*, they may leave *src* no earlier than *t_src* and must arrive at *dst* no later than *t_dst*. For each inhabitant with a job, their daily routine could be described with two tuples in the *src_dst* table. One constraining the time they go from home to work, and the other constraining the opposite.

```

CREATE TEMP TABLE src_dst(
    inhabitant_id INTEGER NOT NULL,
    src            INTEGER NOT NULL,
    dst            INTEGER NOT NULL,
    t_src          INTEGER NOT NULL,
    t_dst          INTEGER NOT NULL,
    PRIMARY KEY(inhabitant_id, src, dst)
);

```

Listing 4: The *src_dst* table.

The resulting path in location-time pairs is stored in the *loc_time* table shown in Listing 5. Each tuple describes that the inhabitant with *inhabitant_id* will arrive at *vertex_id* on the minute of *arrive* attribute and leave on the minute of *leave* attribute. The *dst* and *t_dst* attributes are the same as those two attributes in *src_dst* for convenience.

```

-- Indicate that an inhabitant arrives and leaves at a certain
-- vertex at certain times.
CREATE TEMP TABLE loc_time(
    inhabitant_id INTEGER NOT NULL,
    vertex_id      INTEGER NOT NULL,
    arrive         INTEGER NOT NULL,
    -- This is not forced to be NOT NULL so there wouldn't be
    -- errors with unconnected graph or impossible
    -- constraints.
    leave          INTEGER,
    dst            INTEGER NOT NULL,
    t_dst          INTEGER NOT NULL,
    PRIMARY KEY(inhabitant_id, vertex_id,
    -- arrive)
);

```

Listing 5: The *loc_time* table.

The daily path of an inhabitant is generated by randomly walking from *src* to *dst* while satisfying the constraints. This is implemented using a recursive trigger. Initially, the inhabitant waits at the source vertex for some random time limited by the fact that they must be able to at least reach the destination on time taking the shortest path. These initial location-time pairs are then inserted into *loc_time* as shown in Listing 6.

```

-- Wait randomly at the source vertices and initiate the
↪ recursive trigger.
INSERT INTO loc_time
    SELECT inhabitant_id, src, t_src,
           t_src + ABS(RANDOM()) % (t_dst - t_src + 1
           - (SELECT MIN(d) FROM dist WHERE dist.src =
           ↪ src_dst.src AND dist.dst = src_dst.dst)),
           dst, t_dst
    FROM src_dst;

```

Listing 6: Insertion of initial location-time pairs.

Inserting the initial conditions fires the trigger shown in Listing 7 recursively. For each tuple inserted into *loc_time* where the destination is not reached, a neighboring vertex is randomly selected to be the next vertex that the inhabitant advances to. This vertex, however, has to satisfy the condition that the current time, plus the time cost of going to that vertex, and plus the shortest distance of going from that vertex to the destination does not exceed *t_dst*. A random portion of the time available to spare after taking all these into account is then chosen to be the duration that the inhabitant waits at the neighboring vertex. The new location-time pair is then also inserted into the table.

```

-- Called each time a new location-time decision is made and thus
-- → inserted into the table.
CREATE TEMP TRIGGER insert_loc_time AFTER INSERT ON loc_time
WHEN
    -- Stop when the destination is reached.
    NEW.vertex_id <> NEW.dst
BEGIN
    -- Traverse one edge and wait at the neighboring vertex
    -- → for a random amount of time,
    -- such that the current time, plus the time to traverse
    -- → the edge,
    -- plus the shortest time to go from the neighboring
    -- → vertex to the destination,
    -- and plus the random waiting time does not exceed the
    -- → constraint.
    -- Then randomly choose one plausible edge to go through.
    INSERT INTO loc_time
        SELECT NEW.inhabitant_id, end, NEW.leave +
            cost_min,
            NEW.leave + cost_min + ABS(RANDOM()) %
            (NEW.t_dst - (NEW.leave + cost_min)
            + 1
            - (SELECT MIN(d) FROM dist WHERE
            dist.src = end AND dist.dst =
            NEW.dst)),
            NEW.dst, NEW.t_dst
        FROM edge
        WHERE start = NEW.vertex_id
            AND NEW.leave + cost_min +
            (SELECT MIN(d) FROM dist WHERE dist.src
            = end AND dist.dst = NEW.dst)
            <= NEW.t_dst
        ORDER BY RANDOM()

```

Listing 7: Path generation implemented using a recursive trigger.

