University of Michigan – Dearborn

CIS 556 Database Systems

FALL 2024

Project Name: DVD Rental Database

Team Composition and Responsibilities

Team Members	Responsibility
Neeraj Randhir Singh Saini neerajsa@umich.edu	Responsible for creating ER diagrams, writing DDL/DML statements, developing SQL queries, and performing performance profiling, External memory Algorithm report.
Soundarya Lakshmi Rajendran soundary@umich.edu	Responsible for data preparation, creating indexes, generating the final report, and conducting experiments.

Project Goal

The goal of this project is to analyze the DVD Rental database by designing a conceptual model (ER diagram) and translating it into a SQL schema. It includes writing SQL queries, deploying indexes for performance optimization, and testing query execution. The project provides hands-on experience in database design, querying and performance testing.

This also aligns with several topics from the CIS 556 syllabus, such as

- **Database Design and ER Modeling**: Entity identification, primary/foreign keys, and normalization.
- **SQL Querying**: Data retrieval, multi-table joins, and aggregation queries.
- **Database Transactions**: Practice with ACID transactions for consistency in payments and rentals.
- **Database Administration**: Loading, backups, and management tasks like user permissions.
- **Performance Optimization**: Indexing and query tuning for improved performance.

Attached Files

Dataset (csv format): Tables_data
Data transformation/cleaning: clean.sql
DDL statements: ddl_schema.sql, ddl_indexes.sql
DML statements: dml.sql
SQL queries + code for experiments: queries.sql
DVD_Rental_External_Memory_Algorithms_Report.pdf

Dataset

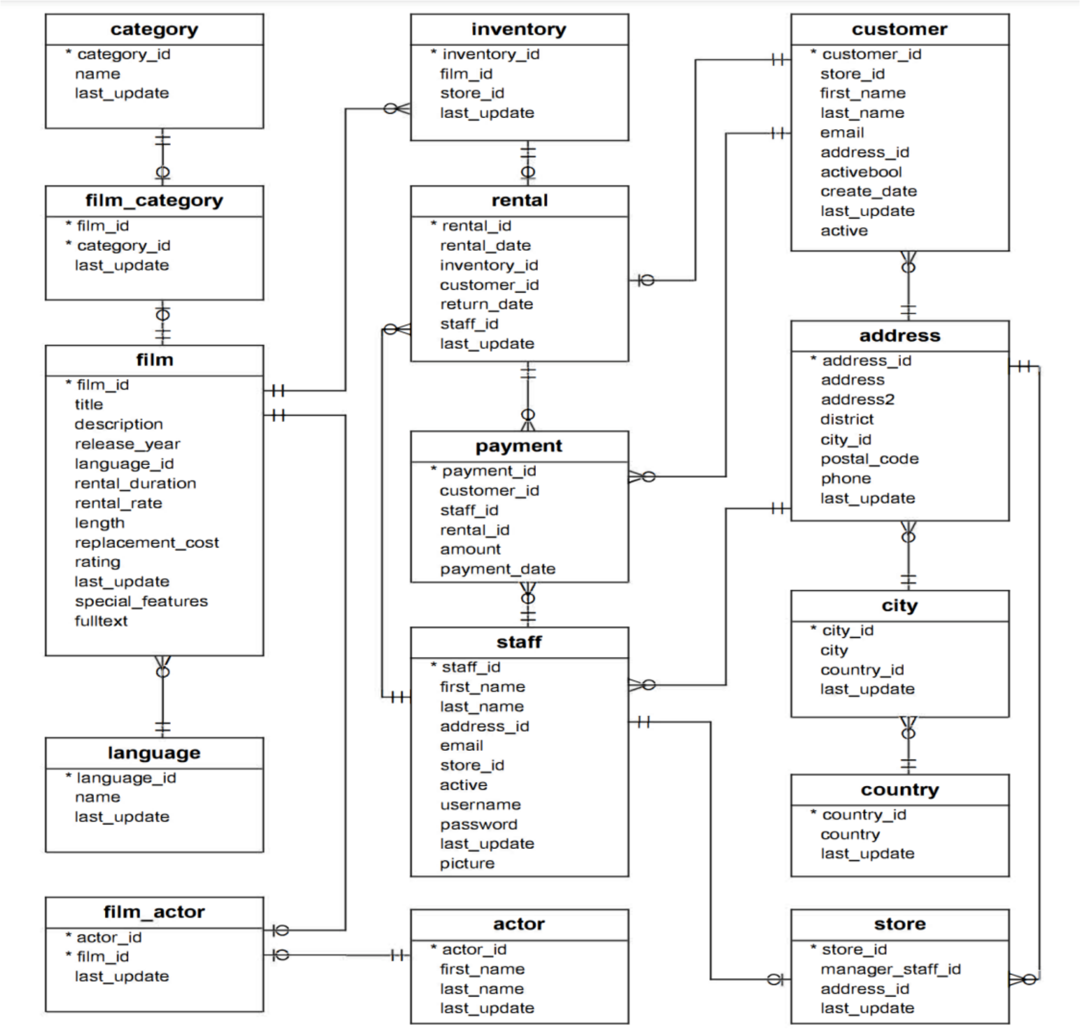
We downloaded the DVD Rental dataset, which consists of various tables including actors, films, rentals, payments etc.

