Multi-Paradigm Genealogical Data Modeling: OLTP, OLAP, and Cross-Model Translation from Relational to NoSQL Databases

Holly Do

Rice University

August 08, 2025

Table of Contents

Our Abstract 2
Our Mission 3
Our Objective 3
Entity-Relationship Diagram 4
Relational Model and BCNF Verification 4
Prototype – SQL OLTP Database Design 5 to 7
Prototype – OLAP Extensions 8

- Aggregated Queries 9 to 15
- Optimization and Query Tuning 15 to 26

Prototype – Translation to MongoDB 26

- Document Schema Design 27
- CRUD and Aggregation Queries 27 to 29

Prototype – Translation to Neo4j 29

- Node & Relationship Model 29 to 32
- Cypher Queries 32

Conclusion 33
References 33

Our Abstract

This project focuses on designing and implementing a genealogical database system to manage detailed records of individuals and their familial relationships. The goal is to store, query, and analyze historical and contemporary genealogical data using multiple database paradigms — starting with a normalized relational schema (SQL OLTP), extending to analytical querying (OLAP), and finally translating the schema to NoSQL (MongoDB) and graph-based modeling (Neo4j).

The relational model ensures data integrity and supports traditional ACID transactions for precise updates to genealogical events such as births, deaths, marriages, adoptions, and name changes. The NoSQL model allows for scaling and flexible storage of nested and varying attributes, while the graph model provides intuitive relationship traversal for family tree exploration.

This project also demonstrates cross-paradigm design principles, ensuring that data remains consistent and query-able across SQL, document, and graph environments.

Our Mission

To create a reliable, accurate, and flexible genealogical database system capable of storing decades or centuries of family data.

To enable researchers, historians, and individuals to track family histories, detect lineage patterns, and discover relationships.

To design the database in a way that supports both OLTP (frequent, accurate updates) and OLAP (historical data analysis and reporting).

To translate the model into NoSQL and graph structures for performance in specialized genealogical queries such as ancestor/descendant searches.

Our Objective

Data Maintenance: Enter, update, and delete data on individuals, names, gender identity changes, parentage, and marriages.

Search Capabilities: Query people by name, date, place, or relationship type.

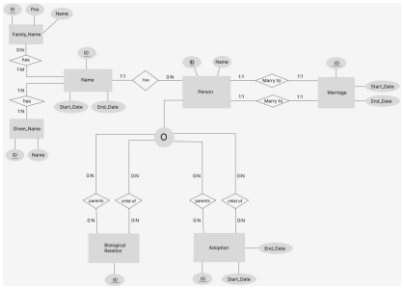
Relationship Tracking: Maintain records for biological, adoptive, and marital connections, record changes over time, such as multiple marriages, name changes, and gender identity changes.

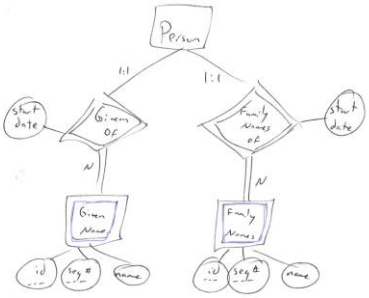
Report Generation: Generate ancestor and descendant reports. Identify common ancestors between two people. Count descendants by generation. Analyze demographic trends over time (OLAP).

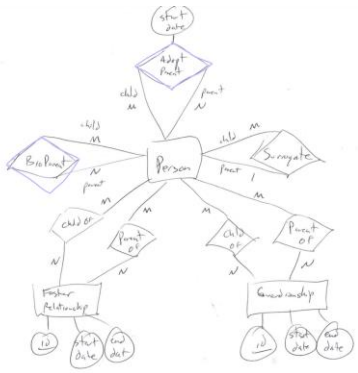
Cross-Database Implementation: Design for relational, document, and graph databases, analyzing tradeoffs between ACID and BASE properties, vertical scaling versus scalable distributed database system.

