Graph Databases for Large-Scale Healthcare Systems: A Framework for Efficient Data Management and Data Services

Yubin Park
University of Texas at Austin
TX, USA
Email: yubin.park@utexas.edu

Mallikarjun Shankar
Oak Ridge National Laboratory
TN, USA
Email: shankarm@ornl.gov

Byung-Hoon Park
Oak Ridge National Laboratory
TN, USA
Email: parkbh@ornl.gov

Joydeep Ghosh
University of Texas at Austin
TX, USA
Email: ghosh@ece.utexas.edu

***Abstract*—Designing a database system for both efficient data management and data services has been one of the enduring challenges in the healthcare domain. In many healthcare systems, data services and data management are often viewed as two orthogonal tasks; data services refer to retrieval and analytic queries such as search, joins, statistical data extraction, and simple data mining algorithms, while data management refers to building error-tolerant and non-redundant database systems. The gap between service and management has resulted in rigid database systems and schemas that do not support effective analytics. We compose a rich graph structure from an abstracted healthcare RDBMS to illustrate how we can fill this gap in practice. We show how a healthcare graph can be automatically constructed from a normalized relational database using the proposed "3NF Equivalent Graph" (3EG) transformation. We discuss a set of real world graph queries such as finding self-referrals, shared providers, and collaborative filtering, and evaluate their performance over a relational database and its 3EG-transformed graph. Experimental results show that the graph representation serves as multiple de-normalized tables, thus reducing complexity in a database and enhancing data accessibility of users. Based on this finding, we propose an ensemble framework of databases for healthcare applications.**

I. Introduction

It is increasingly believed that data-driven approaches can help reduce the current healthcare expenditure growth in the United States [1]. Both private and public healthcare sectors are experiencing drastic changes in their policies and data management systems [2]. While health reforms are expected to slow down the current rate of healthcare spending [3], effective data-driven services have been widely suggested as a solution to prevent cost leakage and waste in healthcare systems [4]. Data-driven solutions could provide better personalized, objective, and cost-effective healthcare, as well as early-detection of anomalous behaviors.

In healthcare systems, data management and data services have been traditionally viewed as two independent tasks. In this paper, data management specifically refers to storing data with error-tolerant and non-redundant structure, and data service means various analytic queries including search, joins, statistical data extraction queries, and simple data mining algorithms. Data schemas for healthcare systems are volatile by their nature, and usually involve a multitude of heterogeneous entities and their relations. To minimize redundancy and dependency, healthcare data are typically stored and managed using their "normalized" forms [5], [6]. Those normalized tables are later either restructured or "de-normalized" for data analytics [7], [8], [9]. Due to the gap between data management and service, relational databases and warehouses conducive for analytics are hard to design and analyze. For example, a normalized healthcare database and its dimensional extensions can consist of dozens of individual tables. If we include temporary tables and views for supporting a variety of different services, then the complexity of the data schema grows significantly. Such complex schemas are bottlenecks to analytic procedures that typically require convoluted ways to get to constituent data. Efficient denormalization techniques are still an open topic of discussion.

In this paper, we abstract a rich graph structure from a typical healthcare relational table structure. We adopt a "graph database" approach to extensively utilize such a graph representation. Unlike traditional relational databases, a graph database directly stores and represents nodes, edges, and properties [10]. A graph database can handle directly a wide range of queries that would otherwise require deep join operations in normalized relational tables [11]. Several examples of such join queries are presented in Section V. These queries involve relationships between healthcare entities and can be utilized in collaborative filtering as well as developing personalized services for healthcare delivery. We evaluate the performance of such queries over a relational database and its equivalent graph representation. Our experimental results suggest that a graph database performs well for these classes of queries and also gives us the benefit of an intuitive query representation and access to data. We observe that a single graph database representation can serve many different types of queries that would otherwise require many separate tables and schemas. A graph database as a form of denormalized table can avoid generating and replicating a large number of tables.

Our ultimate goal is to offer a cost efficient data management framework for healthcare systems. We argue that a graph database is a crucial component of such a framework, rather than a whole replacement to the existing architecture. As we observe in our experiments, traditional relational databases still have many advantages over graph databases. Moreover, each graph database has its own specialty and application needs [12]. For example, Neo4J [13] is suited for online transaction processing (OLTP), while Pregel [14] is designed for high latency, high throughput platforms. An efficient ensemble

of these components will lead to a better system design, supporting a wide range of services efficiently. We summarize our contributions in this paper as follows:

- We propose a graph database design rationale, "3NF Equivalent Graph" (3EG) transform. This technique can automatically construct a graph database from an existing normalized relational database.
- We assemble eight useful service queries for healthcare analytics, and evaluate the performance of these queries over two representative databases: MySQL for a relational database and Neo4J for a graph database.
- We generate realistic and large-scale synthetic healthcare data to simulate typical healthcare databases.
- We propose a blueprint for a cost efficient database architecture in healthcare systems. This proposal requires minimal changes to an existing architecture.