4.2 Victim Selection

The Victim Selection script performs the crucial task of choosing a victim for the killer when executed. This script follows the execution of the previously explained path generator. The selection process considers multiple factors. Foremost among these is the intersection of paths between the killer and potential victims, a pivotal element in determining the feasibility of a murder. Additionally, the script accounts for the killer’s proclivity to target specific segments of the population, as specified in the *killer_chara* table. By integrating these

considerations, the script not only adds an element of realism to the simulation but also introduces a layer of complexity that aligns with the characteristics and tendencies attributed to the killer in the simulation environment.

The script presented in Listing 8 is designed to construct two temporary tables, namely *pot_victim* and *weighed_pot_victim*. The former serves as a representation of potential victims, while the latter refines this selection by sorting potential victims based on their weight, determined by the killer’s characteristics. The *pot_victim* table contains information on inhabitants deemed viable targets for the killer – those with intersecting paths on the given *day* of the simulation. The population of this table involves a query that assesses inhabitants sharing a vertex with the killer at the same time. The utilization of the `max` and `min` functions tracks overlapping time intervals, subsequently facilitating the generation of a time of death.

The initialization of *weighed_pot_victim* is done through a joining of pertinent tables, incorporating information for weight determination, including details from the Home and Workplace tables. The inclusion of the *killer_chara* table facilitates the assignment of weights based on the distinct characteristics of the killer. Within the **WHERE** clause, a **CASE** statement dynamically evaluates the characteristics fulfilled by each inhabitant. The subsequent **SELECT** clause projects pairs of inhabitants and their respective characteristics, accompanied by the corresponding weights extracted from the *killer_chara* table.

Afterwards, *weighed_pot_victim* is used in the final query that groups each inhabitant by their id and have their total weight calculated using an aggregate function. The inhabitant is sorted by descending order based on the calculated weight with information summarized in the **SELECT** clause, the time of death is calculated by randomly selecting a period from the overlapping time interval calculated from *pot_victim*. Using this final query, the victim is chosen by fetching the first tuple from python.

```

DROP TABLE IF EXISTS weighed_pot_victim;
CREATE TEMP TABLE weighed_pot_victim (
    inhabitant_id INTEGER,
    description TEXT,
    chara_weight INTEGER,
    vertex_id INTEGER
);

DROP TABLE IF EXISTS pot_victim;
CREATE TEMP TABLE pot_victim(
    inhabitant_id INTEGER,
    vertex_id INTEGER,
    start_min INTEGER,
    end_min INTEGER
);

INSERT INTO pot_victim
SELECT DISTINCT B.inhabitant_id, B.vertex_id, MAX(A.arrive,
↪ B.arrive), MIN(A.leave, B.leave)
FROM status, loc_time AS A, loc_time AS B
WHERE A.inhabitant_id = status.killer_inhabitant_id AND
      A.vertex_id = B.vertex_id AND
      A.arrive <= B.leave AND
      A.leave >= B.arrive;

INSERT INTO weighed_pot_victim
WITH killer_info AS (
    SELECT inhabitant.* FROM status, inhabitant
    WHERE status.killer_inhabitant_id =
↪ inhabitant.inhabitant_id
)
SELECT DISTINCT inhabitant_id, chara_description, chara_weight,
↪ vertex_id
FROM pot_victim NATURAL JOIN inhabitant AS i NATURAL JOIN home AS
↪ h NATURAL JOIN workplace,
    killer_chara AS k, status

```

Listing 8: Victim Selection script.


```

WHERE status.killer_id = k.killer_id AND
CASE
    WHEN k.chara_description = "low income" THEN
        ⇨ income_level = "low income"
    WHEN k.chara_description = "high income" THEN
        ⇨ income_level = "high income"
    WHEN k.chara_description = "neighbor" THEN EXISTS(
        SELECT * FROM killer_info
        WHERE h.home_building_id =
            ⇨ killer_info.home_building_id
    )
    WHEN k.chara_description = "rapist" THEN EXISTS(
        SELECT * FROM killer_info
        WHERE killer_info.gender <> i.gender
    )
    WHEN k.chara_description = "colleague" THEN
        ⇨ EXISTS(
            SELECT * FROM killer_info WHERE
            ⇨ killer_info.workplace_id =
            ⇨ i.workplace_id
        )
    ELSE EXISTS(
        SELECT * FROM relationship
        WHERE subject_id = killer_inhabitant_id AND
            object_id = inhabitant_id AND
            relationship.description = "Relative"
    )
END;

```

Listing 8: Victim Selection script.