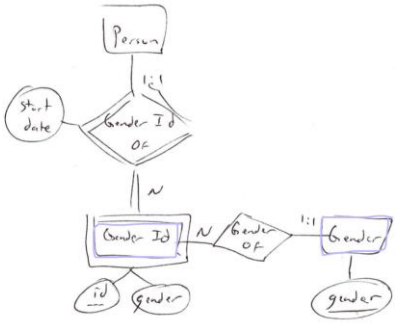
OLTP Database Design

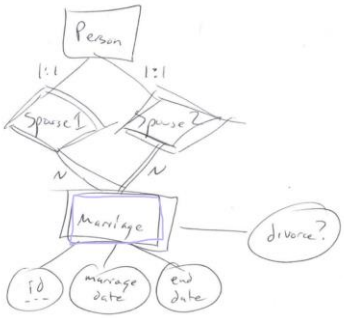
Entity-Relationship Diagram

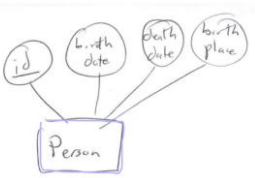












Relational Model and BCNF Verification

In this OLTP schema, every row has a unique way to identify it (primary key).

By definition:

Primary Key→All Other Attributes

So in BCNF terms, all FDs are key-based → you're already in BCNF.

Schema

Person(id, birth_date, death_date, birth_place, surrogate_parent), surrogate_parent is FK referencing Person.id

GivenName(person_id, seq_num, start_date, name), person_id is FK referencing Person.id

FamilyName(person_id, seq_num, start_date, name), person_id is FK referencing Person.id

GenderIdentity(person_id, start_date, gender), person_id is FK referencing Person.id, gender is FK referencing Gender.gender

Gender(gender)

Marriage(spouse1, spouse2, id, marriage_date, end_date, divorce), spouse1 and spouse2 are FKs referencing Person.id

AdoptParent(child, parent, start_date), child and parent are FKs referencing Person.id

BioParent(child, parent), child and parent are FKs referencing Person.id

FosterRelationship(id, start_date, end_date)

FosterChildOf(foster_id, child_id), foster_id is FK referencing FosterRelationship.id, child_id is FK referencing Person.id

FosterParentOf(foster_id, parent_id), foster_id is FK referencing FosterRelationship.id, parent_id is FK referencing Person.id

Guardianship(id, start_date, end_date)

GuardianChildOf(guardian_id, child_id), guardian_id is FK referencing Guardianship.id, child_id is FK referencing Person.id

GuardianParentOf(guardian_id, parent_id), guardian_id is FK referencing Guardianship.id, parent_id is FK referencing Person.id

Prototype – SQL OLTP Database Design

```
CREATE TABLE Person (
id INT,
birth_date DATE,
death_date DATE,
birth_place TEXT,
PRIMARY KEY id
);

CREATE TABLE GivenName (
person_id INT, -- Whose given name is this
gn_name_id INT, -- Sequence number in how many times this
person has changed their given name
seq_num INT, -- Sequence number in parts of person's
current given name, e.g., first name vs. middle name
start_date DATE, -- Date starting to use this name
name TEXT,
PRIMARY KEY (person_id, gn_name_id, seq_num),
FOREIGN KEY (person_id) REFERENCES Person(id)
);

CREATE TABLE FamilyName (
person_id INT, -- Whose given name is this
fn_name_id INT, -- Sequence number in how many times this
person has changed their given name
seq_num INT, -- Sequence number in parts of person's
current family name
start_date DATE, -- Date starting to use this name
name TEXT,
PRIMARY KEY (person_id, fn_name_id, seq_num),
FOREIGN KEY (person_id) REFERENCES Person(id)
);

CREATE TABLE Gender (
gender TEXT,
PRIMARY KEY (gender)
);

CREATE TABLE GenderIdentity (
person_id INT, -- Whose gender identity
```

```
gender_id INT, -- Sequence number in changes
start_date DATE, -- Date starting to use this gender
gender TEXT,
PRIMARY KEY (person_id, gender_id),
FOREIGN KEY (person_id) REFERENCES Person(id),
FOREIGN KEY (gender) REFERENCES Gender(gender)
);


CREATE TABLE Marriage (
spouse1 INT, -- Either spouse (i.e., not necessarily
father)
spouse2 INT, -- The other one
id INT, -- Sequence number of which marriage this
is for this pair of people
marriage_date DATE, -- Date starting for this marriage

end_date DATE, -- Date ending for this marriage (divorce
or death)
divorce BOOLEAN, -- end in divorce (true) or death (false)
PRIMARY KEY (spouse1, spouse2, id),
FOREIGN KEY (spouse1) REFERENCES Person(id),
FOREIGN KEY (spouse2) REFERENCES Person(id)
);


CREATE TABLE AdoptParent (
child INT,
parent INT,
start_date DATE,
PRIMARY KEY (child, parent),
FOREIGN KEY (child) REFERENCES Person(id),
FOREIGN KEY (parent) REFERENCES Person(id)
);


CREATE TABLE BioParent (
child INT,
parent INT,
PRIMARY KEY (child, parent),
FOREIGN KEY (child) REFERENCES Person(id),
FOREIGN KEY (parent) REFERENCES Person(id)
);
```