In Section II, we briefly discuss healthcare databases in general. We then review forms of database normalization, and graph databases. In Section III, we propose a novel graphs database design rationale, 3NF Equivalent Graph (3EG) transformation. In Section IV, we illustrate how we generate realistic synthetic healthcare data. The data is first stored in a relational database format, then converted to a graph format using the 3EG transformation. In Section V, we discuss a set of real world use cases of service queries in healthcare databases. Empirical evaluation results using MySQL and Neo4J are provided in Section VI. We discuss the limitation of the proposed work, and a proposal for efficient data management and utilization in Section VII.

II. Background

In this section, we describe healthcare databases in general. We also visit basic concepts in database normalization and graph databases.

Healthcare Databases. We describe healthcare systems through an illustrative example. Let's consider a situation that a patient (also referred to as a beneficiary) visits a hospital for his illness. A doctor (also referred to as a provider) diagnoses the patient, and performs a certain procedure. Using the patient's insurance information, a claim is later filed to a corresponding organization. If the patient is with a private sector health insurance, the claim will be delivered to the corresponding institution. This filed claim usually contains a list of providers involved, referral information (occasionally), patient information, diagnosis and procedures performed, and cost.

At their core, healthcare databases need to handle all the information associated with this series of events. From this simple example, we identify two different types of central entities: the beneficiary and the provider. The interaction between these two entities results in insurance claims. Claim events consist of multiple types of features: numeric, categorical values, and texts. Note that this example describes only a slice of complex healthcare ecosystems. In practice, there are a host of other types of events and information sources, such as nurses' notes, Electronic Health Records systems, and interaction groups and networks of beneficiaries and providers. It is no accident that traditional healthcare databases tend to have hundreds of tables describing such complex events and dynamics.

Database Normalization. Database normalization, firstly proposed by E. F. Codd, is a fundamental relational database design rationale to minimize redundancy and ill-defined dependencies [15], [16]. By reducing redundancy and dependency within a system, transactional operations, such as additions, deletions, and updates, can be made in a limited set of tables at a time. Normalized tables, in theory, are better protected from transactional anomalies and inconsistencies from dynamic data changes.

Each Normal Form imposes a set of rules. A specific normal form is achieved when its requirements are fulfilled. First Normal Form (1NF) presumes the existence of the "key", and requires domains of a table to be "atomic". In other words, every attribute in a table should not be divided further. For example, a name attribute containing both first and last names is not atomic, since it actually holds two fields. To understand Second Normal Form (2NF), we should introduce the concept of "functional dependency". A field X is said to have a functional dependency on a field Y if and only if values of X are precisely mapped to values of Y. 2NF is accomplished when 1) 1NF is achieved and 2) no non-key element is functionally dependent on a proper subset of key elements. Third Normal Form (3NF), the most popular normal form, adds a further constraint to 2NF requirements. In 3NF, no transitive dependency is allowed between non-key elements. Specifically, if a table has ZIP code and County code as non-key elements, County code should be removed from the table since County code is transitively dependent on ZIP code. The relation between County and ZIP codes needs to be recorded in a separate table to adhere to 3NF. In this paper, we do not consider other higher forms of normalization such as 4NF [17] and 5NF [18], and mainly focus on 3NF. In most practical situations, higher normal forms than 3NF is not preferred due to their unnecessarily specific constraints.

Graph Database. A graph database can be characterized by its distinct data model that differentiates it from traditional relational databases [10]. A data model is a set of conceptual tools to represent and manage data, consisting of three components [19]: 1) data structure types, 2) query operators, and 3) integrity rules. Data in a graph database are stored and represented as graphs, or data structures generalizing the notion of a graph. Each graph database has its own specialized graph query language. (Structured Query Language (SQL) is inappropriate for graph-represented data.) For example, Neo4J uses Cypher language, and many RDF (Resource Description Framework) databases use SPARQL (SPARQL Protocol and RDF Query Language). Finally, integrity rules in a graph database are based on its graph constraints, rather than those from an imposed relational schema.

With increasing complexity of real-world data and growing needs for graph queries, numerous graph databases have been proposed and developed in recent years. Each graph database is designed for special application needs. For example, Neo4J is suited for online transaction processing (OLTP), while Pregel is designed for high latency, high throughput platforms. In this paper, we primarily use Neo4J to explore the graph structure in healthcare systems. Neo4J is a widely used open-source graph

database, implemented in Java. Neo4J is characterized as an "embedded, disk-based, and fully transactional graph database engine". We use Neo4J because of its broad deployment base and its well-documented APIs. We mainly focus on general graph processing capabilities of Neo4J, rather than its specialized functions compared to other graph engines. Experimental evaluations of other graph engines are also practically important, and we leave this for future work.

III. 3EG: 3NF Equivalent Graph Transform

Converting a relational model to another data model has been an important research topic in both software engineering and database system communities. In software engineering, the semantic structure of a relational database is used to redesign its schema. The Entity-Relationship (ER) model [20] has been a popular choice to represent such semantic structures. Several automatic algorithms to convert a relational model to an ER model, also known as "database reverse engineering"[1], have been introduced in [21], [22], [23]. Migrating a relational database to a completely different class of databases became a natural extension of this database reverse engineering. Various database migration approaches can be found in [24], [25], [26]. These approaches include migrations between 1) ER model and relational model, 2) Object-Oriented (OO) model and Relational model, and 3) OO model and ER model.

