Transportation Events Project

Database Theory and Design – CSCI 411

Terrance Wallace, Chris Harstad, and Kenneth Kippley

https://github.com/KINGTUT10101/TransportationEventsProject

Overview

In this project, we designed queries and a website to retrieve transportation events data. The raw data contained node, link, and event information in XML files. The nodes and links described an environment, and the events described a list of actions that took place over a time period.

The main difficulty of this project was designing an efficient SQL database to handle the events data. There were 20 different event types, each of which had different attributes. Only a small handful of attributes were shared among all event types. Solving this problem required clever thinking, pattern recognition, and compromises.

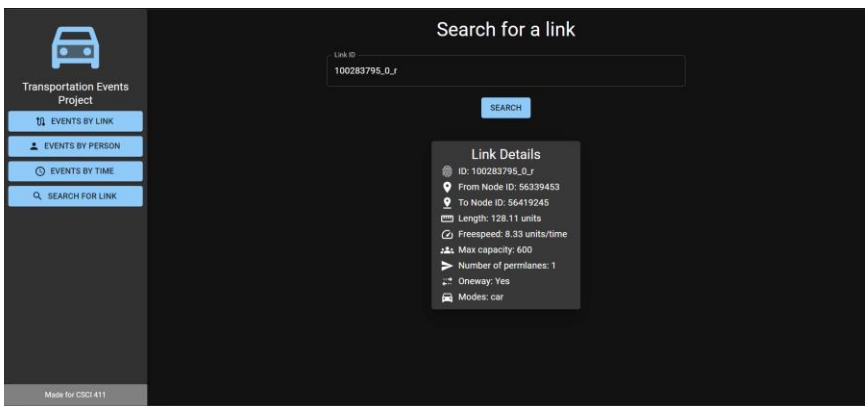

Figure 1 A screenshot of the project's frontend

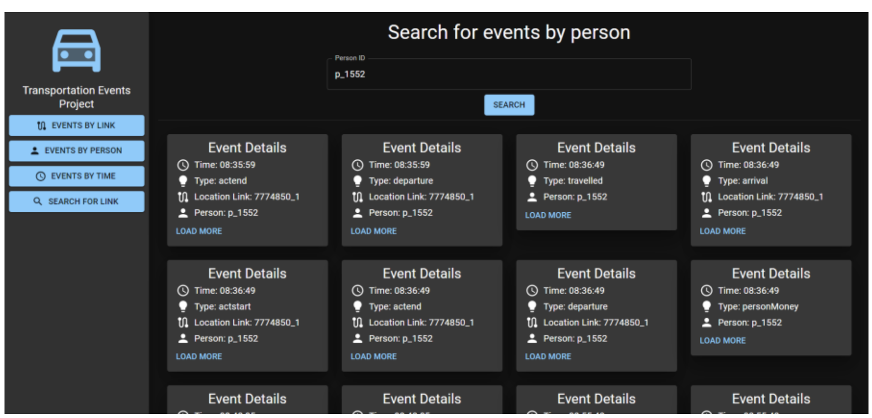

Figure 2 Another screenshot of the frontend

Frontend

The project's frontend was programmed in JavaScript by Terrance. It uses React, React Router, and Material UI. Each frontend function described in question 5 can be found on the sidebar. Clicking each of them will bring you to a new page where you can search the database. The frontend will connect to the backend and fetch the data dynamically when you search for an item. The information is then displayed as a series of cards on the screen. Since the events data has so many variable attributes, the event details cards will only show the common attributes by default. The additional attributes can be loaded and shown dynamically by pressing "show more". The search results also utilize pagination to improve performance. This means that only a small subset of the full result to loaded at any time. This makes navigation easier and reduces memory usage.