4.3 Witness Count

The Witness Count query counts the number of times each inhabitant has been witnessed on a certain vertex. It aims at reflecting that in many detective novels, the police can put up posters asking the general public about whether they have seen suspicious people in certain places. Our implementation is a bit naive as it simply counts the number of witnesses regardless of the “suspiciousness”.

As shown in Listing 9, the query counts the number of times that a given inhabitant a has been on the same vertex as any inhabitant (witness) b at the same time. It checks for temporal overlaps by comparing the time that a and b have arrived and left the given vertex. Simultaneously, the inhabitant a must not be dead before reaching the given vertex and that the witness b must be alive at the end of the turn to tell the tale.

```

SELECT a.inhabitant_id, MIN(a_info.first_name),
↪ MIN(a_info.last_name),
COUNT(b.inhabitant_id) AS c
FROM loc_time AS a
JOIN inhabitant AS a_info
ON a_info.inhabitant_id = a.inhabitant_id
JOIN loc_time AS b
ON a.inhabitant_id <> b.inhabitant_id
AND a.vertex_id = b.vertex_id
AND ((a.arrive <= b.arrive AND b.arrive <= a.leave)
OR (a.arrive <= b.leave AND b.leave <= a.leave))
AND (a_info.dead = FALSE OR a.arrive
<= (SELECT MIN(min_of_death) FROM victim WHERE
↪ victim_id = a.inhabitant_id))
AND b_info.dead = FALSE
JOIN inhabitant AS b_info
ON b_info.inhabitant_id = b.inhabitant_id
WHERE a.vertex_id = ?
GROUP BY a.inhabitant_id
ORDER BY c DESC

```

Listing 9: Witness Count query.

4.4 Victim Commonality

The *commonality* view is a perspective that reveals recurring patterns among the victims within a particular simulation instance. This query extracts and presents attributes along with their corresponding values that have manifested across multiple victims. The significance of this view lies in its ability to succinctly summarize information about the victims throughout the simulation. Consequently, it plays a pivotal role in the profiling of the killer. This functionality proves indispensable not only in the context of the simulation but also mirrors its critical counterpart in real-world scenarios, emphasizing its role in enhancing the understanding and analysis of patterns in criminal cases.

The query is written to be a view because it will be frequently updated and accessed, in every round of the simulation.

Listing 10 presents the definition of the view. Initially, a local query table named *victim_info* is created using the *WITH* clause. This table proves instrumental in subsequent subqueries and is, therefore, invoked within the *WITH* clause. The ensuing query employs a subquery named 'attributes' within the *FROM* clause. 'Attributes' encapsulates a collection of victim information attributes extracted from *victim_info* and projects them in the format of name and value pairs. This operation essentially assembles tuples for each victim alongside their respective attributes. Notably, the process necessitates the use of *UNION* to concatenate diverse attributes that must be considered, ensuring

a comprehensive dataset. Additionally, attributes not of datatype TEXT are explicitly cast to align with the specified schema. After 'attributes' is queried, it is grouped by the name and value pairs and filtered to include only pairs that recurred at least thrice, the pair is then projected by the SELECT clause along with the recurrence and sorted by descending order.

```
CREATE VIEW commonality AS
WITH victim_info AS(
    SELECT * FROM
        inhabitant LEFT OUTER JOIN home USING(home_building_id),
        ⇨ victim
    WHERE inhabitant_id = victim_id
)
SELECT attribute_name, attribute_value, COUNT(*) AS attribute_cnt
FROM (
    SELECT 'name' AS attribute_name, first_name AS
        ⇨ attribute_value FROM victim_info
    UNION ALL
    SELECT 'home_building_id', CAST(home_building_id AS TEXT)
        ⇨ FROM victim_info
    UNION ALL
    SELECT 'workplace_id', CAST(workplace_id AS TEXT) FROM
        ⇨ victim_info
    UNION ALL
    SELECT 'gender', gender FROM victim_info
    UNION ALL
    SELECT 'scene_vertex_id', CAST(scene_vertex_id AS TEXT) FROM
        ⇨ victim_info
    UNION ALL
    SELECT 'min_of_death', CAST(min_of_death AS TEXT) FROM
        ⇨ victim_info
    UNION ALL
    SELECT 'income_level', CAST(income_level AS TEXT) FROM
        ⇨ victim_info
) AS attributes
GROUP BY attribute_name, attribute_value
HAVING COUNT(*) > 2
ORDER BY attribute_cnt DESC;
```

Listing 10: Victim commonality view.

