An Efficient Data Structure for Fast Join Query Processing

Mohammed Hamdi
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA
mhamdi@siu.edu

Feng Yu
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, OH, USA
fyu@ysu.edu

Sarah Alswedani
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA
sarah.swy@siu.edu

Wen-Chi Hou
Department of Computer Science
Southern Illinois University
Carbondale, IL, USA
hou@cs.siu.edu

***Abstract*—In this research, we propose 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, which are collectively called the Join Core. Queries involving arbitrary legitimate sequences of equi-joins, semi-joins, outer-joins, anti-joins, unions, differences, and intersections can all be answered quickly by merely merging these tables without having to perform joins. The Join Core can also be updated dynamically. Preliminary experimental results showed that all test queries began to generate results instantly, and many completed instantly too. The proposed methodology can be very useful for queries with complex joins of large relations as there are fewer or even no relations or intermediate results needed to be retrieved, generated.**

***Keywords*—Query processing; join queries; equi-join; semi-join; outer-join; anti-join; set operations**

I. INTRODUCTION

As hardware technologies advance, the price of disks drops significantly while the capacity increases drastically. Database researchers now have the luxury of exploring innovative ways to utilize these cheap and abundant spaces to improve query processing.

In relational databases, data are spread among relations. The equi-join operation, which includes the natural join, is the most commonly used operator to combine data spread across relations. Other join operators, such as semi-joins, outer-joins, and anti-joins, are also very useful. Unfortunately, these join operations are generally expensive to execute. Complex queries involving multiple joins of large relations can easily take minutes or even hours to compute. Consequently, much effort in the past few decades has been devoted to developing efficient join algorithms [10], [5], [9]. Even today, improving join operations remains a focus of database research [16], [2].

In this research, we propose to pre-store 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, which are collectively called the Join Core.

Join Core is an efficient data structure from which not only the results of all possible equi-joins can be obtained, but also the results of all legitimate combinations of equi-joins, outer-joins, anti-joins, unions, differences, and intersections can be derived. Without having to perform joins, memory consumptions are dramatically reduced. In addition, Join Core can be updated dynamically in the face of updates.

In this research, we also discuss heuristics that can effectively cut down the sizes of 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 join indices 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 [18] 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 the literature, materialized views are, to a certain extent, related to our work as both attempt to use precomputed 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 [15], [13], [6], [8] and how to choose an appropriate set of materialized views to answer a query [7], [1], [14].

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 [12], [17] 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.

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 [16] 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 [18].

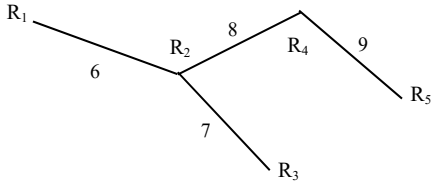
Definition 1. (Join Graph of a Database). Let D be a database with n relations R_1, R_2, …, R_n, and G (V, E) be the join graph of D, where V is a set of nodes that represents the set of relations in D, i.e., $V = \{R_1, R_2, R_3, ..., R_n\}$, and $E = \{\langle R_i, R_j\rangle \mid R_i, R_j \in V, i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

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

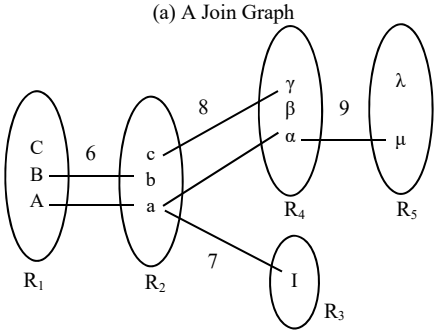
Each join comes with a predicate, omitted in the graph, specifying the requirements that a result tuple of the join must satisfy, e.g., $R_1.attr1=R_2.attr2$. For simplicity, we shall use a join, a join edge, and a join predicate interchangeably. We also assume all relations and join edges are numbered.

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.