Mapping of relational database (RDB) data to RDF has been compressively investigated in recent years [27]. RDF is commonly represented in the form of subject-predicate-object expressions. This triple store representation directly maps to a graph representation: subjects and objects become "nodes", and predicates specify edges. Berners-Lee [28] suggested a general conversion mapping of RDB to RDF:

- An RDB record → a RDF node
- The column name of RDB table → a RDF predicate
- A RDB cell → a value

However, these conversion rules may result in redundant nodes and edges, and in practice, domain-specific ontology maps are incorporated to post-process and refine the resultant RDF.

Automatic conversion of a relational database to a (compact) graph database has not been adequately explored so far. Typically, researchers manually examine the semantic structure of the original relational database, and then carefully port the data to a graph database [29]. Compared to well established RDBMS design principles, a design rationale for a graph database is not well grounded and often *ad hoc*. In this paper, we do not plan to design a graph database from scratch, and focus rather on the fact that most database systems are already in 3NF. We presume the existence of a 3NF relational database, and propose simple conversion rules from 3NF to a graph database. Our conversion rules inherit the Berners-Lee conversion rules, but we derive those rules from the connection between the database normalization theory and graph database.

We focus here on pairwise relationships between nodes, and all the relationships are undirected. The overall ideas of

[1]Conversion from an ER model to a relational schema is called as "database forward engineering".

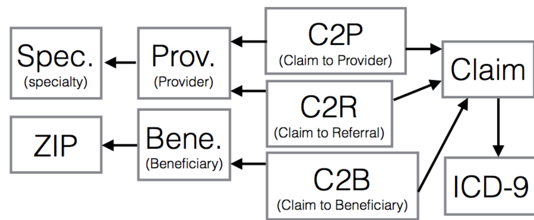

Figure 1. Synthetic Data Generation Process. The directed lines represent reference relations. Synthetic sequence is: Spec, ZIP, Prov → Prov, Bene, Claim → C2P, C2R, C2B.

our conversion rules are as follows: The existence of keys in 1NF intuitively maps to the existence of nodes in a graph database. Relationships (or edges) in a graph database can be analogous to those in a relational database. Properties of nodes can be specified by the requirements in 3NF. The requirements of 3NF table are often concisely described as "every non-key attribute must provide a fact about the key, the whole key, and nothing but the key" [30]. Restated for the graph, non-key elements describe the key (the node in a graph), and this description is only to the key and nothing but the key (a unique connection to the node).

We now formalize our idea of transforming relational tables to a graph database. Consider a table $\mathcal{T} = \{t\}$ where t is a tuple $t = (x, y)$, and a graph $\mathcal{G} = \{\mathcal{N}, \mathcal{R}\}$ where $\mathcal{N}$ and $\mathcal{R}$ are sets of nodes and relationships, respectively. For the ease of notation, suppose x is a primary key in $\mathcal{T}$. 3NF Equivalent Graph (3EG) transformation is a process of applying the following four rules:

- Each tuple $t \in \mathcal{T}$ becomes a node $n \in \mathcal{N}$. Each node n is identified by its table name and primary key: $\text{id}(n) = (\text{name}(\mathcal{T}), x)$ where x is from $t = (x, y)$.
- If y is a foreign key, a relationship $r \in \mathcal{R}$ is created between n and m. In this case, $\text{id}(m) = (\text{name}(\mathcal{T}_y), y)$ and $\mathcal{T}_y$ is a table where y is its primary key.
- If y is a non-key element, y becomes a property of a node n where $\text{id}(n) = (\text{name}(\mathcal{T}), x)$.
- For a set of keys in t, each pairwise relationship maps to an edge between the corresponding vertices.

Each rule has its origin as follows:

- Rule 1 specifies 1NF.
- Rule 3 is based on the 3NF requirement: about the key, the whole key, and nothing but the key.
- Rule 2 and 4 follow from the definition of "relational databases".

The 3EG-transform provides a simple set of rules, which can be instantaneously applied to any 3NF table. By applying these rules, we obtain a mirrored graph database of the existing relational database. We note that the generated graph may not be the optimal design for a specific service purpose. The generated graph only guarantees that the representation of the data is equivalent to the original 3NF database.

IV. SYNTHETIC DATA

For purposes of our evaluation, we generate realistic synthetic healthcare data. Our synthetic data is generated to be consistent with 3NF healthcare databases, and then 3EG-transformed to be loaded on a graph database.