5 Simulation Demo

The following is a minimal user interface created using DearPyGui.[4]

5.1 Main Menu

Figure 4 presents the Main Menu of the simulator, the user has three options: start a new game, load a saved game, and quit the simulator.



Figure 4: Main Menu.

5.2 Load Screen

In figure 5 the Load Screen is presented, here the user can find previously saved games. The user has the ability to load a saved game or delete it.

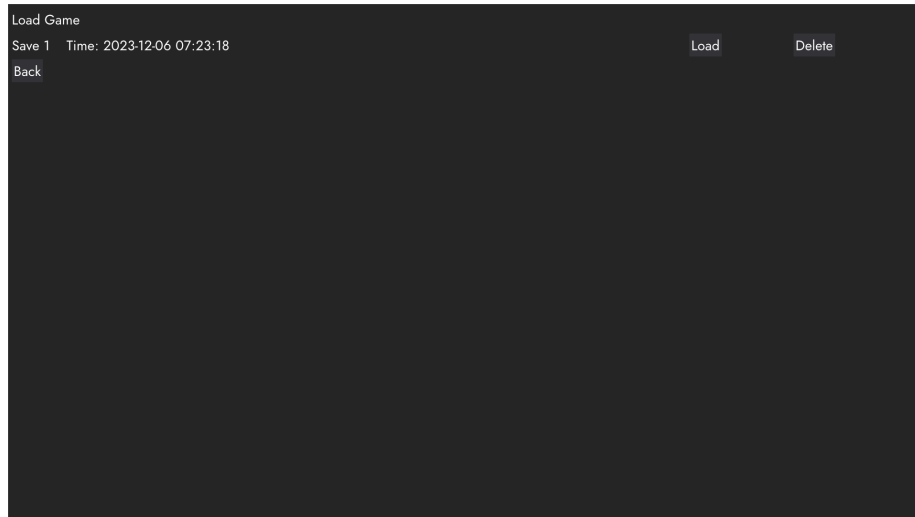


Figure 5: Load Screen.

5.3 Game Window

After loading or starting a game, the user enters the Game Window, which displays a variety of information and tools as presented in Figure 6. On the left of the screen a scrollable map is displayed. This map identifies the individual vertices as white circles, and the buildings as red squares. The white lines connecting each vertex are the edges. Every vertex is labeled by its ID, every building is labeled by its name, and every edge is labeled by its weight (*cost_min* attribute).

On the right hand side of the screen is a query box for searching inhabitants with specific attributes, the search results are on the bottom. The user is free to enter any filter information into the query box. In this case, the user searches for all inhabitants with occupation equal to “Agricultural engineer,” the query results, “Rebecca,” “Mark,” etc. are therefore the inhabitants that have the occupation “Agricultural engineer.”