(a) A Join Graph



(b) Matching of Join Attribute Values

Fig. 1. A join graph and matching tuples.

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.

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 VII.

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.

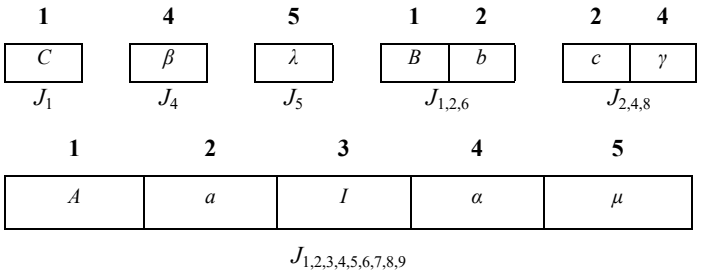


Fig. 2. Join core.

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 $\theta \neq$ "=", 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 $1 \leq k, \ldots, l \leq n$, $t_k \in R_k, \ldots, t_l \in R_l$, 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.

B. 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), $(B, b, -, -)$, $(C, -, -, -)$, $(-, c, -, \gamma)$, and $(-, -, -, \beta)$. The final outer-join with R_5 generates (A, a, I, α, μ), $(B, b, -, -, -)$, $(C, -, -, -, -)$, $(-, c, -, \gamma, -)$, $(-, -, -, \beta, -)$, and $(-, -, -, -, \lambda)$, 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.

C. Answering Queries using Join Core

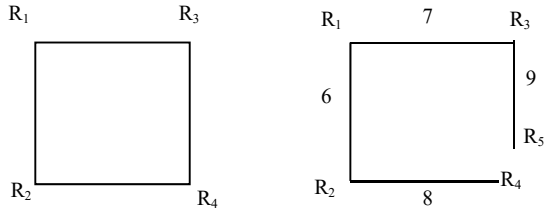
The name of a Join Core table specifies the join predicates satisfied by the match tuples stored in it. On the other hand, a join query specifies predicates that must be satisfied by the result tuples. Therefore, to answer a query is to look for Join Core tables whose names contain the predicates of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



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

Fig. 3. Converting a cyclic graph.

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.

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 $\{u, \ldots, v\} \neq \emptyset$. 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*.

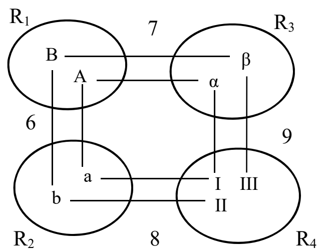
B. Cyclic Join Graph

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.

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 Fig. 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.



(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

Fig. 4. Cyclic join relationship and join core.

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 $\bowtie(\{R_i, \ldots, R_j\}, \{u, \ldots, v\})$, $1 \leq i, \ldots, j \leq n$, 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 [18] for discussions on more complicated issues for cyclic queries.

C. Multiple Join Edges between Relations

It is possible that there is more than one joins 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 $\{(C, -), (B, b), (-, c)\ (A, a)\}$. 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 $J_{1,2,3,4,5,6,7,8,9}$.

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

$R_1 \ltimes R_2$: $S = 2 \in \{k, \ldots, l\}$. 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 $\{(B, b), (-, c) (A, a)\}$.

4) Anti-Join. An anti-join $R_i \triangleright R_j$, 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 $R_i \triangleright R_j$, 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).

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

$R_2 \triangleright R_4$: $S = 2 \in \{k, \ldots, l\} \wedge \neg (8 \in \{k, \ldots, l\})$. 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. $E = E_1 \bowtie\!\!\!\!\ltimes E_2$. 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 $E_1 \rtimes E_2$, $S = S_1$; for $E_1 \ltimes E_2$, $S = S_2$.

4) Anti-Join. $E = E_1 \triangleright E_2$. 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).