We first generate baseline data that will be used to constitute the claim. This baseline data includes location information (as a ZIP table), provider speciality (Spec table), codes for the diagnosis of disease (ICD-9 table) [31], and the available procedures (Proc table). These tables are based on actual codes used in practice; for example, we use the full list of real ICD-9 codes to create our ICD9 table. Next, we synthesize Beneficiary (Bene) and Provider (Prov) tables. The number of beneficiaries and providers are given as parameters to our synthesizer, and the corresponding number of BeneID's and ProvID's are generated. ZIP codes and Specialty codes (only for providers) are randomly assigned to BeneID's and ProvID's. We assume that each beneficiary has 50 claims on average. Based on the number of beneficiaries, we generate 50 times larger number of claims, and assign ClaimID's. Each ClaimID is randomly associated with ICD-9, Procedure codes, and one beneficiary in the claim-beneficiary relationship (C2B) table. A beneficiary is randomly chosen from the already synthesized beneficiary pool. Choosing a beneficiary from millions of beneficiaries, however, needs a bit of attention. To avoid computational complexity, we adopt a tree structured multinomial sampling. We first sample a group of beneficiaries, then select a beneficiary from this group (two level sampling). Every ClaimID has at least one provider, and it can have five providers at maximum, and these associations are recorded in the claim-provider relationship C2P table. With 10% chance, a claim has a referral from another provider, and this information is captured in the claim-referral C2R table. This synthetic data generation configuration is illustrated in Figure 1. Finally, these relational tables are 3EG-transformed, and we obtain the equivalent graph representation.

Figure 2 shows a 3EG-transformed graph schema for typical 3NF healthcare databases. Each primary key in a table is converted to a node. For example, a BeneID in Bene table now maps to a Bene node. A foreign key in a table is connected to a primary key: a ProvID is connected to a ClaimID based on C2P table. Non-key elements in ZIP, Spec, ICD9, and Proc tables are properties of ZIP, Spec, ICD9, and Proc nodes, respectively.

Table I describes eight synthetic datasets used in our experiments. The number of nodes and relationships correspond to the number of beneficiaries and providers when the relational tables are 3EG-transformed. For example, dataset 8 is generated by specifying one million beneficiaries and 100 thousand providers, and after generating the synthetic relational tables; those tables are 3EG-transformed resulting in a total 51 million nodes and 257 million relationships.

Table I. SYNTHETIC DATASETS.

Dataset	#. Bene	#. Prov	#. Claims	#. Nodes	#. Rel.
1	8K	800	400K	425K	2M
2	16K	1.6K	800K	824K	4M
3	31K	3.1K	1.5M	1.62M	8M
4	63K	6.3K	3.1M	3.21M	16M
5	125K	12.5K	6M	7M	33M
6	250K	25K	12M	13M	65M
7	500K	50K	25M	26M	129M
8	1M	100K	50M	51M	257M

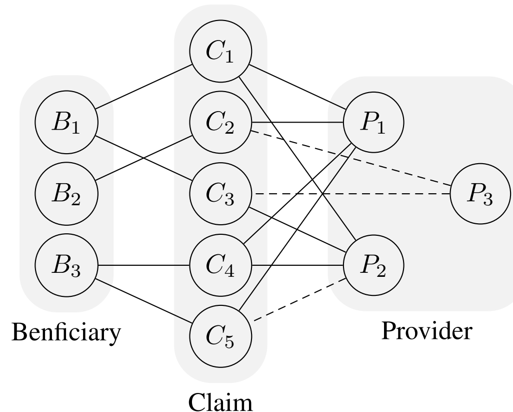

Figure 2. A Snap Shot of Graph Representation. Other types of nodes are not shown to unclutter the diagram.

V. TEST CASES

In this section, we assemble and examine eight use cases that illustrate healthcare-driven data services for analytics.

Queries in Healthcare Systems. We focus primarily on three classes of service queries - those that support direct retrieval, queries supporting relationship mining, and queries that enable personalized web services. In healthcare systems, mining and identifying relationships between different entities are critical in a variety of analytic procedures. The complex connections between the entities in the healthcare system are best exposed by these logically simpler structures like graphs that explicitly highlight underlying relationships. These explicit relationships help understand business as well as delivery structures in provider and beneficiary networks. Understanding the structures helps us better explore for areas of improvement. Personalized web services, on the other hand, have been one of the key challenges in healthcare informatics as described in [32]. The customization of healthcare can help patients make smart decisions on finding physicians, drugs, and insurance plans. We demonstrate through these use cases that similarity search and collaborative filtering can be easily implemented in graph database.

Case 1) Shared Provider. Our first query example is to find out shared providers between two beneficiaries. Figure 3 illustrates the idea. Given two beneficiaries B_a and B_b, the query should return a list of providers $\{P_x\}$ shared by these two beneficiaries.

The corresponding MySQL query is as follows:

```
SELECT tableA.provID FROM (
  SELECT provID FROM C2P INNER JOIN (
    SELECT claimID FROM C2B WHERE beneID=25869
  ) AS tmp ON tmp.claimID=C2P.claimID
) AS tableA INNER JOIN (
  SELECT provID FROM C2P INNER JOIN (
    SELECT claimID FROM C2B WHERE beneID=751751
  ) AS tmp ON tmp.claimID=C2P.claimID
) AS tableB ON tableA.provID=tableB.provID;
```

where C2P and C2B represent claim to provider and claim to

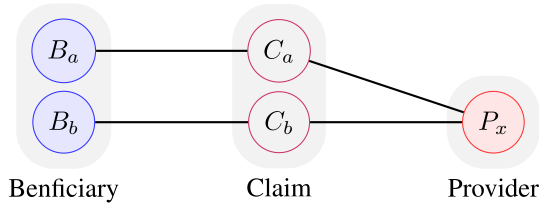

Figure 3. Case 1: A list of shared providers by Beneficiary A and Beneficiary B.

