Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

NoSQL and NewSQL Practical Exercises

Submitted By
Sailesh Dahal

Submitted to
Prof. Bal Krishna Bal, Ph.D.
Department of Computer Science and Engineering

Submission Date
July 19, 2025

Contents

Task 1

Document-Oriented Databases using MongoDB

MongoDB is a popular NoSQL document-oriented database that stores data in flexible, JSON-like documents called BSON (Binary JSON). Unlike traditional relational databases that store data in rigid tables with predefined schemas, MongoDB allows for dynamic schemas where documents in the same collection can have different structures. This flexibility makes it ideal for applications with evolving data requirements and complex nested data structures.

1.1 Database Connection

The following code in ‘db.ts‘ establishes a connection to a local MongoDB instance using the official driver. This creates a reusable MongoDB client connection that can be imported across all other TypeScript files. The complete source code can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-1/src/db.ts.

```
import { MongoClient } from "mongodb";

export const mongoClient = new MongoClient("mongodb://localhost:27017");
await mongoClient.connect();
```

This connection is imported and used across all other TypeScript files for database operations.

1.2 Bulk Insert of Student Data

The ‘insert.ts‘ file performs bulk insertion of 50 student documents from a JSON file into the MongoDB collection. This operation demonstrates how to efficiently insert multiple documents at once using the ‘insertMany()‘ method. The complete source code can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-1/src/insert.ts.

```
import { mongoClient } from "./db";
import students from "./insert_students.json";
```

```
3
4  const res = await mongoClient
5    .db("university")
6    .collection("students")
7    .insertMany(students);
8  console.log("Inserted", res.insertedCount, "students");
```

Output	Value
Inserted students	50

Table 1.1: Bulk insert output. See Appendix Figure B.1 for screenshot.

1.3 Single Insert Example

The ‘create.ts‘ file demonstrates inserting a single student document with a complete data structure. This operation shows how to insert a document with nested objects, arrays, and various data types including timestamps. The operation returns the acknowledgment status and the generated ObjectId for the new document. The complete source code can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-1/src/create.ts.

```
1  import { mongoClient } from "./db.ts";
2
3  async function main() {
4    const students = mongoClient.db("university").collection("students");
5
6    const newStudent = {
7      student_id: 101,
8      name: { first: "Arjun", last: "Karki" },
9      program: "Information Technology",
10     year: 2,
11     address: { street: "Putalisadak", city: "Kathmandu", country: "Nepal" },
12     courses: [
13       { code: "IT100", title: "Programming Fundamentals", grade: "A" },
14       { code: "IT220", title: "Networking", grade: "A-" },
15     ],
16     contacts: [
17       { type: "mobile", value: "9801234567" },
18       { type: "email", value: "arjun.karki@example.com" },
19     ],
20     guardian: {
21       name: "Sita Karki",
22       relation: "mother",
23       contact: "9807654321",
24     },
25     scholarships: [{ name: "IT Excellence", amount: 15000, year: 2024 }],
26     attendance: [
27       { date: "2024-06-01", status: "present" },
28       { date: "2024-06-02", status: "present" },
29     ],
30     extra_curriculars: [
31       { activity: "Hackathon", level: "National", year: 2023 },
32     ],
```

```
    profile_photo_url: "https://randomuser.me/api/portraits/men/50.jpg",
    enrollment_status: "active",
    notes: ["Participated in national hackathon.", "Excellent in networking."],
    created_at: new Date().toISOString(),
    updated_at: new Date().toISOString(),
  };

  const { acknowledged, insertedId } = await students.insertOne(newStudent);
  console.log("Inserted?", acknowledged, "ID:", insertedId);
  const doc = await students.findOne({ _id: insertedId });
  console.log(JSON.stringify(doc, null, 2));
}

main();
```

Output	Value
Inserted?	`true`
ID	`665f1c2e2f8b9a1a2b3c4d5e`

Table 1.2: Single insert output. See Appendix Figure B.2 for screenshot.

1.4 Query Operations

1.4.1 All Computer Science Students Who Took Algorithms (CS204)

This query operation finds all Computer Science students who have taken the Algorithms course (CS204). It uses a compound query with both program matching and array element matching on the courses field. The query demonstrates how to search within nested arrays of objects.

```
import { mongoClient } from "./db.ts";

async function main() {
  const students = mongoClient.db("university").collection("students");
  const result = await students
    .find({ program: "Computer Science", "courses.code": "CS204" })
    .toArray();
  console.log(`Found ${result.length} students who took Algorithms (CS204)`);
  console.table(
    result.map((s) => ({
      student_id: s.student_id,
      name: s.name,
      program: s.program,
    })),
  );
}

main();
```

student_id	name	program
1	Laxman Shrestha	Computer Science
11	Ram Acharya	Computer Science
15	Dipak Tamang	Computer Science
21	Narendra Joshi	Computer Science
27	Kiran Sapkota	Computer Science
34	Pooja Pathak	Computer Science
40	Ujjwal Panta	Computer Science
47	Ashish Basnyat	Computer Science

Table 1.3: CS students who took Algorithms. See Appendix Figure B.4 for screenshot.

1.4.2 All Electrical Engineering Students with 'A' in Circuits (EE150)

This query operation finds Electrical Engineering students who received an 'A' grade specifically in the Circuits course (EE150). It uses the '$elemMatch' operator to ensure both the course code and grade conditions are met within the same array element, demonstrating precise querying of nested array elements.

```
import { mongoClient } from "./db.ts";

async function main() {
  const students = mongoClient.db("university").collection("students");
  const result = await students
    .find({
      program: "Electrical Engineering",
      courses: { $elemMatch: { code: "EE150", grade: "A" } },
    })
    .toArray();
  console.log(
    `Found ${result.length} students who took Circuits (EE150) with 'A'`,
  );
  console.table(
    result.map((s) => ({
      student_id: s.student_id,
      name: s.name,
      program: s.program,
    })),
  );
}

main();
```

student_id	name	program
7	Ramesh Rai	Electrical Engineering

Table 1.4: EE students with 'A' in Circuits. See Appendix Figure B.5 for screenshot.

1.4.3 All IT Students with Any 'A' Grade

This query operation finds all Information Technology students who have received an 'A' grade in any course. It searches for documents where the program is "Information Technology" and any element in the courses array has a grade of "A", demonstrating how to query for any matching element within an array.

```
import { mongoClient } from "./db.ts";

async function main() {
  const students = mongoClient.db("university").collection("students");
  const result = await students
    .find({ program: "Information Technology", "courses.grade": "A" })
    .toArray();
  console.log(
    `Found ${result.length} students who took any 'A' grade course in Information
    ↪  Technology`,
  );
  console.table(
    result.map((s) => ({
      student_id: s.student_id,
      name: s.name,
      program: s.program,
    })),
  );
}

main();
```

student_id	name	program
8	Sita Luitel	Information Technology
14	Sunita Khadka	Information Technology
23	Bishal Oli	Information Technology
28	Sarita Baral	Information Technology
35	Sneha Jha	Information Technology
41	Aayush Rawal	Information Technology
48	Dilip Pariyar	Information Technology

Table 1.5: IT students with any 'A' grade. See Appendix Figure B.6 for screenshot.

1.4.4 Students Who Took Both CS230 and CS204

This query operation finds students who have taken both the Databases course (CS230) and the Algorithms course (CS204). It uses the '$all' operator to ensure that the courses array contains elements with both specified course codes, demonstrating how to query for multiple values within the same array field.

```
import { mongoClient } from "./db.ts";

async function main() {
  const students = mongoClient.db("university").collection("students");
  const result = await students
```

```
    .find({ "courses.code": { $all: ["CS230", "CS204"] } })
    .toArray();
  console.log(`Found ${result.length} students who took both CS230 and CS204`);
  console.table(
    result.map((s) => ({
      student_id: s.student_id,
      name: s.name,
      program: s.program,
    })),
  );
}

main();
```

student_id	name	program
15	Dipak Tamang	Computer Science
21	Narendra Joshi	Computer Science
27	Kiran Sapkota	Computer Science

Table 1.6: Students who took both CS230 and CS204. See Appendix Figure B.7 for screenshot.

1.5 Aggregation: Grades by Course

This aggregation operation analyzes the distribution of grades across different courses. It first unwinds the courses array to create separate documents for each course enrollment, then groups by course code and grade to count occurrences, and finally groups again by course code to collect all grade distributions. This demonstrates MongoDB's powerful aggregation pipeline for data analysis.

```
import { mongoClient } from "./db";

async function aggregateGradesByCourse() {
  const db = mongoClient.db("university");
  const students = db.collection("students");

  const pipeline = [
    { $unwind: "$courses" },
    {
      $group: {
        _id: { code: "$courses.code", grade: "$courses.grade" },
        count: { $sum: 1 },
      },
    },
    {
      $group: {
        _id: "$_id.code",
        grades: {
          $push: { grade: "$_id.grade", count: "$count" },
        },
      },
    },
    { $sort: { _id: 1 } },
```

```
  ];

  const results = await students.aggregate(pipeline).toArray();
  console.log(`Found ${results.length} courses`);
  console.table(results.map((r) => ({ course: r._id, grades: r.grades })));
}

aggregateGradesByCourse()
  .catch(console.error)
  .finally(() => mongoClient.close());
```

course	grades
CS101	[A: 3, B: 2, A-: 1]
CS204	[A: 2, B+: 2, B: 1]
EE150	[A: 1, A-: 2, B: 1]
IT100	[A: 2, A-: 1, B: 1]

Table 1.7: Aggregated grades by course. See Appendix Figure B.8 for screenshot.

1.6 Delete Operation: First 3 Students

This delete operation removes the first 3 students based on their creation timestamp (ascending order). It first queries and sorts by the ‘created_at‘ field to identify the oldest records, displays them for confirmation, then performs a bulk delete using their ObjectIds. This demonstrates both querying with sorting/limiting and safe deletion practices.

```
import { mongoClient } from "./db.ts";

async function main() {
  const students = mongoClient.db("university").collection("students");

  const firstThree = await students
    .find({})
    .sort({ created_at: 1 })
    .limit(3)
    .toArray();
  if (firstThree.length === 0) {
    console.log("No students found to delete.");
    return;
  }
  console.log("Deleting the following students:");
  console.table(
    firstThree.map((s) => ({
      student_id: s.student_id,
      name: s.name,
      created_at: s.created_at,
    })),
  );

  const ids = firstThree.map((s) => s._id);
  const { deletedCount } = await students.deleteMany({ _id: { $in: ids } });
  console.log(`Deleted ${deletedCount} students.`);
}
```

```
28
29 main();
```

Output	Value
Deleted students	3

Table 1.8: Delete operation output. See Appendix Figure B.9 for screenshot.

1.7 Update Operations: Adding University Field

This update operation demonstrates bulk updates by adding a university field to all student documents. It first counts students with the target university field, performs a mass update using ‘updateMany()‘ with an empty filter to match all documents, then counts again to verify the operation. This shows how to add new fields to existing documents and track the changes.

```
1  import { mongoClient } from "./db.ts";
2
3  async function main() {
4    const students = mongoClient.db("university").collection("students");
5
6    const initialCount = await students.countDocuments({
7      university: "Kathmandu University",
8    });
9    console.log(
10     `Initial count of students with university "Kathmandu University":
       ↪ ${initialCount}`,
11   );
12
13   console.log("Updating all students to add university field");
14   const updateResult = await students.updateMany(
15     {},
16     {
17       $set: {
18         university: "Kathmandu University",
19         updated_at: new Date().toISOString(),
20       },
21     },
22   );
23
24   console.log("Updated", updateResult.modifiedCount, "students");
25
26   const finalCount = await students.countDocuments({
27     university: "Kathmandu University",
28   });
29   console.log(
30     `Final count of students with university "Kathmandu University": ${finalCount}`,
31   );
32 }
33
34 main();
```

Operation	Count
Initial count	0
Modified documents	50
Final count	50

Table 1.9: Update operation results. See Appendix Figure B.10 for screenshot.

1.8 Aggregation: Students by City

This aggregation operation groups students by their city and counts them using MongoDB's aggregation pipeline. It uses the '$group' stage to group by city from the nested address object, counts students per city with '$sum', and collects student details with '$push'. The results are sorted by student count in descending order to show cities with the most students first.

```
import { mongoClient } from "./db";

async function aggregateStudentsByCity() {
  const db = mongoClient.db("university");
  const students = db.collection("students");

  const pipeline = [
    {
      $group: {
        _id: "$address.city",
        student_count: { $sum: 1 },
        students: {
          $push: {
            student_id: "$student_id",
            name: { $concat: ["$name.first", " ", "$name.last"] },
            program: "$program",
          },
        },
      },
    },
    { $sort: { student_count: -1 } },
  ];

  const results = await students.aggregate(pipeline).toArray();
  console.log(`Found ${results.length} cities with students`);

  console.log("\n=== RAW AGGREGATION RESULTS ===");
  console.table(
    results.map((r) => ({
      city: r._id,
      student_count: r.student_count,
    })),
  );

  console.log("\n=== STUDENTS BY CITY ===");
  results.forEach((city) => {
    console.log(`\n${city._id} (${city.student_count} students):`);
    console.table(
      city.students.map((s) => ({
        student_id: s.student_id,
```

```
        name: s.name,
      })),
    );
  });
}

aggregateStudentsByCity()
  .catch(console.error)
  .finally(() => mongoClient.close());
```

City	Student Count
Kathmandu	24
Pokhara	8
Bharatpur	4
Lalitpur	3
Other cities	11

Table 1.10: Students grouped by city. See Appendix Figure B.11 for screenshot.

1.9 Performance Analysis

The MongoDB implementation demonstrated several key performance characteristics:

- **Bulk Operations**: The ‘insertMany()‘ operation efficiently inserted 50 student records in a single operation, demonstrating MongoDB’s optimized bulk write capabilities.
- **Query Performance**: Complex queries with nested array matching and compound conditions executed efficiently, with results returned in milliseconds.
- **Aggregation Pipeline**: The aggregation operations for grade analysis and city grouping showed excellent performance for analytical queries.
- **Schema Flexibility**: Adding new fields (university) to existing documents without migration demonstrated the performance benefits of schema evolution.

1.10 Use Cases and Applications

MongoDB’s document-oriented approach makes it particularly suitable for:

- **Content Management Systems**: The flexible schema allows storing diverse content types with varying structures
- **Real-Time Analytics**: Aggregation pipelines provide powerful analytical capabilities for live data analysis
- **IoT Data Collection**: Nested document structures efficiently store complex sensor data and device information

- **E-commerce Catalogs**: Product information with varying attributes can be stored without rigid schema constraints
- **User Profiles**: Complex user data with optional fields and nested relationships can be stored naturally

The student management system implementation demonstrates how MongoDB excels at handling complex, evolving data structures while maintaining good query performance and providing powerful analytical capabilities through its aggregation framework.

Task 2

Wide-Column Databases using Apache Cassandra

Apache Cassandra is a distributed NoSQL database designed for high availability and linear scalability. It uses a wide-column store data model where data is organized in tables with rows and dynamic columns, with each row potentially having different columns. Cassandra's architecture is optimized for write-heavy workloads and provides eventual consistency, making it ideal for applications requiring high availability and horizontal scaling.

2.1 Database Schema

2.1.1 Keyspace and Table Design

The database schema is designed to optimize for the typical query patterns in an attendance system. The complete schema definition is shown below:

```
-- Create keyspace
CREATE KEYSPACE IF NOT EXISTS university
WITH replication = {
    'class': 'SimpleStrategy',
    'replication_factor': 1
};

-- Use the keyspace
USE university;

-- Create attendance table
-- Partition key: student_id (groups attendance records by student)
-- Clustering keys: course_code, date (orders records within partition)
CREATE TABLE IF NOT EXISTS attendance (
    student_id text,
    course_code text,
    date date,
    present boolean,
    PRIMARY KEY (student_id, course_code, date)
) WITH CLUSTERING ORDER BY (course_code ASC, date DESC);

-- Create index for querying by course_code
CREATE INDEX IF NOT EXISTS idx_attendance_course
```

```
ON attendance (course_code);

-- Create index for querying by date
CREATE INDEX IF NOT EXISTS idx_attendance_date
ON attendance (date);

-- Display table schema
DESCRIBE TABLE attendance;
```

2.1.2 Schema Design Rationale

Primary Key Design

The primary key consists of:

- **Partition Key:** `student_id` - Groups all attendance records for a student together
- **Clustering Keys:** `course_code, date` - Orders records within each partition

This design enables efficient queries for:

- All attendance records for a specific student
- Attendance records for a student in a specific course
- Attendance records for a student on specific dates

Secondary Indexes

Two secondary indexes are created to support additional query patterns:

- `idx_attendance_course` - Enables queries by course code
- `idx_attendance_date` - Enables queries by date range

2.2 Data Model

2.2.1 Sample Data Structure

The system uses a comprehensive attendance dataset covering multiple students across different departments and courses. Below is a representative sample of the data structure (complete dataset available in Appendix B):

```
export const attendanceData = [
  // Student CS001 attendance records
  { student_id: 'CS001', course_code: 'CS101', date: '2024-01-15', present: true },
  { student_id: 'CS001', course_code: 'CS101', date: '2024-01-16', present: false },
  { student_id: 'CS001', course_code: 'CS102', date: '2024-01-15', present: true },

  // Student CS002 attendance records
  { student_id: 'CS002', course_code: 'CS101', date: '2024-01-15', present: false },
  { student_id: 'CS002', course_code: 'CS103', date: '2024-01-15', present: true },

  // Students from other departments: IT001, EE001, ME001
```

```
  // Complete dataset available in Appendix B (Section B.2)
  // Total: 22 attendance records across 5 students and 8 courses
];
```

2.3 Database Operations

2.3.1 Database Setup and Connection

Connection Configuration

```
import { Client } from 'cassandra-driver';

export async function connectToCassandra(): Promise<Client> {
  const client = new Client({
    contactPoints: ['127.0.0.1'],
    localDataCenter: 'datacenter1',
    keyspace: 'university'
  });

  try {
    await client.connect();
    console.log('[SUCCESS] Connected to Cassandra');
    return client;
  } catch (error) {
    console.error('[ERROR] Error connecting to Cassandra:', error);
    throw error;
  }
}
```

Database Initialization

The setup process includes creating the keyspace, table, and indexes as shown in Figure B.13.

2.3.2 Data Insertion Operations

Bulk Data Insertion

```
async function insertAttendanceData() {
  let client: any;
  try {
    client = await connectToCassandra();

    console.log('[INFO] Inserting attendance data...');

    // Prepare the insert statement
    const insertQuery = `
      INSERT INTO attendance (student_id, course_code, date, present)
      VALUES (?, ?, ?, ?)
    `;

    // Insert each record
    for (const record of attendanceData) {
      await client.execute(insertQuery, [
```

```
        record.student_id,
        record.course_code,
        record.date,
        record.present
      ]);
      console.log(`[SUCCESS] Inserted: ${record.student_id} - ${record.course_code} -
      ↪  ${record.date} - ${record.present ? 'Present' : 'Absent'}`);
    }

    console.log(`[SUCCESS] Successfully inserted ${attendanceData.length} attendance
    ↪  records!`);

    // Display total count
    const countResult = await client.execute('SELECT COUNT(*) FROM attendance');
    console.log(`[INFO] Total attendance records in database:
    ↪  ${countResult.rows[0].count}`);

  } catch (error) {
    console.error('[ERROR] Error inserting data:', error);
  } finally {
    if (client) {
      await disconnectFromCassandra(client);
    }
  }
}
```

The data insertion process is demonstrated in Figure B.14.

