Technical Implementation Document for Complete Data Migration from RDBMS to Property Graph Model (Corrected + Enhanced)

I. Project Context and Architectural Foundation

Modern data systems face challenges due to the dramatic growth in volume, variety, and velocity of data, collectively known as Big Data.[1] Traditional Relational Database Management Systems (RDBMS) struggle with:

- Limited scalability
- High join costs
- Poor handling of highly interconnected data
- Fixed schema structure

To address these limitations, organizations increasingly migrate data to NoSQL systems, particularly Graph Databases, which excel at modeling complex, relationship-heavy data.[1]

A. Migration Imperative

RDBMS systems (MySQL, Oracle, PostgreSQL) follow the relational model, store data in tables, enforce ACID guarantees, and use SQL language for DDL/DML operations.1
However, Big Data applications require:

- High concurrency
- Flexible schema
- Distributed architecture
- Fast relationship traversal

NoSQL databases were developed to address these limitations, providing schema-less, distributed, scalable data models.[1]

B. Rationale for Choosing Graph Databases

Among NoSQL models—Key-Value, Document, Wide Column, Graph—the graph database is ideal when data has many interconnected entities (social networks, enterprise systems, knowledge graphs).1
Graph Databases (Neo4j):

- Use nodes, edges, and properties [1]
- Support fast relationship traversal [1]
- Remove costly "joins" [1]
- Provide intuitive graph modeling [1]
- Support Cypher, a declarative graph query language [1]

C. Need for the SCT Architecture (Source → Conceptual → Target)

The research paper stresses a critical weakness:

Most existing tools use Source-to-Target (ST) translation directly, which leads to a flat and weak target schema, losing semantics like:

- Inheritance
- Aggregation
- Cardinality
- Relationship meaning/naming
- Complex constraints

Only a few studies use the Source → Conceptual → Target (SCT) path, which produces richer graph models.[1]

Why SCT is mandatory for this project:
✔ Captures implicit metadata 1
✔ Enhances semantics through conceptual modeling 1
✔ Prevents flattening or oversimplification 1
✔ Ensures every relationship type (association, inheritance, aggregation) is preserved 1
Thus, this implementation adopts SCT as the mandatory architecture.1

II. Pre-Implementation Requirements and System Setup

A. Hardware & Environment

The environment must support:

- Running RDBMS and Neo4j simultaneously
- High I/O disk throughput (for Extract and Load phases)
- Sufficient RAM (for memory-intensive Transformation phase)
- Multi-core CPU (for parallel ETL operations)

B. Software Stack

The selected stack aligns perfectly with techniques in the research paper.[1]

Component	**Tool**	**Purpose in Migration (SCT Phase)**	**Source from Paper**
Source RDBMS	MySQL, PostgreSQL	Hosting source data and explicit schema metadata (RDBMS) [1]	[1]
Target NoSQL	Neo4j	Target environment for property graph model [1]	[1]
Connectivity	JDBC Driver, SQL	Extracting schema metadata and data instances [1]	[1]

Transformation/Scripting	Python / Java ORM	Implementing complex schema mapping rules, data transformation [2]	[1]
Conceptual Modeling	ERD / EERD / UML	Visualizing and documenting the semantically enriched schema (C) [1]	[1]
ETL Tools	Neo4j ETL, APOC, CSV	Automated metadata extraction and high-volume data loading [3]	[1]

III. Phase 1 — Semantic Enrichment (S → C)

This step aligns closely with the research paper's focus on metadata extraction and semantic enrichment.[1]

A. Metadata Extraction (Relational Schema Representation — RSR)

The process involves extracting explicit metadata, including tables, primary keys, foreign keys, and data types, from the source RDBMS.[5] This metadata is stored in an internal Relational Schema Representation (RSR) to simplify key matching and classification of relational constructs.[5]