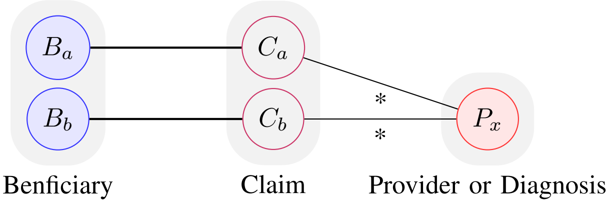

Figure 4. Cases 2, 3, and 4: A list of shared providers or diagnoses by Beneficiary A and Beneficiary B.

beneficiary tables, respectively.

On the other hand, the same Neo4J Cypher query is below:

```
START beneA=node(25869), beneB=node(751751)
MATCH beneA-->()<--prov-->()<--beneB
RETURN prov;
```

We observe that these two queries are quite different in their query lengths. The query constraints appear in "where" and "join" clauses in the MySQL query, and "match" clause in the Neo4J Cypher query. The match clause directly represents the diagram in Figure 3, while join clauses need more consideration to understand properly. The benefit of the graph database comes from not only its direct data representation and storage, but also its intuitive query structure. The compact representation of the graph query facilitates the management, user validation, and exploration of the analytic intent of the query.

Case 2) Loosely Specified Relationship. Our second query relaxes the condition in the first query. In this case, the links between claims and providers are not limited to the C2P table. We ask shared providers between two beneficiaries either by actual visits or by referrals. Figure 4 describes this use case. To ask this query in MySQL, we separately ask three different combinations of relationships: C2P+C2P, C2P+C2R, and C2R+C2R, where C2R is a claim to referral table. Each result should be stored and later union operation should be taken to prepare the final results. In Neo4J, on the other hand, we only need to modify the first query to include the referral relationship between claims and providers.

Case 3) Shared Disease. This query is a slight modification of the first query. We want to find a list of shared diseases between two beneficiaries by looking up their claim records. The provider node in Figure 4 becomes an ICD9 node in this example. This query can be used to identifying and estimating the comorbidity of a certain disease.

Case 4) Any Link between Two Entities: In this example, we find any type of links between two beneficiaries. This link can be a shared provider, shared disease code, or shared

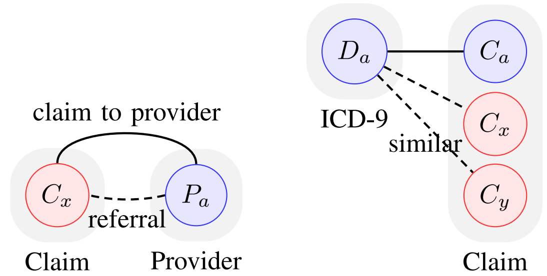

Figure 5. Self Referrals For Provider A (left) and Similar Claims based on Diagnosis (right).

procedure code. This query may discover unexpected links between two entities, providing clues for further investigation on those entities. Note that the provider node in Figure 4 is now unspecified, and can be any of these nodes.

Case 5) Shared Beneficiary: This example asks the exact same type of Case 1 question but from the providers' side. We want to find shared beneficiaries between two providers. Although this may look like repeating Case 1, its internal data operation is not quite the same. The synthetic data in our experiments have different number of beneficiaries and providers. The number of beneficiaries is ten times bigger than the number of providers in all datasets. This difference results in the difference in node degrees between beneficiaries and providers. Provider nodes typically would have ten times higher node degrees than beneficiary nodes.

Case 6) Self Referral. Given a provider, we find self-referred claims in this example. Figure 5 (left) shows this self-referral in claim records. Self-referrals have been serious problems in both private and public healthcare sectors. For example, it is reported that doctors' self-referral cost national health insurance more than 100 million dollars [33]. The self-referral diagram shown in Figure 5 (left) would be the simplest form of such malicious activities.

Case 7) Similar Record. In this use case, we find similar claims based on their diagnosis codes i.e. the same diagnosis code (see Figure 5 (right)). Even this simple looking query might consume enormous time when tables are decomposed into many and each table has a huge number of records. For example, dataset 8 in our experiments has nearly 50 million claim records. Just scanning across all the claim records can be significantly time consuming.

Case 8) Collaborative Filtering. In this example, we design a simple collaborative filtering application using a graph database. Consider a situation in which a patient experiences a new symptom D_a, and wants to find a new doctor. The patient, however, wants to find a doctor P_x who is referred by his favorite doctor P_a. Figure 6 illustrates this process. This recommendation system utilizes accumulated referral history of other claim records. This type of collaborative filtering can enhance the quality in healthcare systems, and also can provide personalized services.

VI. Experimental Evaluation

In this section, we provide experimental results based on our eight use cases. Our experiment platform is a 2.7 GHz

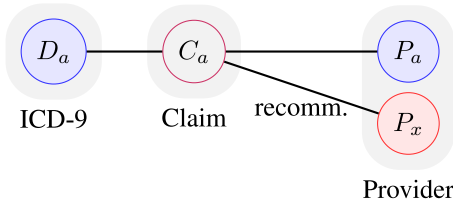

Figure 6. Collaborative Filtering using Referral History.