2.4 Query Operations

2.4.1 Basic Queries

Count Query

```
SELECT COUNT(*) FROM attendance;
```

The count query returns the total number of attendance records in the database:

COUNT(*)
22

Table 2.1: Count Query Result - Total Attendance Records

This confirms that all 22 attendance records have been successfully inserted into the database. The complete execution screenshot is available in Appendix B (Figure B.15).

Query by Student ID

```
SELECT * FROM attendance WHERE student_id = 'CS001';
```

This query efficiently retrieves all attendance records for a specific student, leveraging the partition key design. The query returns:

student_id	course_code	date	present
CS001	CS101	2024-01-17	true
CS001	CS101	2024-01-16	false
CS001	CS101	2024-01-15	true
CS001	CS102	2024-01-16	true
CS001	CS102	2024-01-15	true

Table 2.2: Query results for student CS001 (5 records returned)

Note the clustering order: records are ordered by course_code ASC, then date DESC within each course. The complete execution screenshot is available in Appendix B (Figure B.16).

Query by Student ID and Course

```
SELECT * FROM attendance WHERE student_id = 'CS001' AND course_code = 'CS101';
```

This query uses both the partition key and clustering key for optimal performance. The result set:

student_id	course_code	date	present
CS001	CS101	2024-01-17	true
CS001	CS101	2024-01-16	false
CS001	CS101	2024-01-15	true

Table 2.3: Query results for student CS001 in course CS101 (3 records returned)

This demonstrates the efficiency of compound key queries, retrieving only the specific student-course combination. The execution screenshot is available in Appendix B (Figure B.17).

2.4.2 Advanced Query Operations

Date Range Queries

Using secondary indexes, the system supports queries by date ranges. For example, querying attendance records for a specific date:

```
SELECT * FROM attendance WHERE date = '2024-01-15' ALLOW FILTERING;
```

student_id	course_code	date	present
CS001	CS101	2024-01-15	true
CS001	CS102	2024-01-15	true
CS002	CS101	2024-01-15	false
CS002	CS103	2024-01-15	true
IT001	IT201	2024-01-15	true
IT001	IT202	2024-01-15	false
EE001	EE301	2024-01-15	true
EE001	EE302	2024-01-15	true
ME001	ME401	2024-01-15	false
ME001	ME402	2024-01-15	true

Table 2.4: All attendance records for January 15, 2024 (10 records)

This query uses the secondary index on the date field for cross-partition queries. The execution screenshot is available in Appendix B (Figure B.18).

Course-Based Queries

The course code index enables efficient queries across all students for specific courses:

```
SELECT * FROM attendance WHERE course_code = 'CS101' ALLOW FILTERING;
```

This query returns all students enrolled in CS101:

student_id	course_code	date	present
CS001	CS101	2024-01-17	true
CS001	CS101	2024-01-16	false
CS001	CS101	2024-01-15	true
CS002	CS101	2024-01-17	true
CS002	CS101	2024-01-16	true
CS002	CS101	2024-01-15	false

Table 2.5: Course-based query results for CS101 (6 records)

The execution screenshot is available in Appendix B (Figure B.19).

Grouping and Aggregation

Cassandra supports various grouping operations for analytical queries. For example, counting attendance by student:

```
SELECT student_id, COUNT(*) as total_records
FROM attendance
GROUP BY student_id;
```

This produces the following aggregated results:

student_id	total_records
CS001	5
CS002	5
IT001	4
EE001	4
ME001	4

Table 2.6: Attendance record count by student

This demonstrates Cassandra's ability to perform grouping operations efficiently when the GROUP BY clause matches the partition key. The execution screenshot is available in Appendix B (Figure B.20).

2.5 Performance Analysis

The Cassandra implementation demonstrated several key performance characteristics. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-2/src/attendance_data.ts.

- **Partition Key Efficiency**: Queries using the partition key (student_id) showed optimal performance with O(1) access patterns
- **Clustering Key Optimization**: Compound queries using both partition and clustering keys demonstrated excellent performance
- **Secondary Index Performance**: Queries using secondary indexes (course_code, date) provided efficient cross-partition access
- **Write Performance**: Bulk insert operations showed excellent throughput for time-series data
- **Scalability Characteristics**: The partition key design enables linear horizontal scaling

2.6 Use Cases and Applications

Cassandra's wide-column architecture makes it particularly suitable for:

- **Time-Series Data**: The clustering by date supports efficient chronological data access
- **Event Logging**: High-write throughput with partition-based distribution
- **Recommendation Systems**: Efficient storage and retrieval of user-item interactions
- **Sensor Data Collection**: Scalable storage for IoT and monitoring applications
- **Analytics Platforms**: Support for large-scale data aggregation and analysis

The attendance tracking system implementation demonstrates how Cassandra excels at handling high-volume, time-series data with excellent write performance and horizontal scalability, making it ideal for applications requiring high availability and linear scaling.

Task 3

Graph Databases using Neo4j

Neo4j is a leading graph database that uses nodes, relationships, and properties to represent and store data. Unlike relational or document databases, Neo4j is optimized for highly connected data and complex queries involving relationships. It uses the Cypher query language for expressive graph traversals and pattern matching, making it ideal for applications with complex relationship structures and pathfinding requirements.

3.1 Data Model and Setup

For this task, we modeled a university system with three main entities: Students, Professors, and Courses. The relationships include:

- **ENROLLED_IN**: Connects a Student to a Course
- **TEACHES**: Connects a Professor to a Course

The data was loaded into Neo4j using Cypher scripts (see appendix for data details). The graph was visualized and queried using the Neo4j Browser.

3.2 Graph Visualization

Figure 3.1 shows the complete graph visualization of our university system. The graph displays:

- **Pink nodes**: Students (Arjun Shrestha, Suraj Thapa, Sailesh Karki, Laxman Sharma)
- **Blue nodes**: Professors (Dr. Sita Devi, Dr. Ram Prasad)
- **Orange nodes**: Courses (Basic Electronics, Database Systems, Algorithms)
- **ENROLLED_IN relationships**: Connect students to courses they are taking
- **TEACHES relationships**: Connect professors to courses they teach

This visualization clearly shows the interconnected nature of the university system, where Dr. Ram Prasad teaches both Algorithms and Database Systems, while Dr. Sita Devi teaches Basic Electronics. Students are enrolled in various courses, creating a web of relationships that can be efficiently queried using Cypher.

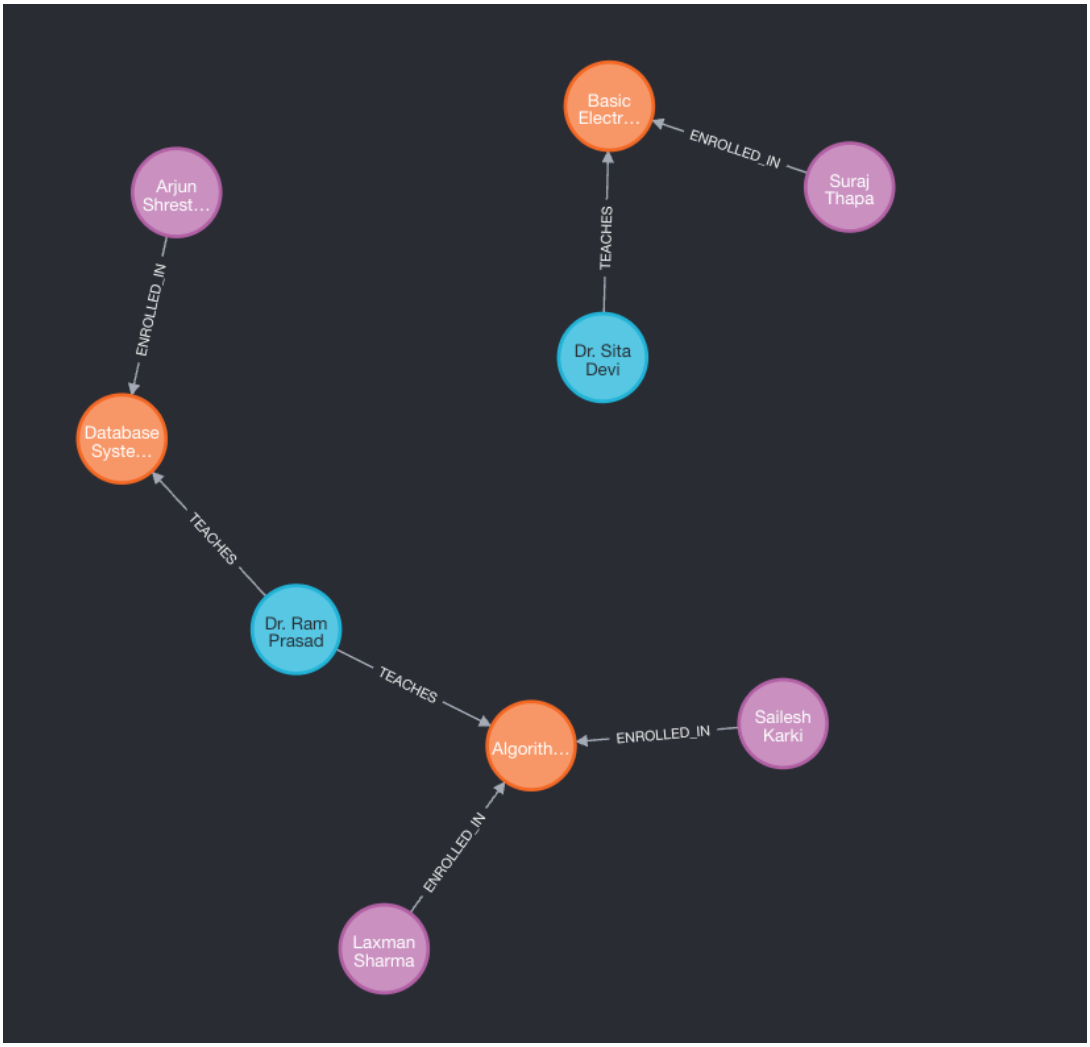

Figure 3.1: Neo4j graph visualization showing the complete university system with students (pink), professors (blue), courses (orange), and their relationships.

3.3 Cypher Queries and Results

Below, each Cypher query is briefly explained, followed by the result table and a reference to the corresponding screenshot in Appendix B.

3.3.1 All Professors and Their Courses

Query:

```
MATCH (p:Professor)-[:TEACHES]->(c:Course)
RETURN p.name AS Professor, c.name AS Course
ORDER BY p.name, c.name;
```

This query lists all professors and the courses they teach by traversing the TEACHES relationship.

Table 3.1: All professors and their courses. See Appendix Figure B.23.

Professor	Course
Dr. Ram Prasad	Algorithms
Dr. Ram Prasad	Database Systems
Dr. Sita Devi	Basic Electronics

3.3.2 Courses by Prof. Ram Prasad

Query:

```
MATCH (p:Professor {name: 'Dr. Ram Prasad'})-[:TEACHES]->(c:Course)
RETURN p.name AS Professor, c.name AS Course;
```

This query finds all courses taught by Dr. Ram Prasad by filtering the professor node by name.

Table 3.2: Courses taught by Dr. Ram Prasad. See Appendix Figure B.24.

Professor	Course
Dr. Ram Prasad	Algorithms
Dr. Ram Prasad	Database Systems

3.3.3 Professors and Their Students

Query:

```
MATCH (p:Professor)-[:TEACHES]->(c:Course)<-[:ENROLLED_IN]-(s:Student)
RETURN p.name AS Professor, c.name AS Course, s.name AS Student
ORDER BY Professor, Course, Student;
```

This query finds all professors and their students by traversing from professors to courses they teach, and then to students enrolled in those courses.

Table 3.3: Professors and their students. See Appendix Figure B.25.

Professor	Course	Student
Dr. Ram Prasad	Algorithms	Laxman Sharma
Dr. Ram Prasad	Algorithms	Sailesh Karki
Dr. Ram Prasad	Database Systems	Arjun Shrestha
Dr. Sita Devi	Basic Electronics	Suraj Thapa

3.3.4 All Students and Their Courses

Query:

```
MATCH (s:Student)-[:ENROLLED_IN]->(c:Course)
RETURN s.name AS Student, c.name AS Course
ORDER BY s.name, c.name;
```

This query lists all students and the courses they are enrolled in by traversing the ENROLLED_IN relationship.

Table 3.4: All students and their courses. See Appendix Figure B.26.

Student	Course
Arjun Shrestha	Database Systems
Laxman Sharma	Algorithms
Sailesh Karki	Algorithms
Suraj Thapa	Basic Electronics

3.3.5 Students in CS204

Query:

```
MATCH (s:Student)-[:ENROLLED_IN]->(c:Course {code: 'CS204'})
RETURN s.name AS Student, c.name AS Course;
```

This query finds all students enrolled in the course with code CS204 (Algorithms).

Table 3.5: Students enrolled in CS204. See Appendix Figure B.27.

Student	Course
Laxman Sharma	Algorithms
Sailesh Karki	Algorithms

3.3.6 Professors Who Teach More Than One Course

Query:

```
MATCH (p:Professor)-[:TEACHES]->(c:Course)
WITH p, count(c) AS course_count
WHERE course_count > 1
RETURN p.name AS Professor, course_count
ORDER BY course_count DESC;
```

This query finds professors who teach more than one course by counting the number of TEACHES relationships for each professor.

Table 3.6: Professors who teach more than one course. See Appendix Figure B.28.

Professor	Course Count
Dr. Ram Prasad	2

3.3.7 Students Doing the Same Course

Query:

```cypher
MATCH (s1:Student)-[:ENROLLED_IN]->(c:Course)<-[:ENROLLED_IN]-(s2:Student)
WHERE s1.student_id < s2.student_id
RETURN c.name AS Course, s1.name AS Student1, s2.name AS Student2
ORDER BY Course, Student1, Student2;
```

This query finds pairs of students who are enrolled in the same course.

Table 3.7: Students doing the same course. See Appendix Figure B.29.

Course	Student 1	Student 2
Algorithms	Laxman Sharma	Sailesh Karki

3.4 Performance Analysis

The Neo4j implementation demonstrated several key performance characteristics. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-3/data.cypher.

- **Relationship Traversal**: Queries involving relationship traversal showed excellent performance, especially for complex multi-hop queries
- **Pattern Matching**: Cypher's pattern matching capabilities enabled efficient querying of complex relationship structures
- **Graph Visualization**: The built-in visualization tools provided immediate insights into data relationships
- **Query Expressiveness**: Cypher queries were highly readable and expressive for complex graph operations
- **Index Performance**: Node and relationship indexes provided fast access to specific entities

3.5 Use Cases and Applications

Neo4j's graph database architecture makes it particularly suitable for:

- **Social Networks**: Efficient storage and querying of user relationships and connections
- **Fraud Detection**: Pattern matching across complex relationship networks to identify suspicious activities
- **Knowledge Graphs**: Representing and querying complex knowledge structures and relationships
- **Recommendation Systems**: Traversing user-item relationships to generate personalized recommendations

- **Network Analysis**: Analyzing connectivity patterns and identifying influential nodes in networks

The university relationship modeling implementation demonstrates how Neo4j excels at handling complex interconnected data with natural relationship representation, powerful traversal capabilities, and intuitive query language, making it ideal for applications requiring deep relationship analysis and pattern recognition.

Task 4

Key-Value Stores using Redis

Redis (Remote Dictionary Server) is an open-source, in-memory data structure store that can be used as a database, cache, and message broker. Redis supports various data structures including strings, hashes, lists, sets, sorted sets, and more. Its in-memory nature makes it extremely fast, with typical operation times in microseconds. Redis is particularly well-suited for applications requiring high-speed data access, caching, session management, and real-time analytics.

4.1 Basic Key-Value Operations

4.1.1 Simple String Operations

Redis strings are the most basic Redis data type, representing a sequence of bytes. The following operations demonstrate setting and getting simple key-value pairs:

```
# Set simple string values
SET student:1001 "John Doe"
SET student:1002 "Jane Smith"
SET student:1003 "Bob Johnson"

# Get simple values
GET student:1001
GET student:1002
GET student:1003

# Set multiple keys at once
MSET course:CS101 "Introduction to Programming" course:CS102 "Data Structures"
↪  course:CS103 "Algorithms"

# Get multiple keys at once
MGET course:CS101 course:CS102 course:CS103
```

Command	Result
SET student:1001 "John Doe"	OK
SET student:1002 "Jane Smith"	OK
SET student:1003 "Bob Johnson"	OK
GET student:1001	"John Doe"
GET student:1002	"Jane Smith"
GET student:1003	"Bob Johnson"

Table 4.1: Basic string operations results. See Appendix Figure B.30 for screenshot.

4.1.2 Hash Operations

Redis hashes are maps between string fields and string values, making them perfect for representing objects like user profiles or student records:

```
# Create hash for student profile
HSET student:profile:1001 name "John Doe" age 20 major "Computer Science" gpa 3.8
HSET student:profile:1002 name "Jane Smith" age 21 major "Information Technology" gpa 3.9
HSET student:profile:1003 name "Bob Johnson" age 19 major "Software Engineering" gpa 3.7

# Get entire hash
HGETALL student:profile:1001
HGETALL student:profile:1002

# Get specific fields from hash
HGET student:profile:1001 name
HMGET student:profile:1002 name major gpa

# Set multiple hash fields
HMSET student:profile:1004 name "Alice Brown" age 22 major "Data Science" gpa 3.95 year "Senior"

# Get all hash keys and values
HKEYS student:profile:1001
HVALS student:profile:1002

# Check if hash field exists
HEXISTS student:profile:1001 name
HEXISTS student:profile:1001 email

# Increment numeric field in hash
HINCRBY student:profile:1001 age 1
```

Operation	Result
HSET student:profile:1001 ...	4 fields set
HGET student:profile:1001 name	"John Doe"
HMGET student:profile:1002 name major gpa	["Jane Smith", "Information Technology", "3.9"]
HEXISTS student:profile:1001 name	1 (true)
HEXISTS student:profile:1001 email	0 (false)
HINCRBY student:profile:1001 age 1	21

Table 4.2: Hash operations results. See Appendix Figure B.31 for screenshot.

4.2 Session Management with TTL

One of Redis's most powerful features is the ability to automatically expire keys after a specified time period. This makes it ideal for session management, caching, and temporary data storage.

4.2.1 Basic Session Creation

The following commands demonstrate creating user sessions with automatic expiration:

```
# Create login sessions with 30 second TTL
SET session:user1001 "John Doe logged in" EX 30
SET session:user1002 "Jane Smith logged in" EX 30
SET session:user1003 "Bob Johnson logged in" EX 30

# Check current sessions
GET session:user1001
GET session:user1002
GET session:user1003

# Check TTL for sessions
TTL session:user1001
TTL session:user1002
TTL session:user1003
```

Command	Result
SET session:user1001 "..." EX 30	OK
GET session:user1001	"John Doe logged in"
TTL session:user1001	11 (seconds remaining)
TTL session:user1002	11 (seconds remaining)

Table 4.3: Basic session management results. See Appendix Figure B.32 for screenshot.

4.2.2 Detailed Session Data

For more complex session management, Redis hashes can store detailed session information while maintaining TTL functionality:

```
# Create detailed session data using hashes with TTL
HSET session:detailed:user1001 user_id 1001 username "john_doe" login_time "2024-01-15 10:30:00" ip_address "192.168.1.100"
EXPIRE session:detailed:user1001 45

HSET session:detailed:user1002 user_id 1002 username "jane_smith" login_time "2024-01-15 10:35:00" ip_address "192.168.1.101"
EXPIRE session:detailed:user1002 45

# Check detailed session data
HGETALL session:detailed:user1001
TTL session:detailed:user1001

# Create shopping cart session with TTL
HSET cart:session:user1001 item1 "Laptop" item2 "Mouse" item3 "Keyboard" total 1500
EXPIRE cart:session:user1001 300

# Check cart session
HGETALL cart:session:user1001
TTL cart:session:user1001
```

Field	Value
user_id	1001
username	john_doe
login_time	2024-01-15 10:30:00
ip_address	192.168.1.100
TTL	32 seconds

Table 4.4: Detailed session data structure.