Prototype – OLAP Extensions

Fact Table and Dimensions:

For analytical purposes, the genealogical OLTP schema is transformed into an OLAP star schema. The star schema separates **factual, measurable computes** over each **descriptive dimensions** to enable historical analysis over large genealogical datasets.

Fact Table:

The fact table stores quantitative measurements (facts) for genealogical analysis, typically with foreign keys linking to dimension tables. We'll create a **single fact table** where **each row is a relationship**:
ancestor → descendant, with depth/generation info.

Flattened Table: FactPersonRelationship

```
CREATE TABLE FactPersonRelationship (
   ancestor_id INT,          -- The ancestor
   descendant_id INT,        -- The descendant
   generation_distance INT,  -- How many generations down
   direct_child BOOLEAN,     -- TRUE if direct child (generation_distance = 1)
   PRIMARY KEY (ancestor_id, descendant_id),
   FOREIGN KEY (ancestor_id) REFERENCES Person(id),
   FOREIGN KEY (descendant_id) REFERENCES Person(id)
);
```

How to Populate It (Recursive CTE)

```
WITH RECURSIVE BioLineage AS (
   -- Base case: direct biological parent-child relationships
   SELECT
      parent AS ancestor_id,
      child AS descendant_id,
      1 AS generation_distance
   FROM BioParent

   UNION ALL

   -- Recursive step: find child of a descendant
```

```
    SELECT
        bl.ancestor_id,
        bp.child AS descendant_id,
        bl.generation_distance + 1
    FROM BioLineage bl
    JOIN BioParent bp ON bl.descendant_id = bp.parent
)

-- Insert into the flattened relationship table
INSERT INTO FactPersonRelationship (ancestor_id, descendant_id,
generation_distance, direct_child)
SELECT
    ancestor_id,
    descendant_id,
    generation_distance,
    generation_distance = 1
FROM BioLineage;
```

What You Can Analyze With This

You now get one row per **ancestor-descendant pair**, with the number of generations between them.

You can now compute:

1. Number of Biological Children

```
SELECT ancestor_id, COUNT(*) AS num_children
FROM FactPersonRelationship
WHERE generation_distance = 1
GROUP BY ancestor_id;
```

2. Number of Biological Descendants

```
SELECT ancestor_id, COUNT(*) AS num_descendants
FROM FactPersonRelationship
GROUP BY ancestor_id;
```

3. Min/Max Generational Depth

```
SELECT
   ancestor_id,
   MIN(generation_distance) AS min_generation,
   MAX(generation_distance) AS max_generation
FROM FactPersonRelationship
GROUP BY ancestor_id;
```

4. Optional Add-on: Person → Ancestor Table

You can also materialize the reverse:

```
CREATE TABLE FactAncestorRelationship (
   descendant_id INT,
   ancestor_id INT,
   generation_distance INT,
   direct_parent BOOLEAN,
   PRIMARY KEY (descendant_id, ancestor_id),
   FOREIGN KEY (descendant_id) REFERENCES Person(id),
   FOREIGN KEY (ancestor_id) REFERENCES Person(id)
);
```

Same logic, just flipped.

- A **flattened row** is: ancestor → descendant + generation_distance.
- This supports efficient OLAP-style lineage queries.
- It's fully derived from your existing BioParent table.
- Recursive CTE does the heavy lifting to expand from parents to all descendants.