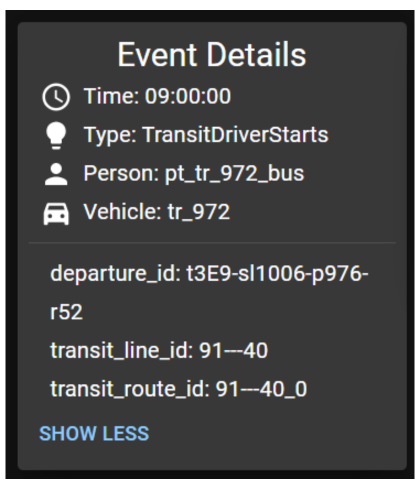

Figure 3 An event details card with the additional attributes already shown

```
async function handleLoadMore () {
  setLoadMore (true) // Disable the load more button

  let { event_type, event_id } = data

  await axios.get(`/api/specialEventData/${encodeURIComponent(event_type)}/${event_id}`).
    setSpecialData (response.data)
    let specialAttributes = getEventAttributes(response.data)
    setInitialAttributes (specialAttributes.slice(0, specialAttributeLimit))
    setExtraAttributes (specialAttributes.slice(specialAttributeLimit))

    if (specialAttributes.length === 0) {
      setShowToast (true)
    }
    else {
      setShowMore (true)
    }

  }).catch ((err) => {
    alert ("Error fetching data\n" + err)
    setInitialAttributes ([])
    setExtraAttributes ([])
  })
};
```

Figure 4 Event details component function that dynamically loads the additional attributes

```
async function getData () {
  setWaiting (true)

  await axios.get(`/api/person/${encodeURIComponent(personID)}?page=${page}&count=${itemsPerPage}`).then((response) => {
    setEventsData(response.data);
  }).catch ((err) => {
    alert ("Error fetching content (please check that the ID is correct)\n" + err)
    setEventsData ([])
  })

  setWaiting (false)
}
```

Figure 5 Search page function that loads the common attributes for each event shown on the current page

```
async function nextPage (event, value) {
  setPage (value)
  // Data will update automatically in the useEffect hook
  // This fixes a bug where users had to click the page twice to update the data
}

React.useEffect (()=>{
  if (firstRender){
    setFirstRender (false)
  }
  else {
    getData ()
  }
}, [page])

React.useEffect (()=>{
  if (!waiting) {
    window.scrollTo(0, 0)
  }
}, [waiting])
```

Figure 6 Search page pagination logic. It is hooked up to a Material UI pagination element (not shown)

Backend

As for the backend, Kenneth implemented the original version and Terrance implemented pagination and a route that fetches additional event attributes. A big problem we faced was JavaScript's string length limitation. This would crash the backend if the query returned too much data. To solve this, we added pagination. Now, all the queries require page and item count parameters. Our initial version also queried the database for all the events attributes, which created huge query results with tons of null values. To solve this, we now only return the attributes common to each event type. If the frontend needs the additional attributes (like when the user clicks "show more"), it must request that data manually. Combined, these methods greatly reduce the amount of data we need to send over the network.

```
router.get('/event/:linkID', async (req, res) => {
  try{
    // Get the page and offset for the query
    if(!parseInt(req.query.count) || req.query.count<0) req.query.count = 20;
    const limit = req.query.count;
    if(!parseInt(req.query.page) || req.query.page<1) req.query.page = 1;
    const offset = req.query.count*(req.query.page - 1);

    const lid = req.params.linkID;

    const result = await db.query(`SELECT *
                                   FROM event_data
                                   WHERE link_id = $1
                                   ORDER BY event_time, event_id
                                   LIMIT $2 OFFSET $3;`,[lid,limit,offset]);
    if (result.rows.length === 0) res.status(404).send('Not Found');
    else res.status(200).send(result.rows);
  }catch (error) {
    console.error('Error', error);
    res.status(500).send('Server error');
  }
});
```

Figure 7 An API route that gets a page of events data containing only their common attributes

```
router.get('/count/event/:linkID', async (req, res) => {
  try{
    const lid = req.params.linkID;

    const result = await db.query(`SELECT COUNT(*)
                                   FROM event_data
                                   WHERE link_id = $1;`,[lid]);
    if (result.rows.length === 0) res.status(404).send('Not Found');
    else res.status(200).send(result.rows[0]);
  }catch (error) {
    console.error('Error', error);
    res.status(500).send('Server error');
  }
});
```