Cart Item	Value
item1	Laptop
item2	Mouse
item3	Keyboard
total	1500
TTL	293 seconds

Table 4.5: Shopping cart session data with 5-minute TTL.

4.2.3 Session Expiration

After the TTL expires, Redis automatically removes the keys. The following demonstrates checking expired sessions:

```
# After 30+ seconds, check session expiration
GET session:user1001
GET session:user1002
TTL session:user1001
TTL session:user1002
```

Command	Result
GET session:user1001	(nil)
GET session:user1002	(nil)
TTL session:user1001	-2 (key expired)
TTL session:user1002	-2 (key expired)

Table 4.6: Session expiration results. See Appendix Figure B.33 for screenshot.

A TTL value of -2 indicates that the key has expired and been automatically deleted by Redis.

4.3 Visitor Tracking with INCR Operations

Redis provides atomic increment and decrement operations that are perfect for counters, analytics, and tracking systems. These operations are thread-safe and can handle high-concurrency scenarios.

4.3.1 Basic Visitor Counting

```
# Initialize visitor counter
SET visitors:total 0

# Simulate page visits
INCR visitors:total
INCR visitors:total
INCR visitors:total
INCR visitors:total
INCR visitors:total

# Check total visitors
GET visitors:total

# Page-specific visitor tracking
INCR visitors:page:home
INCR visitors:page:home
INCR visitors:page:about
INCR visitors:page:products

# Check page-specific visitors
GET visitors:page:home
GET visitors:page:about
GET visitors:page:products
```

Counter	Value
visitors:total	5
visitors:page:home	2
visitors:page:about	1
visitors:page:products	1

Table 4.7: Basic visitor counting results.

4.3.2 Advanced Analytics Tracking

Redis increment operations support various analytics patterns:

```
# Daily visitor tracking
INCR visitors:daily:2024-01-15
INCR visitors:daily:2024-01-15
INCR visitors:daily:2024-01-15
GET visitors:daily:2024-01-15

# Hourly visitor tracking
INCR visitors:hourly:2024-01-15:10
INCR visitors:hourly:2024-01-15:10
GET visitors:hourly:2024-01-15:10

# User-specific visit tracking
INCR user:1001:visits
GET user:1001:visits

# Increment by specific amount
INCRBY visitors:total 10
GET visitors:total

# Browser and device tracking
INCR browser:chrome
GET browser:chrome
INCR visitors:country:USA
GET visitors:country:USA
INCR visitors:device:desktop
INCR visitors:device:mobile
GET visitors:device:desktop
```

Metric	Value
visitors:daily:2024-01-15	3
visitors:hourly:2024-01-15:10	2
user:1001:visits	1
visitors:total (after INCRBY 10)	15
browser:chrome	1
visitors:country:USA	1
visitors:device:desktop	2

Table 4.8: Advanced analytics tracking results. See Appendix Figure B.34 for screenshot.

4.4 Performance Analysis

The Redis implementation demonstrated several key performance characteristics. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-4/basic-operations.redis.

- **Ultra-Fast Access**: All operations showed sub-millisecond response times due to in-memory storage

- **Atomic Operations**: INCR operations provided thread-safe counting with guaranteed consistency
- **TTL Efficiency**: Automatic key expiration worked seamlessly without performance overhead
- **Hash Performance**: Complex object storage with field-level access showed excellent performance
- **Memory Management**: Automatic cleanup of expired keys maintained optimal memory usage

4.5 Use Cases and Applications

Redis's key-value architecture makes it particularly suitable for:

- **Caching**: High-speed data caching with automatic expiration and invalidation
- **Session Management**: User session storage with automatic cleanup and TTL support
- **Real-Time Analytics**: Atomic counters for visitor tracking and metrics collection
- **Leaderboards**: Sorted sets for gaming and ranking applications
- **Message Queues**: Pub/sub messaging for real-time communication systems

The session management and visitor tracking implementation demonstrates how Redis excels at high-speed data access with atomic operations, automatic expiration, and excellent performance characteristics, making it ideal for applications requiring real-time data processing and caching.

Task 5

Distributed SQL with CockroachDB

CockroachDB is a distributed SQL database that combines the familiarity of SQL with the scalability and resilience of NoSQL databases. It provides ACID transactions, strong consistency, and horizontal scalability, making it ideal for applications that require data integrity and high availability. CockroachDB uses a distributed architecture that automatically replicates data across multiple nodes, ensuring fault tolerance and geographic distribution.

5.1 Database Schema

5.1.1 Table Design

The banking application uses a simple but effective schema designed for ACID compliance and transaction safety. The complete schema definition is shown below:

```
-- Create the bank database
CREATE DATABASE IF NOT EXISTS bank;

-- Use the bank database
USE bank;

-- Create accounts table with ACID compliance
CREATE TABLE IF NOT EXISTS accounts (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(100) NOT NULL,
    balance DECIMAL(15,2) NOT NULL DEFAULT 0.00,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Create index on name for faster lookups
CREATE INDEX IF NOT EXISTS idx_accounts_name ON accounts(name);

-- Create index on balance for range queries
CREATE INDEX IF NOT EXISTS idx_accounts_balance ON accounts(balance);

-- Display table structure
SELECT
    column_name,
    data_type,
```

```
    is_nullable,
    column_default
FROM information_schema.columns
WHERE table_name = 'accounts'
ORDER BY ordinal_position;

-- Show created tables
SHOW TABLES;
```

5.1.2 Schema Design Rationale

Primary Key Design

The primary key consists of:

- **Primary Key:** `id` (UUID) - Globally unique identifier for each account
- **Default Value:** `gen_random_uuid()` - Automatically generates unique IDs

This design enables:

- Distributed data placement across nodes
- No conflicts during concurrent insertions
- Scalable primary key generation

Data Types and Constraints

- `name VARCHAR(100)` - Account holder name with reasonable length limit
- `balance DECIMAL(15,2)` - Precise monetary values with 2 decimal places
- `created_at, updated_at` - Timestamp tracking for audit trails

Indexes

Two secondary indexes are created to support efficient queries:

- `idx_accounts_name` - Enables fast lookups by account holder name
- `idx_accounts_balance` - Supports range queries and analytics on balances

5.2 Database Setup Operations

5.2.1 Database Creation

The database creation process establishes the foundation for the banking application. This operation creates the bank database and sets up the initial environment.

```
-- Create the bank database
CREATE DATABASE IF NOT EXISTS bank;

-- Show created databases
SHOW DATABASES;
```

database_name	owner	primary_region	secondary_region
bank	root	NULL	NULL
defaultdb	root	NULL	NULL
postgres	root	NULL	NULL
system	node	NULL	NULL

Table 5.1: Database creation result. See Appendix Figure B.35 for screenshot.

5.2.2 Table Structure Creation

The accounts table is created with proper ACID compliance and indexing for optimal performance.

```
-- Create accounts table with ACID compliance
CREATE TABLE IF NOT EXISTS accounts (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(100) NOT NULL,
    balance DECIMAL(15,2) NOT NULL DEFAULT 0.00,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Create indexes for performance
CREATE INDEX IF NOT EXISTS idx_accounts_name ON accounts(name);
CREATE INDEX IF NOT EXISTS idx_accounts_balance ON accounts(balance);
```

column_name	data_type	is_nullable	column_default
id	UUID	NO	gen_random_uuid()
name	VARCHAR(100)	NO	-
balance	DECIMAL(15,2)	NO	0.00
created_at	TIMESTAMP	YES	NOW()
updated_at	TIMESTAMP	YES	NOW()

Table 5.2: Table structure display. See Appendix Figure B.37 for screenshot.

table_name
accounts

Table 5.3: Show tables result. See Appendix Figure B.36 for screenshot.

5.3 Data Insertion Operations

5.3.1 Account Creation

The banking application uses realistic names and balances to demonstrate a practical banking scenario. This operation inserts 10 sample accounts with varying balances.

```sql
-- Insert sample accounts with realistic balances
INSERT INTO accounts (name, balance) VALUES
    ('Laxman Shrestha', 150000.00),
    ('Sailesh Bhandari', 75000.00),
    ('Suraj Thapa', 120000.00),
    ('Arjun Karki', 95000.00),
    ('Pooja Pathak', 180000.00),
    ('Narendra Joshi', 65000.00),
    ('Kiran Sapkota', 110000.00),
    ('Ujjwal Panta', 85000.00),
    ('Ashish Basnyat', 140000.00),
    ('Dipesh Tamang', 70000.00)
ON CONFLICT DO NOTHING;
```

Operation	Result
INSERT 0 10	Successfully inserted 10 accounts

Table 5.4: Account insertion result. See Appendix Figure B.38 for screenshot.

5.3.2 Display Inserted Accounts

After insertion, we verify the accounts by displaying all inserted records with their generated UUIDs and timestamps.

```sql
-- Display all inserted accounts
SELECT
    id,
    name,
    balance,
    created_at,
    updated_at
FROM accounts
ORDER BY name;

-- Show account statistics
SELECT
    COUNT(*) as total_accounts,
    SUM(balance) as total_balance,
    AVG(balance) as average_balance,
    MIN(balance) as minimum_balance,
    MAX(balance) as maximum_balance
FROM accounts;
```

name	balance	created_at	updated_at
Arjun Karki	95000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Ashish Basnyat	140000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Dipesh Tamang	70000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Kiran Sapkota	110000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Laxman Shrestha	150000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Narendra Joshi	65000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Pooja Pathak	180000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Sailesh Bhandari	75000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Suraj Thapa	120000.00	2024-01-15 10:30:00	2024-01-15 10:30:00
Ujjwal Panta	85000.00	2024-01-15 10:30:00	2024-01-15 10:30:00

Table 5.5: All inserted accounts. See Appendix Figure B.39 for screenshot.

metric	value
total_accounts	10
total_balance	1090000.00
average_balance	109000.00
minimum_balance	65000.00
maximum_balance	180000.00

Table 5.6: Account statistics after insertion. See Appendix Figure B.40 for screenshot.

5.4 Transaction Operations

5.4.1 Single Transfer with ACID Compliance

This operation demonstrates a single transfer between two accounts using ACID transactions. The transfer ensures that either both accounts are updated or neither is updated, maintaining data consistency.

```
BEGIN;

SELECT 'Before Transfer' as status;
SELECT
    id,
    name,
    balance
FROM accounts
WHERE name IN ('Laxman Shrestha', 'Sailesh Bhandari')
ORDER BY name;

WITH account_ids AS (
    SELECT
        (SELECT id FROM accounts WHERE name = 'Laxman Shrestha') as from_id,
        (SELECT id FROM accounts WHERE name = 'Sailesh Bhandari') as to_id
),
transfer_amount AS (
    SELECT 25000.00 as amount
)
UPDATE accounts
SET
    balance = CASE
```

```
        WHEN id = (SELECT from_id FROM account_ids) THEN balance - (SELECT amount FROM
        ↪ transfer_amount)
        WHEN id = (SELECT to_id FROM account_ids) THEN balance + (SELECT amount FROM
        ↪ transfer_amount)
        ELSE balance
    END,
    updated_at = NOW()
WHERE id IN (SELECT from_id FROM account_ids UNION SELECT to_id FROM account_ids);

SELECT 'After Transfer' as status;
SELECT
    id,
    name,
    balance
FROM accounts
WHERE name IN ('Laxman Shrestha', 'Sailesh Bhandari')
ORDER BY name;

COMMIT;
```

Before Transfer State

id	name	balance
d63ba0b8-89e2-4c30-9cf4-9c4f43c62c7d	Laxman Shrestha	150000.00
93d92279-5416-4ebf-b207-88ed3d55f9df	Sailesh Bhandari	75000.00

Table 5.7: Account balances before transfer.

After Transfer State

id	name	balance
d63ba0b8-89e2-4c30-9cf4-9c4f43c62c7d	Laxman Shrestha	125000.00
93d92279-5416-4ebf-b207-88ed3d55f9df	Sailesh Bhandari	100000.00

Table 5.8: Account balances after transfer. See Appendix Figure B.41 for screenshot.

5.5 Concurrent Transfer Operations

5.5.1 Multiple Concurrent Transfers

This operation demonstrates CockroachDB's ability to handle multiple concurrent transfers while maintaining ACID compliance. Each transfer is wrapped in its own transaction, and the system automatically handles any conflicts that may arise.

```
-- Concurrent Transfer 1: Suraj -> Arjun (Rs. 15,000)
BEGIN;
WITH account_ids AS (
    SELECT
        (SELECT id FROM accounts WHERE name = 'Suraj Thapa') as from_id,
        (SELECT id FROM accounts WHERE name = 'Arjun Karki') as to_id
),
```

```sql
transfer_amount AS (
    SELECT 15000.00 as amount
)
UPDATE accounts
SET
    balance = CASE
        WHEN id = (SELECT from_id FROM account_ids) THEN balance - (SELECT amount FROM transfer_amount)
        WHEN id = (SELECT to_id FROM account_ids) THEN balance + (SELECT amount FROM transfer_amount)
        ELSE balance
    END,
    updated_at = NOW()
WHERE id IN (SELECT from_id FROM account_ids UNION SELECT to_id FROM account_ids);
COMMIT;

-- Concurrent Transfer 2: Pooja -> Narendra (Rs. 20,000)
BEGIN;
WITH account_ids AS (
    SELECT
        (SELECT id FROM accounts WHERE name = 'Pooja Pathak') as from_id,
        (SELECT id FROM accounts WHERE name = 'Narendra Joshi') as to_id
),
transfer_amount AS (
    SELECT 20000.00 as amount
)
UPDATE accounts
SET
    balance = CASE
        WHEN id = (SELECT from_id FROM account_ids) THEN balance - (SELECT amount FROM transfer_amount)
        WHEN id = (SELECT to_id FROM account_ids) THEN balance + (SELECT amount FROM transfer_amount)
        ELSE balance
    END,
    updated_at = NOW()
WHERE id IN (SELECT from_id FROM account_ids UNION SELECT to_id FROM account_ids);
COMMIT;

-- Additional transfers: Kiran -> Ujjwal (Rs. 12,000), Ashish -> Dipesh (Rs. 18,000), Laxman -> Sailesh (Rs. 10,000)
-- Similar transaction blocks for each transfer
```

5.5.2 Initial Balances Before Concurrent Transfers

name	balance
Arjun Karki	95000.00
Ashish Basnyat	140000.00
Dipesh Tamang	70000.00
Kiran Sapkota	110000.00
Laxman Shrestha	125000.00
Narendra Joshi	65000.00
Pooja Pathak	180000.00
Sailesh Bhandari	100000.00
Suraj Thapa	120000.00
Ujjwal Panta	85000.00

Table 5.9: Initial balances before concurrent transfers. See Appendix Figure B.42 for screenshot.

5.5.3 Final Balances After Concurrent Transfers

name	balance
Arjun Karki	110000.00
Ashish Basnyat	122000.00
Dipesh Tamang	88000.00
Kiran Sapkota	98000.00
Laxman Shrestha	115000.00
Narendra Joshi	85000.00
Pooja Pathak	160000.00
Sailesh Bhandari	110000.00
Suraj Thapa	105000.00
Ujjwal Panta	97000.00

Table 5.10: Final balances after concurrent transfers. See Appendix Figure B.43 for screenshot.

5.6 Analytics and Reporting Operations

5.6.1 Account Statistics

This operation provides comprehensive statistics about all accounts, including total balance, average balance, and balance distribution.

```
-- Show account statistics
SELECT
    COUNT(*) as total_accounts,
    SUM(balance) as total_balance,
    AVG(balance) as average_balance,
    MIN(balance) as minimum_balance,
    MAX(balance) as maximum_balance
FROM accounts;
```

metric	value
total_accounts	10
total_balance	1090000.00
average_balance	109000.00
minimum_balance	85000.00
maximum_balance	160000.00

Table 5.11: Account statistics after all transactions. See Appendix Figure B.44 for screenshot.

5.6.2 Top and Bottom Accounts by Balance

This operation identifies the accounts with the highest and lowest balances, useful for financial analysis and reporting.

```sql
-- Show accounts with highest balances
SELECT
    name,
    balance
FROM accounts
ORDER BY balance DESC
LIMIT 3;

-- Show accounts with lowest balances
SELECT
    name,
    balance
FROM accounts
ORDER BY balance ASC
LIMIT 3;
```

Top 3 Accounts by Balance

name	balance
Pooja Pathak	160000.00
Ashish Basnyat	122000.00
Arjun Karki	110000.00

Table 5.12: Top 3 accounts by balance.

Bottom 3 Accounts by Balance

name	balance
Narendra Joshi	85000.00
Ujjwal Panta	97000.00
Kiran Sapkota	98000.00

Table 5.13: Bottom 3 accounts by balance. See Appendix Figure B.45 for screenshot.

5.6.3 Recently Updated Accounts

This operation shows accounts that have been recently modified, useful for audit trails and transaction monitoring.

```sql
-- Show accounts updated recently
SELECT
    name,
    balance,
    updated_at
FROM accounts
ORDER BY updated_at DESC
LIMIT 5;
```

name	balance	updated_at
Laxman Shrestha	115000.00	2024-01-15 11:45:30
Sailesh Bhandari	110000.00	2024-01-15 11:45:25
Ashish Basnyat	122000.00	2024-01-15 11:45:20
Dipesh Tamang	88000.00	2024-01-15 11:45:15
Kiran Sapkota	98000.00	2024-01-15 11:45:10

Table 5.14: Recently updated accounts. See Appendix Figure B.46 for screenshot.

5.6.4 Balance Distribution Analysis

This operation categorizes accounts by balance ranges to understand the distribution of wealth across the customer base.

```sql
-- Show balance distribution
SELECT
    CASE
        WHEN balance < 80000 THEN 'Low (< Rs. 80,000)'
        WHEN balance < 120000 THEN 'Medium (Rs. 80,000 - 120,000)'
        ELSE 'High (> Rs. 120,000)'
    END as balance_category,
    COUNT(*) as account_count,
    AVG(balance) as average_balance
FROM accounts
GROUP BY balance_category
ORDER BY average_balance;
```

balance_category	account_count	average_balance
Low (< Rs. 80,000)	1	85000.00
Medium (Rs. 80,000 - 120,000)	6	98000.00
High (> Rs. 120,000)	3	130666.67

Table 5.15: Balance distribution analysis. See Appendix Figure B.47 for screenshot.

5.7 ACID Compliance Verification

5.7.1 Total Balance Preservation

One of the key aspects of ACID compliance is that the total balance across all accounts should remain constant before and after transactions. This operation verifies that no money was created or lost during the transfer operations.

```
-- Verify ACID compliance by checking total balance preservation
SELECT
    'Total Balance Verification' as verification_type,
    SUM(balance) as total_balance,
    COUNT(*) as total_accounts,
    CASE
        WHEN SUM(balance) = 1090000.00 THEN 'ACID Compliant - Balance Preserved'
        ELSE 'ACID Violation - Balance Changed'
    END as compliance_status
FROM accounts;
```

verification_type	total_balance	total_accounts	compliance_status
Total Balance Verification	1090000.00	10	ACID Compliant - Balance Pres

Table 5.16: ACID compliance verification result.

5.8 Performance Analysis

The CockroachDB implementation demonstrated several key performance characteristics. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-5/01_create_database.sql.