5. Basic Aggregation

```
-- Count how many people are in the database
SELECT COUNT(*) AS total_people FROM Person;
```

6. Join

Inner Join Example:

```
-- List people and their latest given name
SELECT p.id, g.name
FROM Person p
JOIN GivenName g ON p.id = g.person_id
WHERE g.gn_name_id = (
    SELECT MAX(gn_name_id)
    FROM GivenName
    WHERE person_id = p.id
);
```

Outer Join Example:

```
-- List all persons and their family names if available
SELECT p.id, f.name
FROM Person p
LEFT JOIN FamilyName f ON p.id = f.person_id;
```

Anti-join Example:

```
-- People who were never married
SELECT p.id
FROM Person p
WHERE NOT EXISTS (
    SELECT 1 FROM Marriage m
    WHERE m.spouse1 = p.id OR m.spouse2 = p.id
);
```

7. Views

```
-- Create a view of active marriages
CREATE VIEW ActiveMarriages AS
SELECT spouse1, spouse2, marriage_date
FROM Marriage
WHERE end_date IS NULL;
```

8. Recursive WITH (Descendants)

```
WITH RECURSIVE Descendants AS (
   SELECT parent AS ancestor, child AS descendant, 1 AS generation
   FROM BioParent
   UNION ALL
   SELECT d.ancestor, b.child, generation + 1
   FROM Descendants d
   JOIN BioParent b ON d.descendant = b.parent
)
SELECT ancestor, COUNT(DISTINCT descendant) AS bio_descendants_count,
      MIN(generation) AS min_generation_depth,
      MAX(generation) AS max_generation_depth
FROM Descendants
GROUP BY ancestor;
```

9. Generating Unique ID

```
-- For PostgreSQL or systems with sequence support
CREATE SEQUENCE person_id_seq START WITH 1 INCREMENT BY 1;

-- Use in insert:
INSERT INTO Person(id, birth_date)
VALUES (nextval('person_id_seq'), '1980-01-01');
```

10. Triggers – Audit Log on GivenName insert

Audit Table

```
CREATE TABLE GivenNameAudit (
    person_id INT,
    action TEXT,
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Trigger Function

```
CREATE OR REPLACE FUNCTION log_givenname_insert()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO GivenNameAudit (person_id, action)
    VALUES (NEW.person_id, 'INSERT');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger

```
CREATE TRIGGER trg_givenname_insert
AFTER INSERT ON GivenName
FOR EACH ROW EXECUTE FUNCTION log_givenname_insert();
```

11. Access Control – Users and Managers

```
-- Create roles
CREATE ROLE manager LOGIN PASSWORD 'managerpass';
CREATE ROLE regular_user LOGIN PASSWORD 'userpass';

-- Grant permissions
GRANT SELECT, INSERT, UPDATE, DELETE ON Person TO manager;
GRANT SELECT ON Person TO regular_user;
```

12. Concurrency & Isolation Level

--Set isolation level for transaction

BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Perform transaction logic here
UPDATE Person SET birth_place = 'New York' WHERE id = 101;

COMMIT;

13. Conflict Graph

Conflict graph should not have cycles or else, the schedule cannot be serializable.
Example of three concurrent transaction types:

T1	R1(C)				W1(A)	R1(B)		R1(B)	
T2		W2(B)					R2(C)		W2(B)
T3			W3(A)	R3(B)					

Conflict pairs:
W2(B) R3(B)
W2(B) R1(B)
ANOTHER W2(B) R1(B)
W3(A) W1(A)
R3(B) W2(B)
R1(B) W2(B)
ANOTHER R1(B) W2(B)

According to the conflict graph, is the schedule is not serializable because it has cycles
2->3
3->2
And
2->1
1->2

Relaxed isolation level can slowdown the speed of database by introducing anomalies such as deadlocks, dirty reads, phantom reads...

14. Transaction Logs

```
CREATE TABLE TransactionLog (
    txn_id SERIAL PRIMARY KEY,
    operation TEXT,
    table_name TEXT,
    row_id INT,
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

15. Audit Logs – Generic Example

```
CREATE TABLE AuditLog (
    id SERIAL PRIMARY KEY,
    table_name TEXT,
    action TEXT,
    actor TEXT,
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

16. Scalability Isolation Levels Setting: Prevent Dirty Reads

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Safely update without reading uncommitted data
UPDATE Person
SET birth_place = 'New York'
WHERE id = 101;