Figure 8 An API route that counts the total number of events associated with a search. It's used within the pagination system in the frontend

```
router.get('/specialEventData/:eventType/:eventID', async (req, res) => {
  try{
    const eid = req.params.eventID; // Event ID
    const sourceTable = typeMap[req.params.eventType] // Source table for special columns

    const result = await db.query(`SELECT *
                                   FROM ${sourceTable}
                                   WHERE event_id = $1
                                   LIMIT 1;`,[eid]);
    if (result.rows.length === 0) res.status(404).send('Not Found');
    else res.status(200).send(result.rows[0]);
  }catch (error) {
    console.error('Error', error);
    res.status(500).send('Server error');
  }
});
```

Figure 9 An API route that gets the additional attributes for a specific event

Database design

As for the database, we decided to use PostgreSQL since we had used it for our project in CSCI 414. Chris set up and implemented the database. Implementing tables for the node and link data was straightforward. However, we faced issues when we analyzed the event data. There were 20 event types, each of which had different attributes. Only a handful of attributes were common to each type. The main problem was designing an efficient database that would avoid wasting space on redundant data. We originally considered using a huge table that would contain every event type. However, this would've wasted lots of space with null values. We also considered making a table for each event type, but this would've been slow to query due to all the joins that would be needed.

After some clever observations, we settled on our final design that uses only 14 tables. Kenneth noted that many of the event types mirrored each other and could be combined into one table. Terrance also suggested adding a table to contain the attributes common to each type. We also noted that the most common event types by far were actstart and actend, which contained a link attribute. We added this to the common attributes table to speed up performance when searching for events by link ID. Finally, we added some indexes to speed up query performance event further.

```
/*Indexes*/
CREATE INDEX ev_time ON event_data USING BTREE(event_time);
CREATE INDEX ev_link ON event_data USING BTREE(link_id);
```

Figure 10 The database indexes we used

Our final ER diagram can be found on the next page.