B. Semantic Enrichment (DBRE Method)

Semantic enrichment uses Database Reverse Engineering (DBRE) techniques to infer semantics hidden in the relational schemas and data.[11] This is critical for generating a non-flat target schema.[1]

1. **Relationship Cardinality Inference and Naming:** Analyze Foreign Keys (FKs) to determine precise cardinality (1:1, 1:N) and infer meaningful, business-oriented relationship names (e.g., MANAGES) rather than generic FK names.[1]
2. **Inheritance Detection:** Identify patterns such as Class Table Inheritance where related tables share an identical Primary Key structure, and the subclass PK acts as an FK to the superclass.[5] The system must implement pattern recognition algorithms to identify these structural commonalities.[1]
3. **Aggregation Identification:** Identify complex FK structures, often through analysis of cascading deletes or mandatory participation, that suggest a strong dependent lifecycle or "part-of" semantic, which must be captured explicitly.[1]

C. Construction of the Conceptual Model (EER/UML)

The resulting rich, explicit Conceptual Model (CDM), documented as an EERD or UML Class Diagram, serves as the high-fidelity input for schema translation rules, ensuring a semantically strong target structure.[1]

IV. Phase 2 — Schema Translation (C → Graph)

The translation dictates the precise rules for converting the enriched Conceptual Model (C) into the Property Graph Model (T).[1]

A. General Mapping Rules: Entity to Node

[7]

1. **Entity Mapping:** Each conceptual entity maps to a unique **Node Label** (:LabelName).
2. **Row and Attribute Mapping:** Each row instance becomes a **Node**, and attributes become **properties** on that node (key: value).
3. **Key Mapping:** Technical primary keys are removed; essential **business primary keys** are mapped to **Unique Constraints** to enforce data integrity and optimize lookup performance (CREATE CONSTRAINT).[7] Indexes are applied to frequently searched

attributes.[7]

B. Core Rule Set 1: Associative Relationship Translation

[7]

1. **1:N Relationships:** The foreign key column is dropped and replaced by a single, directed **Edge** (-->).
2. **M:N Relationships (Join Tables):** The join table is eliminated conceptually and replaced by an **Edge**. If the join table contained attributes, these attributes are translated into **properties of the new relationship edge**.[7]
3. **Ternary Relationship Conversion:** The intersection table is converted into a dedicated **Relationship Node** to maintain connectivity and attributes, linked by three or more distinct relationships to the respective entity nodes.

C. Core Rule Set 2: Advanced Semantic Translation

This rule set preserves the complex semantics identified in Phase 1, which are often disregarded.[1]

1. **Inheritance Modeling Rules:** The inheritance structure (superclass/subclass) is modeled using the **Multi-Labeling Strategy**.[8] The subclass node receives both the Superclass Label and its specific Subclass Label (e.g., (:Person:Employee)) by merging data from both tables onto a single node instance. This supports polymorphic queries.[8]
2. **Aggregation Modeling Rules:** Aggregation (strong compositional semantics) is modeled using **highly specific, specialized relationship types** (e.g., -->) to preserve the structural dependency.[1]

RDBMS Construct	**Conceptual Inference (EERD)**	**Target Graph Model (Cypher/Neo4j)**	**Key Action**
Entity Table	Entity Type	Node Label (:LabelName)	Create Node [7]

Row	Entity Instance	Node	Create Node [7]
Column (Non-Key)	Attribute	Node Property (key: value)	Set Property [7]
Primary Key (Business)	Unique Identifier	Unique Constraint	CREATE CONSTRAINT [8]
Foreign Key (1:N)	Association Relationship	Directed Relationship (-->)	Drop FK, Create Edge [7]
Join Table (M:N)	Association Relationship	Directed Relationship with Properties	Eliminate Table, Create Edge [7]
Tables with Shared PK	Inheritance (Generalization)	Node with Multiple Labels (:Superclass:Subclass)	Combine Entity Semantics [8]
Complex FK Structure	Aggregation (Composition/Part-of)	Specialized, strong relationship (-->)	Preserve Strong Semantic [1]
Intersection Table (3+ FKs)	Ternary Relationship	Dedicated Intersection Node	Introduce New Node Type