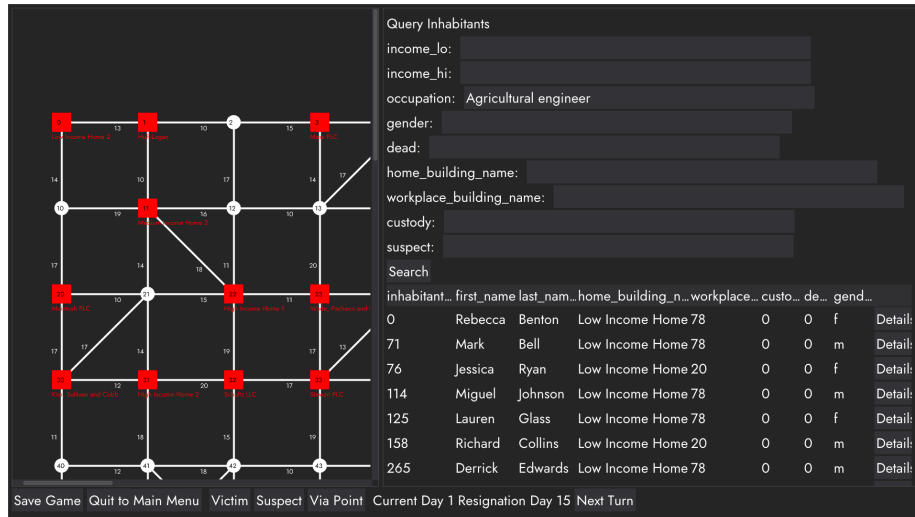


Figure 6: Game Window.

5.4 Building Detail and Inhabitant Detail

When the buildings on the map is clicked, a screen displaying the information of the building is provided as seen on the left view of Figure 7. It displays components such as the name of the building, “Mcintosh PLC” in this case, whether if it is under lockdown, the occupations that work in this building, and the inhabitants that visit this building. The user is capable of placing a building into lockdown using “Toggle Lockdown,” which disables all edges involving this building and thus limiting the paths of various inhabitants in the next turn.

When the user wish to view the details of an inhabitant, they can click on the “Details” button to the right of every inhabitant tuple. When clicked, a screen on the right will pop up displaying the details of the selected inhabitant; in the case of Figure 7, we are looking at details of “Rebecca Benton.” On this screen, there are a few tools that can be used, such as “Toggle Suspect,” such sets or unsets the inhabitant as a suspect, or “Accuse” which means this inhabitant is selected as the person the user believes is the killer, the program will provide a feedback of whether if the user has selected the correct inhabitant as the killer as shown in Figure 8.

Mcintosh PLC Close Toggle Lockdown				Inhabitant Detail Close Accuse Toggle Suspect			
Workplace, not under lockdown				inhabitant_id 0			
Occupations				first_name Rebecca			
occupation_name	income	arrive_min	leave_min	last_name Benton			
Air cabin crew	25000	540	1080	custody 0			
Psychotherapist, child	74000	480	960	dead 0			
Osteopath	26000	600	1080	gender f			
Inhabitants				home_building_name Low Income Home 2			
inhab_id	first_name	last_name	home_building_name	home_lockdown 0			
14	Christy Allison	Medium Income	Incor 27	workplace_building_name Moss PLC			
16	Kristin Vasquez	Low Income	Inc 26	workplace_lockdown 0			
96	Manuela Weave	Low Income	Inc 25	occupation_name Agricultural engineer			
142	Gloria Lewis	Low Income	Inc 26	income 10000			
180	Ashley Atkins	Low Income	Inc 25	arrive_min 540			
195	Kimberly Bradford	Medium Income	Incor 27	leave_min 1080			
295	Deborah Ayers	Medium Income	Incor 27	Relationships			
414	Michael Smith	Low Income	Inc 25	inhabitant_id object_first_name object_last_name description			
446	Christine Chen	Low Income	Inc 26	71 Mark Bell Colleague Details			
461	Dawn Jackson	Medium Income	Incor 27	114 Miguel Johnson Colleague Details			
Save Game Quit to Main Menu Victim Suspect Via Point				Current Day 1 Resignation Day 15 Next Turn			

Figure 7: Building Detail and Inhabitant Detail.

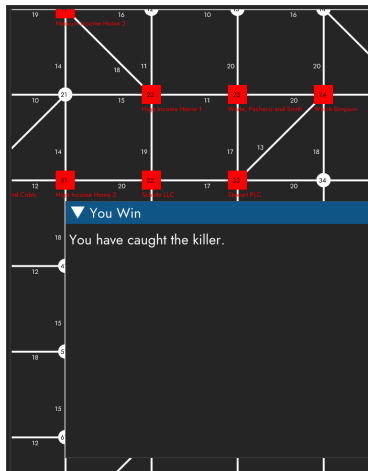
				Inhabitant Detail Close Accuse Toggle Suspect			
				inhabitant_id 999			
				first_name Light			
				last_name Yagami			
				custody 0			
				dead 0			
				gender m			
				home_building_name Low Income Home 2			
				0			
				0			
				Trade union research officer			
				27000			
				600			
				960			
				object_first_name object_last_name description			
<div>You Win</div> <div>You have caught the killer.</div>							
Save Game Quit to Main Menu Victim Suspect Via Point				Current Day 3 Resignation Day 15 Next Turn			

Figure 8: Winning the game.