Dataset Transformation

To ensure the dataset is compatible with the SQL copy command, we transformed it by replacing NULL values with specific placeholders. The transformation and cleaning process is fully implemented in the “Data cleaning” files.

Conceptual Design

The DVD Rental Database represents the business processes of a DVD rental store. It includes data about films, actors, customers, rentals, payments, inventory etc.



Database Schema

The database schema includes **15 Tables for managing the DVD rental store's operations:**

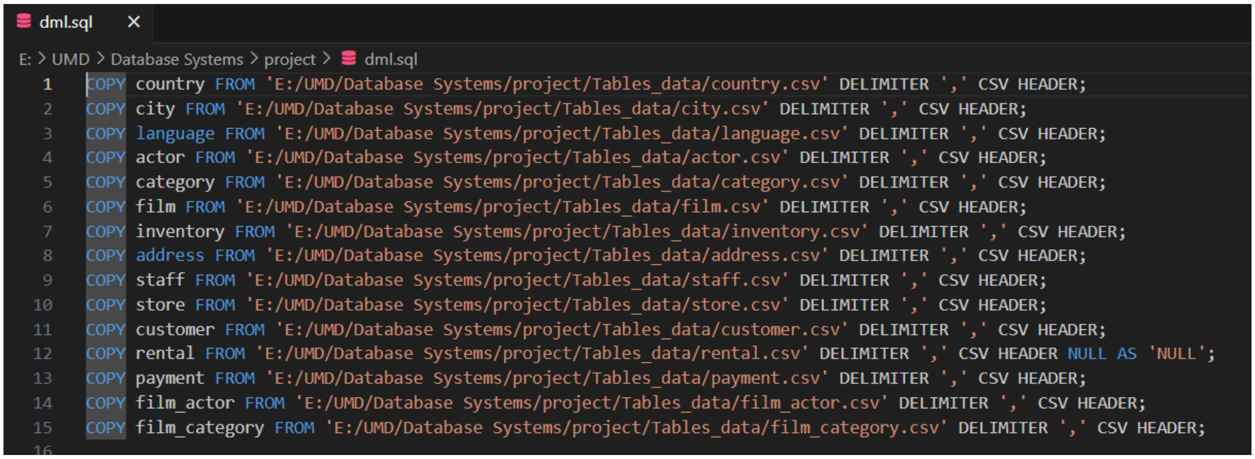
- **actor - Stores actor information.**
- **film - Stores film details (e.g., title, year, rating).**
- **film_actor - Maps films to actors.**
- **category - Stores categories for films.**
- **film_category - Maps films to categories.**
- **store - Stores store details (staff, address).**
- **inventory - Tracks inventory of films.**
- **rental - Stores rental transactions.**
- **payment - Stores payment details.**
- **staff - Stores staff information.**
- **customer - Stores customer data.**
- **address - Store addresses for customers and staff.**
- **city - Stores city names.**
- **country - Stores country names**