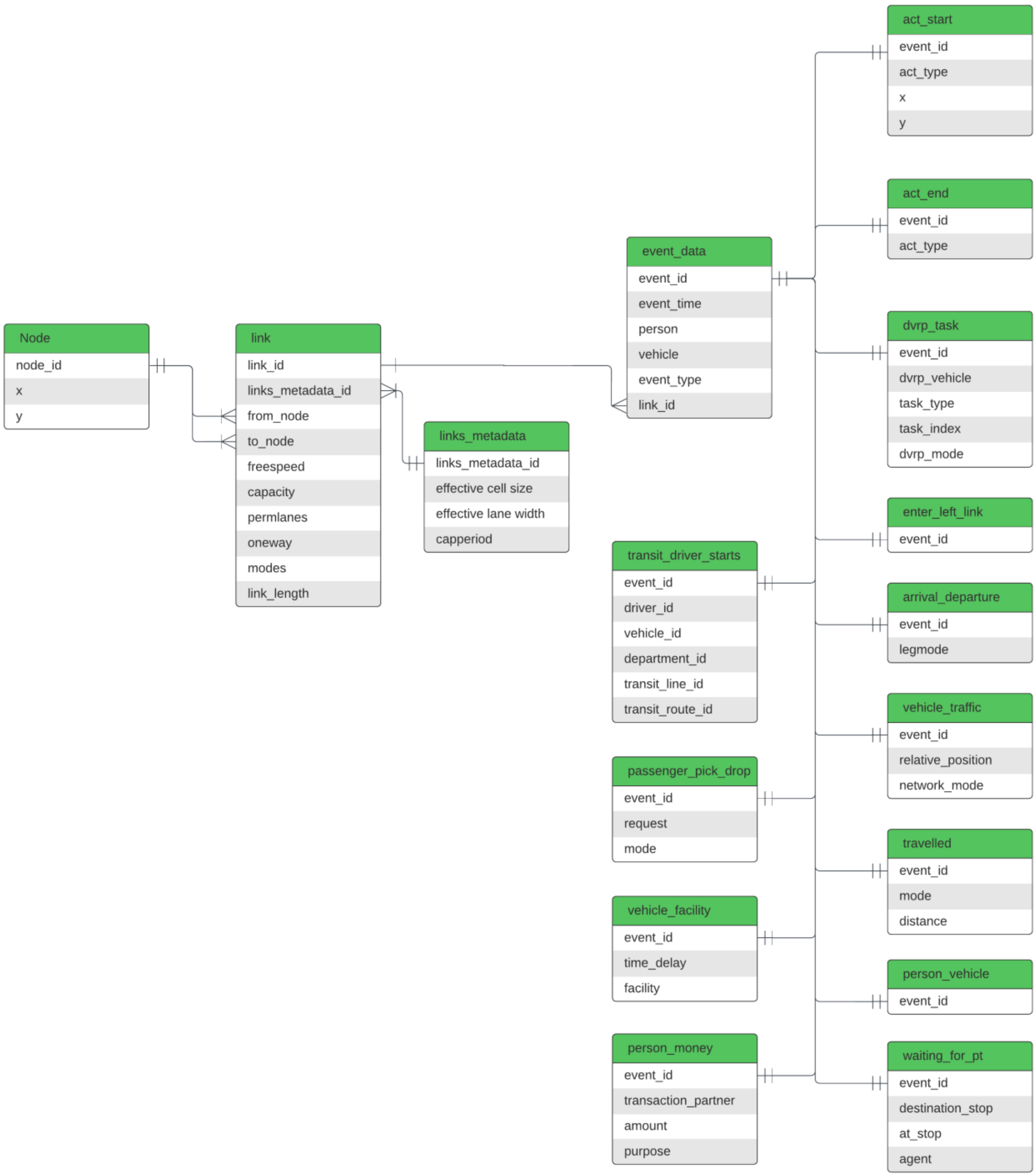

Figure 11 The ER diagram for the entire project

Data importing

The next problem we faced was importing the data. We used pgAdmin to interact with our database, but it didn't support importing XML data. To solve this, Terrance converted the node and link data into CSV files using Microsoft Excel. From there, it was trivial to import that data using the pgAdmin UI.

