A Denormalization Approach to Answering Join Queries

Mohammed Hamdi[1], Kavya Narne[1], Feng Yu[2], Nourah Janbi[3],
Yousef Alsenani[1], and Wen-Chi Hou[1]
Department of Computer Science, Southern Illinois University,
Carbondale, IL, USA
{mhamdi, kavya, yalsenani, hou}@cs.siu.edu
Department of Computer Science and Information Systems,
Youngstown State University
Youngstown, OH, USA
fyu@ysu.edu
Department of Information Technology, Faculty of Computers and IT – Khulais
University of Jeddah, Jeddah, Saudi Arabia
nfjanbi@ju.edu.sa

***Abstract*— Relational databases may not be an efficient solution to store highly connected data. Moreover, graph traversals over high-connected data require complex join operations. These join operations are generally very expensive and hard to compute. For these reasons, a new and efficient data structure, called Join Core for fast join query processing is emerging. In this paper, we present a methodology to store equi-join relationships of tuples on inexpensive and space abundant devices, such as disks, to facilitate query processing. The equi-join relationships are captured, grouped, and stored as various tables on disks. The approach assists the join queries to be answered quickly by merely merging these tables without having to perform expensive joins. As Join Core and Neo4j graph database deal with highly connected data, experiments are performed to compare the time and space consumptions between them. Preliminary experimental results showed that Join Core outperforms Neo4j when complex queries are processed.**

Keywords—Query Processing, Join Queries, Graph Databases, Equi-Join.

I. INTRODUCTION

In many applications such as semantic web, social and computer networks, and in geographic applications data are more and highly connected and have a natural representation as a graph. In these contexts, relational databases may not be suitable for those highly connected data where data are spread among relations, and it is hard to capture and group the join relationships among data over traditional systems [24]. Moreover, graph traversals over high-connected data involve complex join operations [24]. These join operations are generally very expensive and hard to compute. Complex queries involving multiple joins of large relations can easily take minutes or even hours to compute over the target database.

For the above reasons, an anti-relational approach, called Join Core, has been proposed. Join Core pre-stores the equi-join relationships of tuples to facilitate query processing. We have designed a simple method to capture the equi-join relationships in the form of maximally extended match tuples. A simple and novel naming technique has been designed to group and store the equi-join relationships in tables on disks.

Vérifier avec BD graph

In this research, we develop a fast join query processing technique that can be several orders of magnitude faster than the traditional techniques. This technique also can ease the job of the query optimizer because there are fewer or no joins to perform and provide less resources consumptions, e.g., CPU and memory. A number of experiments have done to compare the performance of Join Core and Neo4j. The experimental results show that processing queries with Join Core is faster than with Neo4j. This is because there is no need to perform join operations at run time with Join Core while in Neo4j, the path traversal operations depend upon the complexities of the relationships of tuples

In this paper, we also propose effective methods that can significantly reduce the space consumption of the Join Cores. We believe the benefits of Join Core, namely instant responses, fast query processing, and small memory consumptions, are well worth the additional storage space incurred.

The rest of the paper is organized as follows. Section II surveys work in materialized views and Section III introduces the terminology. Section IV shows a sample Join Core and how it can be used to answer equi-join queries. Section V lays down the theoretical foundation for answering equi-join queries using the Join Core. Section VI extends the framework to queries with other types of joins and set operations. Section VII analyzes the time and space consumptions of the Join Core, and discusses measures to reduce the space consumption. Section VIII reports experimental results. Finally, conclusions are presented in Section IX.

Due to space limitation, readers are referred to [22] for an extended version of the paper that includes detailed discussions on dynamic maintenance of the Join Core, proofs of theorems, applications to bag semantics, literature survey, and experimental results.

II. LITERATURE SURVEY

In this section we discuss briefly the literature survey. Materialized views, join indices, and graph databases are related to our work as both attempt to pre-compute data to facilitate query processing.

Materialized views generally focus on SPJ (Select-Project-Join) queries and, perhaps, with final grouping and aggregate functions. The select and project operations in the views confine and complicate the uses of the views. As a result, much research has focused on how to select the most beneficial views to materialize [19], [15], [8], [10] and how to choose an appropriate set of materialized views to answer a query [9], [1], [16].

Materialized views materialize selected query results while Join Core materializes selected equi-join relationships. Therefore, materialized views may benefit queries that are relevant to the selected queries, while Join Core can benefit queries that are related to the selected equi-join relationships, which include queries with arbitrary sequences of equi-, semi-, outer-, anti-joins and set operators.

A join index [14], [21] for a join stores the (equi-)join result in a concise manner as pairs of identifiers of tuples that would match in the join operation. It has been shown that joins can be performed more efficiently with join indices than the traditional join algorithms. However, it still requires at least one scan of the operand relations, writes and reads of temporary files (as large as the source relations), and generating intermediate result relations (for queries with more than one join). One the other hand, with Join Core, join results are readily available without accessing any source or intermediate relation. Very little memory and computations are required. In addition, join indices are not useful to other join operators, such as outer-joins and anti-joins.

Graph databases use the graph data model to structure and perform the main database systems operations (Create, Read, Update, and Delete). Graph data model has two basic elements: node and relationship. Unlike the relational databases, the graph databases store the relationships as entities which make it more flexible and scalable. This is because when the data model expands or business requirement changes, it is easier to add connection (relationship) between entities [24].

Graph databases also use the graph model to pre-store the join relationships of tuples and query connections at creation time and make them readily available for any later join query operation [24]. This can result no penalties for complex join queries at runtime as the Join Core does. They use the index-free so that the query processing time depends on the searched graph length rather than the total size of the graph. However, the path traversal operations in the complex relationships of nodes sometimes decelerate the query processing time. In contrast, the result size of the query, not the complexity of join query determines the query processing time with Join Core.

III. TERMINOLOGY

In this paper, we assume all the data model and queries are based on the set semantics. Readers are referred to an extended version of the paper [22] for discussions on the bag semantics. The equi-join operator is the most commonly used operator to combine data spread across relations. Other useful joins, such as the semi-join, outer-join, and anti-join, are all related to the equi-join. Therefore, we shall first lay down the theoretical foundation of Join Core based on the equi-join, and then extend the framework to other joins in Section 6. Hereafter, we shall use, for simplicity, a join for an equi-join, unless otherwise stated.

A join graph is commonly used to describe the equi-join relationships between pairs of relations. These relationships are generally defined before the database has been created. Certainly, one can also include other frequently referenced ad-hoc equi-join relationships in the graph.

For simplicity, we assume there is at most one equi-join relationship between each pair of relations. This assumption is relaxed in [22].

Definition 1. (Join Graph of a Database). Let D be a database with n relations $R_1, R_2, \ldots, R_n$, and $G(V, E)$ be the join graph of D, where V is a set of nodes that represents the set of relations in D, i.e., $V = \{R_1, R_2, R_3, ..., R_n\}$, and $E = \{\langle R_i, R_j\rangle \mid R_i, R_j \in V, i \neq j)\}$, is a set of edges, in which each represents an equi-join relationship that has been defined between R_i and R_j, $i \neq j$.s

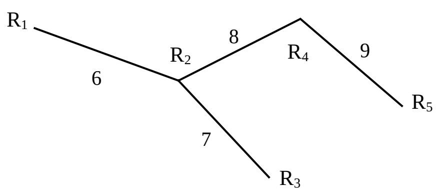
If the join graph is not connected, one can consider each connected component separately. Therefore, we shall assume all join graphs are connected.