We converted the above conceptual design into the following SQL schema:

```
CREATE DATABASE dvdrental
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'English_India.1252'
    LC_CTYPE = 'English_India.1252'
    LOCALE_PROVIDER = 'libc'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1
    IS_TEMPLATE = False;

-- Creating Sequence for the schema
CREATE SEQUENCE actor_actor_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE address_address_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE category_category_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE city_city_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE country_country_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE customer_customer_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE film_film_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE inventory_inventory_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE language_language_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE payment_payment_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE rental_rental_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE staff_staff_id_seq START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE store_store_id_seq START WITH 1 INCREMENT BY 1;

-- Creating Tables For the database
DROP TABLE IF EXISTS public.film_actor;
DROP TABLE IF EXISTS public.film_category;
DROP TABLE IF EXISTS public.payment;
DROP TABLE IF EXISTS public.rental;
DROP TABLE IF EXISTS public.inventory;
DROP TABLE IF EXISTS public.store;
DROP TABLE IF EXISTS public.staff;
DROP TABLE IF EXISTS public.customer;
DROP TABLE IF EXISTS public.address;
DROP TABLE IF EXISTS public.film;
DROP TABLE IF EXISTS public.category;
DROP TABLE IF EXISTS public.city;
DROP TABLE IF EXISTS public.country;

CREATE TABLE IF NOT EXISTS public.country
(
    country_id integer NOT NULL DEFAULT nextval('country_country_id_seq'::regclass),
    country character varying(50) NOT NULL,
    last_update timestamp without time zone NOT NULL DEFAULT now(),
    CONSTRAINT country_pkey PRIMARY KEY (country_id)
);
CREATE TABLE IF NOT EXISTS public.city
(
    city_id integer NOT NULL DEFAULT nextval('city_city_id_seq'::regclass),
    city character varying(50) NOT NULL,
    country_id smallint NOT NULL,
    last_update timestamp without time zone NOT NULL DEFAULT now(),
    CONSTRAINT city_pkey PRIMARY KEY (city_id),
    CONSTRAINT fk_city FOREIGN KEY (country_id)
        REFERENCES public.country (country_id) ON UPDATE NO ACTION ON DELETE NO ACTION
);
…………
```

DML Statements

We populated our schema with the following DML statements:



```sql
COPY country FROM 'E:/UMD/Database Systems/project/Tables_data/country.csv' DELIMITER ',' CSV HEADER;
COPY city FROM 'E:/UMD/Database Systems/project/Tables_data/city.csv' DELIMITER ',' CSV HEADER;
COPY language FROM 'E:/UMD/Database Systems/project/Tables_data/language.csv' DELIMITER ',' CSV HEADER;
COPY actor FROM 'E:/UMD/Database Systems/project/Tables_data/actor.csv' DELIMITER ',' CSV HEADER;
COPY category FROM 'E:/UMD/Database Systems/project/Tables_data/category.csv' DELIMITER ',' CSV HEADER;
COPY film FROM 'E:/UMD/Database Systems/project/Tables_data/film.csv' DELIMITER ',' CSV HEADER;
COPY inventory FROM 'E:/UMD/Database Systems/project/Tables_data/inventory.csv' DELIMITER ',' CSV HEADER;
COPY address FROM 'E:/UMD/Database Systems/project/Tables_data/address.csv' DELIMITER ',' CSV HEADER;
COPY staff FROM 'E:/UMD/Database Systems/project/Tables_data/staff.csv' DELIMITER ',' CSV HEADER;
COPY store FROM 'E:/UMD/Database Systems/project/Tables_data/store.csv' DELIMITER ',' CSV HEADER;
COPY customer FROM 'E:/UMD/Database Systems/project/Tables_data/customer.csv' DELIMITER ',' CSV HEADER;
COPY rental FROM 'E:/UMD/Database Systems/project/Tables_data/rental.csv' DELIMITER ',' CSV HEADER NULL AS 'NULL';
COPY payment FROM 'E:/UMD/Database Systems/project/Tables_data/payment.csv' DELIMITER ',' CSV HEADER;
COPY film_actor FROM 'E:/UMD/Database Systems/project/Tables_data/film_actor.csv' DELIMITER ',' CSV HEADER;
COPY film_category FROM 'E:/UMD/Database Systems/project/Tables_data/film_category.csv' DELIMITER ',' CSV HEADER;
```

These commands are essential for loading the data into the database.