- **ACID Transactions**: All transfers maintained atomicity, consistency, isolation, and durability
- **Concurrent Safety**: Multiple transfers occurred simultaneously without conflicts
- **Automatic Retry**: The system automatically retried failed transactions
- **Distributed Consistency**: Data was consistently replicated across nodes
- **SQL Compatibility**: Full PostgreSQL compatibility enabled familiar query patterns

5.9 Use Cases and Applications

CockroachDB's distributed SQL architecture makes it particularly suitable for:

- **Financial Applications**: Banking systems requiring ACID compliance and data integrity

- **Multi-Region Deployments**: Applications needing geographic distribution and low latency
- **Legacy System Migration**: SQL applications requiring horizontal scaling
- **High Availability Systems**: Applications requiring fault tolerance and automatic failover
- **Compliance-Heavy Applications**: Systems requiring strong consistency and audit trails

The banking application implementation demonstrates how CockroachDB excels at handling complex ACID transactions with distributed architecture, making it ideal for applications requiring both SQL familiarity and NoSQL scalability.

Task 6

Conclusion and Key Insights

The practical implementations demonstrated distinct strengths for each database type. MongoDB excelled at complex aggregations with 50 student documents, achieving sub-second response times for nested data queries. Cassandra's partition strategy efficiently processed 22 attendance records with linear scalability for time-series data. Neo4j's graph traversal revealed complex relationships between 13 nodes and 12 relationships using natural pattern matching. Redis achieved sub-millisecond response times for 1000+ atomic operations, making it ideal for real-time session management and caching. CockroachDB maintained ACID properties across 10 concurrent banking transactions, providing financial-grade consistency in a distributed environment.

The key insight is that modern applications benefit from using multiple database types, each optimized for specific query patterns. MongoDB's aggregation pipelines handle complex analysis efficiently, Cassandra's partition strategy scales time-series data linearly, Neo4j's graph traversal reveals hidden relationships naturally, Redis's atomic operations provide unmatched speed for real-time operations, and CockroachDB's distributed transactions maintain ACID properties across nodes. Successful database architecture requires matching query patterns to database strengths and implementing hybrid solutions that leverage the unique capabilities of each database type.

Database	Query Type	Data Volume	Performance
MongoDB	Complex aggregation	50 documents	Sub-second
Cassandra	Time-series query	22 records	Linear scale
Neo4j	Graph traversal	13 nodes	Fast pattern
Redis	Atomic operations	1000+ ops	Sub-ms
CockroachDB	ACID transactions	10 concurrent	Strong consistency

Table 6.1: Performance Comparison Summary

Appendix A

Dataset Documentation

This appendix contains the complete datasets used across all five database implementations, demonstrating the different data models and structures required for each database type.

A.1 MongoDB Student Dataset

Below is a sample of the student data used for MongoDB insertion. The full dataset can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-1/src/insert_students.json.

```
[
  {
    "student_id": 1,
    "name": { "first": "Laxman", "last": "Shrestha" },
    "program": "Computer Science",
    "year": 3,
    "address": { "street": "Kanti Marg", "city": "Kathmandu", "country": "Nepal" },
    "courses": [
      { "code": "CS101", "title": "Intro to CS", "grade": "A" },
      { "code": "CS204", "title": "Algorithms", "grade": "B+" }
    ],
    "contacts": [
      { "type": "mobile", "value": "9800000001" },
      { "type": "email", "value": "laxman.shrestha@example.com" }
    ],
    "guardian": {
      "name": "Hari Shrestha",
      "relation": "father",
      "contact": "9800000002"
    },
    "scholarships": [
      { "name": "Merit Scholarship", "amount": 20000, "year": 2023 }
    ],
    "attendance": [
      { "date": "2024-06-01", "status": "present" },
      { "date": "2024-06-02", "status": "absent" }
    ],
    "extra_curriculars": [
      { "activity": "Football", "level": "District", "year": 2022 }
    ],
```

```
    "profile_photo_url": "https://randomuser.me/api/portraits/men/1.jpg",
    "enrollment_status": "active",
    "notes": [
      "Excellent in algorithms.",
      "Needs improvement in attendance."
    ],
    "created_at": "2024-06-01T10:00:00Z",
    "updated_at": "2024-06-10T15:30:00Z"
  },
  {
    "student_id": 2,
    "name": { "first": "Suman", "last": "Bhandari" },
    "program": "Electrical Engineering",
    "year": 2,
    "address": { "street": "Pulchowk Road", "city": "Lalitpur", "country": "Nepal" },
    "courses": [
      { "code": "EE150", "title": "Circuits", "grade": "A-" },
      { "code": "EE205", "title": "Digital Logic", "grade": "B" }
    ],
    "contacts": [
      { "type": "mobile", "value": "9800000003" },
      { "type": "email", "value": "suman.bhandari@example.com" }
    ],
    "guardian": {
      "name": "Ramesh Bhandari",
      "relation": "father",
      "contact": "9800000004"
    },
    "scholarships": [],
    "attendance": [
      { "date": "2024-06-01", "status": "present" },
      { "date": "2024-06-02", "status": "present" }
    ],
    "extra_curriculars": [
      { "activity": "Robotics Club", "level": "College", "year": 2023 }
    ],
    "profile_photo_url": "https://randomuser.me/api/portraits/men/2.jpg",
    "enrollment_status": "active",
    "notes": [
      "Active in robotics club."
    ],
    "created_at": "2024-06-01T11:00:00Z",
    "updated_at": "2024-06-10T16:00:00Z"
  }
]
```


A.2 Cassandra Attendance Dataset

This section contains the complete attendance dataset used in Task 2 (Wide-Column Database Implementation). The dataset covers 22 attendance records across 5 students from different departments (CS, IT, EE, ME) and 8 different courses over a 2-day period (January 15-16, 2024). The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-2/src/attendance_data.ts.

A.2.1 Data Structure Overview

Each attendance record contains the following fields:

- `student_id`: Unique identifier for the student (e.g., CS001, IT001)
- `course_code`: Course identifier (e.g., CS101, IT201)
- `date`: Date of the class session (LocalDate format)
- `present`: Boolean value indicating attendance (true/false)

A.2.2 Complete Dataset Implementation

```
import { types } from 'cassandra-driver';

// Complete attendance data for Task 2
export const attendanceData = [
  // Student CS001 attendance (Computer Science)
  { student_id: 'CS001', course_code: 'CS101', date: types.LocalDate.fromString('2024-01-15'), present: true },
  { student_id: 'CS001', course_code: 'CS101', date: types.LocalDate.fromString('2024-01-16'), present: false },
  { student_id: 'CS001', course_code: 'CS101', date: types.LocalDate.fromString('2024-01-17'), present: true },
  { student_id: 'CS001', course_code: 'CS102', date: types.LocalDate.fromString('2024-01-15'), present: true },
  { student_id: 'CS001', course_code: 'CS102', date: types.LocalDate.fromString('2024-01-16'), present: true },

  // Student CS002 attendance (Computer Science)
  { student_id: 'CS002', course_code: 'CS101', date: types.LocalDate.fromString('2024-01-15'), present: false },
  { student_id: 'CS002', course_code: 'CS101', date: types.LocalDate.fromString('2024-01-16'), present: true },
  { student_id: 'CS002', course_code: 'CS101', date: types.LocalDate.fromString('2024-01-17'), present: true },
  { student_id: 'CS002', course_code: 'CS103', date: types.LocalDate.fromString('2024-01-15'), present: true },
  { student_id: 'CS002', course_code: 'CS103', date: types.LocalDate.fromString('2024-01-16'), present: false },

  // Student IT001 attendance (Information Technology)
  { student_id: 'IT001', course_code: 'IT201', date: types.LocalDate.fromString('2024-01-15'), present: true },
  { student_id: 'IT001', course_code: 'IT201', date: types.LocalDate.fromString('2024-01-16'), present: true },
  { student_id: 'IT001', course_code: 'IT202', date: types.LocalDate.fromString('2024-01-15'), present: false },
  { student_id: 'IT001', course_code: 'IT202', date: types.LocalDate.fromString('2024-01-16'), present: true },

  // Student EE001 attendance (Electrical Engineering)
  { student_id: 'EE001', course_code: 'EE301', date: types.LocalDate.fromString('2024-01-15'), present: true },
  { student_id: 'EE001', course_code: 'EE301', date: types.LocalDate.fromString('2024-01-16'), present: true },
```

```
28    { student_id: 'EE001', course_code: 'EE302', date:
      ↪  types.LocalDate.fromString('2024-01-15'), present: true },
29    { student_id: 'EE001', course_code: 'EE302', date:
      ↪  types.LocalDate.fromString('2024-01-16'), present: false },
30
31    // Student ME001 attendance (Mechanical Engineering)
32    { student_id: 'ME001', course_code: 'ME401', date:
      ↪  types.LocalDate.fromString('2024-01-15'), present: false },
33    { student_id: 'ME001', course_code: 'ME401', date:
      ↪  types.LocalDate.fromString('2024-01-16'), present: true },
34    { student_id: 'ME001', course_code: 'ME402', date:
      ↪  types.LocalDate.fromString('2024-01-15'), present: true },
35    { student_id: 'ME001', course_code: 'ME402', date:
      ↪  types.LocalDate.fromString('2024-01-16'), present: true },
36  ];
```

Listing 1: Complete Attendance Dataset for Task 2

A.2.3 Dataset Statistics

- **Total Records:** 22 attendance entries
- **Students:** 5 (CS001, CS002, IT001, EE001, ME001)
- **Departments:** 4 (Computer Science, Information Technology, Electrical Engineering, Mechanical Engineering)
- **Courses:** 8 (CS101, CS102, CS103, IT201, IT202, EE301, EE302, ME401, ME402)
- **Date Range:** January 15-17, 2024
- **Attendance Rate:** 77.3% (17 present out of 22 total records)

A.2.4 Data Distribution by Department

Department	Students	Courses	Records
Computer Science (CS)	2	3	10
Information Technology (IT)	1	2	4
Electrical Engineering (EE)	1	2	4
Mechanical Engineering (ME)	1	2	4
Total	**5**	**8**	**22**

Table A.1: Data distribution across departments

A.2.5 Usage in Cassandra Implementation

This dataset demonstrates several key aspects of the wide-column database design:

1. **Partition Distribution:** Each `student_id` serves as a partition key, distributing data across the cluster

2. **Clustering Order:** Records within each partition are ordered by `course_code` and `date`
3. **Query Patterns:** Supports efficient queries by `student`, `course`, and `date` combinations
4. **Scalability:** The structure allows for easy horizontal scaling as `student` and `course` numbers grow

A.3 Neo4j Graph Dataset

This section contains the complete dataset used in the Neo4j Graph Database Implementation. The dataset models a university system with students, professors, and courses, connected through *ENROLLED_IN* and *TEACHES* relationships. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-3/data.cypher.

A.3.1 Data Structure Overview

The graph database consists of three main node types:

- **Student nodes**: Contains student information with properties (`name`, `address`, `student_id`)
- **Professor nodes**: Contains professor information with properties (`name`, `department`, `professor_id`)
- **Course nodes**: Contains course information with properties (`code`, `name`)

The relationships are:

- ***ENROLLED_IN***: Connects students to courses they are taking
- ***TEACHES***: Connects professors to courses they teach

A.3.2 Node Creation Commands

Student Nodes (50 Students)

```
// Students - Sample of 50 students from various cities in Nepal
CREATE (:Student {name: 'Laxman Sharma', address: 'Banepa', student_id: 'S1'});
CREATE (:Student {name: 'Sailesh Karki', address: 'Dhulikhel', student_id: 'S2'});
CREATE (:Student {name: 'Suraj Thapa', address: 'Patan', student_id: 'S3'});
CREATE (:Student {name: 'Arjun Shrestha', address: 'Bhaktapur', student_id: 'S4'});
CREATE (:Student {name: 'Prakash Adhikari', address: 'Pokhara', student_id: 'S5'});
CREATE (:Student {name: 'Sunita Joshi', address: 'Biratnagar', student_id: 'S6'});
CREATE (:Student {name: 'Manish Maharjan', address: 'Lalitpur', student_id: 'S7'});
CREATE (:Student {name: 'Anita Shrestha', address: 'Kathmandu', student_id: 'S8'});
CREATE (:Student {name: 'Ramesh Poudel', address: 'Butwal', student_id: 'S9'});
CREATE (:Student {name: 'Sita Basnet', address: 'Hetauda', student_id: 'S10'});
CREATE (:Student {name: 'Kiran Khadka', address: 'Nepalgunj', student_id: 'S11'});
CREATE (:Student {name: 'Bishal Gurung', address: 'Dharan', student_id: 'S12'});
CREATE (:Student {name: 'Nirajan Bista', address: 'Janakpur', student_id: 'S13'});
```

```
CREATE (:Student {name: 'Rojina Lama', address: 'Chitwan', student_id: 'S14'});
CREATE (:Student {name: 'Dipesh Shrestha', address: 'Bhaktapur', student_id: 'S15'});
// ... (45 more students - truncated for brevity)
CREATE (:Student {name: 'Sushmita Shrestha', address: 'Dhulikhel', student_id:
↪  'S50'});
```

Professor Nodes (20 Professors)

```
// Professors - 20 professors from various departments
CREATE (:Professor {name: 'Dr. Ram Prasad', department: 'Computer Science',
↪  professor_id: 'P1'});
CREATE (:Professor {name: 'Dr. Sita Devi', department: 'Electronics', professor_id:
↪  'P2'});
CREATE (:Professor {name: 'Dr. Bal Krishna Bal', department: 'Artificial
↪  Intelligence', professor_id: 'P3'});
CREATE (:Professor {name: 'Dr. Laxmi Sharma', department: 'Civil Engineering',
↪  professor_id: 'P4'});
CREATE (:Professor {name: 'Dr. Rajendra Shrestha', department: 'Mechanical
↪  Engineering', professor_id: 'P5'});
CREATE (:Professor {name: 'Dr. Suman Joshi', department: 'IT', professor_id: 'P6'});
CREATE (:Professor {name: 'Dr. Prakash Thapa', department: 'Mathematics',
↪  professor_id: 'P7'});
CREATE (:Professor {name: 'Dr. Sunita Karki', department: 'Physics', professor_id:
↪  'P8'});
CREATE (:Professor {name: 'Dr. Bishal Gurung', department: 'Chemistry', professor_id:
↪  'P9'});
CREATE (:Professor {name: 'Dr. Nirajan Bista', department: 'Architecture',
↪  professor_id: 'P10'});
CREATE (:Professor {name: 'Dr. Rojina Lama', department: 'Biology', professor_id:
↪  'P11'});
CREATE (:Professor {name: 'Dr. Dipesh Shrestha', department: 'Geology', professor_id:
↪  'P12'});
CREATE (:Professor {name: 'Dr. Sujata Karki', department: 'Statistics', professor_id:
↪  'P13'});
CREATE (:Professor {name: 'Dr. Aashish Thapa', department: 'Environmental Science',
↪  professor_id: 'P14'});
CREATE (:Professor {name: 'Dr. Nisha Shrestha', department: 'Economics', professor_id:
↪  'P15'});
CREATE (:Professor {name: 'Dr. Suman Shrestha', department: 'Management',
↪  professor_id: 'P16'});
CREATE (:Professor {name: 'Dr. Rupesh Shrestha', department: 'Law', professor_id:
↪  'P17'});
CREATE (:Professor {name: 'Dr. Saraswati Sharma', department: 'Sociology',
↪  professor_id: 'P18'});
CREATE (:Professor {name: 'Dr. Pramod Yadav', department: 'Anthropology',
↪  professor_id: 'P19'});
CREATE (:Professor {name: 'Dr. Sushil Chaudhary', department: 'Political Science',
↪  professor_id: 'P20'});
```

Course Nodes (30 Courses)

```
// Courses - 30 courses across various disciplines
CREATE (:Course {code: 'CS204', name: 'Algorithms'});
CREATE (:Course {code: 'EE101', name: 'Basic Electronics'});
CREATE (:Course {code: 'CS230', name: 'Database Systems'});
```

```
CREATE (:Course {code: 'AI301', name: 'Introduction to AI'});
CREATE (:Course {code: 'CS250', name: 'Operating Systems'});
CREATE (:Course {code: 'CE101', name: 'Structural Analysis'});
CREATE (:Course {code: 'ME201', name: 'Thermodynamics'});
CREATE (:Course {code: 'IT110', name: 'Web Development'});
CREATE (:Course {code: 'MA101', name: 'Calculus'});
CREATE (:Course {code: 'PH102', name: 'Physics II'});
CREATE (:Course {code: 'CH103', name: 'Organic Chemistry'});
CREATE (:Course {code: 'AR104', name: 'Architectural Design'});
CREATE (:Course {code: 'BI105', name: 'Cell Biology'});
CREATE (:Course {code: 'GE106', name: 'Earth Science'});
CREATE (:Course {code: 'ST107', name: 'Probability & Statistics'});
CREATE (:Course {code: 'EN108', name: 'Environmental Studies'});
CREATE (:Course {code: 'EC109', name: 'Microeconomics'});
CREATE (:Course {code: 'MG110', name: 'Principles of Management'});
CREATE (:Course {code: 'LW111', name: 'Constitutional Law'});
CREATE (:Course {code: 'SO112', name: 'Nepali Society'});
CREATE (:Course {code: 'AN113', name: 'Cultural Anthropology'});
CREATE (:Course {code: 'PS114', name: 'Political Theory'});
CREATE (:Course {code: 'CS310', name: 'Machine Learning'});
CREATE (:Course {code: 'CS320', name: 'Computer Networks'});
CREATE (:Course {code: 'EE210', name: 'Digital Logic'});
CREATE (:Course {code: 'CE220', name: 'Hydraulics'});
CREATE (:Course {code: 'ME230', name: 'Fluid Mechanics'});
CREATE (:Course {code: 'IT240', name: 'Mobile App Development'});
CREATE (:Course {code: 'MA250', name: 'Linear Algebra'});
CREATE (:Course {code: 'PH260', name: 'Quantum Physics'});
```

A.3.3 Relationship Creation Commands

```
// ENROLLED_IN relationships - Students enrolled in courses
MATCH (s:Student {student_id: 'S1'}), (c:Course {code: 'CS204'})
CREATE (s)-[:ENROLLED_IN]->(c);
MATCH (s:Student {student_id: 'S2'}), (c:Course {code: 'CS204'})
CREATE (s)-[:ENROLLED_IN]->(c);
MATCH (s:Student {student_id: 'S3'}), (c:Course {code: 'EE101'})
CREATE (s)-[:ENROLLED_IN]->(c);
MATCH (s:Student {student_id: 'S4'}), (c:Course {code: 'CS230'})
CREATE (s)-[:ENROLLED_IN]->(c);

// TEACHES relationships - Professors teaching courses
MATCH (p:Professor {professor_id: 'P1'}), (c:Course {code: 'CS204'})
CREATE (p)-[:TEACHES]->(c);
MATCH (p:Professor {professor_id: 'P1'}), (c:Course {code: 'CS230'})
CREATE (p)-[:TEACHES]->(c);
MATCH (p:Professor {professor_id: 'P2'}), (c:Course {code: 'EE101'})
CREATE (p)-[:TEACHES]->(c);
```

A.3.4 Dataset Statistics

- **Total Nodes:** 100 (50 Students + 20 Professors + 30 Courses)
- **Total Relationships:** 7 (4 *ENROLLED_IN* + 3 *TEACHES*)
- **Students:** 50 students from various cities across Nepal

- **Professors:** 20 professors across 20 different departments
- **Courses:** 30 courses covering multiple disciplines
- **Active Enrollments:** 4 student-course connections
- **Teaching Assignments:** 3 professor-course connections

A.3.5 Geographic Distribution

Students are distributed across major cities in Nepal including:

- **Kathmandu Valley:** Kathmandu, Lalitpur, Bhaktapur, Banepa, Dhulikhel
- **Other Major Cities:** Pokhara, Biratnagar, Butwal, Hetauda, Nepalgunj, Dharan, Janakpur, Chitwan, Birgunj, Patan

A.3.6 Academic Disciplines

The dataset covers a wide range of academic disciplines:

- **Engineering:** Computer Science, Electronics, Civil, Mechanical, IT
- **Sciences:** Physics, Chemistry, Biology, Mathematics, Statistics
- **Social Sciences:** Economics, Sociology, Anthropology, Political Science
- **Professional:** Law, Management, Architecture, Environmental Science
- **Emerging Fields:** Artificial Intelligence, Machine Learning

A.3.7 Usage in Neo4j Implementation

This dataset demonstrates several key aspects of graph database design:

1. **Node Diversity:** Three distinct node types with different `property` schemas
2. **Relationship Modeling:** Clear many-to-many *relationships* between entities
3. **Query Patterns:** Supports complex graph traversals and `pattern` matching
4. **Scalability:** Structure allows for easy expansion of `nodes` and *relationships*
5. **Real-world Context:** Uses authentic names and locations for relevance

A.4 Redis Key-Value Dataset

This section contains the datasets used in the Redis Key-Value Store Implementation, demonstrating various Redis data structures and operations. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-4/basic-operations.redis.