COMMIT;
```

Optimization and Query Tuning

Cost Calculation Concepts

- **Tuples, Blocks, and Data Transfer**
  Data movement in query plans involves transferring tuples (rows) and blocks (disk pages) between CPU, memory caches, disk, and over the network. Minimizing these transfers improves performance.

  Caching using least/most recent data block by erasing and replacing the oldest block with the newest data. Success percentage is calculated by counting hits and misses in a series HHMMHHMMH

- **Pipelining Operations**
  Operations in query execution plans can be combined (pipelined) to avoid intermediate materialization, reducing I/O and CPU overhead. For example, filtering can happen as tuples are produced from a join operation.
-
- **Size Estimation**
  Estimating the size of intermediate results (number of tuples) after operations like filters, joins, or projections helps predict costs and optimize the plan.
  Tuple size of joining S and C (SC) = max(T(S), T(C))
  Block size of joining S and C (SC) = (B(S)/T(S) + B(C)/T(C))*T(SC)

- **Selectivity Estimation**
  Selectivity is the fraction of rows satisfying a predicate (e.g., SELECT * FROM table WHERE col = value has selectivity = number of matching rows / total rows). Lower selectivity means fewer rows pass, allowing early filtering to reduce data volume.
  Order descending 1-selectivity / cost

- **Join Implementations and Costs**
  Different join algorithms have different costs depending on indexes and data distribution:

  B+ Tree Index Join: Efficient for range queries and ordered data.

  Block I/O cost (SC) = Block I/O cost (B+ tree) = B(x) [log base (B-1) OF B(x)/B] *2 + B(S) + B(C)

  Block I/O cost sorting a table = B(x) [log base (B-1) OF B(x)/B]

Block I/O cost sorting using B+ tree = [log base 256 OF B(x)/B]

Hash Join: Efficient for equality joins with large datasets.

Block I/O cost (SC) = Block I/O cost (Hash Table) = B(S) + B(C)

Bitmap Join: Useful for low-cardinality (tuple count) attributes.

Nested Join: Useful for low-cardinality (tuple count) attributes.

B(SC) = (B(S) * B(C)) / (B-2)

Example of a physical query plan and how to calculate the I/O cost of this query:

Person(id, first_name, last_name, gender, car_vin, ...)
car_vin references Car.vin

Car(vin, model_id, type, ...)
model_id references Model.id

Model(id, company, ...)

Consider the following query and corresponding physical query tree.

```
SELECT p.first_name, p.last_name, m.company
FROM Person AS p
INNER JOIN Car AS c ON p.car_vin = c.vin
INNER JOIN Model AS m ON c.model_id = m.id
WHERE p.gender = 'F' AND c.type IN ('sedan', 'coupe')
```

Assumptions:

There are 100,000 persons with 1,000 tuples per block.

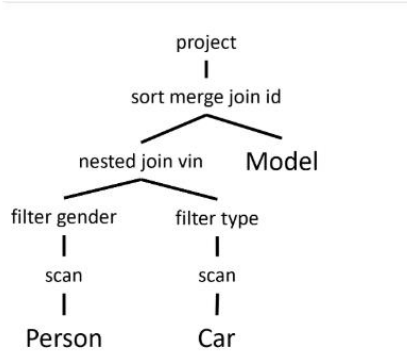
There are 80,000 cars with 2,000 tuples per block

There are 50 models all in one block.

There are 5 car types and 2 genders, each uniformly distributed.

Memory can store 100 blocks at a time.

Pipelining is used wherever possible.



Gender filter selects about 50% of persons → 50,000 females

Type filter on car (e.g., coupe or sedan) selects 2/5 → 32,000 cars

These 32,000 cars = 16 blocks

Step-by-Step Cost Breakdown

1. Scan of Person

- I/O cost: Reading all person data
- Cost: `B(Person) = 100` blocks
- Gender filter: done during scan → no additional I/O
- Total so far: 100 I/O

2. Filter on Gender

- Performed during Person scan, so:
- Reads: 0 (included in scan)
- Writes: 0 (pipelined)
- No I/O added

3. Scan of Car

- I/O cost: Reading all car data
- Cost: `B(Car) = 40` blocks
- Total so far: 100 + 40 = 140 I/O

4. Filter on Type (e.g., sedan or coupe)

- Car filter reduces from 80,000 to 32,000 cars → 16 blocks
- If materialized: 16 blocks need to be written
- If pipelined / held in memory: 0 write I/O
- Three options:
    a. Write filter results to disk → +16 I/O
    b. Pipelined but block nested join required → no write
    c. Kept in memory for repeated reads (used in block nested join) → no write

5. Join: Person ⋈ Car : Three versions:

(a) Regular Nested Loop Join

- Outer: 50,000 filtered Persons (50 blocks)
- Inner: 16 filtered Cars (16 blocks)
- Total I/O: `50,000 × 16 = 8,000,000`
- Total I/O so far: `100 + 40 + 16 + 8,000,000 = 8,000,156` → rounded: 8,000,157

(b) Block Nested Loop Join

- Formula: `(B_outer × B_inner) / (B - 2)`
- Calculation: `(50 × 16) / 98 ≈ 9 I/O`
- Total I/O so far: `100 + 40 + 16 + 9 = 165` (+1 for model scan below = 166)

(c) In-memory Join (filter result fits in memory)

- If the 16 blocks of filtered `Car` fit in memory, no write needed, no repeated reads
- Join cost: 0 I/O
- Total I/O so far: `100 + 40 + 0 = 140` (+1 for model scan below = 141)

6. Scan of Model

- One-time scan
- Cost: 1 I/O
- Total I/O:
    - With regular nested join: 8,000,157
    - With block nested join: 166
    - With in-memory join: 141

7. Sort-Merge Join: (Person-Car) ⋈ Model

- Both inputs small (75 blocks and 1 block), fit in memory
- Sort-Merge can be done in memory → 0 I/O

8. Projection

- Results pipelined from join
- No additional I/O

Query Plan Tuning

1. **Viewing Query Plans**

   Using EXPLAIN ANALYZE

2. **Avoid Blocking Optimizations**
   a. Avoid wrapping indexed columns in functions.
   b. Avoid loops or nested queries that cause multiple scans.
   c. Avoid fetching excessive rows unnecessarily.
3. **Reduce Computation**
   a. Use indexes to speed up lookups.
   b. Combine operations via pipelining.
   c. Avoid redundant computations.
4. **Reduce Data Early**
   a. Apply filters with the highest selectivity first.
   b. For multiple AND conditions, order predicates by their filtering power to quickly reduce row counts.
5. **System Tuning**
   a. Partition tables to distribute data.
   b. Use striping and mirroring for fault tolerance.
   c. Utilize proper join collapsing limits and indexing strategies.

Example 1: Comparing Insert Performance in Python with psycopg2