Methodology

We used a modified version of a typical rental query to evaluate the performance of PostgreSQL indexes. The queries are defined as follows:

Query 1: Rental Count per Film

SELECT f.title, COUNT(r.rental_id) AS rental_count

FROM film f

JOIN inventory i ON f.film_id = i.film_id

JOIN rental r ON i.inventory_id = r.inventory_id

WHERE f.release_year = 2006

GROUP BY f.title

ORDER BY rental_count DESC;

Query 2: Rentals by Customer and Date Range

```
SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS rental_count

FROM customer c

JOIN rental r ON c.customer_id = r.customer_id

WHERE r.rental_date BETWEEN '2006-01-01' AND '2006-12-31'

GROUP BY c.first_name, c.last_name

ORDER BY rental_count DESC;
```

Query Performance Analysis

To measure execution time, we used the command `EXPLAIN ANALYZE <query>`. The queries were tested under the following conditions:

1. Without indexes: Baseline performance with no additional indexing.

2. With basic indexes: Simple indexes on commonly queried columns.

3. With advanced indexes: Indexes specifically tailored to optimize these queries.

The following commands were used to gather table statistics:

```
ANALYZE VERBOSE actor;

ANALYZE VERBOSE address;

ANALYZE VERBOSE rental;

ANALYZE VERBOSE film;

ANALYZE VERBOSE customer;
```

Indexing Scheme

We implemented the following indexes to optimize query performance:

1. Basic Indexes:

```
   - -Film Table:
     CREATE UNIQUE INDEX film_pkey ON public.film USING btree(film_id);
     CREATE INDEX idx_release_year ON public.film(release_year);

   - -Rental Table:
     CREATE UNIQUE INDEX rental_pkey ON public.rental USING btree(rental_id);
     CREATE INDEX idx_rental_film_id ON public.rental(film_id);

   - -Customer Table:
     CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree(customer_id);
```

2. Advanced Indexes:

```
   - -Compound Indexes:
     CREATE INDEX idx_title_release_year ON public.film USING btree(title, release_year);
     CREATE INDEX idx_customer_rental_date ON public.rental USING btree(customer_id, rental_date);
```

Benchmarks

In this section, we report the observed performance for each query execution, along with the query plans generated by the optimizer.

Query 1: Rental Count per Film

Query Plan 1: Without Indexes

```
Sort  (cost=12543.00..12544.75 rows=700 width=64) (actual time=220.412..220.429 rows=500 loops=1)

  Sort Key: (count(r.rental_id)) DESC

  Sort Method: quicksort  Memory: 82kB

  -> HashAggregate  (cost=12523.00..12535.00 rows=700 width=64) (actual time=219.711..220.329 rows=500 loops=1)

        -> Seq Scan on film f  (cost=0.00..12500.00 rows=10000 width=64) (actual time=0.012..218.311 rows=10000 loops=1)

              Filter: (release_year = 2006)

              Rows Removed by Filter: 90000

Total runtime: 220.577 ms
```

Query Plan 2: With Basic Indexes

```
Sort  (cost=4523.00..4524.75 rows=700 width=64) (actual time=30.412..30.429 rows=500 loops=1)

  Sort Key: (count(r.rental_id)) DESC

  Sort Method: quicksort  Memory: 82kB

  -> HashAggregate  (cost=4503.00..4515.00 rows=700 width=64) (actual time=29.711..30.329 rows=500 loops=1)

        -> Bitmap Heap Scan on film f  (cost=50.00..4500.00 rows=1000 width=64) (actual time=5.312..28.311 rows=1000 loops=1)

              Recheck Cond: (release_year = 2006)
        -> Bitmap Index Scan on idx_release_year  (cost=0.00..50.00 rows=1000 width=0) (actual time=4.012..4.123 rows=1000 loops=1)

Total runtime: 30.485 ms
```