Each join comes with a predicate, omitted in the graph, specifying the requirements that a result tuple of the join must satisfy, e.g., $R_1.attr1=R_2.attr2$. For simplicity, we shall use a join, a join edge, and a join predicate interchangeably. We also assume all relations and join edges are numbered.

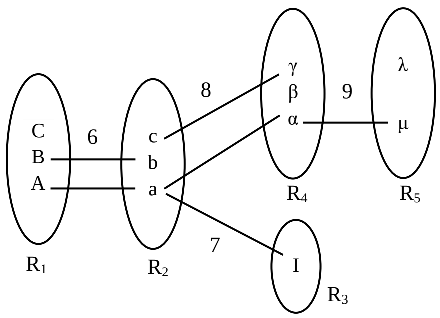
Example 1. (*Join Graph*). Fig. 1(a) shows the join graph of a database with five relations R_1, R_2, R_3, R_4, and R_5, connected by join edges, numbered from 6 to 9.

To round out the theoretical framework, we shall introduce a concept, called the *trivial* (*equi-*)*join*. Each tuple in a relation R_i can be considered as a result tuple of a trivial join between R_i and itself with a join predicate $R_i.key = R_i.key$, where *key* is the (set of) key attribute(s) of R_i. Trivial join predicates are not shown explicitly in the join graphs. All join edges in Fig. 1(a), such as 6, 7, 8, and 9, are non-trivial or regular joins.

We have reserved predicate number i, $1 \leq i \leq 5$, for trivial join predicate i, which is automatically satisfied by every tuple in relation R_i. The concept of trivial join predicates will be useful later when we discuss a query that contains outer-joins, anti-joins, or no joins. Hereafter, all joins and join predicates refer to non-trivial ones, unless otherwise stated.



(a) A Join Graph



(b) Matching of Join Attribute Values

Figure 1. A Join Graph and Matching Tuples

To conserve space, a database and its join graph refer to only the parts of the database and join graphs that are of our interest and for which we intend to build Join Cores. We will discuss other space conservation measures in Section 8.

Definition 2. (Join Queries). Let $\bowtie(\{R_i, \ldots, R_j\}, E')$ be a join query, representing joins of the set of relations $\{R_i, \ldots, R_j\} \subseteq V$, $1 \leq i, \ldots, j \leq n$, with respect to the set of join predicates $E' \subseteq E$ among them.

Definition 3. (Join Graph of a Join Query). The join graph of a join query $\bowtie(\{R_i, \ldots, R_j\}, E')$, denoted by $G'(V', E')$, is a connected subgraph of $G(V, E)$, where $V' = \{R_i, \ldots, R_j\} \subseteq V$, and $E' \subseteq E$ is the set of join predicates specified in the query.

The join graph of a join query is also called *a query graph*. We shall exclude queries that must execute Cartesian products or θ-joins, where θ ≠ "=", from discussion as Join Core cannot facilitate executions of such operators.

Example 2. (*Matching of Join Attribute Values*). Fig. 1(b) shows the matching of join attribute values between tuples. Tuples are represented by their IDs in the figure. That is, R_1 has 3 tuples, A, B, C, i.e., $R_1 = \{A, B, C\}$. $R_2 = \{a, b, c\}$, $R_3 = \{I\}$, $R_4 = \{\alpha, \beta, \gamma\}$, $R_5 = \{\mu, \lambda\}$.

The edges between tuples represent matches of join attribute values. For example, tuples A and B of R_1 match tuples a and b of R_2, respectively. Tuple a has two other matches, I of R_3 and α of R_4. c of R_2 matches γ of R_4, and α matches μ of R_5.

Definition 4. ((Maximally) Extended Match Tuple). Given a database $D = \{R_1, \ldots, R_n\}$ and its join graph G, an extended match tuple $(t_k, \ldots, t_l)$, where $1 \leq k, \ldots, l \leq n$, $t_k \in R_k$, …, $t_l \in R_l$, and $R_k, \ldots, R_l$ are all distinct relations, represents a set of tuples $\{t_k, \ldots, t_l\}$ that generates a result tuple in $\{t_k\} \bowtie \ldots \bowtie \{t_l\}$. A maximally extended match tuple $(t_k, \ldots, t_l)$, is an extended match tuple if no tuple t_m in R_m ($\notin \{R_k, \ldots, R_l\}$) matches any of the tuples $t_k, \ldots, t_l$ in join attribute values.

It can be observed that in Fig. 1(b), (A, a, I, α, μ) is a maximally extended match tuple. The same can be said of (B, b) because the match cannot be extended by any tuple in relations other than R_1 and R_2. Similarly, (c, γ), as well as (C), (β), and (λ), is also a maximally extended match tuple.

IV. JOIN CORE STRUCTURE AND CONSTRUCTION

In this section, we show an example of a Join Core and explain how it is structured and used to answer equ-join queries.

A. Join Core Structure and Naming

Consider Fig. 1 again. The join relationships we wish to store are (A, a, I, α, μ), (B, b), (c, γ), (C), (β), and (λ), each representing a maximally extended match tuple. We intend to store these maximally extended match tuples in various tables based on the join predicates, both trivial and non-trivial ones, they satisfy. These tables form the *Join Core*.

Example 3. (*Sample Join Core*). Fig. 2 shows the Join Core for the database in Fig. 1. The attributes of the Join Core tables, i.e., 1, 2, 3, 4, and 5, represent the sets of (interested) attributes of R_1, R_2, R_3, R_4, and R_5, respectively, and are called the R_1, R_2, …, R_5 components of the tables.

(B, b) is stored in $J_{1,2,6}$ because (B, b) satisfies join predicate 6, and trivial predicates 1 ($B \in R_1$) and 2 ($b \in R_2$). Similarly, (c, γ) is stored in $J_{2,4,8}$ and (A, a, I, α, μ) is stored in $J_{1,2,3,4,5,6,7,8,9}$. C ($\in R_1$), β ($\in R_4$), and λ ($\in R_5$) satisfy only trivial predicates and thus are stored in J_1, J_4, and J_5, respectively.

Assume join predicate numbers 1, …, n are reserved for trivial joins between $R_1, \ldots, R_n$ and themselves, respectively, and non-trivial predicates are numbered from n+1 to n+e, where e is the number of join edges in the join graph.

Definition 5. (Join Core). A join Core is composed of a set of tables $J_{k, \ldots, l}$, $1 \leq k, \ldots, l \leq n+e$, each of which stores a set of maximally extended match tuples that satisfy *all and only* the join predicates $k, \ldots, l$. Each table $J_{k, \ldots, l}$ is called a *Join Core table* (or *relation*). The indices $k, \ldots l$ of the table $J_{k, \ldots, l}$ is called the name of the table for convenience.