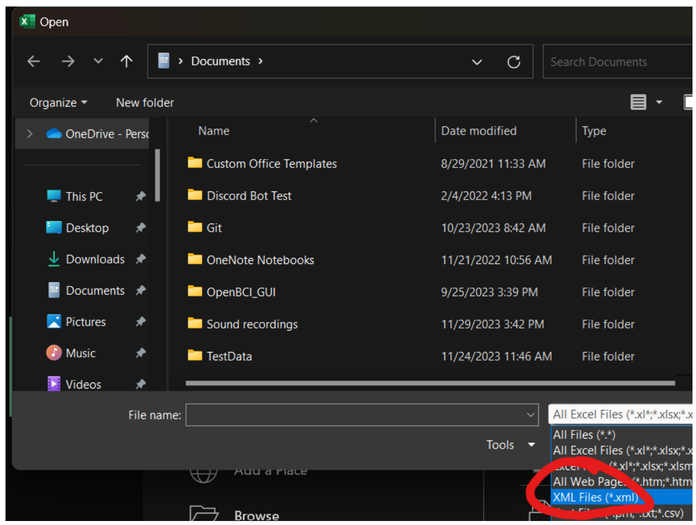

Figure 12 The option to import XML data into Excel

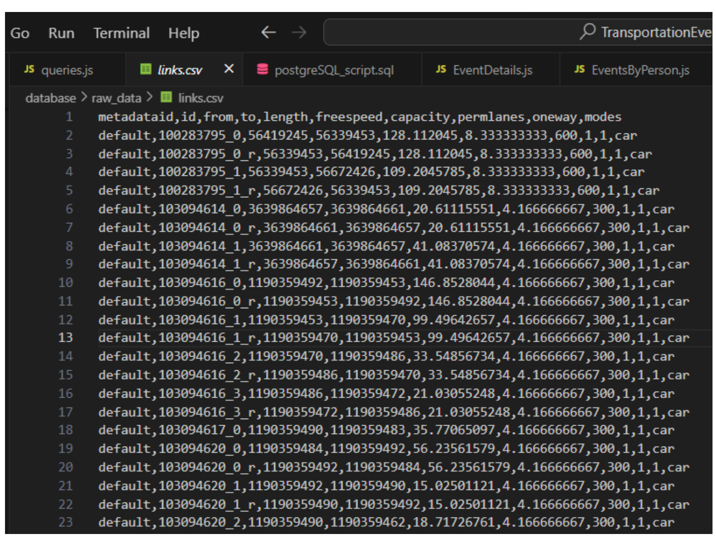

Figure 13 Link CSV data that was created in Excel

However, the event data was not so easy. Our database structure made it impossible to do an easy import, so Terrance designed a JavaScript program to import the data instead. The program would look at the event attributes and map them into the correct database table. This was done by taking the event type and using it to index an object containing the table names. We also used this technique to remap the attribute names to the proper column names (we had to change some of the column names to make them compatible with SQL).

```javascript
import pkg from 'pg';
import fs from 'fs';
import xml2js from 'xml2js';

import typeMap from './typeMap.js';
import commonColumnMap from './commonColumnMap.js';
import specialColumnMap from './specialColumnMap.js';

async function processXmlData() {
  console.log ("Starting the program...")

  // Connect to pgSQL
  const { Pool } = pkg;
  const pool = new Pool({
    user: 'postgres',
    host: 'localhost',
    database: 'transportProject',
    password: 'password',
    port: 5432,
  });

  pool.on('error', (err, client) => {
    console.error('Unexpected error on idle client', err);
    process.exit(-1);
  });

  const client = await pool.connect();

  console.log ("Connected to the database!")
  console.log ("Reading XML file...")
```

Figure 14 The code that connects to the database and reads the event data into memory

```javascript
for (const { $: eventData } of eventsArr) {
  let commonColumnNames = [], commonColumnValues = []
  let specialColumnNames = [], specialColumnValues = []
  let columnNums
  let eventType = eventData.type
  let childTableName = typeMap[eventData.type]
  let parentQueryResult, childQueryResult

  if (!childTableName)
    throw "Undefined event type:\n" + JSON.stringify (eventData, null, 2)

  console.log (line, eventType, childTableName)
  line += 1

  // Iterate over the attributes
  for (let [key, value] of Object.entries(eventData)) {
    // Map the key to the database alias
    if (Object.hasOwn(commonColumnMap, key)) {
      let mappedKey = commonColumnMap[key]
      commonColumnNames.push (mappedKey)
      commonColumnValues.push (value)
    }
    else if (Object.hasOwn(specialColumnMap[eventType], key)) {
      let mappedKey = specialColumnMap[eventType][key]
      specialColumnNames.push (mappedKey)
      specialColumnValues.push (value)
    }
    else {
      throw "Undefined column '" + key + "' for type '" + eventType + "':\n" +  + JSON.stringify (eventData, null, 2)
    }
  }
```

Figure 15 The code that maps the events and event attributes

```
try {
  // Insert into parent events_data table
  columnNums = commonColumnNames.map((_, index) => `$${index + 1}`).join (", ")
  parentQueryResult = await client.query(
    `INSERT INTO event_data (${commonColumnNames.join (", ")}) VALUES (${columnNums}) RETURNING event_id`,
    commonColumnValues,
  );
}
catch (err) {
  console.log ("Error importing data into parent table")
  console.log (err)
  process.exit (-1)
}

try {
  // Add foreign key to child data
  specialColumnNames.push ("event_id")
  specialColumnValues.push (parentQueryResult.rows[0].event_id)

  // Insert into child table
  columnNums = specialColumnNames.map((_, index) => `$${index + 1}`).join (", ")
  childQueryResult = await client.query(
    `INSERT INTO ${childTableName} (${specialColumnNames.join (", ")}) VALUES (${columnNums})`,
    specialColumnValues,
  );
}
```

Figure 16 The code that inserts each event into the common table and their respective child table

Extra queries

The query results from question 4 were completed by Chris. Their results are shown below.

Retrieve all events for a person with ID = " p_9031"

```
SELECT event_id, event_time, event_type FROM event_data
WHERE person = 'p_9031';
```

Data Output Messages Notifications

	event_id [PK] integer	event_time numeric	event_type text
1	1	20.0	actend
2	2	20.0	departure
3	3	20.0	PersonEntersVehicle
4	4	20.0	vehicle enters traffic
5	1716	1356.0	vehicle leaves traffic
6	1717	1356.0	PersonLeavesVehi...
7	1718	1356.0	arrival
8	1719	1356.0	actstart
9	3176369	70876.0	actend
10	3176370	70876.0	departure
11	3176371	70876.0	PersonEntersVehicle
12	3176411	70876.0	vehicle enters traffic
13	3227293	72158.0	vehicle leaves traffic
14	3227294	72158.0	PersonLeavesVehi...
15	3227295	72158.0	arrival
16	3227296	72158.0	actstart