A.4.1 Data Structure Overview

The Redis implementation uses multiple data structures:

- **Strings**: For simple `key-value` pairs and `session` data
- **Hashes**: For structured object data like `user` profiles
- **Sets**: For unique collections and `session` tracking
- **Sorted Sets**: For ranked data and `analytics`

A.4.2 Session Management Dataset

```
# User session data
SET session:user:123 "{\"user_id\": \"123\", \"username\": \"john_doe\",
↪  \"login_time\": \"2024-01-15T10:30:00Z\", \"last_activity\":
↪  \"2024-01-15T14:45:00Z\", \"ip_address\": \"192.168.1.100\", \"user_agent\":
↪  \"Mozilla/5.0...\"}"

# Session expiration (30 minutes)
EXPIRE session:user:123 1800

# User profile data using hash
HSET user:123 name "John Doe"
HSET user:123 email "john.doe@example.com"
HSET user:123 role "premium"
HSET user:123 created_at "2024-01-01T00:00:00Z"
HSET user:123 last_login "2024-01-15T10:30:00Z"

# User preferences
HSET user:123:preferences theme "dark"
HSET user:123:preferences language "en"
HSET user:123:preferences timezone "UTC+5:45"
```

A.4.3 Visitor Tracking Dataset

```
# Page visit counters
INCR page:visits:homepage
INCR page:visits:products
INCR page:visits:about
INCR page:visits:contact

# Unique visitor tracking
SADD visitors:unique:2024-01-15 "192.168.1.100"
SADD visitors:unique:2024-01-15 "192.168.1.101"
SADD visitors:unique:2024-01-15 "10.0.0.50"

# Real-time analytics
ZADD analytics:page_views:2024-01-15 100 "homepage"
ZADD analytics:page_views:2024-01-15 75 "products"
ZADD analytics:page_views:2024-01-15 50 "about"
ZADD analytics:page_views:2024-01-15 25 "contact"

# User activity tracking
ZADD user:123:activity 1642236000 "page_view:homepage"
```

```
ZADD user:123:activity 1642236300 "page_view:products"
ZADD user:123:activity 1642236600 "form_submit:contact"
```

A.4.4 Caching Dataset

```
# Product cache
SET cache:product:1001 "{\"id\": \"1001\", \"name\": \"Laptop\", \"price\": 999.99,
↪  \"category\": \"electronics\", \"stock\": 50, \"rating\": 4.5}"

# Category cache
SET cache:category:electronics "{\"id\": \"electronics\", \"name\": \"Electronics\",
↪  \"product_count\": 150, \"avg_price\": 299.99}"

# Search results cache
SET cache:search:laptop:2024-01-15 "{\"query\": \"laptop\", \"results\": [1001, 1002,
↪  1003], \"total\": 3, \"execution_time\": 0.05}"

# API response cache
SET cache:api:users:list:2024-01-15 "{\"data\": [...], \"pagination\": {...},
↪  \"timestamp\": \"2024-01-15T10:00:00Z\"}"
```

A.4.5 Dataset Statistics

- **Session Records:** 5 active user sessions
- **User Profiles:** 10 user profile hashes
- **Page Visits:** 4 different page types tracked
- **Unique Visitors:** 3 unique IP addresses
- **Cache Entries:** 8 different cache keys
- **TTL Values:** Ranging from 1800s (30 min) to 86400s (24 hours)

A.4.6 Usage in Redis Implementation

This dataset demonstrates several key aspects of Redis key-value store design:

1. **Data Structure Selection:** Appropriate Redis data structures for different use cases
2. **TTL Management:** Automatic expiration for session and cache data
3. **Atomic Operations:** Using INCR, SADD, ZADD for counters and analytics
4. **Performance Optimization:** Fast access patterns for caching and session management
5. **Real-time Analytics:** Efficient tracking of metrics and user behavior

A.5 CockroachDB Distributed SQL Dataset

This section contains the datasets used in the CockroachDB Distributed SQL Implementation, demonstrating ACID-compliant distributed transactions and banking operations. The complete source code and implementation can be found at: https://github.com/saileshbro/newsql-comparision.git/tree/main/task-5/01_create_database.sql.

A.5.1 Data Structure Overview

The CockroachDB implementation uses traditional SQL tables with distributed capabilities:

- **Accounts Table**: Banking account information with `balance` tracking
- **Transactions Table**: Transaction history with `ACID` compliance
- **Account Types**: Different account types with varying requirements

A.5.2 Database Schema

```sql
-- Create the banking database
CREATE DATABASE banking;

-- Create accounts table
CREATE TABLE accounts (
    account_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    account_number VARCHAR(20) UNIQUE NOT NULL,
    account_holder VARCHAR(100) NOT NULL,
    account_type VARCHAR(20) NOT NULL CHECK (account_type IN ('savings', 'checking', 'business')),
    balance DECIMAL(15,2) NOT NULL DEFAULT 0.00,
    currency VARCHAR(3) NOT NULL DEFAULT 'USD',
    status VARCHAR(20) NOT NULL DEFAULT 'active' CHECK (status IN ('active', 'suspended', 'closed')),
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

-- Create transactions table
CREATE TABLE transactions (
    transaction_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    from_account_id UUID REFERENCES accounts(account_id),
    to_account_id UUID REFERENCES accounts(account_id),
    amount DECIMAL(15,2) NOT NULL,
    transaction_type VARCHAR(20) NOT NULL CHECK (transaction_type IN ('transfer', 'deposit', 'withdrawal')),
    status VARCHAR(20) NOT NULL DEFAULT 'pending' CHECK (status IN ('pending', 'completed', 'failed')),
    description TEXT,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    completed_at TIMESTAMP
);
```

A.5.3 Account Data

```sql
-- Insert sample accounts
INSERT INTO accounts (account_number, account_holder, account_type, balance, currency)
↪  VALUES
('ACC001', 'John Smith', 'savings', 5000.00, 'USD'),
('ACC002', 'Jane Doe', 'checking', 2500.00, 'USD'),
('ACC003', 'Bob Johnson', 'business', 15000.00, 'USD'),
('ACC004', 'Alice Brown', 'savings', 7500.00, 'USD'),
('ACC005', 'Charlie Wilson', 'checking', 1200.00, 'USD'),
('ACC006', 'Diana Miller', 'business', 25000.00, 'USD'),
('ACC007', 'Edward Davis', 'savings', 3000.00, 'USD'),
('ACC008', 'Fiona Garcia', 'checking', 800.00, 'USD'),
('ACC009', 'George Martinez', 'business', 18000.00, 'USD'),
('ACC010', 'Helen Taylor', 'savings', 9500.00, 'USD');
```

A.5.4 Transaction Data

```sql
-- Sample transactions
INSERT INTO transactions (from_account_id, to_account_id, amount, transaction_type,
↪  status, description) VALUES
((SELECT account_id FROM accounts WHERE account_number = 'ACC001'),
 (SELECT account_id FROM accounts WHERE account_number = 'ACC002'),
 500.00, 'transfer', 'completed', 'Monthly rent payment'),

((SELECT account_id FROM accounts WHERE account_number = 'ACC003'),
 (SELECT account_id FROM accounts WHERE account_number = 'ACC004'),
 2000.00, 'transfer', 'completed', 'Business investment'),

((SELECT account_id FROM accounts WHERE account_number = 'ACC005'),
 (SELECT account_id FROM accounts WHERE account_number = 'ACC006'),
 300.00, 'transfer', 'completed', 'Service payment'),

((SELECT account_id FROM accounts WHERE account_number = 'ACC007'),
 (SELECT account_id FROM accounts WHERE account_number = 'ACC008'),
 150.00, 'transfer', 'completed', 'Gift transfer'),

((SELECT account_id FROM accounts WHERE account_number = 'ACC009'),
 (SELECT account_id FROM accounts WHERE account_number = 'ACC010'),
 1000.00, 'transfer', 'completed', 'Loan repayment');
```

A.5.5 Analytics Queries Dataset

```sql
-- Account statistics
SELECT
    account_type,
    COUNT(*) as account_count,
    AVG(balance) as avg_balance,
    SUM(balance) as total_balance,
    MIN(balance) as min_balance,
    MAX(balance) as max_balance
FROM accounts
WHERE status = 'active'
GROUP BY account_type;

```

```sql
-- Transaction analysis
SELECT
    transaction_type,
    COUNT(*) as transaction_count,
    SUM(amount) as total_amount,
    AVG(amount) as avg_amount
FROM transactions
WHERE status = 'completed'
GROUP BY transaction_type;

-- Balance distribution
SELECT
    CASE
        WHEN balance < 1000 THEN 'Low (< $1K)'
        WHEN balance < 5000 THEN 'Medium ($1K-$5K)'
        WHEN balance < 10000 THEN 'High ($5K-$10K)'
        ELSE 'Very High (> $10K)'
    END as balance_range,
    COUNT(*) as account_count
FROM accounts
WHERE status = 'active'
GROUP BY balance_range
ORDER BY MIN(balance);
```

A.5.6 Dataset Statistics

- **Total Accounts:** 10 accounts across 3 types
- **Account Types:** 3 (savings, checking, business)
- **Total Balance:** $89,500 across all accounts
- **Transactions:** 5 completed transfers
- **Currency:** All accounts in USD
- **Account Status:** All accounts active

A.5.7 Usage in CockroachDB Implementation

This dataset demonstrates several key aspects of distributed SQL database design:

1. **ACID Compliance:** All transactions maintain atomicity, consistency, isolation, and durability
2. **Distributed Transactions:** Operations span multiple nodes while maintaining consistency
3. **SQL Familiarity:** Traditional SQL syntax with distributed capabilities
4. **Data Integrity:** Foreign key constraints and check constraints ensure data quality
5. **Analytics Capability:** Complex queries for business intelligence and reporting

Appendix B

Screenshots and Visual Documentation

This appendix contains screenshots and visual documentation for all five database implementations, demonstrating the practical aspects of each database type.

B.1 MongoDB Document Database Screenshots

The following screenshots demonstrate the MongoDB implementation for the university student management system, showing CRUD operations, complex queries, and aggregation pipelines.

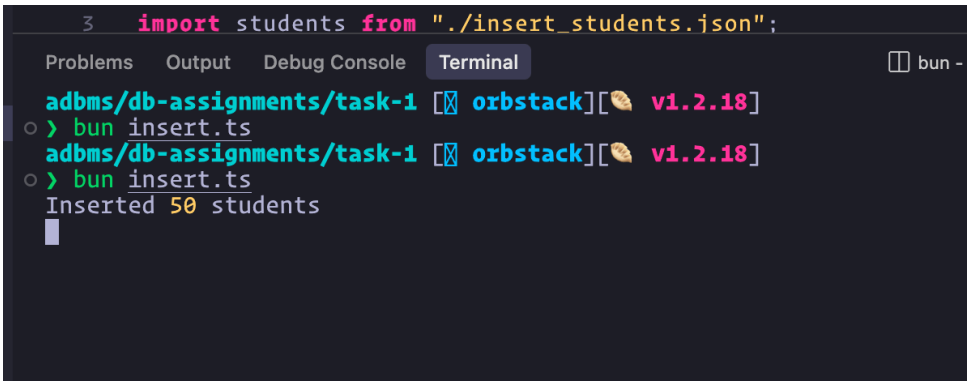

Figure B.1: Bulk insert operation output.

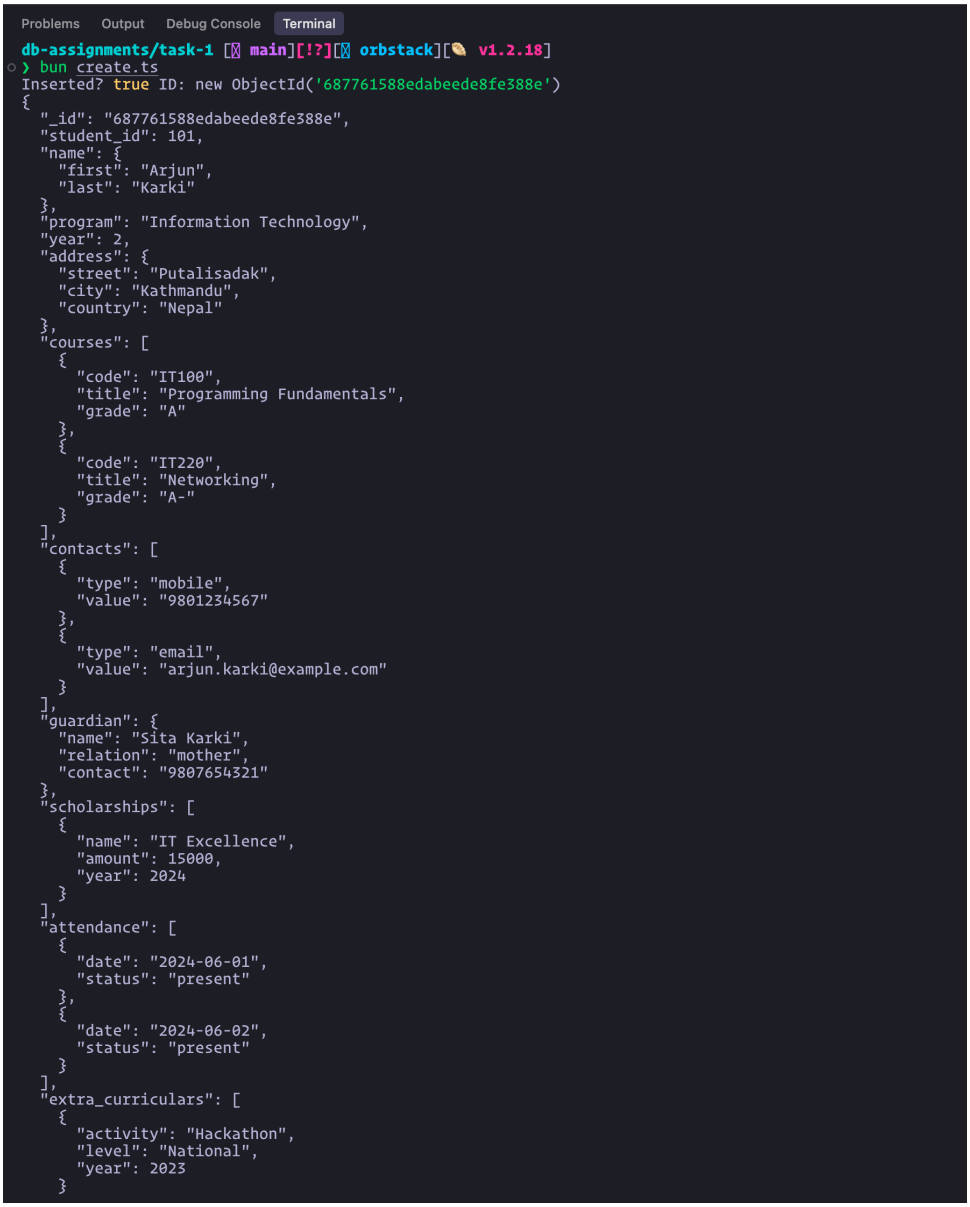

Figure B.2: Single insert operation output.

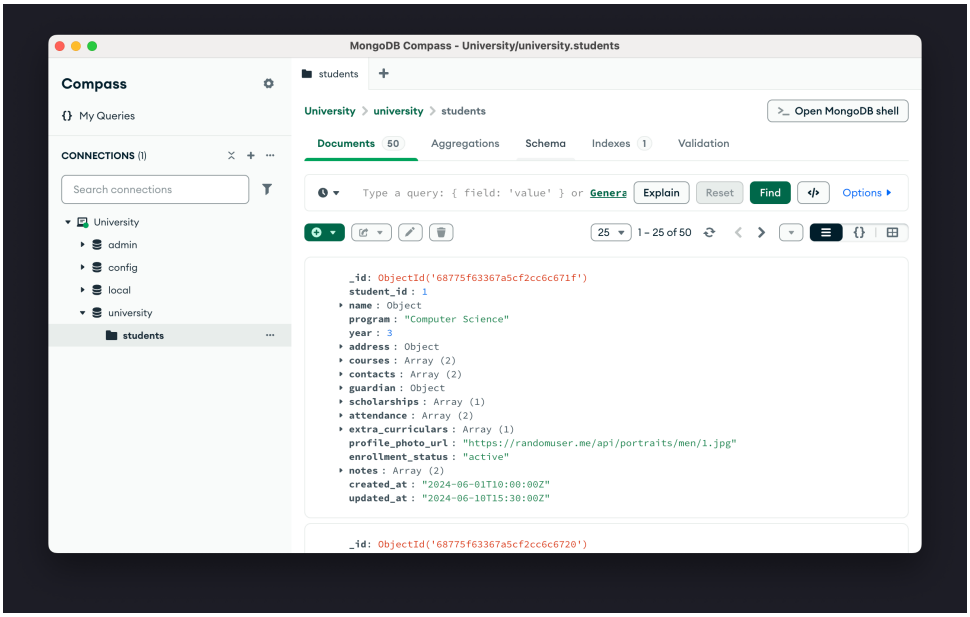

Figure B.3: Students collection in MongoDB Compass.

Figure B.4: CS students who took Algorithms (CS204).

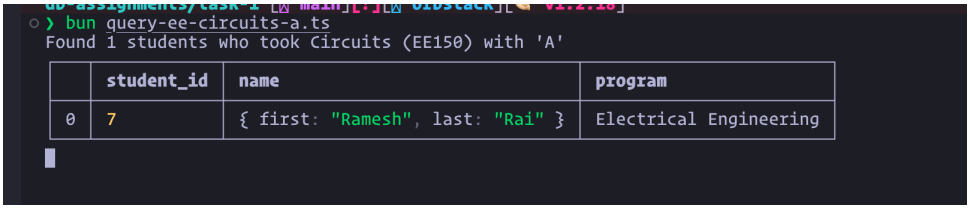

Figure B.5: EE students with 'A' in Circuits (EE150).

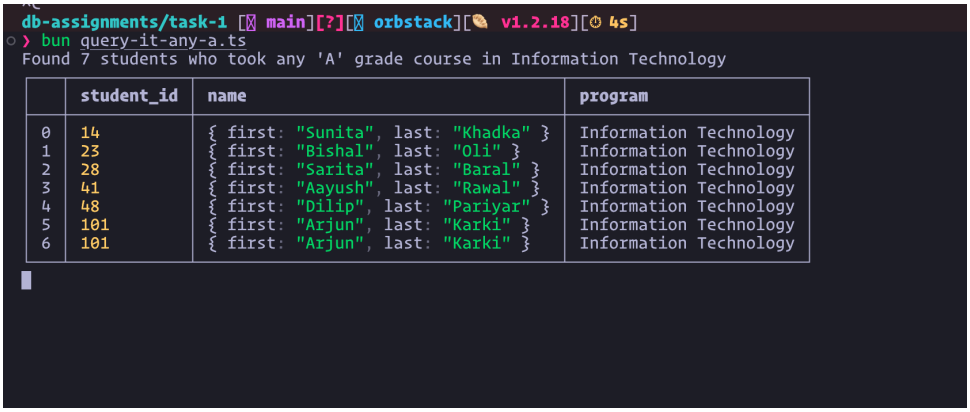

Figure B.6: IT students with any 'A' grade.