For simplicity, we shall call a maximally extended match tuple in a Join Core table a match tuple, to be differentiated from a tuple in a regular relation.

1
C

J_1

4
β

J_4

5
λ

J_5

1	**2**
B	b

$J_{1,2,6}$

2	**4**
c	γ

$J_{2,4,8}$

1	**2**	**3**	**4**	**5**
A	a	I	α	μ

$J_{1,2,3,4,5,6,7,8,9}$

Figure 2. Join Core

B. Answering Queries using Join Core

The name of a Join Core table specifies the join predicates satisfied by the match tuples stored in it. On the other hand, a join query specifies predicates that must be satisfied by the result tuples. Therefore, to answer a query is to look for Join Core tables whose names contain the predicates of the query.

Consider Fig.1 and 2 and the query $\bowtie(\{R_1, R_2, R_3, R_4, R_5\}, \{6, 7, 8, 9\})$. The components of the result tuples must satisfy predicates 6, 7, 8, and 9. In addition, the components themselves also satisfy trivial predicates 1, 2, 3, 4, 5. Thus, we look for Join Core tables whose names contain predicates 1, 2, 3, 4, 5, 6, 7, 8, and 9. That is, $\bowtie(\{R_1, R_2, R_3, R_4, R_5\}, \{6, 7, 8, 9\}) = J_{1,2,3,4,5,6,7,8,9}$.

As for $\bowtie(\{R_1, R_2\}, \{6\})$, while $J_{1,2,6}$ certainly contains some result tuples, $J_{1,2,3,4,5,6,7,8,9}$ also contains some result tuples because tuples in $J_{1,2,3,4,5,6,7,8,9}$ also satisfy 1, 2, and 6. That is, $\bowtie(\{R_1, R_2\}, \{6\}) = \pi_{1,2}(J_{1,2,6}) \cup \pi_{1,2}(J_{1,2,3,4,5,6,7,8,9})$. Similarly, $\bowtie(\{R_2, R_4\}, \{8\}) = \pi_{2,4}(J_{2,4,8}) \cup \pi_{2,4}(J_{1,2,3,4,5,6,7,8,9})$; $\bowtie(\{R_2, R_3\}, \{7\}) = \pi_{2,3}(J_{1,2,3,4,5,6,7,8,9})$.

It even holds for queries containing no non-trivial joins. For example, $R_1 = \pi_1 J_1 \cup \pi_1(J_{1,2,6}) \cup \pi_1(J_{1,2,3,4,5,6,7,8,9})$, $R_2 = \pi_2(J_{1,2,6}) \cup \pi_2(J_{2,4,8}) \cup \pi_2(J_{1,2,3,4,5,6,7,8,9})$, $R_3 = \pi_3(J_{1,2,3,4,5,6,7,8,9})$, $R_4 = \pi_4 J_4 \cup \pi_4(J_{2,4,8}) \cup \pi_4(J_{1,2,3,4,5,6,7,8,9})$, and $R_5 = \pi_5 J_5 \cup \pi_5(J_{1,2,3,4,5,6,7,8,9})$. It is observed that R_i can be reconstructed from the Join Core, which implies that a Join Core can itself be the database, if one wishes to not store the relations in traditional ways.

Notice that when a non-trivial join predicate, such as 6, is satisfied by a match tuple, the associated trivial predicates on its operand relations, i.e., 1 and 2, are also satisfied automatically. Therefore, there is no need to match the trivial predicates of a query with the Join Core table names. That is, given a join query with a non-empty set of predicates $\{u, \ldots, v\}$, the result tuples can be found in Join Core tables whose names contain $u, \ldots, v$, without regard to trivial predicates. Trivial predicates cannot be ignored when a query contains no non-trivial joins, such as those described above or contains outer- or anti-joins, discussed later.

Duplicates need not be eliminated in individual $\pi_{i, \ldots, j}(J_{k, \ldots, l})$ above; they can be eliminated all at once when match tuples are merged in the final union operations. To identify duplicate result tuples, a simple hashing scheme is sufficient. Note that this is the only place that requires major memory consumption (in building a hash table).

The database system can begin to generate result tuples once the first block of a relevant Join Core table is read into memory, that is, instantly. The total computation time is also drastically reduced because there are no (or fewer) joins to perform.

C. Join Core Construction

Now, let us discuss how to construct a Join Core for a database. Tuples that find no match in one join may find matches in another join. For example, b finds no match in $R_2 \bowtie R_3$, but finds a match B in $R_1 \bowtie R_2$. Unfortunately, such join relationships can be lost in successive joins, for example, in $(R_1 \bowtie R_2) \bowtie R_3$.

Full outer-joins, or simply outer-joins, retain matching tuples as well as dangling tuples, and thus can capture all the join relationships. Any graph traversal method can be used here as long as it incurs no Cartesian products during the traversal.

For illustrative purpose, we assume a breadth-first traversal is adopted here. Relations are numbered based on the order encountered in the traversal. An outer-join is performed for each join edge. The output of the previous outer-join is used as an input to the next outer-join. The result tuples are distributed to Join Core tables based on the join predicates, both trivial and non-trivial ones, they have satisfied in the traversal.

Example 4. (*Join Core Construction*). Assume a breadth-first traversal of the join graph (Fig. 1(a)) from R_1 is performed. An outer-join is first performed between R_1 and R_2. It generates (intermediate) result tuples (A, a), (B, b), (C, -), and (-, c). The next outer-join with R_3 generates (A, a, I), (B, b, -), (C, -, -) and (-, c, -). Then, the outer-join with R_4 generates (A, a, I, α), (B, b, -, -), (C, -, -, -), (-, c, -, γ), and (-, -, -, β). The final outer-join with R_5 generates (A, a, I, α, μ), (B, b, -, -, -), (C, -, -, -, -), (-, c, -, γ, -), (-, -, -, β, -), and (-, -, -, -, λ), which are written, without nulls, to $J_{1,2,3,4,5,6,7,8,9}$, $J_{1,2,6}$, J_1, $J_{2,4,8}$, J_4, and J_5, respectively, based on the join predicates they satisfy.

V. ANSWERING EQUI-JOIN QUERIES

In this section, we formally discuss how a join query can be answered using the Join Core. First, we consider databases with acyclic join graphs, followed by databases with cyclic join graphs.

A. Acyclic Join Graph

As illustrated in the previous section, join queries with acyclic join graphs can be answered by simply extracting the requested components from Join Core tables whose names contain the join predicates specified in the queries.

Readers are referred to [18] for formal proofs of all the theorems.

*Theorem 1***.** Let $\bowtie(\{R_i, \ldots, R_j\}, \{u, \ldots, v\})$ be joins of the set of relations $\{R_i, \ldots, R_j\}$ with respect to a set of join predicates $\{u, \ldots, v\} \neq \varnothing$. Let e be the number of join edges in the join graph,

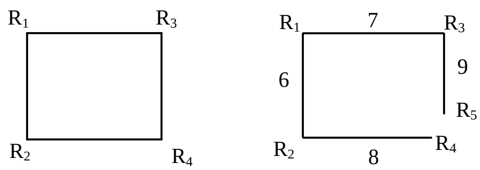
$\bowtie(\{R_i, \ldots, R_j\}, \{u, \ldots, v\}) = \cup_{\{k, \ldots, l\} \supseteq \{u, \ldots, v\}} \pi_{i, \ldots, j}(J_{k, \ldots, l})$

where $1 \le i, \ldots, j \le n$, $1 \le k, \ldots, l, u, \ldots, v \le n+e$.