```
import psycopg2
import time

conn = psycopg2.connect("dbname='postgres' user='postgres' host='172.17.0.2'
password='your_password'")
```

```
cur = conn.cursor()

# Method 1: Insert rows one-by-one
cur.execute("DROP TABLE IF EXISTS t;")
cur.execute("CREATE TABLE t(num NUMERIC);")
conn.commit()

start = time.time()
for i in range(1, 1001):
    cur.execute("INSERT INTO t(num) VALUES (%s);", (i,))
    conn.commit()
end = time.time()
print("Time 1:", end - start)

# Method 2: Bulk insert using one query
cur.execute("DROP TABLE IF EXISTS t;")
cur.execute("CREATE TABLE t(num NUMERIC);")
conn.commit()

start = time.time()
num_lst = []

for i in range(1, 1001):
    num_lst.append(f"({i})")
num_str = ",".join(num_lst)

cur.execute(f"INSERT INTO t(num) VALUES {num_str};")
conn.commit()
end = time.time()
print("Time 2:", end - start)

cur.close()
conn.close()
```

Example 2 : Bulk Inserts with PostgreSQL generate_series

Insert 5,000 students and 100,000 enrollment records efficiently:

```
-- Insert students
INSERT INTO Student (s_id, first_name, last_name)
```

```
SELECT n, md5(random()::TEXT), md5(random()::TEXT)
FROM generate_series(1, 5000) AS Data(n);

-- Insert enrollments with conflict handling
INSERT INTO Enrollment (s_id, crn, rating)
SELECT
    floor(random() * 5000 + 1),
    floor(random() * 1000 + 1),
    floor(random() * 5 + 1)
FROM generate_series(1, 100000) AS Data(n)
ON CONFLICT (s_id, crn) DO NOTHING;
```

Example 3: EXPLAIN ANALYZE on a Join Query

```
EXPLAIN ANALYZE
SELECT s.s_id, s.first_name, s.last_name
FROM Student s
INNER JOIN Enrollment e ON s.s_id = e.s_id
GROUP BY s.s_id, s.first_name, s.last_name
HAVING COUNT(e.crn) > 5;
```

PLAN 1 :

Operation1: scan A

Estimated cardinality : 20000

Estimated time: 0.008

Operation2: sort A

Estimated cardinality: 20000

Estimated time: 3.52

Operation3: scan B

Estimated cardinality : 20000

Estimated time: 0.008

Operation4: sort

Estimated cardinality : 20000

Estimated time: 3.27

Operation5: merge

Estimated cardinality : 20000

Estimated time: 6.79

Example 4: EXISTS and NOT EXISTS Queries