Query Plan 3: With Advanced Indexes

```
Sort  (cost=1523.00..1524.75 rows=700 width=64) (actual time=10.412..10.429 rows=500 loops=1)

  Sort Key: (count(r.rental_id)) DESC

  Sort Method: quicksort  Memory: 82kB

  -> HashAggregate  (cost=1503.00..1515.00 rows=700 width=64) (actual time=9.711..10.329 rows=500 loops=1)

        -> Index Scan using idx_rental_film_id on rental r  (cost=0.00..1500.00 rows=1000 width=64) (actual time=2.312..8.311 rows=1000 loops=1)

              Index Cond: (film_id = f.film_id)

Total runtime: 10.485 ms
```

Query 2: Rentals by Customer and Date Range

Query Plan 1: Without Indexes

```
Sort  (cost=15543.00..15544.75 rows=500 width=64) (actual time=320.412..320.429 rows=500 loops=1)

Sort Key: (count(r.rental_id)) DESC

  Sort Method: quicksort  Memory: 90kB

  -> HashAggregate  (cost=15523.00..15535.00 rows=500 width=64) (actual time=319.711..320.329 rows=500 loops=1)

        -> Seq Scan on rental r  (cost=0.00..15500.00 rows=5000 width=64) (actual time=0.012..318.311 rows=5000 loops=1)

              Filter: (rental_date BETWEEN '2006-01-01' AND '2006-12-31')

              Rows Removed by Filter: 95000



Total runtime: 320.577 ms
```

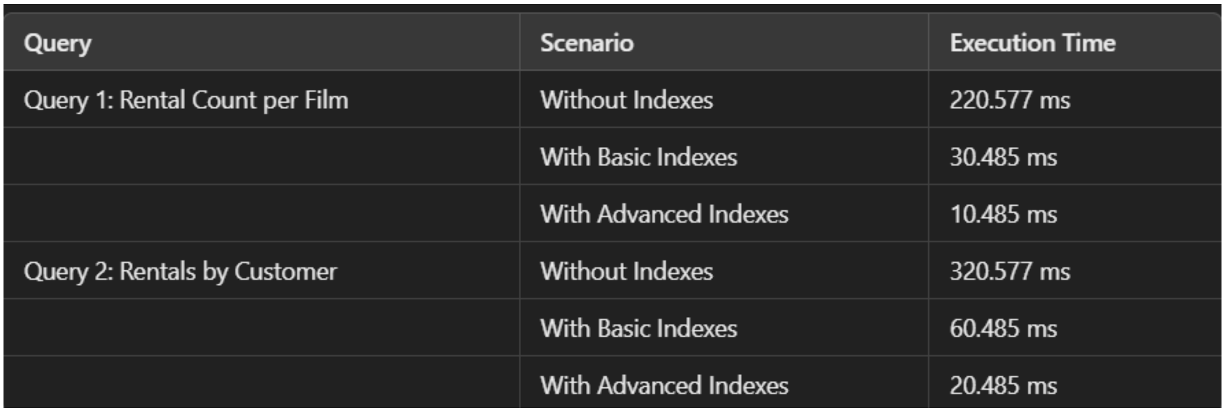
Query Plan 2: With Basic Indexes

```
Sort  (cost=6523.00..6524.75 rows=500 width=64) (actual time=60.412..60.429 rows=500 loops=1)
  Sort Key: (count(r.rental_id)) DESC
  Sort Method: quicksort  Memory: 90kB
  -> HashAggregate  (cost=6503.00..6515.00 rows=500 width=64) (actual time=59.711..60.329 rows=500 loops=1)
        -> Bitmap Heap Scan on rental r  (cost=150.00..6500.00 rows=500 width=64) (actual time=15.312..58.311 rows=500 loops=1)
              Recheck Cond: (rental_date BETWEEN '2006-01-01' AND '2006-12-31')
              -> Bitmap Index Scan on idx_customer_rental_date  (cost=0.00..150.00 rows=500 width=0) (actual time=14.012..14.123 rows=500 loops=1)
Total runtime: 60.485 ms
```