Here, we shall call $\{k, \ldots, l\} \supseteq \{u, \ldots, v\}$ or equivalently, $k \in \{u, \ldots, v\} \wedge \ldots \wedge l \in \{u, \ldots, v\}$ shall be called (*table name*) *selection criteria*.

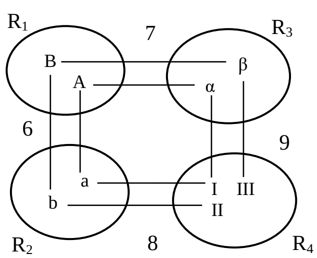
B. Cyclic Join Graph

Fig. 3(a) shows a cyclic join graph. When a relation is visited in a, for example, breadth-first traversal, its attributes are added to the resulting schema. In a cyclic join graph however, a node may be visited more than once. For example, R_4 is visited through edge $\langle R_2, R_4\rangle$ for the first time, and then through $\langle R_3, R_4\rangle$ for the second time when the cycle forms. To differentiate matches associated with different edges, we shall create two copies of R_4, named R_4 (the original name) and R_5 (the next available relation number). Note that this is effectively converting a cyclic graph into an acyclic one. We shall call all copies of R_4, i.e., R_4 and R_5, alias relations of R_4. Note that a *cycle-completing* relation, such as R_4, may replicate more than once if it completes more than one cycle in the traversal. Fig. 3(b) shows the converted graph.



(a) A Cyclic Join Graph (b) A Converted Join Graph

Figure 3. Converting A Cyclic Graph



(a) Cyclic Join Relationship

1	2	3	4	5
A	*a*	*α*	*I*	*I*
B	*b*	*β*	*II*	*III*

$J_{1,2,3,4,5,6,7,8,9}$

(b) Join Core Tables

Figure 4. Cyclic Join Relationship and Join Core

With a cyclic join graph converted into an acyclic one, a Join Core can be constructed in the same way as before. However, to determine whether an extended match tuple contains a cycle or not, we need to check if the alias components have the same value.

Example 5. (*Answering Cyclic Join Queries*). Fig. 4 shows the join relationships and the Join Core for Figure 3. Consider a cyclic join query: $\bowtie(\{R_1, R_2, R_3, R_4\}, \{6, 7, 8, 9\})$. To ensure that it is the same tuple in the cycle-completing relation that satisfies both predicates 8 and 9, the alias components R_4 and R_5 must be the same. That is, a selection condition, $\sigma_{4=5}$, must be imposed. Thus, $\bowtie(\{R_1, R_2, R_3, R_4\}, \{6, 7, 8, 9\}) = \pi_{1,2,3,4}(\sigma_{4=5}(J_{1,2,3,4,5,6,7,8,9})) = \{(A, a, \alpha, I)\}$. On the other hand, (B, b, β, II, III) does not contain an answer to the query because its R_4 and R_5 components (i.e., *II* and *III*) are not the same.

Consequently, cycles in a query graph can be treated like ordinary acyclic join predicates, with the exception that additional constraints on the equalities of alias components must be added.

Theorem 2. Let $\bowtie(\{R_i, \ldots, R_j\}, \{u, \ldots, v\})$, $1 \le i, \ldots, j \le n$, be a query contains cycles.
$\bowtie(\{R_i, \ldots, R_j\}, \{u, \ldots, v\}) = \cup_{\{k, \ldots, l\} \supseteq \{u, \ldots, v\}} \pi_{i, \ldots, j}(\sigma_F(J_{k,\ldots,l}))$

Readers are referred to [22] for discussions on more complicated issues for cyclic queries.

C. Multiple Join Edges Between Relations

It is possible that there is more than one join edge between a pair of relations. This situation can be easily resolved by treating it as a cycle.

Example 6. (*Multiple Edges between Relations*) Assume there are two join edges, e_1 and e_2, between R_1 and R_2. Then, one can pick any relation, say R_2, as the cycle completing relation, replicate it, and call the replica R_3. Finally, let e_1 be the edge between R_1 and R_2, and e_2 be the edge between R_1 and R_3.

VI. QUERIES WITH OTHER JOINS

Now, a join can be an equi-, semi-, outer- or anti-join. A join generates result tuples dependent upon whether the equi-join predicate between the operand relations are satisfied (in an equi- or semi-join) or not satisfied (in an anti-join). A little deliberation reveals that match tuples that do not satisfy an equi-join predicate can be found in Join Core tables whose names do not contain that predicate, recalling that Join Core table names specify *all and only* the equi-join predicates satisfied. An outer-join generates a result tuple no matter whether the equi-join predicate is satisfied or not.

A join query consisting of a sequence of join operators has a *query predicate* that is a logical combination of the individual predicates of constituent joins. We attempt to obtain query result tuples from Join Core tables whose names satisfy the query predicates. Here, we focus on how to formula the query predicates as (*table name*) *selection criteria* for Join Core tables that contain the query result tuples. For example, satisfying predicate p is rewritten as $p \in \{k, \ldots, l\}$, where $\{k, \ldots, l\}$ is the set of indices of a Join Core table name.

Afterward, specific handlings, such as removal of unwanted attributes, equality checking for alias components (for cycle-completing relations), and padding null values for "missing" attributes (for outer-joins), are performed. For simplicity, we shall only briefly describe these afterward handlings.

A. Single-Join Queries

We start by deriving the selection criteria, denoted by S, for queries with only one join operator. Let p be the equi-join predicate between R_i and R_j. Consider R_i *op* R_j, where *op* is either an equi-join, semi-join, outer-join, or anti-join.

1) Equi-Join. As discussed, to compute $R_i \bowtie R_j$ with a join predicate p, we look for Join Core tables $J_{k,\ldots,l}$ whose indices contain p, i.e., $S = p \in \{k, \ldots, l\}$. As mentioned, trivial predicates i and j need not, but can, be included in S because they are satisfied automatically and must have appeared as part of the names of the tables satisfying p.

2) Semi-Join. The left semi-join $R_i \ltimes R_j$ and right semi-join $R_i \rtimes R_j$ extract only the R_i and R_j components from $R_i \bowtie R_j$, respectively. Here, we shall not be concerned about the projection operations. Consequently, the selection criterion S for a semi-join is the same as that for an equi-join, that is, $S = p \in \{k, \ldots, l\}$.