Total rows: 16 of 16 Query complete 00:00:00.369

Figure 17 The query result for question 4A

For link ID = " 7735018_0", get link details with the node information for that link

```sql
SELECT l.*, t.x AS to_node_x, t.y AS to_node_y, f.x AS from_node_x, f.y AS from_node_y
FROM link l
JOIN node t ON (to_node = t.node_id)
JOIN node f ON (from_node = f.node_id)
WHERE link_id = '7735018_0';
```

Data Output Messages Notifications

	oneway boolean	modes text	link_length numeric	to_node_x numeric	to_node_y numeric	from_node_x numeric	from_node_y numeric
1	true	car	343.5966216	510244.6288842118	3713514.882330475	510426.68306076934	3713226.5933797597

Total rows: 1 of 1 Query complete 00:00:00.117

Figure 18 The query result for question 4B (NOTE: some of the columns on the left are cut off in the screenshot)

Calculate the total distance traveled by each person who walked. Display the results in descending order of distance.

```sql
SELECT person, SUM(distance) AS distance_walked
FROM event_data NATURAL JOIN travelled
WHERE mode = 'walk'
GROUP BY person
ORDER BY distance_walked DESC;
```

Data Output Messages Notifications

	person text	distance_walked numeric
1	p_8184	16394.887295374370
2	p_6202	16058.248299577164
3	p_6245	15780.549187682871
4	p_7411	15569.705841220799
5	p_7000	15557.774280892080
6	p_7632	15479.994141218812
7	p_7904	15417.813740717151
8	p_6159	15382.873319712829
9	p_8031	15328.3597754106975
10	p_8287	15236.954185475804
11	p_8132	15072.2844318279325
12	p_7113	15054.086292802284
13	p_7112	15042.692773030163
14	p_7360	14868.694159455872
15	p_7076	14728.285656950641
16	p_6873	14715.046567270541

Total rows: 3566 of 3566 Query complete 00:00:00.881

Figure 19 The query result for question 4C

Calculate the average time it takes for persons to complete their "actend" activities. Display the results in ascending order of average time.

```sql
SELECT
    end_events.person,
    AVG(start_events.event_time - end_events.event_time) AS average_time_to_next_actstart
FROM
    event_data AS end_events
JOIN
    event_data AS start_events ON end_events.person = start_events.person
WHERE
    end_events.event_type = 'actend'
    AND start_events.event_type = 'actstart'
    AND start_events.event_time > end_events.event_time
GROUP BY
    end_events.person
ORDER BY
    average_time_to_next_actstart ASC;
```

Data Output Messages Notifications

	person text	average_time_to_next_actstart numeric
1	p_2156	3.0000000000000000
2	p_2232	7.0000000000000000
3	p_1106	7.0000000000000000
4	p_262	8.0000000000000000
5	p_738	14.0000000000000000
6	p_1248	34.0000000000000000
7	p_808	68.0000000000000000
8	p_1191	69.0000000000000000

Total rows: 1000 of 9184 Query complete 00:00:01.426

Figure 20 The query result for question 4D

Retrieve the earliest "departure" time for each person who used the "car" mode.

```sql
SELECT person, MIN(event_time) AS earliest_time
FROM event_data NATURAL JOIN arrival_departure
WHERE legmode = 'car' AND event_type = 'departure'
GROUP BY person;
```

Data Output Messages Notifications

	person text	earliest_time numeric
1	p_2550	26445.0
2	p_2551	20491.0
3	p_2552	24645.0
4	p_2553	31399.0
5	p_2554	24372.0
6	p_2555	31876.0
7	p_2556	59165.0
8	p_2557	29113.0
9	p_2558	22536.0
10	p_2559	31584.0
11	p_2560	23717.0
12	p_2561	25011.0
13	p_2562	55350.0
14	p_2563	20223.0
15	p_2564	29048.0
16	p_2565	28271.0

Total rows: 1000 of 6590 Query complete 00:00:00.484

Figure 21 The query result for question 4E