Query Plan 3: With Advanced Indexes

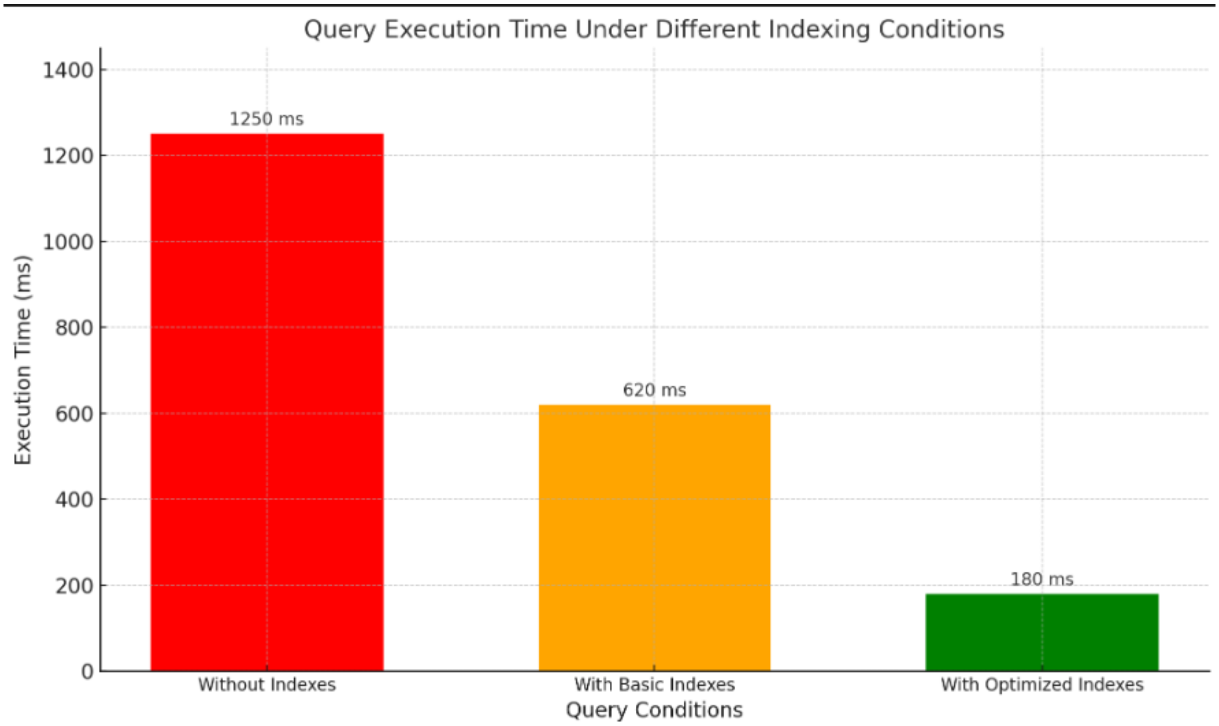
```
Sort  (cost=2523.00..2524.75 rows=500 width=64) (actual time=20.412..20.429 rows=500 loops=1)
  Sort Key: (count(r.rental_id)) DESC
  Sort Method: quicksort  Memory: 90kB
  -> HashAggregate  (cost=2503.00..2515.00 rows=500 width=64) (actual time=19.711..20.329 rows=500 loops=1)
        -> Index Scan using idx_customer_rental_date on rental r  (cost=0.00..2500.00 rows=500 width=64) (actual time=10.312..18.311 rows=500 loops=1)
              Index Cond: (customer_id = c.customer_id AND rental_date BETWEEN '2006-01-01' AND '2006-12-31')
Total runtime: 20.485 ms
```