3) Outer-Join. While computing R_i ⟗ R_j during the construction of the Join Core, each pair of tuples satisfying predicate p forms an output tuple. In addition, each non-matching tuple from either R_i (satisfying the trivial predicate i) or R_j (satisfying the trivial predicate j) also forms an output tuple. Consequently, to answer the query R_i ⟗ R_j, we look for Join Core tables $J_{k,\ldots,l}$ such that $(i \in \{k, \ldots, l\} \wedge (\neg (p \in \{k, \ldots, l\}))) \vee (j \in \{k, \ldots, l\} \wedge (\neg(p \in \{k, \ldots, l\}))) \vee p \in \{k, \ldots, l\}$, where $\neg$ is the logical "not" operator and $\vee$ is the logical "or" operator. Since $p \in \{k, \ldots, l\}$ implies $i \in \{k, \ldots, l\} \wedge j \in \{k, \ldots, l\}$, the selection criteria S can be simplified to $S= i \in \{k, \ldots, l\} \vee j \in \{k, \ldots, l\}$. Trivial predicates i and j cannot be omitted from S because no non-trivial predicates that reference i and j are satisfied.

A left outer-join R_i ⟕ R_j asks for matching tuple pairs and non-matching tuples from R_i. Therefore, $S= i \in \{k, \ldots, l\}$. Similarly, for a right outer-join R_i ⟖ R_j, $S=j \in \{k, \ldots, l\}$.

After identifying the Join Core tables, tuples that do not find a match in the other operand relation need to be padded with null values for those attributes of the other relation.

Example 7. (*Outer-Join*). Let us consider Fig. 1 and 2.

R_1 ⟗ R_2: $S= 1 \in \{k, \ldots, l\} \vee 2 \in \{k, \ldots, l\}$. Only J_1, $J_{1,2,6}$, $J_{2,4,8}$, and $J_{1,2,3,4,5,6,7,8,9}$ satisfy S. The answer is $\{(C, \text{-}), (B, b), (\text{-}, c)\ (A, a)\}$. Note that tuples in J_1 and J_8 need to be padded with null values for the set of attributes of the other operand relations, while unwanted components 3, 4, and 5 need to be removed from $J_{1,2,3,4,5,6,7,8,9}$.

R_1 ⟕ R_2: $S= 1 \in \{k, \ldots, l\}$. Only J_1, $J_{1,2,6}$, $J_{1,2,3,4,5,6,7,8,9}$ satisfy S, and the result is $\{(C, \text{-}), (B, b), (A, a)\}$.

R_1 ⟖ R_2: $S= 2 \in \{k, \ldots, l\}$. Only $J_{1,2,6}$, $J_{2,4,8}$, $J_{1,2,3,4,5,6,7,8,9}$ satisfy S, and the result is $\{(B, b), (\text{-}, c)\ (A, a)\}$.

4) Anti-Join. An anti-join $R_i \rhd R_j$, defined as $R_i - (R_i \ltimes R_j)$, returns tuples in R_i that do not find a match in R_j. When the outer-join for the edge p was performed during the construction of the Join Core, such tuples (from R_i) must have found no match in R_j and were stored in tables whose names contain i, but not p. Therefore, to answer the query $R_i \rhd R_j$, we look for $J_{k,\ldots,l}$, $i \in \{k, \ldots, l\} \wedge \neg (p \in\{k, \ldots, l\})$, namely, $S= i \in \{k, \ldots, l\} \wedge \neg (p \in\{k, \ldots, l\})$. Trivial predicate i cannot be omitted.

Example 8. (*Anti-Join*).

$R_1 \rhd R_2$: $S= 1 \in \{k, \ldots, l\} \wedge \neg (6 \in \{k, \ldots, l\})$. Only J_1 satisfies and the answer is $\{C\}$.

$R_2 \rhd R_4$: $S= 2 \in \{k, \ldots, l\} \wedge \neg (8 \in \{k, \ldots, l\})$. Only $J_{1,2,6}$ satisfies and the answer is $\{b\}$.

B. Multi-Join Queries

A Join Core consists of regular and extended Join Core tables. For simplicity, we shall not mention explicitly what types of Join Core tables the query predicates are applied to. Readers are advised that if the query is of Type (i), then the selection criteria should be applied to both types of Join Core tables; otherwise, they should only be applied to regular Join Core tables.

Let $E = E_1\ op\ E_2$, where E, E_1, and E_2 are expressions that contain arbitrary legitimate sequences of equi-, semi, outer- and anti-join operators, and *op* is one of these join operators with a join predicate p. We assume the query graphs for E, E_1, and E_2 are all connected subgraphs of G. Let S_1 and S_2 be the selection criteria on the Join Core tables for E_1 and E_2, respectively, and S the criteria for E. We discuss how to derive S from S_1 and S_2.

1) Equi-Join. Consider $E = E_1 \bowtie E_2$. Each tuple in E is a concatenation of a pair of extended matches in E_1 and E_2 that satisfy p, and such "longer" extended matches must have been captured by successive outer-joins (and complementary joins for cycle-completing relations) performed during the Join Core construction and stored in Join Core tables whose names satisfy $S_1 \wedge S_2 \wedge p\in \{k, \ldots, l\}$. On the other hand, the components of each tuple in such Join Core tables that satisfy S_1 and S_2 must be result tuples of E_1 and E_2, respectively. In addition, the two components satisfy the join predicate p and thus can generate a result tuple in E. Thus, $S = S_1 \wedge S_2 \wedge p \in \{k, \ldots, l\}$.

2) Semi-Join. $E = E_1 \ltimes E_2$ and $E = E_1 \rtimes E_2$. As explained, a semi-join is basically an equi-join, except that only the attribute values of one of the operands is retained. Thus, $S = S_1 \wedge S_2 \wedge p \in \{k,\ldots,l\}$.

3) Outer-Join. $E = E_1$ ⟗ E_2. Tuples in E represent extended matches that come from non-matching tuples of E_1 and E_2, and matching pairs of E_1 and E_2. All these extended match tuples in E were captured by successive outer-joins (and complementary joins for cycle-completing relations) performed during construction of the Join Core and stored in tables whose names satisfy $(S_1 \wedge (\neg p\in \{k, \ldots, l\})) \vee (S_2 \wedge (\neg p\in \{k, \ldots, l\})) \vee (S_1 \wedge S_2 \wedge p\in \{k, \ldots, l\})$, which can be simplified to $S_1 \vee S_2$ because $p\in \{k, \ldots, l\}$ implies $S_1 \wedge S_2$. On the other hand, each tuple in a Join Core table whose name satisfies $S_1 \vee S_2$ must provide a result tuple to E_1, E_2, or E. Thus, $S = S_1 \vee S_2$. Similarly, for E_1 ⟕ E_2, $S = S_1$; for E_1 ⟖ E_2, $S = S_2$.

4) Anti-Join. $E = E_1 \rhd E_2$. Tuples in E are extended matches in E_1 that do not find matches in E_2. Thus, tuples in E must have been captured by successive outer-joins (and complementary joins) performed and stored in Join Core tables whose names satisfy S_1 but not $(S_2 \wedge p\in \{k, \ldots, l\})$. One the other hand, Join Core tables whose names satisfy S_1 but not $(S_2 \wedge p\in \{k, \ldots, l\})$ contain tuples of E_1 that do not join with tuples in E_2, which are exactly the result tuples of E. That is, $S = S_1 \wedge \neg(S_2 \wedge p \in \{k, \ldots, l\})$.