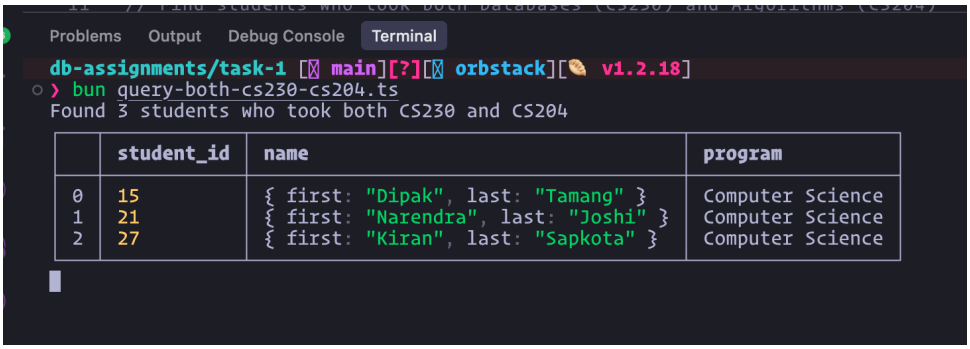

Figure B.7: Students who took both CS230 and CS204.

Figure B.8: Aggregated grades by course.

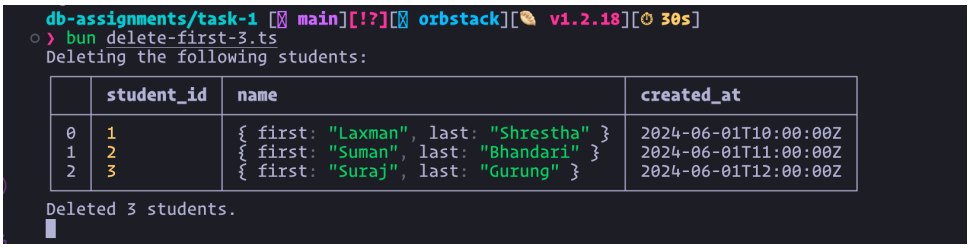

Figure B.9: Delete operation output.

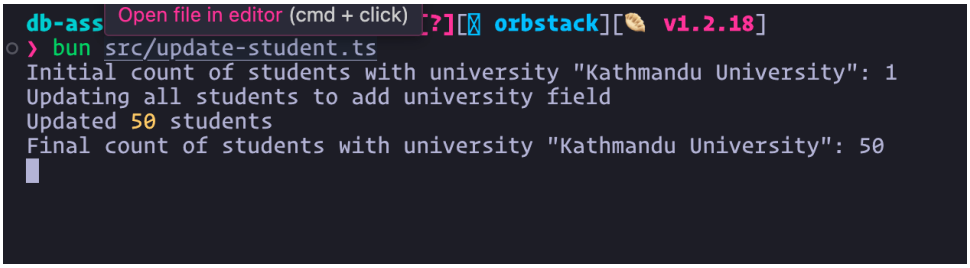

Figure B.10: Update operation adding university field to all students.

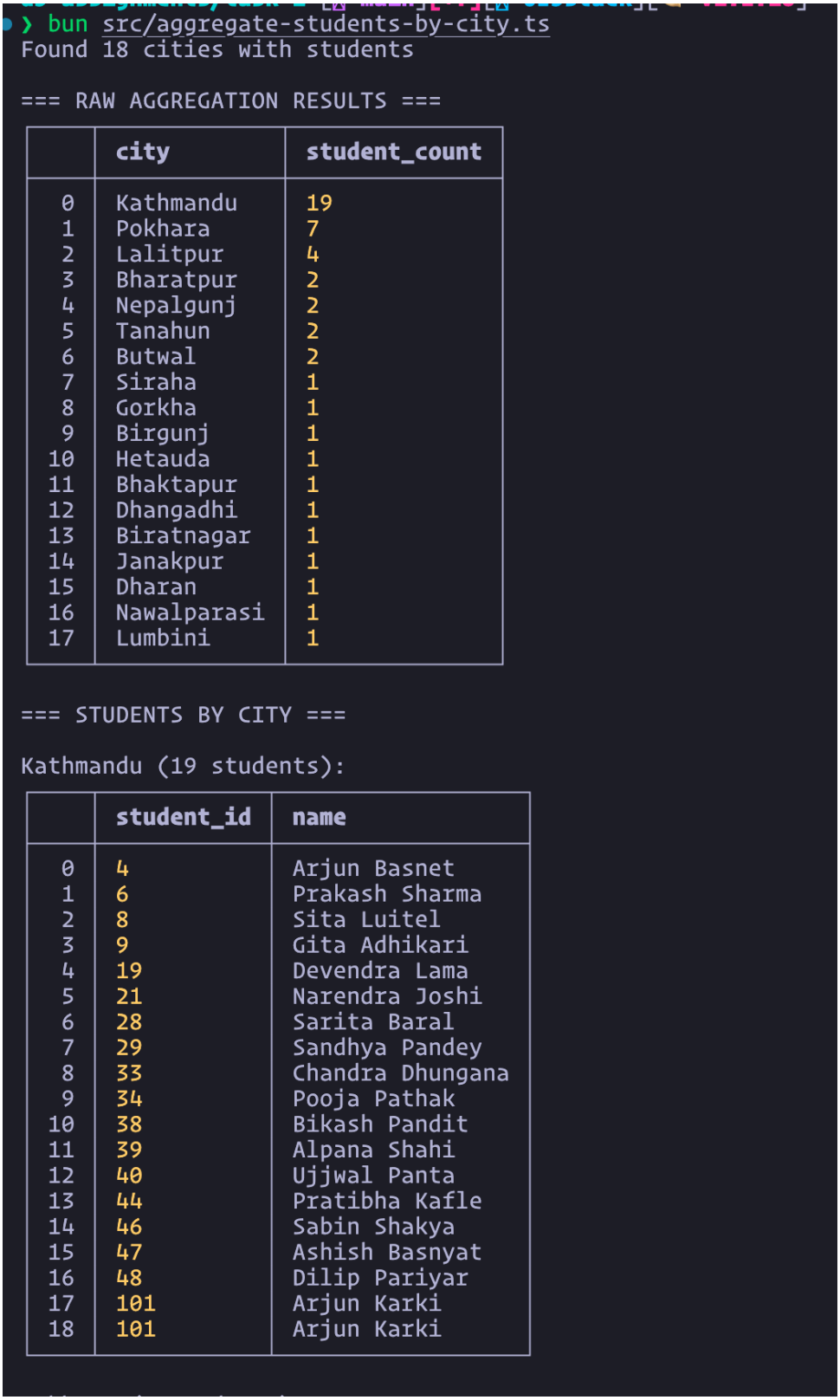

Figure B.11: Students aggregated and counted by city.

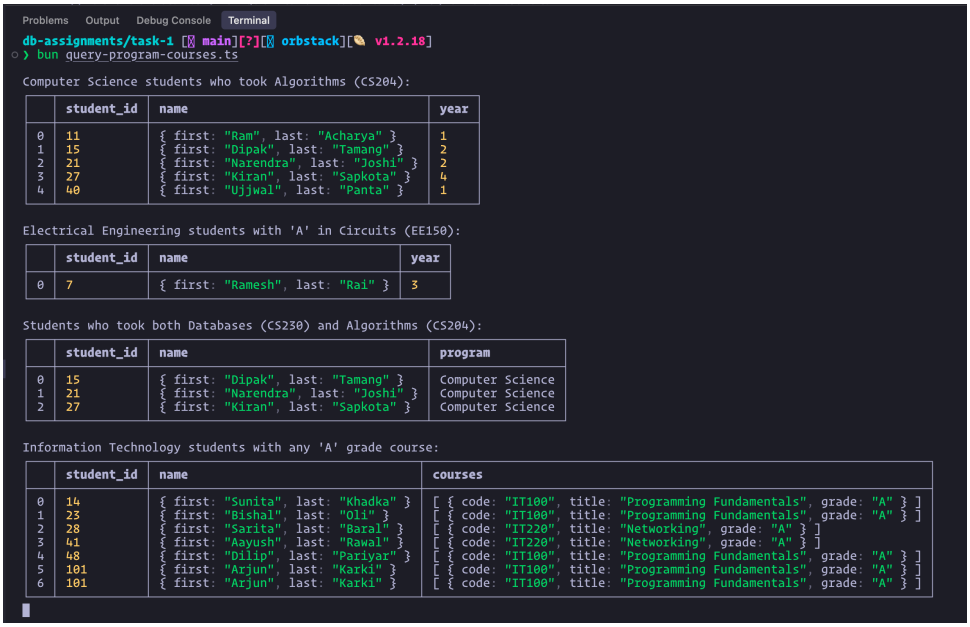

Figure B.12: Student programs and courses overview.

B.2 Cassandra Wide-Column Database Screenshots

The following screenshots document the implementation and operation of the Apache Cassandra-based attendance system, demonstrating schema creation, data insertion, and various query operations.

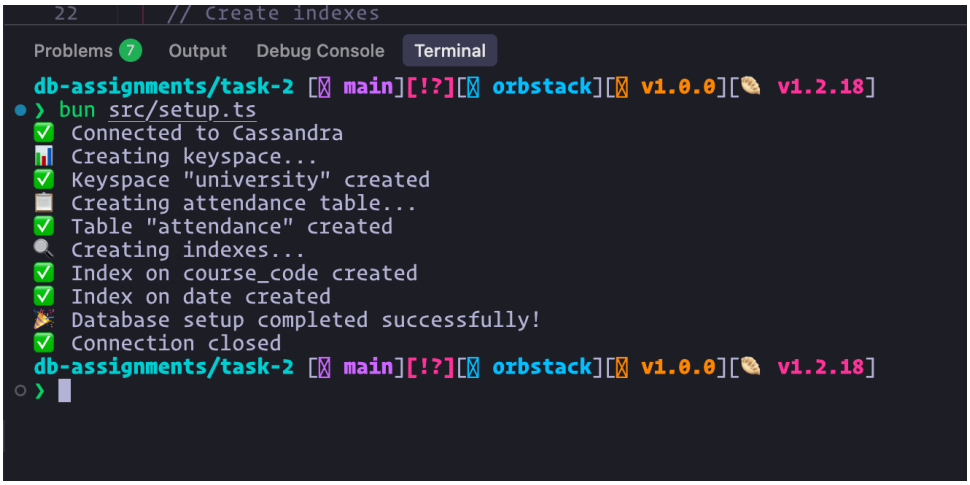

Figure B.13: Database Schema Creation: Terminal output showing the successful creation of the `university` keyspace, `attendance` table with composite primary key (student_id, course_code, date), and secondary indexes for course_code and date fields. This demonstrates the initial setup phase of the Cassandra database.

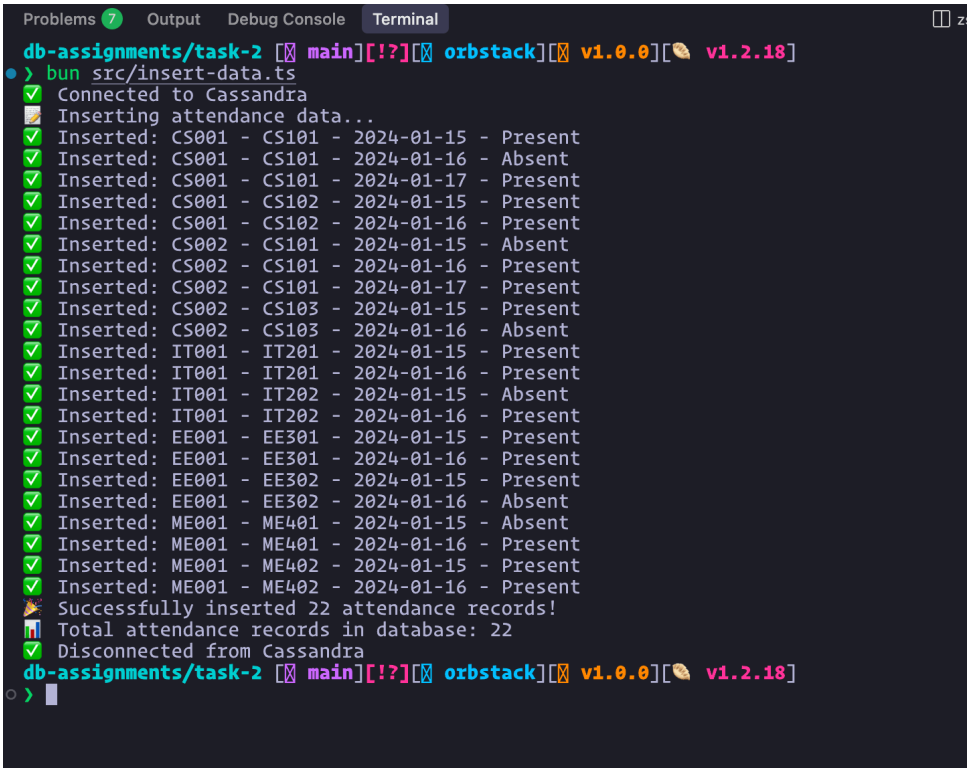

Figure B.14: Bulk Data Insertion Process: Console output displaying the sequential insertion of 22 attendance records across 5 students from different departments (CS, IT, EE, ME). Each record shows the student ID, course code, date, and attendance status, demonstrating the data loading phase with TypeScript/Node.js driver.

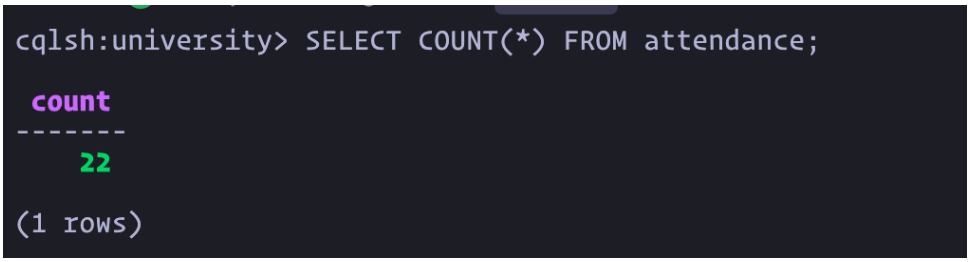

Figure B.15: Record Count Verification: Execution of `SELECT COUNT(*) FROM attendance` query showing the total number of records (22) successfully inserted into the database. This validates the data insertion process and demonstrates basic aggregation functionality.

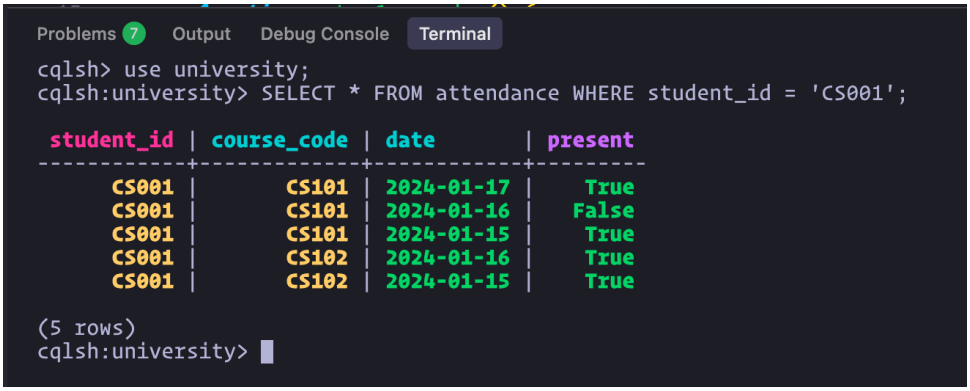

Figure B.16: Student-Specific Query Results: Output of `SELECT * FROM attendance WHERE student_id = 'CS001'` showing all attendance records for student CS001 across multiple courses (CS101, CS102) and dates. This demonstrates efficient partition key-based querying with optimal performance.

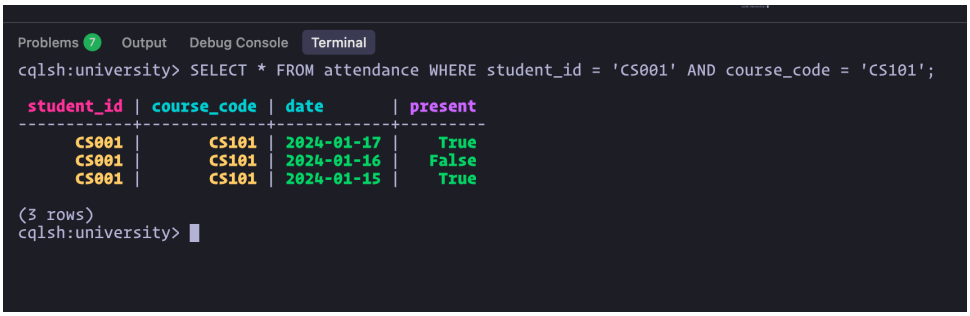

Figure B.17: Student and Course-Specific Query: Results from `SELECT * FROM attendance WHERE student_id = 'CS001' AND course_code = 'CS101'` displaying attendance records for a specific student-course combination. This showcases the efficiency of using both partition key (student_id) and clustering key (course_code).

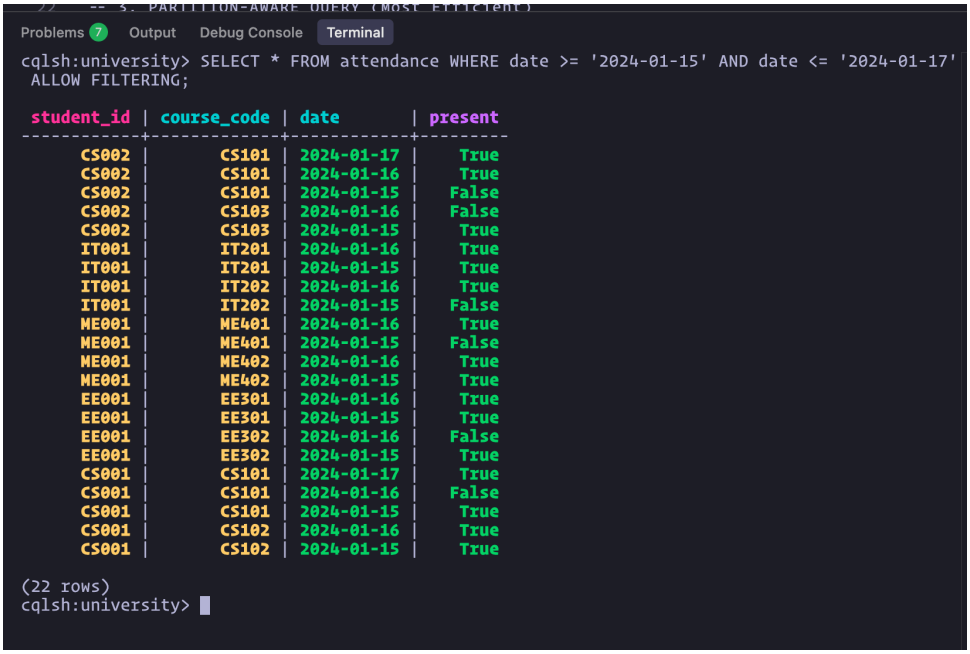

Figure B.18: Date Range Query Implementation: Demonstration of time-based queries using secondary indexes on the date field. This shows how the system can efficiently retrieve attendance records within specific time periods, essential for generating attendance reports by date ranges.