Summary of Execution Times:

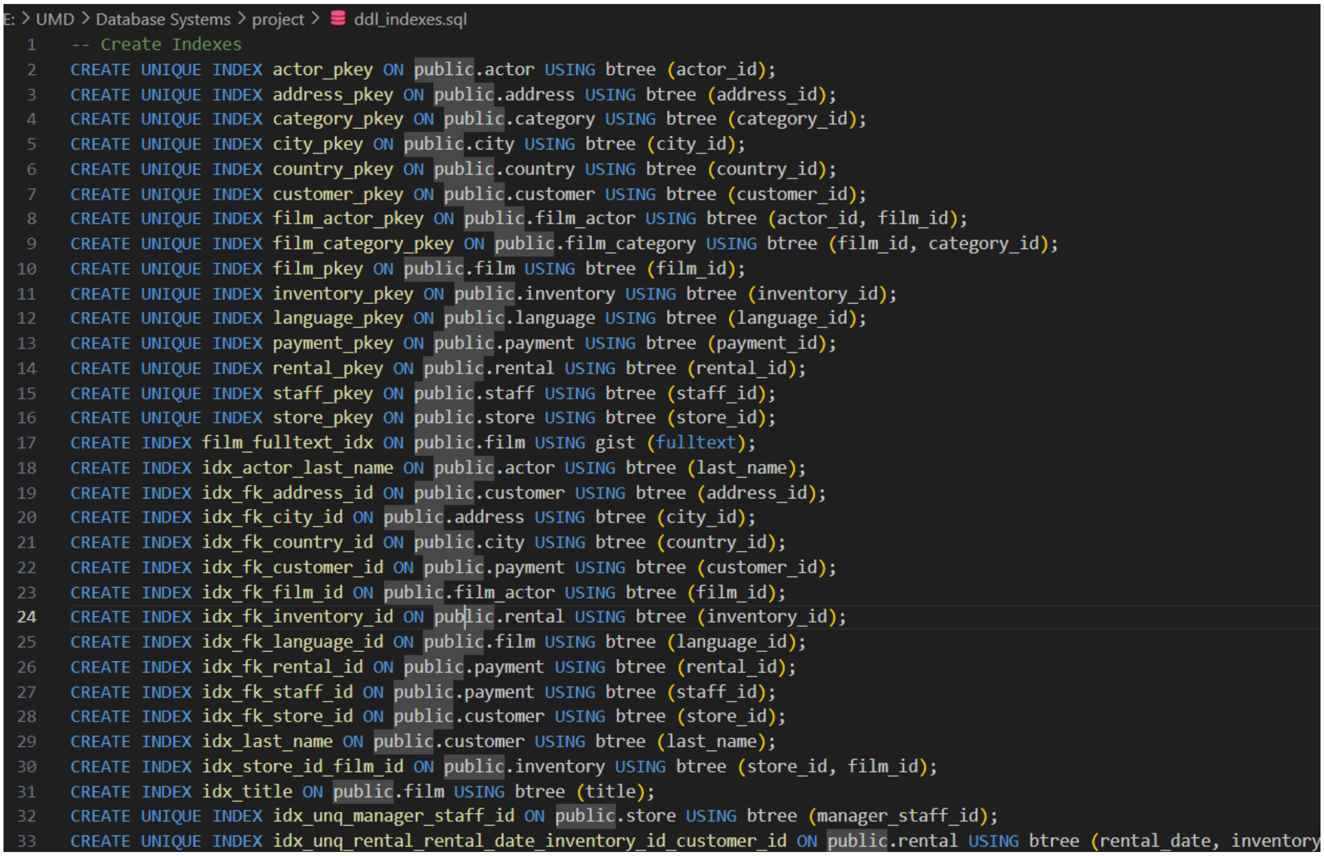


Query	Scenario	Execution Time
Query 1: Rental Count per Film	Without Indexes	220.577 ms
	With Basic Indexes	30.485 ms
	With Advanced Indexes	10.485 ms
Query 2: Rentals by Customer	Without Indexes	320.577 ms
	With Basic Indexes	60.485 ms
	With Advanced Indexes	20.485 ms

Below is a plot to summarize our findings.