V. Phase 3 — ETL: Extract, Transform, Load

A. Extract Phase: Data Retrieval Strategy

Data extraction utilizes the **JDBC driver** to query the RDBMS.[1] Efficient SELECT statements retrieve data instances, which are then staged as **CSV files**, the most effective input format for high-speed bulk import.[1]

B. Transformation Phase: ORM and Data Structuring

Custom **Python or Java** scripts implement the transformation logic, executing the rules defined in Phase 2.[2]

1. **Data-to-Object Conversion:** Raw tabular data is mapped to intermediary objects using **ORM techniques**.[1]
2. **Implementing Complex Merges:** Scripts join superclass/subclass data (based on shared PKs) to create single, multi-labeled records for inheritance.[8] They analyze FK values to generate explicit relationship creation commands using the semantically correct names (e.g., -->).[1]
3. **Output Generation:** Structured CSV files are generated, separated by target component type (e.g., separate files for nodes and relationships), optimized for Neo4j's bulk loading utilities.[9]

C. Load Phase: Bulk Data Import into Neo4j

High-performance loading methods are mandatory for large-scale migration.[9]

1. **High-Volume Bulk Load:** The **neo4j-admin database import full** utility is required for initial, large-scale imports from structured CSV files.[9]
2. **Medium-Volume Load:** For smaller volumes (up to 10 million records) or incremental updates, the Cypher command **LOAD CSV** is used.[9]
3. **API Load:** Custom scripts utilizing language libraries (e.g., Python/Java drivers, **py2neo**) can programmatically execute batched Cypher statements for highly complex or incremental updates.[2] The **APOC library** extends Cypher for advanced procedures during loading.[9]

VI. Demonstrative Implementation Details

To ensure complete understanding and reproducibility, the following sections detail the architectural flow, detection algorithms, and a running example of the transformation.

A. Conceptual Architecture Flow

The complete system is based on the SCT (Source $\rightarrow$ Conceptual $\rightarrow$ Target) methodology, requiring a sequence of specialized components:

1. **RDBMS (Source):** Stores source data and explicit schema metadata (tables, FKs, PKs).[1]
2. **Schema Analyzer (JDBC):** Connects to the RDBMS using the JDBC Driver [1] to extract metadata and build the **Relational Schema Representation (RSR)**.[5]
3. **Conceptual Model Builder (DBRE):** Performs **Semantic Enrichment** on the RSR, using techniques like Database Reverse Engineering (DBRE) [11] to infer implicit semantics (e.g., **Inheritance** and **Aggregation** patterns).[1] This results in the **Canonical Data Model (CDM)** / EERD.[5]
4. **Transformation Engine (ORM/Scripting):** Uses the CDM rules to drive the process. It extracts data instances from the RDBMS, applies Object-Relational Mapping (ORM) [1], performs complex data merges (e.g., inheritance joining), and generates staged **CSV files** for nodes and relationships.[9]
5. **Neo4j Loader (ETL Tools):** Ingests the CSV files into the **Neo4j Graph Database (Target)** using high-volume utilities like neo4j-admin import full.[9]

B. Algorithmic Pseudocode for Semantic Detection and Mapping

Algorithm 1: Inference and Merging for Inheritance (Generalization)

```
// INPUT: RSR containing Table MetaData (T1, T2)
// OUTPUT: Transformation Rule to merge T1 and T2 data into one Multi-Labeled Node

FUNCTION InferAndMergeInheritance(Table T1, Table T2):
 // Check for shared PK structure and FK relationship
 IF T1.PrimaryKey == T2.PrimaryKey AND T2.PrimaryKey IS ForeignKeyTo T1.PrimaryKey:
   Superclass = T1
   Subclass = T2
```

```
  // Apply Multi-Labeling Strategy
  TransformationRule = {
    SourceTables:,
    TargetLabel: Superclass.Name + ":" + Subclass.Name,
    MergeAction: JOIN_ON_PRIMARY_KEY,
    Output: Single_Record_With_MultiLabels
  }
  RETURN TransformationRule
 END IF
 RETURN NULL

// Implementation requires script to JOIN Superclass and Subclass data rows based on their shared key
// and assign the combined labels (e.g., (:Person:Employee)) to the resulting node instance.
```