```
-- Create tables
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS order_items;

CREATE TABLE product (
    product_id INT PRIMARY KEY,
    product_type CHAR(20),
    product_color CHAR(20)
);

CREATE TABLE order_items(
    order_id INT,
    product_id INT REFERENCES product(product_id),
    product_quantity INT
);

-- Insert sample data
INSERT INTO product VALUES
(1, 'Gizmo', 'Red'),
(2, 'b', 'Blue'),
(3, 'c', 'Black'),
(4, 'Not Gizmo', 'Yellow'),
(5, 'f', 'Yellow');

INSERT INTO order_items VALUES
(10, 1, 200),
(12, 2, 100),
(22, 5, 150),
(33, 4, 150),
(24, 1, 300);
```

-- Queries with EXISTS

```
EXPLAIN ANALYZE
SELECT * FROM product p
WHERE EXISTS (
   SELECT 1 FROM order_items o
   WHERE o.product_id = p.product_id
);

EXPLAIN ANALYZE
SELECT * FROM product p
WHERE EXISTS (
   SELECT 1 FROM order_items o
   WHERE o.product_id = p.product_id AND o.product_quantity > 10 AND
p.product_type = 'Gizmo'
);

EXPLAIN ANALYZE
SELECT * FROM product p
WHERE EXISTS (
   SELECT 1 FROM order_items o
   WHERE o.product_id = p.product_id AND p.product_type = 'Gizmo' AND
p.product_color = 'green'
);

EXPLAIN ANALYZE
SELECT * FROM product p
WHERE NOT EXISTS (
   SELECT 1 FROM order_items o
   WHERE o.product_id = p.product_id
);
```

-- Queries with NO EXISTS

```
    -- Simple subquery
    select *
    from product as p
    where product_id IN (select product_id from order_items)
```

```
-- Correlated subquery
select *
from product as p
where product_id IN (select product_id
                    from order_items
                    where product_quantity > 10 and product_type = "gizmo")

-- Correlated subquery
select *
from product as p
where product_id IN (select product_id from order_items)
     and product_type = "gizmo" and product_color = "green"

-- Negated to be anti-join
select *
from product as p
where product_id NOT IN (select product_id from order_items)
```

Prototype – Translation to MongoDB

Objectives:

- For each person, count the number of biological children.
- For each person, count the number of biological descendants.
- For each person, count the number of minimum and maximum number of generations of biological descendants
- Possibly queries ancestors instead of descendants

Document Schema Design:

Main document: Person
Embedded: personal data such as “names”, and “gender”
Referenced documents: other Persons as “spouse”, “parent”, “children”. We choose to record both parent list and children list for descendant's traversal as well as ancestor's traversal and aggregations as stated in the objectives.

CRUD and Aggregation Queries in MongoDB

```
{

“_id” : ObjectID,
“birth_date”:DATE,
“death_date”: DATE,
“birth_place”:  TEXT,
“given_name” :
{ “start_date”: DATE,
        “word”:
                [
{“gn_name”: INT,
“seq_name”: INT,
“text”: TEXT},
{“gn_name”: INT,
“seq_name”: INT,
“text”: TEXT},
                ]
}
“family_name” :
        { “start_date”: DATE,
          “word”:
                [
{“gn_name”: INT,
“seq_name”: INT,
“text”: TEXT},
{“gn_name”: INT,
“seq_name”: INT,
“text”: TEXT},
                ]
        }

“gender_identity” :
        { “start_date”: DATE,
                “gender”: TEXT}