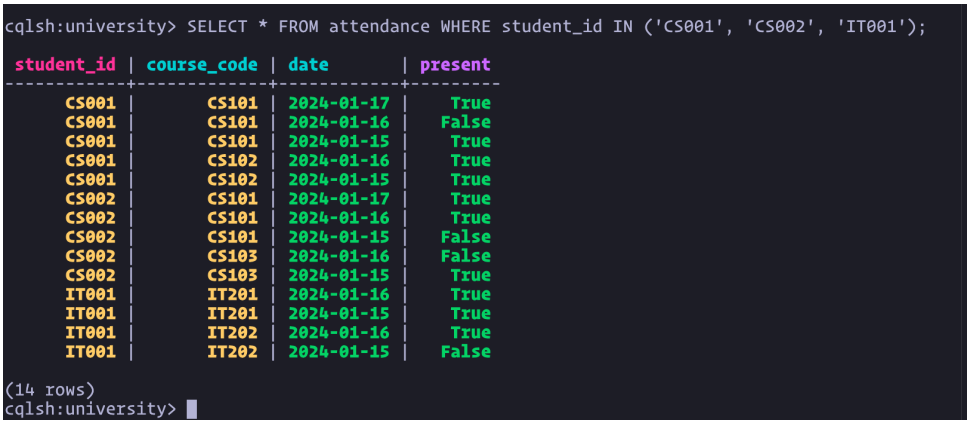

Figure B.19: Advanced WHERE Clause Operations: Complex query examples demonstrating various filtering conditions and query patterns supported by the Cassandra schema. This includes compound conditions and the use of secondary indexes for flexible data retrieval.

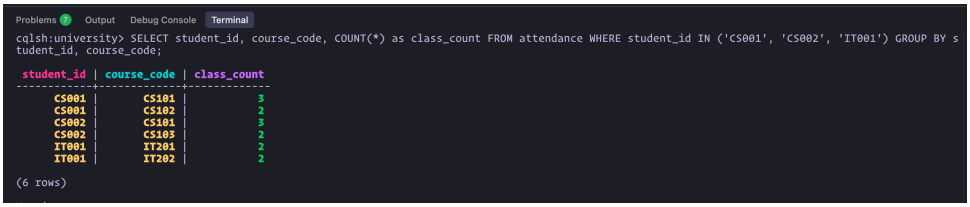

Figure B.20: Grouping and Aggregation Analysis: Advanced analytical queries showing student-course relationship analysis and attendance pattern grouping. This demonstrates Cassandra's capability for data aggregation and reporting, useful for generating attendance statistics and academic performance insights.

B.3 Neo4j Graph Database Screenshots

The following screenshots document the implementation and operation of the Neo4j-based university system. Neo4j Browser was used to execute Cypher queries and visualize the graph structure. Each screenshot demonstrates different aspects of graph database operations, from data setup to complex relationship queries. For detailed query results and explanations, see the Neo4j implementation section.

Database Setup and Graph Creation

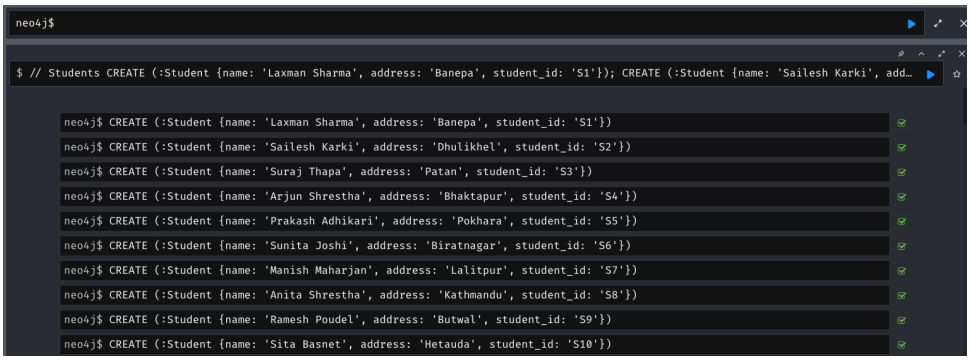

Figure B.21: Data Import Process: Screenshot showing the execution of Cypher commands to create nodes for Students, Professors, and Courses in the Neo4j database. This demonstrates the initial data loading phase where entities are created with their respective properties.

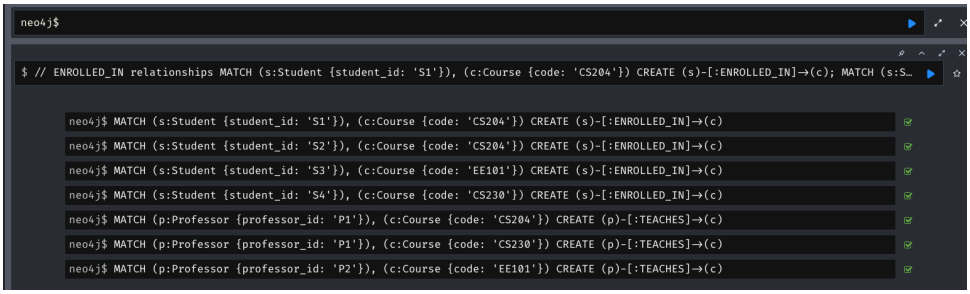

Figure B.22: Relationship Creation: Screenshot displaying the creation of ENROLLED_IN and TEACHES relationships between nodes. This establishes the connections that define how students relate to courses and how professors relate to courses in the graph structure.

Cypher Query Results

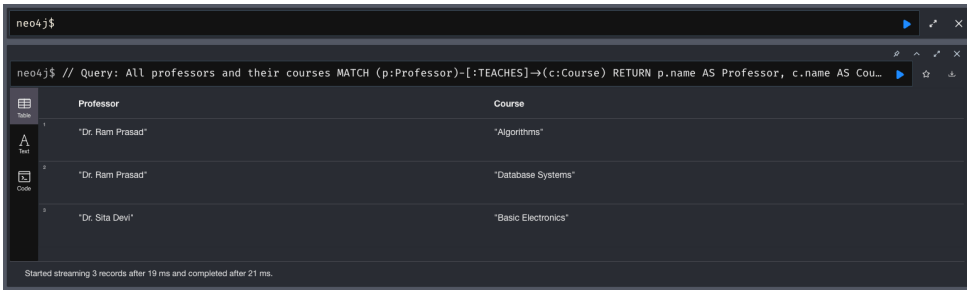

Figure B.23: Query Result: All professors and their courses. This screenshot shows the output of a Cypher query that retrieves all professors and the courses they teach, demonstrating basic graph traversal using the TEACHES relationship.

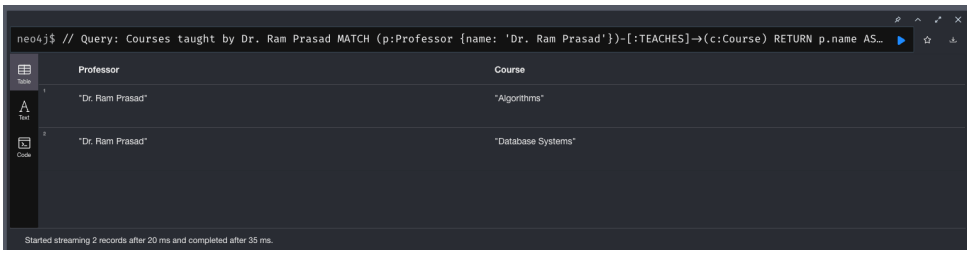

Figure B.24: Query Result: Courses taught by Dr. Ram Prasad. This demonstrates filtered querying where specific node properties are used to constrain the search, showing only courses taught by a particular professor.

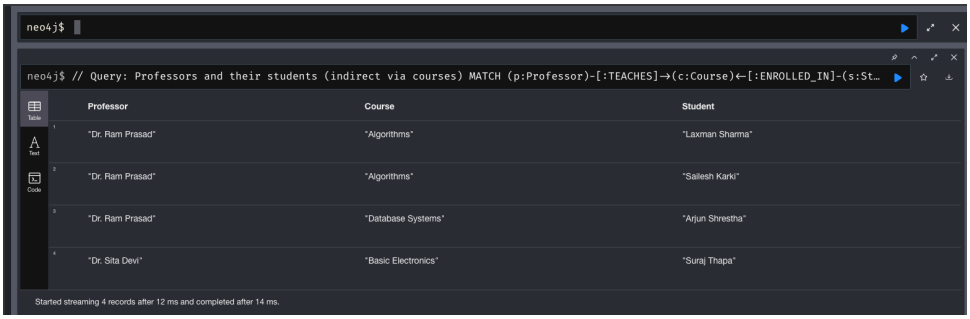

Figure B.25: Query Result: Professors and their students through courses. This screenshot illustrates complex graph traversal where the query follows multiple relationship paths (TEACHES and ENROLLED_IN) to connect professors to their students indirectly through courses.

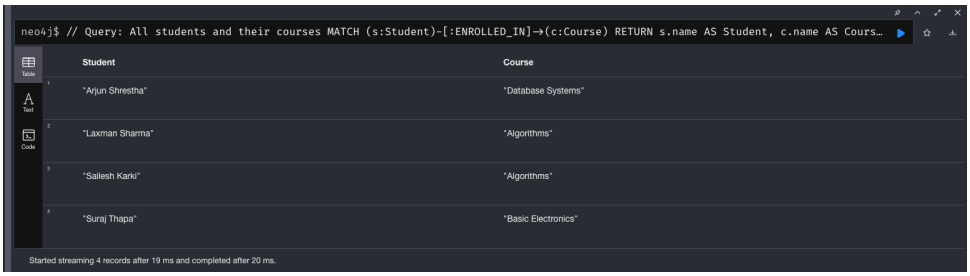

Figure B.26: Query Result: All students and their enrolled courses. This shows the reverse traversal of the ENROLLED_IN relationship, listing students and the courses they are taking.

Figure B.27: Query Result: Students enrolled in CS204 (Algorithms). This demonstrates property-based filtering at the course level, showing how to find all students enrolled in a specific course using course codes.

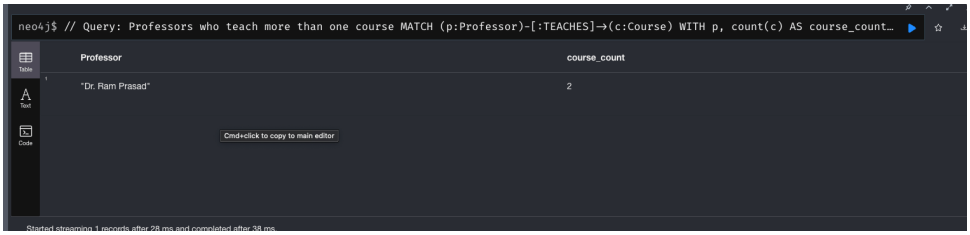

Figure B.28: Query Result: Professors teaching multiple courses. This screenshot shows an aggregation query that counts relationships per professor and filters results, demonstrating Neo4j's capability for analytical queries.

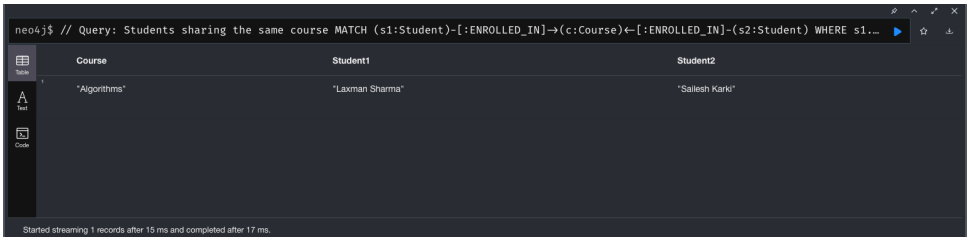

Figure B.29: Query Result: Students enrolled in the same course. This demonstrates a more complex pattern matching query that finds pairs of students connected through the same course, showcasing Neo4j's ability to discover indirect relationships.

B.4 Redis Key-Value Store Screenshots

The following screenshots document the implementation and operation of Redis for key-value store operations, session management with TTL, and visitor tracking using atomic increment operations.

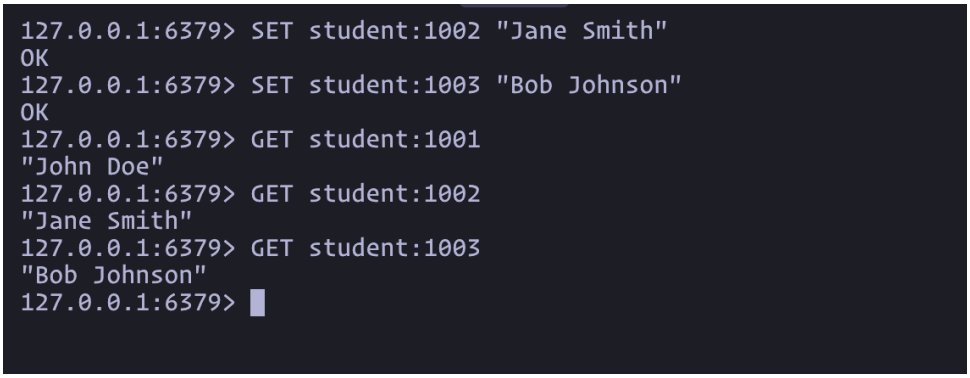

Figure B.30: Basic Redis String Operations: SET and GET commands demonstration. This shows simple key-value pair storage and retrieval operations, which form the foundation of Redis data manipulation.

```
127.0.0.1:6379> HSET student:profile:1002 name "Jane Smith" age 21 major "Information Technology" gpa 3.9
(integer) 0
127.0.0.1:6379> HSET student:profile:1001 name "John Doe" age 20 major "Computer Science" gpa 3.8
(integer) 0
127.0.0.1:6379> HSET student:profile:1003 name "Bob Johnson" age 19 major "Software Engineering" gpa 3.7
(integer) 0
127.0.0.1:6379> HGETALL student:profile:1001
1) "name"
2) "John Doe"
3) "age"
4) "20"
5) "major"
6) "Computer Science"
7) "gpa"
8) "3.8"
127.0.0.1:6379> HGETALL student:profile:1002
1) "name"
2) "Jane Smith"
3) "age"
4) "21"
5) "major"
6) "Information Technology"
7) "gpa"
8) "3.9"
127.0.0.1:6379> HGET student:profile:1001 name
"John Doe"
127.0.0.1:6379> HMGET student:profile:1002 name major gpa
1) "Jane Smith"
2) "Information Technology"
3) "3.9"
127.0.0.1:6379> HMSET student:profile:1004 name "Alice Brown" age 22 major "Data Science" gpa 3.95 year "Senior"
OK
127.0.0.1:6379> HKEYS student:profile:1001
1) "name"
2) "age"
3) "major"
4) "gpa"
127.0.0.1:6379> HVALS student:profile:1002
1) "Jane Smith"
2) "21"
3) "Information Technology"
4) "3.9"
127.0.0.1:6379> HEXISTS student:profile:1001 name
(integer) 1
127.0.0.1:6379> HEXISTS student:profile:1001 email
(integer) 0
127.0.0.1:6379> HINCRBY student:profile:1001 age 1
(integer) 21
127.0.0.1:6379> HGETALL student:profile:1001
1) "name"
2) "John Doe"
3) "age"
4) "21"
```

Figure B.31: Redis Hash Operations: HSET, HGETALL, HGET, and related hash commands. This demonstrates Redis's ability to store structured data using field-value pairs within a single key, ideal for object representation.

```
127.0.0.1:6379> SET session:user1001 "John Doe logged in" EX 30
OK
127.0.0.1:6379> SET session:user1002 "Jane Smith logged in" EX 30
OK
127.0.0.1:6379> SET session:user1003 "Bob Johnson logged in" EX 30
OK
127.0.0.1:6379> GET session:user1001
"John Doe logged in"
127.0.0.1:6379> GET session:user1002
"Jane Smith logged in"
127.0.0.1:6379> TTL session:user1001
(integer) 11
127.0.0.1:6379> TTL session:user1002
(integer) 11
127.0.0.1:6379> HSET session:detailed:user1001 user_id 1001 username "john_doe" login_time "2024-01-15 10:30:00" ip_address "192.168.1.100"
(integer) 4
127.0.0.1:6379> EXPIRE session:detailed:user1001 45
(integer) 1
127.0.0.1:6379> HSET session:detailed:user1002 user_id 1002 username "jane_smith" login_time "2024-01-15 10:35:00" ip_address "192.168.1.101"
(integer) 4
127.0.0.1:6379> EXPIRE session:detailed:user1002 45
(integer) 1
127.0.0.1:6379> HGETALL session:detailed:user1001
1) "user_id"
2) "1001"
3) "username"
4) "john_doe"
5) "login_time"
6) "2024-01-15 10:30:00"
7) "ip_address"
8) "192.168.1.100"
127.0.0.1:6379> TTL session:detailed:user1001
(integer) 32
127.0.0.1:6379> EXPIRE session:user1001 60
(integer) 0
127.0.0.1:6379> TTL session:user1001
(integer) -2
127.0.0.1:6379> HSET cart:session:user1001 item1 "Laptop" item2 "Mouse" item3 "Keyboard" total 1500
(integer) 4
127.0.0.1:6379> EXPIRE cart:session:user1001 300
(integer) 1
127.0.0.1:6379> HGETALL cart:session:user1001
1) "item1"
2) "Laptop"
3) "item2"
4) "Mouse"
5) "item3"
6) "Keyboard"
7) "total"
8) "1500"
127.0.0.1:6379> TTL cart:session:user1001
(integer) 293
127.0.0.1:6379> î
```

Figure B.32: Session Management with TTL: Creating user sessions with automatic expiration. This shows Redis's TTL functionality for managing temporary data like user sessions, shopping carts, and cached content.