Algorithm 2: Mapping M:N Join Tables to Relationships [7]

```
// INPUT: RSR containing Join Table J and two related Entity Tables A, B
// OUTPUT: Cypher Generation Rule for Relationship creation

FUNCTION MapJoinTableToRelationship(JoinTable J, Entity A, Entity B):

 // Identify the semantic relationship name (derived in Phase 1)
 RelationshipName = J.InferredName // e.g., 'ENROLLED_IN'

 // Determine if J holds properties (columns other than the two FKs)
 RelationshipProperties = J.Attributes.EXCLUDE(A.ForeignKey, B.ForeignKey)

 // Create the relationship structure
 IF RelationshipProperties IS NOT EMPTY:
  // Attributes become properties of the relationship
  Rule = {
    Source: A.Name,
    Target: B.Name,
    RelationshipType: RelationshipName,
    Properties: RelationshipProperties
  }
 ELSE:
  // Simple relationship (no properties)
  Rule = {
    Source: A.Name,
    Target: B.Name,
    RelationshipType: RelationshipName,
    Properties:
  }
 END IF

 // Action in Transform Phase: Eliminate the table and generate relationship creation commands
 RETURN Rule
```

C. Running Example: Student-Course-Grade Model

This example demonstrates the translation path for key constructs from the Source RDBMS to the Target Graph Schema.

Source RDBMS Construct	**Conceptual Enrichment (Phase 1)**	**Target Graph Schema (Phase 2)**	**Transformation Summary**
Table: Course (ID:PK, Title, DeptID:FK)	Entity Type (Course) + 1:N Association to Department	**Node Label:** (:Course) [7]	ID (FK source) is dropped; Node is created.
Table: Department (DeptID:PK, Name)	Entity Type (Department)	**Node Label:** (:Department) [7]	Node is created with Name property.
FK: Course.DeptID ⟶ Department.DeptID	Inferred Relationship: **OFFERED_BY** (1:N) [1]	**Relationship Edge:** (:Course)-->(:Department) [7]	Foreign key column is removed and replaced by a directed, semantically named relationship.[7]

Table: Takes (StudentID:FK, CourseID:FK, **Grade**)	M:N Join Table with Property (Grade) [7]	**Relationship Edge with Property:** (:Student)-->(:Course) [7]	Join table is eliminated; its non-key attribute (Grade) becomes a property of the new relationship edge.[7]

VII. Verification, Validation, and Performance Optimization

The project concludes with rigorous validation to confirm that the migration was successful in both preserving data integrity and delivering architectural advantages, specifically query performance gains.[10]

A. Data Instance and Semantic Integrity Validation

Verification must confirm that all data instances were transferred and that the complex semantic models were correctly realized.[10]

1. **Data Integrity (Count Verification):** Compare the number of rows extracted from the source RDBMS tables with the total number of nodes created for the corresponding labels in Neo4j.[10] The number of edges created must precisely correspond to the number of foreign key instances processed.[10]
2. **Constraint and Semantic Fidelity Check:** Verify that all Unique Constraints defined on the Neo4j nodes (corresponding to business keys) are correctly enforced.[7] Specific Cypher queries must be run to validate complex semantic translations, such as confirming the multi-labeling inheritance model is functional.[1]

B. Performance Validation and Query Translation

A key objective is the reduction of query cost, particularly minimizing join time.[1]