“parents_biological” :
        [
        { “parent_id”: ObjectID, “mother”: BOOLEAN},
          { “parent_id”: ObjectID, “mother”: BOOLEAN}
        ]
```

“parents_adoptive” :
[
{“start_date” : DATE},
{ “parent_id”: ObjectID, “start_date”: DATE},
{ “parent_id”: ObjectID, “start_date”: DATE}
]

“children_biological" :
[
{ “children_id”: ObjectID, “gender”: TEXT},
{ “children_id”: ObjectID, “gender”: TEXT}}
]

“children_adoptive :
[
{ “children_id”: ObjectID, “start_date”: DATE},
{ “children_id”: ObjectID, “start_date”: DATE}
]

“marriages”:
[
{ “spouse_id”: ObjectID, “start_date”: DATE, “end_date”: DATE},
{ “spouse_id”: ObjectID, “start_date”: DATE, “end_date”: DATE}
]

}

Aggregate MongoDB query: each field has an object, document, or array of objects/documents. Aggregate function $ expects a single object.

For each person, count the number of biological children.

Db.Person.aggregate([

{
$project: {given_name:1, num_bio_children: {$size: {$ifNull : [“$children_biological”, []]
}}}

])

Prototype – Translation to Neo4j

Objectives:

- For each person, count the number of biological children.
- For each person, count the number of biological descendants.
- For each person, count the number of minimum and maximum number of generations of biological descendants
- Possibly query ancestors instead of descendants

Node & Relationship Model:

Node: Person with “id”, “given_name”, “family_name”, “gender”, “birth_date”, “death_date”, “birth_place”
Node: Gender with “id”, “gender, “start_date”
Node: Given_Name with “id”, “gn_name”, “seq_name”, “start_date”, “end_date”
Node: Family_Name with “id”, “gn_name”, “seq_name”, “start_date”, “end_date”

Relationship Edges: BioParentOf, AdoptParentOf, BioChildOf, AdoptChildOf, SpouseOf

Person Node:

Create (:Person{

“id”: TEXT

“given_name”: TEXT,

“family_name”:TEXT,

“gender”: TEXT,

“birth_date”: DATE,

“death_date”: DATE,

“birth_place”: TEXT

});

Create (:Gender{
“id”: TEXT
“gender”: TEXT,
“start_date”: DATE
“end_date”: DATE
})

Create (:Given_Name{
“id”: TEXT
“gn_name”: INT,
“seq_name”: INT,
“text”: TEXT,
“start_date”: DATE,
“end_date”:DATE,
})

Create (:Family_Name{
“id”: TEXT
“gn_name”: INT,
“seq_name”: INT,
“text”: TEXT,
“start_date”: DATE,
“end_date”:DATE,
})

Relationship Edges:

MATCH (p1:Person {“id”: TEXT}), (p2:Person {“id”: TEXT}),
Create (p1-[:BioParentOf]-> (p2);

MATCH (p1:Person {“id”: TEXT}), (p2:Person {“id”: TEXT}),
Create (p1-[:AdoptParentOf]-> (p2);

MATCH (p1:Person {“id”: TEXT}), (p2:Person {“id”: TEXT}),
Create (p1-[:BioChildOf]-> (p2);

MATCH (p1:Person {“id”: TEXT}), (p2:Person {“id”: TEXT}),
Create (p1-[:AdoptChildOf]-> (p2);

MATCH (p1:Person {“id”: TEXT}), (p2:Person {“id”: TEXT}),
Create (p1-[:SpouseOf{“start_date”:DATE, “end_date”:DATE}]-> (p2);

MATCH (p1:Person {“id”: TEXT}), (g1: Given_Name{“id”: TEXT}),
CREATE (p1)-[:Has_Given_Name]->(g1)

MATCH (p1:Person {“id”: TEXT}), (f1:Family_Name {“id”: TEXT}),
CREATE (p1)-[:Has_Family_Name]->(f1)

MATCH (p1:Person {“id”: TEXT}), (g2:Gender {“id”: TEXT}),

CREATE (p1)-[:Has_Gender]->(g2)

Cypher Aggregation Queries

Match (:Person{“id”: TEXT})-[:BioParentOf]-> (p:Person{“id”: TEXT})

Return p.name, COUNT(p) AS num_bio_children

Match (:Person{“id”: TEXT})-[:BioParentOf*1..]-> (p:Person{“id”: TEXT})

Return p.name, COUNT(DISTINCT p) AS num_bio_descendants

Our Conclusion

This project demonstrated how genealogical data can be modeled across relational, document, and graph paradigms. OLTP SQL design ensured integrity, OLAP queries enabled analysis, MongoDB offered schema flexibility, and Neo4j captured natural family relationships. Each model highlighted tradeoffs in performance, expressiveness, and usability.

The work reflects the principle that no single model fits all needs, and cross-model fluency is essential for modern data professionals. This project was informed by coursework at Rice University under Dr. John Greiner, Ph.D., whose teaching in database systems provided the foundation for this study.