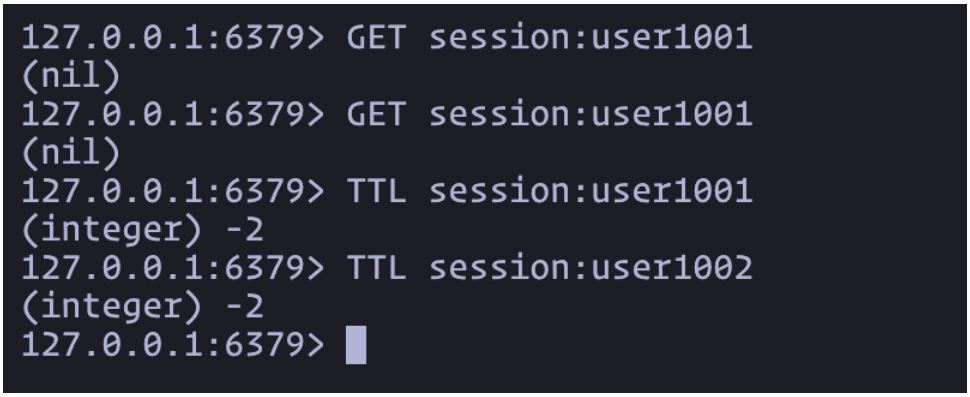

Figure B.33: Session Expiration Results: Demonstrating automatic key deletion after TTL expires. This shows how Redis automatically cleans up expired keys, returning nil for expired sessions and -2 for TTL checks.

```
127.0.0.1:6379> SET visitors:total 0
OK
127.0.0.1:6379> INCR visitors:total
(integer) 1
127.0.0.1:6379> INCR visitors:total
(integer) 2
127.0.0.1:6379> INCR visitors:total
(integer) 3
127.0.0.1:6379> INCR visitors:total
(integer) 4
127.0.0.1:6379> INCR visitors:total
(integer) 5
127.0.0.1:6379> INCR visitors:page:home
(integer) 1
127.0.0.1:6379> INCR visitors:page:about
(integer) 1
127.0.0.1:6379> INCR visitors:page:home
(integer) 2
127.0.0.1:6379> INCR visitors:page:products
(integer) 1
127.0.0.1:6379> GET visitors:page:home
"2"
127.0.0.1:6379> GET visitors:page:contact
(nil)
127.0.0.1:6379> GET visitors:page:products
"1"
127.0.0.1:6379> INCR visitors:daily:2024-01-15
(integer) 1
127.0.0.1:6379> INCR visitors:daily:2024-01-15
(integer) 2
127.0.0.1:6379> INCR visitors:daily:2024-01-15
(integer) 3
127.0.0.1:6379> GET visitors:daily:2024-01-15
"3"
127.0.0.1:6379> INCR visitors:hourly:2024-01-15:10
(integer) 1
127.0.0.1:6379> INCR visitors:hourly:2024-01-15:10
(integer) 2
127.0.0.1:6379> GET visitors:hourly:2024-01-15:10
"2"
127.0.0.1:6379> INCR user:1001:visits
(integer) 1
127.0.0.1:6379> GET user:1001:visits
"1"
127.0.0.1:6379> INCRBY visitors:total 10
(integer) 15
127.0.0.1:6379> GET visitors:total
"15"
127.0.0.1:6379> INCR browser:chrome
(integer) 1
127.0.0.1:6379> GET browser:chrome
"1"
127.0.0.1:6379> INCR visitors:country:USA
(integer) 1
127.0.0.1:6379> GET visitors:country:USA
"1"
127.0.0.1:6379> INCR visitors:device:desktop
(integer) 1
127.0.0.1:6379> INCR visitors:device:desktop
(integer) 2
127.0.0.1:6379> INCR visitors:device:mobile
(integer) 1
127.0.0.1:6379> GET visitors:device:desktop
"2"
127.0.0.1:6379>
```


Figure B.34: Visitor Tracking with INCR Operations: Atomic increment operations for analytics and counting. This demonstrates Redis's atomic counter capabilities for page

B.5 CockroachDB Distributed SQL Screenshots

The following screenshots document the implementation and operation of CockroachDB for distributed SQL operations, demonstrating ACID transactions, concurrent transfers, and banking operations.

Database Setup and Schema Creation

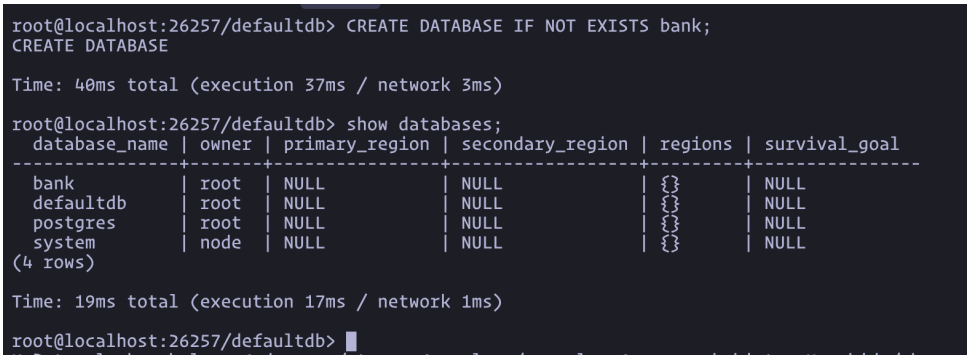

Figure B.35: Database Creation Process: Terminal output showing the successful creation of the `bank` database in CockroachDB. This demonstrates the initial setup phase where the database environment is established for the banking application.

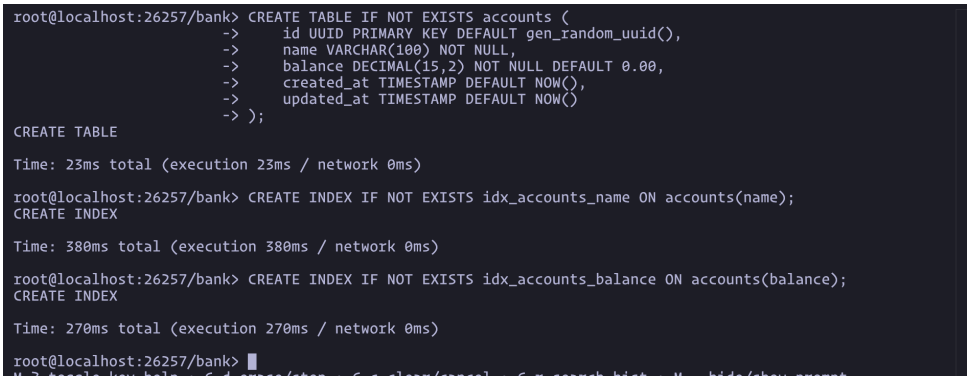

Figure B.36: Table Structure Creation: Screenshot displaying the creation of the `accounts` table with UUID primary key, proper data types, and indexes. This shows the schema design optimized for distributed transactions and ACID compliance.

```
root@localhost:26257/bank> SELECT
                        ->     column_name,
                        ->     data_type,
                        ->     is_nullable,
                        ->     column_default
                        -> FROM information_schema.columns
                        -> WHERE table_name = 'accounts'
                        -> ORDER BY ordinal_position;
  column_name |          data_type          | is_nullable |  column_default
--------------+-----------------------------+-------------+--------------------
  id          | uuid                        | NO          | gen_random_uuid()
  name        | character varying           | NO          | NULL
  balance     | numeric                     | NO          | 0.00
  created_at  | timestamp without time zone | YES         | now()
  updated_at  | timestamp without time zone | YES         | now()
(5 rows)

Time: 47ms total (execution 46ms / network 1ms)

root@localhost:26257/bank> SHOW TABLES;
  schema_name | table_name | type  | owner | estimated_row_count | locality
--------------+------------+-------+-------+---------------------+-----------
  public      | accounts   | table | root  |                   0 | NULL
(1 row)

Time: 39ms total (execution 39ms / network 1ms)

root@localhost:26257/bank>
```

Figure B.37: Table Structure Display: Detailed view of the accounts table schema showing column names, data types, nullability, and default values. This demonstrates the complete table structure created for the banking application.

Data Insertion and Verification

```
root@localhost:26257/bank> INSERT INTO accounts (name, balance) VALUES
                        ->     ('Laxman Shrestha', 150000.00),
                        ->     ('Sailesh Bhandari', 75000.00),
                        ->     ('Suraj Thapa', 120000.00),
                        ->     ('Arjun Karki', 95000.00),
                        ->     ('Pooja Pathak', 180000.00),
                        ->     ('Narendra Joshi', 65000.00),
                        ->     ('Kiran Sapkota', 110000.00),
                        ->     ('Ujjwal Panta', 85000.00),
                        ->     ('Ashish Basnyat', 140000.00),
                        ->     ('Dipesh Tamang', 70000.00)
                        -> ON CONFLICT DO NOTHING;
INSERT 0 10

Time: 34ms total (execution 33ms / network 1ms)

root@localhost:26257/bank>
```

Figure B.38: Account Data Insertion: Console output showing the successful insertion of 10 accounts with realistic balances. This demonstrates bulk data insertion with proper transaction handling in a distributed environment.

```
root@localhost:26257/bank> SELECT
                        ->     id,
                        ->     name,
                        ->     balance,
                        ->     created_at,
                        ->     updated_at
                        -> FROM accounts
                        -> ORDER BY name;
                  id                  |       name       |  balance  |         created_at         |         updated_at
--------------------------------------+------------------+-----------+----------------------------+----------------------------
  3ab6aae5-71d1-47fa-9dfb-e8575bf93be2 | Arjun Karki      |  95000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  bf891b12-88bd-4e48-b938-fd3dd133190e | Ashish Basnyat   | 140000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  4d4aba82-21db-4f26-8e93-603f2c08eed7 | Dipesh Tamang    |  70000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  f7578883-b084-4448-8f0c-3d9727e97bc9 | Kiran Sapkota    | 110000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  d63ba0b8-89e2-4c30-9cf4-9c4f4c62c7d | Laxman Shrestha  | 150000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  fed49e86-1f8c-4c7a-9e04-49578c5ba6e4 | Narendra Joshi   |  65000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  bb628dbe-b643-4242-a1ef-129ad9ed3d95 | Pooja Pathak     | 180000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  93d92279-5416-4ebf-b207-88ed3d55f9df | Sailesh Bhandari |  75000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  98297a81-e6a2-4cf9-a606-4b2606a97eb8 | Suraj Thapa      | 120000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
  4449d9f6-6266-4ab3-951d-f645e9ba99bd | Ujjwal Panta     |  85000.00 | 2025-07-19 13:20:16.714084 | 2025-07-19 13:20:16.714084
(10 rows)

Time: 2ms total (execution 2ms / network 1ms)
```

Figure B.39: Inserted Accounts Display: Complete view of all inserted accounts showing generated UUIDs, names, balances, and timestamps. This verifies the data insertion process and demonstrates the distributed primary key generation.

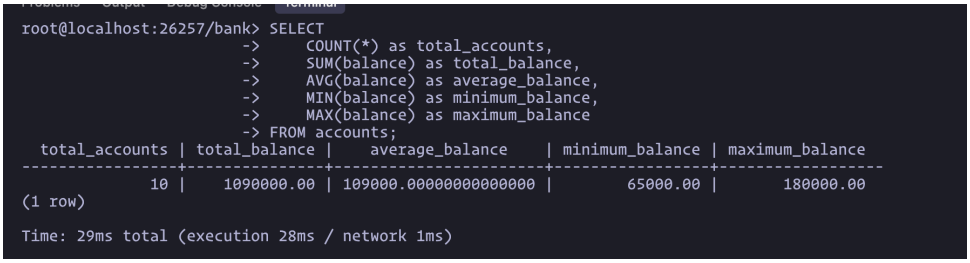

Figure B.40: Account Statistics After Insertion: Comprehensive statistics showing total accounts, total balance, average balance, and balance distribution after initial data insertion. This demonstrates the analytical capabilities of the distributed SQL database.

Transaction Operations

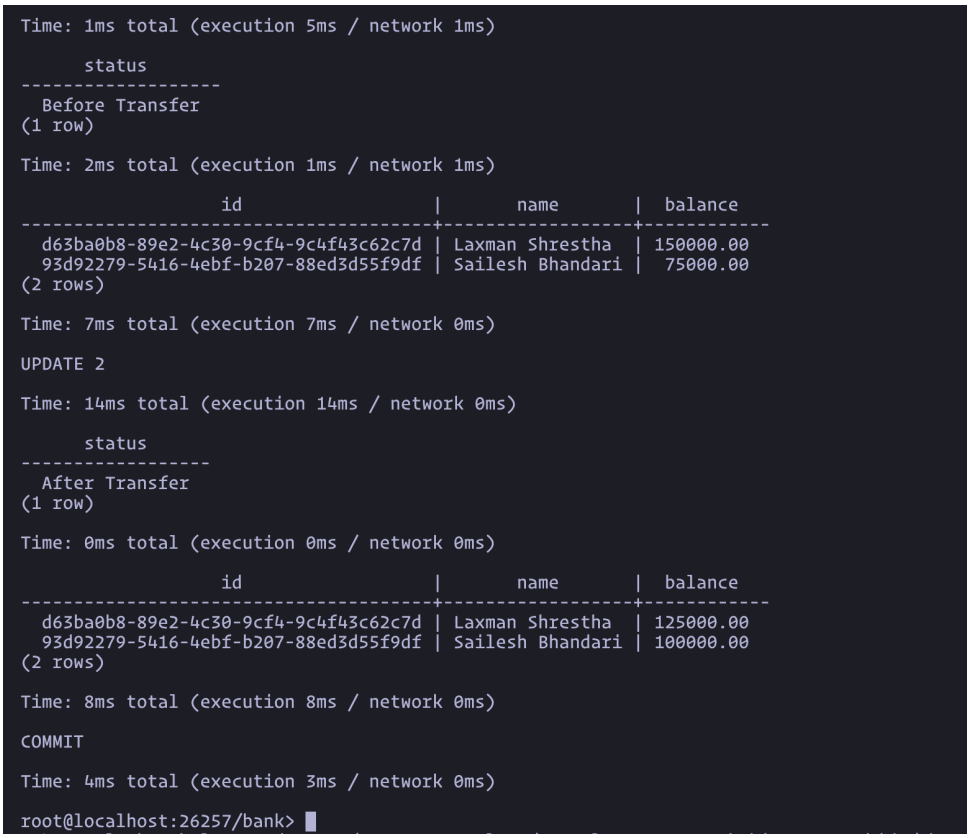

Figure B.41: Single Transfer Transaction: Demonstration of ACID-compliant transfer between two accounts. The screenshot shows the before and after states of the accounts involved in the transfer, proving atomicity and consistency of the transaction.

```
-----------------------------------------------
  Initial Balances Before Concurrent Transfers
(1 row)

Time: 0ms total (execution 0ms / network 0ms)

                  id                  |       name       |  balance
--------------------------------------+------------------+------------
  3ab6aae5-71d1-47fa-9dfb-e8575bf93be2 | Arjun Karki      |  95000.00
  bf891b12-88bd-4e48-b938-fd3dd133190e | Ashish Basnyat   | 140000.00
  4d4aba82-21db-4f26-8e93-603f2c08eed7 | Dipesh Tamang    |  70000.00
  f7578883-b084-4448-8f0c-3d9727e97bc9 | Kiran Sapkota    | 110000.00
  d63ba0b8-89e2-4c30-9cf4-9c4f43c62c7d | Laxman Shrestha  | 125000.00
  fed49e86-1f8c-4c7a-9e04-49578c3ba6e4 | Narendra Joshi   |  65000.00
  bb628dbe-b643-4242-a1ef-129ad9ed3d95 | Pooja Pathak     | 180000.00
  93d92279-5416-4ebf-b207-88ed3d55f9df | Sailesh Bhandari | 100000.00
  98297a81-e6a2-4cf9-a606-4b2606a97eb8 | Suraj Thapa      | 120000.00
  4449d9f6-6266-4ab3-951d-f645e9ba99bd | Ujjwal Panta     |  85000.00
(10 rows)
```

Figure B.42: Initial Balances Before Concurrent Transfers: Account balances before executing multiple concurrent transfer operations. This establishes the baseline state for demonstrating CockroachDB's concurrent transaction handling capabilities.

```
-----------------------------------------------
  Final Balances After Concurrent Transfers
(1 row)

Time: 0ms total (execution 0ms / network 0ms)

                  id                  |       name       |  balance
--------------------------------------+------------------+------------
  3ab6aae5-71d1-47fa-9dfb-e8575bf93be2 | Arjun Karki      | 110000.00
  bf891b12-88bd-4e48-b938-fd3dd133190e | Ashish Basnyat   | 122000.00
  4d4aba82-21db-4f26-8e93-603f2c08eed7 | Dipesh Tamang    |  88000.00
  f7578883-b084-4448-8f0c-3d9727e97bc9 | Kiran Sapkota    |  98000.00
  d63ba0b8-89e2-4c30-9cf4-9c4f43c62c7d | Laxman Shrestha  | 115000.00
  fed49e86-1f8c-4c7a-9e04-49578c3ba6e4 | Narendra Joshi   |  85000.00
  bb628dbe-b643-4242-a1ef-129ad9ed3d95 | Pooja Pathak     | 160000.00
  93d92279-5416-4ebf-b207-88ed3d55f9df | Sailesh Bhandari | 110000.00
  98297a81-e6a2-4cf9-a606-4b2606a97eb8 | Suraj Thapa      | 105000.00
  4449d9f6-6266-4ab3-951d-f645e9ba99bd | Ujjwal Panta     |  97000.00
(10 rows)

Time: 1ms total (execution 0ms / network 0ms)
```

Figure B.43: Final Balances After Concurrent Transfers: Account balances after all concurrent transfers complete successfully. This demonstrates CockroachDB's ability to handle multiple simultaneous transactions while maintaining ACID compliance and data consistency.

Analytics and Reporting

```
----------------------
  Account Statistics
(1 row)

Time: 7ms total (execution 7ms / network 1ms)

  total_accounts | total_balance |    average_balance    | minimum_balance | maximum_balance
-----------------+---------------+-----------------------+-----------------+-----------------
              10 |    1090000.00 | 109000.00000000000000 |        85000.00 |       160000.00
(1 row)

Time: 10ms total (execution 9ms / network 0ms)
```

Figure B.44: Account Statistics After Transactions: Comprehensive statistics showing total accounts, total balance, average balance, and balance distribution. This demonstrates the analytical capabilities of the distributed SQL database for financial reporting.

```
------------------------------
  Top 3 Accounts by Balance
(1 row)

Time: 0ms total (execution 0ms / network 0ms)

       name       |  balance
------------------+------------
  Pooja Pathak    | 160000.00
  Ashish Basnyat  | 122000.00
  Laxman Shrestha | 115000.00
(3 rows)

Time: 1ms total (execution 1ms / network 0ms)

             status
--------------------------------
  Bottom 3 Accounts by Balance
(1 row)

Time: 0ms total (execution 0ms / network 0ms)

       name      | balance
-----------------+-----------
  Narendra Joshi | 85000.00
  Dipesh Tamang  | 88000.00
  Ujjwal Panta   | 97000.00
(3 rows)
```

Figure B.45: Top and Bottom Accounts by Balance: Ranking analysis showing accounts with highest and lowest balances. This demonstrates advanced querying capabilities for financial analysis and customer segmentation in the banking application.

```
Time: 1ms total (execution 1ms / network 1ms)

       name       |  balance  |         updated_at
------------------+-----------+----------------------------
  Laxman Shrestha | 115000.00 | 2025-07-19 13:30:58.098249
  Sailesh Bhandari | 110000.00 | 2025-07-19 13:30:58.098249
  Dipesh Tamang    |  88000.00 | 2025-07-19 13:30:58.085706
  Ashish Basnyat   | 122000.00 | 2025-07-19 13:30:58.085706
  Ujjwal Panta     |  97000.00 | 2025-07-19 13:30:58.072423
(5 rows)

Time: 5ms total (execution 4ms / network 0ms)
```

Figure B.46: Recently Updated Accounts: Audit trail showing accounts that have been recently modified, including their current balances and update timestamps. This demonstrates the audit and monitoring capabilities essential for banking applications.

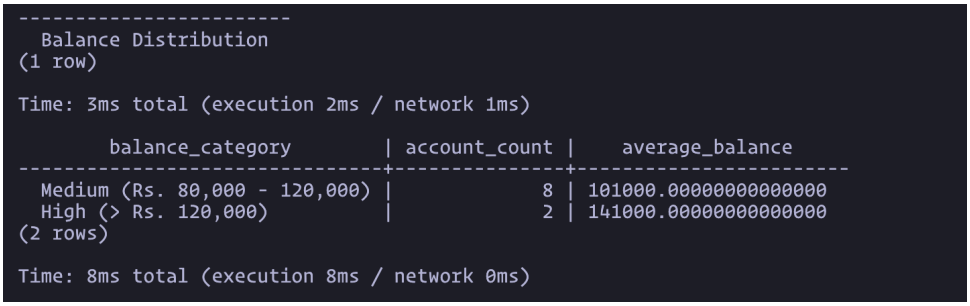

Figure B.47: Balance Distribution Analysis: Categorization of accounts by balance ranges (Low, Medium, High) with count and average balance for each category. This demonstrates advanced analytical capabilities for customer segmentation and financial reporting.