Example 9. (*Multi-Anti-Join Queries*).

$(R_1 \bowtie R_2) \rhd R_3$: S= 6 ∈ $\{k, \ldots, l\} \wedge \neg$ (7 ∈ $\{k, \ldots, l\}$). Only $J_{1,2,6}$ satisfies S and the answer is $\{(B, b)\}$.

$(R_2 \rhd R_1) \rhd (R_4 \bowtie R_5)$: S=(2 ∈ $\{k, \ldots, l\} \wedge \neg$ (6 ∈ $\{k, \ldots, l\}$)) $\wedge \neg$ (9 ∈ $\{k, \ldots, l\} \wedge$ 8∈ $\{k, \ldots, l\}$). Only $J_{2,4,8}$ satisfies S, and the answer is $\{(c)\}$.

Theorem 3. Let $E = E_1$ *op* E_2, where E, E_1, and E_2 are arbitrary legitimate expressions that contain equi-, semi-, outer- and anti-joins, and *op* is one of these join operations with a join predicate p. Let S_1 and S_2 be the selection criteria for identifying Join Core tables from which the resulting tuples of E_1 and E_2 can be derived, respectively. Then, the selection criteria S for E is (*i*) if $op = \bowtie$, $S = S_1 \wedge S_2 \wedge p \in \{k, \ldots, l\}$; (*ii*) if $op = \ltimes$ or $\rtimes$, $S = S_1 \wedge S_2 \wedge p \in \{k, \ldots, l\}$; (*iii*) if op = ⟗, $S = S_1 \vee S_2$; if op = ⟕; $S = S_1$; if op = ⟖, $S = S_2$; (*iv*) if $op = \rhd$, $S = S_1 \wedge \neg(S_2 \wedge p \in \{k, \ldots, l\})$.

C. Join Queries with Intersections, Unions, and Differences

Here, we consider join queries with commonly encountered set operators, intersections, unions, and differences. Note that an intersection can be treated as an equi-join in which the join attribute is the primary key. Here, we assume that the join graph includes edges specifying the equalities of primary keys between two schema compatible relations.

Let p be a join predicate specifying the equality of primary key attributes of two schema compatible relations. The intersection operation requires matches in the key values. Consequently, the resulting tuples of $R_i \cap R_j$ can only be found in Join Core tables $J_{k, \ldots, l}$ whose names contain predicate p i.e., $S = p \in \{k, \ldots, l\}$. This is exactly the same selection criterion as that for an equi-join or a (left or right) semi-join. As for the union operation, the resulting tuples of $R_i \cup R_j$ can be found in Join Core tables whose names contain trivial predicate i or j, i.e., $S = i \in \{k, \ldots, l\} \vee j \in \{k, \ldots, l\}$, the same selection criteria as for a full outer-join. Similarly, for the difference operation, the resulting tuples of $R_i - R_j$ can be found in Join Core tables whose indices contain the trivial predicate i, but not j, i.e., $S = i \in \{k, \ldots, l\} \wedge \neg (j \in \{k, \ldots, l\})$, the same selection criteria as for an anti-join.

By the same reasoning as presented in the previous section (*B*) and Theorem 3, we can extend the usage of Join Core tables to queries with arbitrary legitimate sequences of unions, differences, and intersections, in addition to equi-, semi-, outer- and anti-joins. The theorem follows.

Theorem 4. Let $E = E_1$ *op* E_2, where E, E_1, and E_2 are arbitrary legitimate expressions that contain equi-joins, semi-joins, outer-joins, anti-joins, unions, differences, and intersections, and *op* is one of these operations with a join predicate p. Let S_1 and S_2 be the selection criteria for identifying Join Core tables from which the result tuples of E_1 and E_2 can be derived, respectively. Then, the selection criteria S for E is (*i*) if $op = \bowtie$ or $\cap$, $S = S_1 \wedge S_2 \wedge p \in \{k, \ldots, l\}$; (*ii*) if $op = \ltimes$ or $\rtimes$, $S = S_1 \wedge S_2 \wedge p \in \{k, \ldots, l\}$; (*iii*) if op = ⟗ or $\cup$, $S = S_1 \vee S_2$; if op = ⟕, $S = S_1$; if op = ⟖, $S = S_2$; (*iv*) if $op = \rhd$ or $-$, $S = S_1 \wedge \neg(S_2 \wedge p \in \{k, \ldots, l\})$.

VII. COST ANALYSIS

In this section, we analyze the time and space consumption of using Join Core. In addition, we also discuss measures to reduce the size of Join Core.

A) Time Consumptions

1)Disk Accesses Time

To answer a query, Join Core tables containing the result tuples are read into memory. Thus, the total number of disk accesses is dependent upon the size of the query result, not the complexity of the query.

2) CPU Time

Once desired Join Core tables are read into memory, all that is remaining is to perform equality checking between alias components (of cycle-completing relations), pad "missing" attributes with null values (for outer-join operations), and eliminate unwanted attributes and duplicates. All these tasks should take only a very small amount of CPU time.

B) Space Consumptions

To simplify discussions, we assume no dangling tuple exists in any of the equi-joins in the graph, which represents a worst case space consumption scenario since dangling tuples can shorten the matches. We further assume that in each join, all tuples of a relation find exactly the same number of matches in the other relation, namely a uniformity assumption on the matching of a join.

Consider a join between R_i (with T_i tuples), and R_j (with T_j tuples). We shall call T_j/T_i, denoted as r_{ij}, *the join ratio* of T_i with respect to T_j, that is, the average number of matches found in R_j for each tuple in R_i. In a one-many relationship from R_i to R_j, $r_{ij} \geq 1$. On the other hand, in a many-one relationship from R_i to R_j, $T_j/T_i \leq 1$. Since each tuple in R_i still can find one match in R_j, as we have assumed no dangling tuples exist in the joins, r_{ij} is set to 1 (i.e., r_{ij}=1) when $T_j/T_i \leq$ 1.

To estimate the size of a Join Core, we first estimate the total number of match tuples, denoted by M, in the Join Core, and multiply it by the length of each match tuple.

To estimate the number of different matches, we can start from any relation, say R_i, by setting $M = T_i$, and then marking R_i as visited. For each edge $\langle R_i, R_j \rangle$, where R_i is a visited node while R_j is not, $M = M \times r_{ij}$. Once all relations are visited, the final M is the estimate.

Now, let us compute the length of each match tuple. Let e be the number of join edges and n the number of relations in the join graph. Each outer-join adds the set of attributes of one relation to the schema of the output, recalling the construction of a Join Core. Therefore, the final output of the outer-joins consists of the values of the attributes of e+1 relations, $e+1 \geq n$. For simplicity of analysis, we assume tuples in all relations have the same or a similar length L. Therefore, the size the Join Core is

$$M\times(e+1)\times L \qquad (1)$$

As compared to the database size $T_{\text{avg}}\times n\times L$, where $T_{avg}=Avg\{T_1, \ldots, T_n\}$ is the average number of tuples in a relation.