Intel Core i5 machine with 4 GB 1333 MHz DDR2 RAM. We use 5.5.28 MySQL Community Server for a relational database, and 1.9.M02 Neo4J Community Server. To access MySQL, we use MySQLdb Python interface [2]. For Neo4J, we use its REST API to access its data. Specifically, we use py2neo Python interface[3] and Cypher query language. Basic performance tunings for both databases are as follows. For MySQL, we used the InnoDB storage engine from MySQL, and all the tables are fully indexed (InnoDB uses B-Tree). For Neo4J, we increased the heap size of JVM to be 2GB. All the results are measured after cache warm up for both databases.

We generate 50 random queries for each case, and measure their query execution times. Test cases are sequentially executed from Case 1 to Case 8. Figure 7 and 8 show the query performance measured when 50 million nodes and 250 million relationships are loaded (dataset 8). To see whether random queries are sequentially (or temporally) dependent or not, we plot their execution time over the sequence of queries in Figure 7. Temporal dependencies may arise depending on databases' cache utilization. Except MySQL's case 2 performances, the query performance is stable across the query sequence. For Case 2, we observe that MySQL internally optimizes the Case 2 query as it gets the same type of queries. Except the first three use cases, Neo4J outperforms MySQL in its execution time. Figure 8 shows the same results from a different angle. At this time, we try to see relative query execution times between cases. Case 7 and 8 are especially slow in MySQL, while Neo4J shows a stable performance over different queries.

Figure 9 and 10 illustrate the changes in query performance for each case for increasingly sized datasets. Figure 9 shows the query scalability by case. MySQL outperforms for the first three queries in all datasets but Neo4J outperforms in the rest of the queries. Figure 10 shows the results with a different grouping. We compare the query performance within each database. We can observe that Neo4J also shows quadratic performance degradation as the size of data becomes large. However, Neo4J's speed of degradation is much smaller than MySQL's speed.

The query performance of Neo4J degrades with respect to the degree of a node, while the performance of MySQL is related to the size of joining tables. This finding becomes apparent in Figure 10. In our synthetic data generation scheme, the beneficiaries and providers have the same average degree numbers across different datasets.

$$E[\text{degree}(B)] = 50 + \alpha \text{ and } E[\text{degree}(P)] = 500 + \beta$$

[2]http://mysql-python.sourceforge.net/MySQLdb.html
[3]http://py2neo.org/

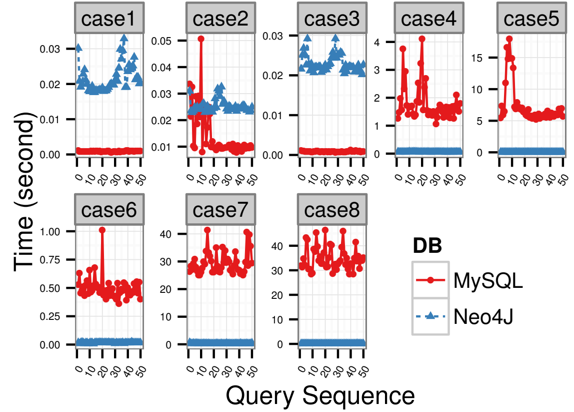

Figure 7. Sequential query performance when 50 million nodes and 250 million relationships are loaded. Except first three cases, Neo4J processes the rest of queries faster than MySQL.

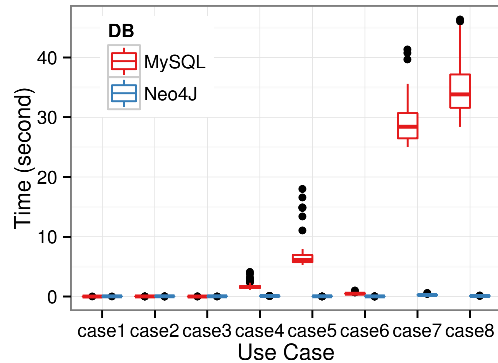

Figure 8. Performance box plot when 50 million nodes and 250 million relationships are loaded. MySQL becomes significantly slower especially in Case 7 and 8.

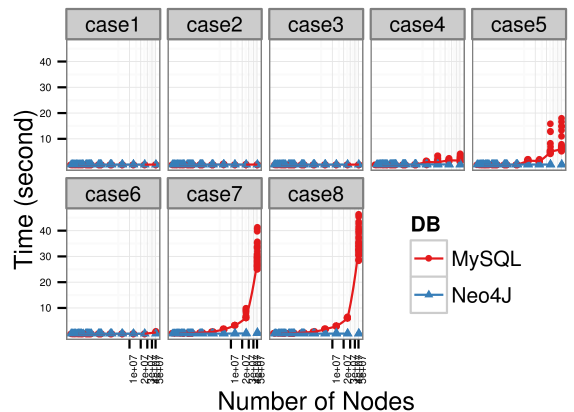

Figure 9. Data Size vs. Query Processing Time. Neo4J shows almost constant performance over a set of different sized datasets, while MySQL does not scale well in many use cases.