5.5 Other Features

The Game Window also provides several other features such as Victim Window, Suspect Window and the “Via Point” query. When the “Victim” button is clicked, the information on the inhabitants killed in this simulation is displayed. In Figure 9, three inhabitants are killed and the attributes shared among the three inhabitants are displayed, the table represents the name of the attribute,

value of the attribute and how many times the attribute has occurred among the victims.

Clicking on “Suspect” opens a window that displayed the information on the inhabitants placed in the suspect list, the example displayed on the top right corner of Figure 9 shows the information of “Rebecca” who is marked as a suspect by the user.

Clicking on “Via Point” opens another window with a query box. Here, the user can enter a starting vertex, ending vertex, and a time frame. The results of the query shown in Figure 9 are vertices that an inhabitant is capable of visiting in a path, given that the inhabitant starts on vertex 1, and has 60 minutes to get to vertex 10.

Relationships of an inhabitant can be seen by scrolling down in the “Details” window of an inhabitant; on the right of Figure 10 shows us the relationships an inhabitant has. The ID of the “object” is displayed along with his/her name and a description of the relationship.

The “Witness” information can also be found on the “Details” window of a building. Similarly with relationship, the user has to scroll down in the Details window to find the inhabitants that were seen visiting this building in a particular turn. An example of this is also provided to the left of Figure 10.

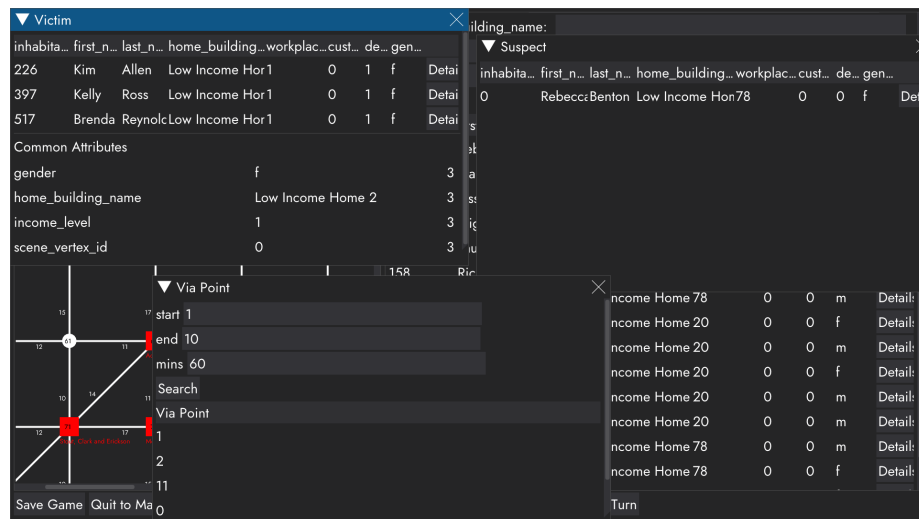


Figure 9: Victim Window, Suspect Window, and Via Point Window.

911	Clinton	Zimmerman	Medium	Incor	27	0	0	m	Det.
Witness Counts									
inhabitant_id	first_name	last_name	count						
481	Kelsey	Jenkins	36	Details					
352	Kari	Gordon	35	Details					
635	Sarah	Harris	23	Details					
446	Christopher	Chen	21	Details					
793	Shawn	Webster	20	Details					
16	Kristin	Vasquez	19	Details					
757	Lisa	Foster	18	Details					
862	Jaime	Lopez	18	Details					
14	Christy	Allison	17	Details					
461	Dawn	Jackson	17	Details					
477	Tammy	Thompson	17	Details					
648	Darrell	Harris	17	Details					
677	Keith	Wood	17	Details					
679	Joseph	Solis	16	Details					
801	Anthony	Coleman	16	Details					
96	Manuel	Weaver	15	Details					
Relationships									
inhabitant_id	object_first_name	object_last_name	description						
71	Mark	Bell	Colleague	Details					
114	Miguel	Johnson	Colleague	Details					
125	Lauren	Glass	Colleague	Details					
265	Derrick	Edwards	Colleague	Details					
439	Donald	Rice	Enemy	Details					
450	Jonathan	Benton	Relative	Details					
518	Travis	Brown	Friend	Details					
774	Christopher	Clark	Colleague	Details					
780	Jacqueline	Hernandez	Colleague	Details					

Figure 10: Witness Counts and Relationships query.