Note that when all relations are of similar sizes, i.e., $T_{avg}\approx T_1 \approx\ldots\approx T_n$, all r_{ij}'s ≈ 1 and $M\approx T_{avg}$. In addition, if the graph has no (or few) cycles, i.e., $e+1=(\approx)\ n$, the Join Core size would be close to the database size, that is, $M\times (e+1)\times L\approx T_{\text{avg}}\times n\times L$, which is the best case scenario.

.

C) Space Reduction Methods

Many data compression techniques [4], [5], [13] can be used to compress the Join Core. Here, we shall only discuss methods that are specifically related to the reduction of the Join Core structure.

Storing all join relationships of a complex graph can consume large amounts of space. Here, we discuss heuristics that can significantly reduce the space consumption of the Join Cores, however, at the price of incurring additional join operations. Further research is still needed to analyze the cost and benefits of these heuristics.

(H1). Store only useful relations, relationships, and attribute values. Statistics and knowledge on the usages of relations, relationships, and attributes may be available or can be collected to assist in making such decisions.

(H2). Remove smaller relations from a join graph. Smaller relations, in terms of the numbers of tuples in the relations, need replicate their tuples more times to generate M match tuples, which will make updates (on smaller relations) more expensive. In addition, if a removed relation is referenced in a join query, then a join operation must be performed. Removing smaller relations incurs less penalty because joins with smaller relations are faster to perform. Moreover, smaller relations have better chances of fitting in memory to make the joins faster.

(H3). Remove cycle-completing relations. Removal of a cycle-completing relation from a graph implies removal of all its aliases too, which can significantly reduce the storage consumption. Since any graph traversal method can be used in construction a Join Core, one is given the opportunity to select "good" relations to be cycle-completing relations. Here, we recommend relations that are small (following H2) and, if possible, complete multiple cycles.

1) Constructing Join Core with Space Constraint

Without detailed cost-benefit measures, here is a simple way to construct a Join Core that satisfies a given space limit. First, one can, following (H1), remove unwanted relations, relationships, and attributes if a priori knowledge or statistics are available. If the Join Core is still too large, one can consider removing a smallest relation, following (H2), or a cycle-completing relation, following (H3), until the desirable size is met.

VI. NEO4G GRAPH DATABASE

In this section, we discuss Neo4j in details as it will be used in later performance evaluation against Join Core.

Neo4j is most popular graph databases according to Forrester [24]. It is an open-source graph database management system that provides high scalability and read/write performance [24]. The high performance is mainly owing to the use of both a native processing and storage model. Native processing model is referred to the leverage of index-free adjacency (where related nodes are physically connected to each other) in graph database. The use of index-free means that the query time depends on the searched graph length rather than the total size of the graph [4]. On the other hand, native storage model refers to the underlying physical structure of the database, where nodes and relationships are stored in a graph structure. This technology ensures that the graph database is optimized by storing related entities close to each other [24].

Neo4j employs the property graph data model [24]. Property graph model consists of nodes, relationship, properties and labels [7]. Both node and relationship holds a number of properties that are stored in the form of key-value pairs. Relationship links nodes to each other and each relationship has a name, direction and start and end nodes. Labels tag nodes to group them, and to identify their role in the dataset. Fig. 5 explains the consumer complaints against the company's products and sub-products, and issues that rose and the company's responses to. Each and every node is associated with the labels and the properties.

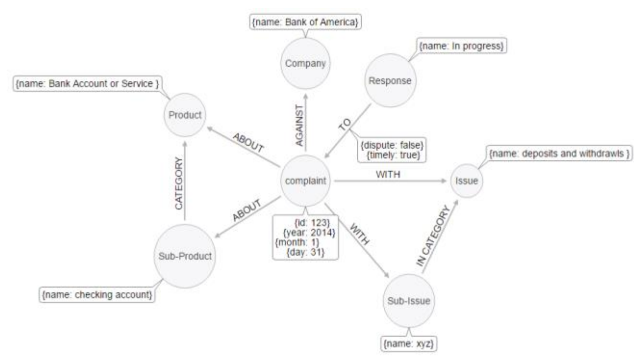


Figure 5. The labelled property graph Model

In Neo4j each type of element is stored in a separate data store. For instance, physical file neostore.nodestore.db contains all nodes in the dataset where neostore.propertystore.db and neostore. relationshipstore.db stores properties and relationship respectively [7]. Records inside node and relationship data store are fixed in size which accelerates record lookups in the file, as any known record ID can be used compute the record's location in the file.

Neo4j can be queried in many ways, such as Traverser API and Cypher query language [17]. Cypher is a declarative graph query language provides an efficient way to create, update and query the graph database [3]. It is considered as a powerful language that focuses on what to get rather than how to get it. Cypher's structure is inspired by SQL to make it easier and more familiar for the SQL users, although it is focuses on finding and describing patterns in the graph.

Cypher uses clauses (like most query languages) to query from the graph, a simple read query would be consist of MATCH, WHERE and RETURN clauses. Cypher execution engine optimize and turn each query into an execution plan. The plan is a pattern of number of connected operators, where each operator is responsible for a small section of the query execution.

VII. EXPERIMENTAL RESULTS

We have implemented the proposed methodology and performed experiments to compare its performance efficiency with a Neo4j graph database. We have used the Neo4j 3.0.6 community edition to perform experiments. In this preliminary study, we will use only the simplest set up to see how the proposed method alone can improve query processing, leaving other performance improving factors to future work. All experiments are performed on a laptop computer with a 2.40 GHz CPU, 8GB RAM, and a 1 TB hard drive.

A) Dataset

We generate 1.2GB consumer complaint dataset for experiments. Fig. 6 shows the join graph of consumer complaint dataset with arrows indicating the join relationships. Consumer complaints dataset has 4 relations i.e. Product, Issue, Response and Complaint. The dataset is loaded into Neo4j where graph data are stored in a number of different files. The same dataset is stored as Join Core tables in the proposed method, which is implemented in the Java programing language.

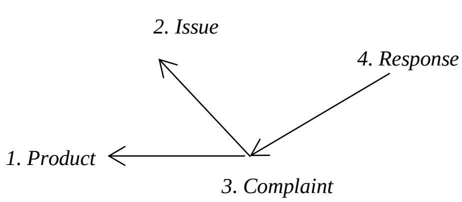


Figure 6. Consumer complaints Join Graph

B) Space Consumptions

As shown in Table 1, the dataset sizes after processing the dataset in Join Core tables and loading all data and join relationships between tuples into Neo4j are 3, and 2.4GB, respectively for the 1.2GB consumer complaint dataset. The larger size of Join Core table is due to the replications of tuples. In Neo4j, the increase in the size is due to summation of actual size of database, ratio of size of graph.db to index and ratio of size of graph.db to schema [24].

Table 1. Space Consumptions

Consumer Complaint Dataset	Join Core	Neo4j
1.2 GB	3 GB	2.4 GB

2) Query Processing Time

We measure the elapsed time of the test queries that come with the consumer complaint datasets. While keeping (most of) the selections and projections, we remove any "group by", "order by", "limit", aggregate functions, etc., from the queries so that we can focus mainly on the join query processing. We add "distinct" to the queries as we have implicitly assumed the set semantics in the paper.