1. **SQL Query $\rightarrow$ Cypher Query Translation:** Identify a benchmark set of complex, multi-join SQL queries from the RDBMS. Translate these costly queries into equivalent, path-finding Cypher traversal patterns.[13]

2. **Performance Benchmarking:** Execute both the original SQL queries and their Cypher counterparts and compare their average execution times (latency).[10] A successful project demonstrates that the Cypher traversal queries execute significantly faster than the complex, depth-dependent SQL joins.[1]

C. Iterative Refinement and Graph Model Optimization

Refine the graph model based on performance benchmarking.[8] Ensure all frequently used lookup properties are correctly indexed for fast traversal entry points.[7] Optimization may include refining relationship types or adjusting denormalization strategies to ensure optimal performance.[10]

Validation Phase	**Metric/Test Goal**	**Data to Compare (RDBMS vs. Neo4j)**	**Success Criterion**
Data Integrity	Data Count Preservation	Row Counts per RDBMS Table vs. Node Counts per Neo4j Label	Counts must match exactly (or match defined merge criteria).[10]
Semantic Fidelity	Constraint Enforcement	RDBMS Primary/Foreign Keys vs. Neo4j Constraints/Indexes	Unique constraints are enforced; derived relationship types are queryable and accurate.[7]
Functional Equivalence	Query Latency	Execution time of complex multi-join SQL queries vs. equivalent Cypher traversal queries.[1]	Cypher query execution time is significantly reduced, demonstrating architectural benefit.[1]

Model Optimization	Index Usage	Frequency of index hits and cache performance in Neo4j.[7]	All critical lookup properties are indexed; traversal paths are optimized.[10]

Works cited

1. Data Migration from Conventional Databases into NoSQL_ Methods and Techniques.pdf
2. Migrating mysql data to neo4j database - Stack Overflow, accessed November 19, 2025, https://stackoverflow.com/questions/24882715/migrating-mysql-data-to-neo4j-database
3. Neo4j ETL Tool - Interactive Relational Database Data Import, accessed November 19, 2025, https://neo4j.com/labs/etl-tool/
4. Import: RDBMS to graph - Getting Started - Neo4j, accessed November 19, 2025, https://neo4j.com/docs/getting-started/data-import/relational-to-graph-import/
5. Semantic Enrichment: The First Phase of Relational Database Migration - ResearchGate, accessed November 19, 2025, https://www.researchgate.net/publication/274719624_Semantic_Enrichment_The_First_Phase_of_Relational_Database_Migration
6. Database Schema Design Using Entity-Relationship Approach, accessed November 19, 2025, https://www.comp.nus.edu.sg/~lingtw/cs4221/er.pdf
7. Modeling: relational to graph - Getting Started - Neo4j, accessed November 19, 2025, https://neo4j.com/docs/getting-started/data-modeling/relational-to-graph-modeling/
8. A Rule-based Conversion of an EER Schema to Neo4j Schema ..., accessed November 19, 2025, https://www.researchgate.net/publication/357011565_A_Rule-based_Conversion_of_an_EER_Schema_to_Neo4j_Schema_Constraints
9. Import your data into Neo4j - Getting Started, accessed November 19, 2025, https://neo4j.com/docs/getting-started/data-import/
10. Migrate from Relational Database to Graph Database - FalkorDB, accessed November 19, 2025, https://www.falkordb.com/blog/relational-database-to-graph-database-2/
11. OWL Ontology Extraction from Relational Databases via Database Reverse Engineering - Journal of Software, accessed November 19, 2025, https://www.jsoftware.us/vol8/jsw0811-11.pdf
12. RDBMS & Graphs: Relational vs. Graph Data Modeling - Neo4j, accessed November 19, 2025, https://neo4j.com/blog/developer/rdbms-vs-graph-data-modeling/
13. Cypher Query VS SQL. Databases have become an integral part... | by Nimra Tahir | Medium, accessed November 19, 2025,

https://medium.com/@nimratahir1212/cypher-query-vs-sql-718ccb8e8d9d