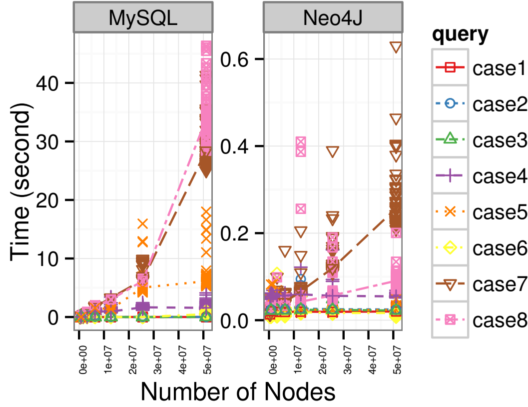

Figure 10. Data Size vs. Query Processing Time. Neo4J's performance also degrades as the size of the data grows. However, its degradation speed is much slower than MySQL's degradation speed.

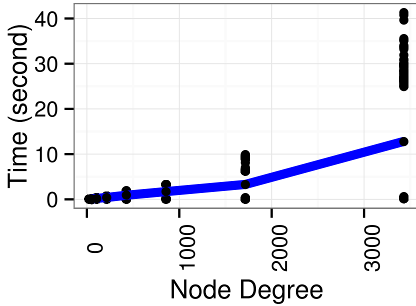

Figure 11. Node Degree vs. Query Processing Time (Case 7).

where α and β indicate the extra relationships such as disease codes and ZIP codes. Therefore, the queries involving only beneficiaries and providers show almost the same performance regardless of the data sizes. Unlike other cases, disease nodes are newly introduced in Case 7 and 8. The total number of diseases are fixed in the ICD-9 definition, thus the degree of a disease node becomes a function of the number of claims:

$$E[\text{degree}(D)] = \frac{|\mathcal{P}|}{|\mathcal{D}|} \approx \frac{|\mathcal{P}|}{15000}$$

where $|\mathcal{P}|$ and $|\mathcal{D}|$ represent the total number of providers and diseases, respectively. The denominator 15K is the total number of the ICD-9 disease codes. Figure 11 illustrates the relationship between node degree and query processing time in Case 7. This result suggests that a graph database can scale to very large-scale healthcare data because large node degrees can be accommodated; and also that the performance characteristics can be revealed by the graph parameters such as vertex degree.

Since these eight queries are formulated to explore the links and relationships between individual entities, an argument can be made that they are biased toward graph-like queries for healthcare systems. Focussing on explicitly non-graph-friendly queries is left for future work.

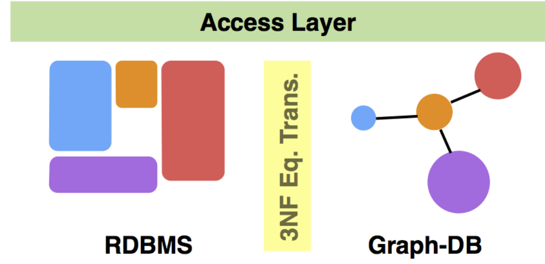

Figure 12. Cost Efficient Ensemble Framework. A graph database is mirrored using the 3EG transform. Access layer mediates data users to access both RDBMS and graph databases depending on their objectives.

VII. Discussion and Ensemble Framework

Graph Database as a De-normalized Tables. The experimental results in Section VI suggests that a graph database can handle a wide range of graph queries even with big data. In a relational database, these queries require heavy join operations even with full indexing. On the other hand, a graph database suited for this type of query processing avoids explicit intermediate and dummy table creation. A graph database generated using the proposed 3EG-transformation may be able to serve as hundreds of different de-normalized tables.

Ensemble Framework. Compared to the rich history and verified stability of relational databases, graph database technology is still relatively new. The use cases in this paper focus on queries that are amenable to graph databases. However, other query classes will still rely on the foundational database operations such as select, project, union, and set difference, which are extensively optimized in relational databases. Furthermore, relationship databases are extensively optimized for classic services like sorting, listing, and all pairs' joins.

Since each database model has its own strength and weakness, we anticipate the development of a pragmatic ensemble approach in healthcare systems. Figure 12 illustrates this basic approach. This approach uses the 3EG transform to automatically construct a graph database from an existing normalized relational database. The access layer guides data users to refer to either or both the RDBMS and graph databases depending on their objectives. This concept is not in itself new because many existing healthcare enterprise data systems are built on layers of interdependent databases [34]. The graph database component will be an additional resource in such enterprise architectures and can even be placed in a decentralized data warehousing environment [35]. Although more work is required to formulate the access layer's service and query routing strategies, we firmly believe that such an approach can substantially help both data management and data services in healthcare systems.

Acknowledgment

This work is sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory. Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the U.S. D.O.E. under the contract no. DE-AC05-00OR22725.

References

[1] C. Meier, “A role for data: An observation on empowering stakeholders,” *American Journal of Preventive Medicine*, vol. 44, no. 1, pp. S5–S11, 2013.

[2] Institute of Medicine, “Transformation of health system needed to improve care and reduce costs,” Press Release, 9 2012.

[3] D. M. Cutler, K. Davis, and K. Stremikis, “The Impact of Health Reform on Health System Spending,” *Commonwealth Fund Issue Brief*, 2010.