5.6 Save Screen

Finally, as shown in Figure 11, when “Save Game” is clicked, the user is allowed to save a particular instance of the simulation either by overwriting a previously save instance using “Overwrite,” or save as a new instance using “New Save.”

Save Game

Save 1 Time: 2023-12-06 07:23:18

Overwrite

Delete

New Save

Back

Figure 11: Save Screen.

6 Comparison Against Existing Application: SQL Murder Mystery

The SQL Murder Mystery is an interactive educational tool developed to impart proficiency in Structured Query Language (SQL).[1] Originating from the Knight Lab at Northwestern University, this gamified platform engages users in a murder mystery scenario, tasking them with the role of a detective tasked with solving a crime by querying a relational database. While its concept is similar to the project we have created, we believe our simulator offers a much more extensive database and functionality that requires a series of complex queries to implement. It is also noteworthy to mention that the application developed by Knight Lab is geared towards individuals with the purpose of learning introductory SQL queries while our project aims to train the user's deductive skills. This difference can be most explicitly seen from the user interface that our group had developed for the purpose of adding a layer of abstraction to the user.

7 Conclusion

The Detective Simulation project successfully encapsulates the intricate aspects of a murder mystery into an immersive game centered around database querying. Through the presented database schema and its refinement process, the structural foundation that enables the game's functionalities is established. Pivotal queries like Path generation and Victim Selection are explored, elucidating the interplay between inhabitants, the murderer, and the overarching simulation environment. These queries constitute the simulation's computational backbone, facilitating intricate behaviors and mirroring real-world complexity.

Additionally, the project demonstrates proficiency in employing recursive triggers to implement complex logic like randomized Path Generation. The use of views to summarize information, as shown by the Victim Commonality example, also reflects an apt utilization of database constructs for analysis.

The graphical user interface, while rudimentary, grants users investigative tools to deduce clues about the murderer's identity. The emulator environment, coupled with the relationship queries, allows methodological tracking of suspects. Saving/loading game states also enables progression over multiple sessions.

Overall, the project presents a comprehensive system melding deductive reasoning and database skills. While scope remains for refinement of UI/UX elements and better feedback to the player after wrong accusations, it stands as a robust foundation readily extensible to more advanced implementations. The work exhibits database design and querying capabilities applicable in both real-world software systems and interactive applications.

8 Contributions

- Database schema version 1:
 - Collective effort amongst Isaac, Yuxiang, and Shanruo
 - Annotations written by Shanruo
- Database schema version 2:
 - ER modeling designed by Isaac
 - Changes to final database schema reflected by Yuxiang
- Database data generator:
 - building, income_range, home, occupation and workplace written by Yuxiang
 - vertex, edge, inhabitant, killer, killer_chara, status, relationship, victim written by Isaac
- Query implementation:
 - Shortest path, inhabitant location-time pair generation, witness count, and plausible via point implemented by Yuxiang
 - Victim selection, victim commonality, inhabitant query implemented by Isaac
 - Isaac also experimented with implementing the inhabitant location-time pair generation in procedures, which is not used in the final game
- Gameplay implementation:
 - Simple queries that provide basic information to the users are implemented by both Yuxiang and Isaac
 - Load-save functionality is written by Yuxiang
 - Checking the end-game condition and a view for edges that are not blocked is written by Isaac
- Front-End UI:
 - Individual effort by Yuxiang

References

- [1] SQLite. (2022, April 27). *Datatypes In SQLite*. Retrieved from <https://www.sqlite.org/datatype3.html>
- [2] Wikipedia. (2023, December 2). *Dijkstra's algorithm*. Retrieved from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- [3] SQLite Forum. (2021, May 1). *Recursive query to find the shortest path*. Retrieved from <https://sqlite.org/forum/info/7c89903050369164>
- [4] Dear PyGui. (2023). GitHub Repository. Retrieved from <https://github.com/hoffstadt/DearPyGui>

References

- [1] Knight Lab. (2019). *SQL Murder Mystery*. Retrieved from <https://mystery.knightlab.com/>