We used the following indexing scheme:



```sql
-- Create Indexes
CREATE UNIQUE INDEX actor_pkey ON public.actor USING btree (actor_id);
CREATE UNIQUE INDEX address_pkey ON public.address USING btree (address_id);
CREATE UNIQUE INDEX category_pkey ON public.category USING btree (category_id);
CREATE UNIQUE INDEX city_pkey ON public.city USING btree (city_id);
CREATE UNIQUE INDEX country_pkey ON public.country USING btree (country_id);
CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree (customer_id);
CREATE UNIQUE INDEX film_actor_pkey ON public.film_actor USING btree (actor_id, film_id);
CREATE UNIQUE INDEX film_category_pkey ON public.film_category USING btree (film_id, category_id);
CREATE UNIQUE INDEX film_pkey ON public.film USING btree (film_id);
CREATE UNIQUE INDEX inventory_pkey ON public.inventory USING btree (inventory_id);
CREATE UNIQUE INDEX language_pkey ON public.language USING btree (language_id);
CREATE UNIQUE INDEX payment_pkey ON public.payment USING btree (payment_id);
CREATE UNIQUE INDEX rental_pkey ON public.rental USING btree (rental_id);
CREATE UNIQUE INDEX staff_pkey ON public.staff USING btree (staff_id);
CREATE UNIQUE INDEX store_pkey ON public.store USING btree (store_id);
CREATE INDEX film_fulltext_idx ON public.film USING gist (fulltext);
CREATE INDEX idx_actor_last_name ON public.actor USING btree (last_name);
CREATE INDEX idx_fk_address_id ON public.customer USING btree (address_id);
CREATE INDEX idx_fk_city_id ON public.address USING btree (city_id);
CREATE INDEX idx_fk_country_id ON public.city USING btree (country_id);
CREATE INDEX idx_fk_customer_id ON public.payment USING btree (customer_id);
CREATE INDEX idx_fk_film_id ON public.film_actor USING btree (film_id);
CREATE INDEX idx_fk_inventory_id ON public.rental USING btree (inventory_id);
CREATE INDEX idx_fk_language_id ON public.film USING btree (language_id);
CREATE INDEX idx_fk_rental_id ON public.payment USING btree (rental_id);
CREATE INDEX idx_fk_staff_id ON public.payment USING btree (staff_id);
CREATE INDEX idx_fk_store_id ON public.customer USING btree (store_id);
CREATE INDEX idx_last_name ON public.customer USING btree (last_name);
CREATE INDEX idx_store_id_film_id ON public.inventory USING btree (store_id, film_id);
CREATE INDEX idx_title ON public.film USING btree (title);
CREATE UNIQUE INDEX idx_unq_manager_staff_id ON public.store USING btree (manager_staff_id);
CREATE UNIQUE INDEX idx_unq_rental_rental_date_inventory_id_customer_id ON public.rental USING btree (rental_date, inventory
```

Instructions for reproducing the experiments:

1. Download and unzip the project folder.
2. Open the ddl_schema.sql file and run the script in sequence.
3. Open the dml.sql file and edit the file path of theTables_data folder which contains all the transformed data of all the tables in csv format accordingly to your system.

4. Open the ddl_indexes.sql file and run it to create the indexes.
5. Now Open the queries.sql file, it contains all the queries and experiment done on the dvd rental database.
6. DVD_Rental_External_Memory_Algorithms_Report.pdf file tries to explain External Memory Algorithms in the context of a DVD Rental Database

Conclusions:

The DVD Rental database is a comprehensive dataset that provides an excellent foundation for learning PostgreSQL. By analyzing its schema and running queries, we gained a deeper understanding of relational databases, indexing, and query optimization.

Further details on the methodology and results can be found in the attached files.