Join Core tables are read from disks into memory for processing, and the result tuples are written back to the disks. Elapsed time measures the time from beginning to end, after writing all result tuples to the disks.

Table 2 shows the query processing time. In the first column, the ID of the consumer complaint query is shown first, followed by the relations involved in the join operations. For simplicity, relations are referenced by the numbers assigned to them in Fig. 6. All times are measured in milliseconds.

With Join Cores, all queries saw their first responses instantly. As explained, all it takes is the retrieval of a block of a relevant Join Core table into memory and simple manipulations before output it after simple manipulations.

The result size of the query, not the complexity determines the query processing time because the join result is readily available in the Join Core. Queries 2 and 3 best illustrate this characteristic of Join core. Query 2 has only one

Table 2. Time Consumptions

Query		Join Core Elapsed Time (ms)	Neo4j Elapsed Time (ms)	Result Tuples
1	R_2	18	714	68
2	$\bowtie \{R_1, R_4\}$	253	31,824	1,976
3	$\bowtie \{R_1, R_2, R_3\}$	13	262	18
4	$\bowtie \{R_1, R_2, R_4\}$	49	317	447
5	R_2	19	319	95
6	R_3	18	130	5
7	R_4	17	860	7
8	$\bowtie \{R_2, R_3, R_4\}$	19	752	84
9	$\bowtie \{R_2, R_4\}$	19	856	106

join but generates large numbers of result tuples. On the other hand, Query 3 has two joins, but generates much smaller numbers of result tuples. Therefore, it took much longer to process Query 2 than Query 3. As shown in Table 3, in join core, it took 253milliseconds to process Query 2 for 1.2 GB dataset, but it took only 13milliseconds, respectively, to process Query 3. Since there were no joins to perform in the join core, many queries completed instantly. Whereas in Neo4j, path traversal operations in the complex relationships of nodes determines the query processing time because in Neo4j it first travels through the relationship table and then retrieves the resultant tuples from the disk. Queries 1 and 6 are best to illustrate this characteristic of Neo4j. Both queries 1 and 6 have no joins. But the complexities of relationships involved in the query makes to retrieve small number of result tuples in more time compared to query 1 which retrieved large number of result tuples in less time compared to query 6.

From the results, it can be inferred that, if the query has the complex relationships in it then Neo4j takes more time than the join core. From the above experimental results, it can be observed that when the queries require large joins one can use join core, which is portable and can be used for any application domain irrespective of the API. Join Core retrieve results instantly when compared to the Neo4j. From this study, it can be concluded that to process complex queries join core is the best option irrespective of the size of the data.

Another advantage of the proposed methodology is that it does not consume much memory. All it needs is to build a hash table for the final duplicate elimination.

We believe the instant responses, fast query processing, and small memory consumption of the Join Core are well worth its required additional storage space.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, an anti-relational approach, called Join Core, has been presented. Join Core technique stores the equi-join relationships of tuples on various tables. The join queries can be answered quickly by merely merging these tables without having to perform expensive joins. We use Neo4j as graph database to perform experiments and compare its time and space consumptions with a Join Core. Preliminary experimental results showed that Join Core outperforms Neo4j when complex join queries are processed. This is because in Join Core, there was no need to perform join operations at run time while in Neo4j, the path traversal operations depend upon the complexities of the relationships of tuples.

REFERENCES

[1] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava, “Answering queries using views”, In ACM PODS Conf., 1995, pp. 95-104.

[2] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. and Sander, “Relational joins on graphics processors”, In ACM SIGMOD Conf., 2008, pp. 511-524.

[3] B. Kenny, Understanding How Neo4j Cypher Queries are Evaluated: http://www.kennybastani.com/2014/07/understanding-how-neo4j-cypher-queries.html

[4] C. Kim, E. Sedlar, and J. Chhugani, “Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs”, In VLDB Conf., 2009, pp. 1378-1389.

[5] D. Abadi, S. Madden, and M. Ferreira, “Integrating Compression and Execution in Column-Oriented Database Systems”, In SIGMOD, 2006, pp. 671–682.

[6] D. DeWitt, and R. Gerber, “Multiprocessor hash-based join algorithms”, In VLDB, 1985, pp. 151–164.

[7] E. Eifrem, J. Webber, I. Robinson, Graph Databases. 2nd Edition “O’Reilly Media, Inc.", 2015

[8] H. Karloff, and M. Mihail, “On the complexity of the view-selection problem”, In ACM PODS Conf., 1999, pp. 167-173.

[9] J. Goldstein, and P.-A. Larson, “Optimizing queries using materialized views: a practical, scalable solution”, In ACM SIGMOD, 2001, pp. 331-342.

[10] J. Yang, K. Karlapalem, and Q. Li, “Algorithms for materialized view design in data warehousing environment”, In VLDB, 1997, pp. 25-29.

[11] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, “Application of hash to data base machine and its architecture”, New Generation Computing 1(1), 1983, pp. 63-74.

[12] M. W. Blasgen, and K. P. Eswaran, “Storage and access in relational data bases”, IBM Systems Journal 16.4, 1977, pp. 363-377.

[13] M. Zukowski, S. Héman, N. Nes, and P. Boncz, “Super-scalar RAM-CPU cache compression”, In ICDE, 2006, - http://doi.org/10.1109/ICDE.2006.150.

[14] P. Valduriez, “Join indices”, ACM Transactions on Database Systems (TODS), 1987, 12(2), pp. 218-246.

[15] R. Derakhshan, F. Dehne, O. Korn, and B. Stantic, “Simulated Annealing for Materialized View Selection in Data Warehousing Environment”, In Databases and applications, 2006, pp. 89-94.

[16] R. Pottinger, and A. Levy, “A scalable algorithm for answering queries using views”, In VLDB Conf., 2000, pp. 484-495.

[17] R. Kaliyar , Graph Databases: A Survey, International Conference on Computing, Communication and Automation (ICCCA2015), p785-790

[18] R. De Virgilio, A. Maccioni, R. Torlone, R. 2013. Converting Relational to Graph Databases. In First International Workshop on Graph Data Management Experiences and Systems (GRADES ’13), pp. 1–6, New York, New York, USA, ACM Press.

[19] S. Agarawal, S. Chaudhuri, and V. Narasayya, “Automated Selection of Materialized Views and Indexes for SQL Databses”, In VLDB , 2000, pp. 496-505.

[20] S. Chu, M. Balazinska, and D. Suciu, “From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System”, In ACM SIGMOD Conf., 2015, pp. 63-78.

[21] Z. Li, K. A. Ross, “Fast joins using join indices”, The VLDB Journal—The International Journal on Very Large Data Bases”, 1999, 8(1), pp. 1-24.

[22] H. Mohammed, Y. Feng, H. Wen-Chi, “Answering Queries Instantly”, https://goo.gl/19ajmF.

[23] TPC-H, http://www.tpc.org/information/benchmarks.asp.

[24] Graph database (Neo4J): https://neo4j.com