[4] D. M. Berwick and A. D. Hackbarth, “Eliminating Waste in US Health Care,” *The Journal of the American Medical Association*, vol. 307, no. 14, pp. 1513–1516, 2012.

[5] C. J. Date, *An Introduction to Database Systems*. Addison-Wesley, 2000.

[6] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill, 2002.

[7] Z. Wei, J. Dejun, G. Pierre, C.-H. Chi, and M. van Steen, “Service-Oriented Data Denormalization for Scalable Web Applications,” in *Proceedings of the 17th International World Wide Web Conference*, 2008, pp. 267–276.

[8] G. L. Sanders and S. Shin, “Denormalization Effects on Performance of RDBMS,” in *Proceedings of the 34th Hawaii International Conference on System Sciences*, vol. 3, 2001, p. 3013.

[9] J. Ravi, Z. Yu, and W. Shi, “A survey on dynamic Web content generation and delivery technique,” *Journal of Network and Computer Applications*, vol. 32, no. 5, pp. 943–960, 2009.

[10] R. Angles and C. Gutierrez, “Survey of Graph Database Models,” *ACM Computing Surveys*, vol. 40, no. 1, 2008.

[11] A. Khan, Y. Wu, and X. Yan, “Emerging graph queries in linked data,” in *Proceedings of IEEE 28th International Conference on Data Engineering*, 2012, pp. 1218–1221.

[12] B. Shao, H. Wang, and Y. Xia, “Managing and mining large graphs: Systems and implementations,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 589–592.

[13] Neo4J. [Online]. Available: http://www.neo4j.org/

[14] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A System for Large-Scale Graph Processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[15] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[16] ——, “Further normalization of the data base relational model,” *IBM Research Report RJ909*, 1971.

[17] R. Fagin, “Multivalued dependencies and a new normal form for relational databases,” *ACM Transactions on Database Systems*, vol. 2, no. 1, 1977.

[18] ——, “Normal forms and relational database operators,” in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, 1979, pp. 153–160.

[19] E. F. Codd, “Data Models in Database Management,” in *Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling*, 1980, pp. 112–114.

[20] P. P. shan Chen, “The Entity-Relationship Model: Toward a Unified View of Data,” *ACM Transactions on Database Systems*, vol. 1, pp. 9–36, 1976.

[21] R. H. L. Chiang, T. M. Barron, and V. C. Storey, “Reverse engineering of relational databases: Extraction of an EER model from a relational database,” *Data and Knowledge Engineering*, vol. 12, pp. 107–142, 1994.

[22] V. M. Markowitz and J. A. Makowsky, “Identifying extended entity-relationship object structures in relational schemas,” *Transactions on Software Engineering*, vol. 16, no. 8, 1990.

[23] W. J. Premerlani and M. R. Blaha, “An Approach for Reverse Engineeing of Relational Databases,” *Communications of the ACM*, vol. 37, 1994.

[24] C. Fahrner and G. Vossen, “A survey of database design tranformations based on the entity-relationship model,” *Data and Knowledge Engineering*, vol. 15, 1995.

[25] A. Maatuk, A. Ali, and N. Rossiter, “Relational database migration: A perspective,” *Database and Expert Systems Applications*, vol. 5181, 2008.

[26] D. Lee, M. Mani, and W. W. Chu, “Effective Schema Conversions between XML and Relational Models,” in *Proceedings of European Conference on Artificial Intelligence (ECAI)*, 2002.

[27] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. T. Jr., S. Auer, J. Sequeda, and A. Ezzat, “A survey of current approaches for mapping of relational databases to rdf,” *World Wide Web Consortium Tech Report*, 2009, http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf.

[28] T. Berners-Lee, “Relational databases on the semantic web,” 1998, http://www.w3.org/DesignIssues/RDB-RDF.html.

[29] Neo4J. [Online]. Available: http://blog.neo4j.org/2012/02/webinar-follow-up-how-to-get-started.html

[30] W. Kent, “A Simple Guide to Five Normal Forms in Relational Database Theory,” *Communications of the ACM*, vol. 26, no. 2, 1983.

[31] World Health Organization, “International classification of diseases (ICD),” 1994, http://www.who.int/classifications/icd/en/.

[32] H. U. Prokosch and T. Ganslandt, “Perspectives for Medical Informatics,” *Methods of Information in Medicine*, vol. 48, no. 1, pp. 38–44, 2009.

[33] United States Government Accountability Office, “Higher Use of Advanced Imaging Services by Providers Who Self-Refer Costing Medicare Millions.” [Online]. Available: http://www.gao.gov/assets/650/648989.pdf

[34] C. G. Chute, S. A. Beck, T. B. Fisk, and D. N. Mohr, “The enterprise data trust at mayo clinic: a semantically integrated warehouse of biomedical data,” *Journal of the American Medical Informatics Association*, vol. 17, no. 2, pp. 131–135, 2010.

[35] S. Hanß, T. Schaaf, T. Wetzel, C. Hahn, T. Schrader, and T. Tolxdorff, “Integration of Decentralized Clinical Data in a Data Warehouse,” *Methods of Information in Medicine*, vol. 48, no. 5, pp. 414–418, 2009.