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

$(R_2 \triangleright R_1) \triangleright (R_4 \bowtie R_5)$: $S = (2 \in \{k, \ldots, l\} \wedge \neg (6 \in \{k, \ldots, l\})) \wedge \neg (9 \in \{k, \ldots, l\} \wedge 8 \in \{k, \ldots, l\})$. 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 $op = \bowtie\!\!\!\!\ltimes$, $S = S_1 \vee S_2$; if $op = \bowtie\!\!\!\!\rtimes$; $S = S_1$; if $op = \bowtie\!\!\!\!\ltimes$, $S = S_2$; (*iv*) if $op = \triangleright$, $S = S_1 \wedge \neg(S_2 \wedge p \in \{k, \ldots, l\})$.

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 $op = \rtimes$ or $\cup$, $S = S_1 \vee S_2$; if $op = \bowtie$, $S = S_1$; if $op = \ltimes$, $S = S_2$; (*iv*) if $op = \triangleright$ 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

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

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. Join Core Size Estimation by Example

In the following, we shall use the TPC-H benchmark dataset to illustrate the use of the estimation formula (1) and check its accuracy. Although each database has its unique features, the TPC-H dataset may give us a general idea how large the Join Core can be because "the data populating the database in TPC-H have been chosen to have broad industry-wide relevance" [19].

Fig. 5 shows the join graph of the TPC-H dataset, which will also be used in our experimental section. For simplicity, relations are numbered from 1 to 8. The arrows indicate many-one relationships. In the 1GB dataset, the largest relation "lineitem" (i.e., R_3) has 6,000,000 tuples, while the smallest one "region" (R_8) has only 5 tuples. The average length of a tuple is 128 bytes.

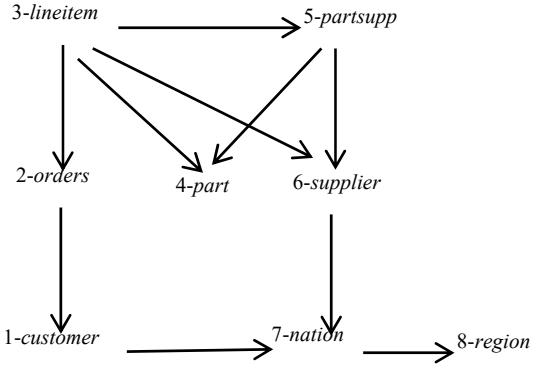


Fig. 5. TPC-H join graph.

Example 10. (*Estimating the Join Core Size*). We start from the largest relation "lineitem", following the arrows to visit smaller relations, until all edges are traversed. Note that all join ratios are 1 along the way as we always visit smaller relations. Thus, the total number of match tuples in the Join Core is 6,000,000 (=*M*). There are 11 edges and the average length of tuple is 128. So, the Join Core size is estimated, following (1), as 6,000,000×12×128= 9.1GB. As shown later in the experimental section, the actual Join Core is 4GB. The overestimation is due to there being many dangling tuples in the joins, shortening the length of match tuples.

In general, the Join Core could be several time larger than the database. However, as the disk space is becoming cheaper and cheaper, the space requirements should not be a big issue.

D. Space Reduction Methods

Many data compression techniques [3], [4], [11] 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.

VIII. Experimental Results

We have implemented the proposed methodology and performed experiments to compare its time and space consumptions with a MySQL database system. Many factors, such as the number of CPUs, disks (and types of disks, magnetic or SSD), etc., can affect the performance of query processing. 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 1.60 GHz CPU, 8GB RAM, and a 1 TB hard drive.

A. Datasets

We generate 1, 4, and 10GB TPC-H datasets for experiments. Fig. 5 shows the join graph of the TPC-H datasets with arrows indicating many-one relationships. The datasets are stored as relations in a MySQL database and as Join Core tables in the proposed method, which is implemented in the Java programing language.

TABLE I. Space Consumptions

Join Core Size	Datasets		
	1GB	**4GB**	**10GB**
Full	4 GB	13.8 GB	39.7 GB
Reduced 1	2.3 GB	7.1 GB	20.1 GB
Reduced 2	1.7 GB	5.4 GB	15.8 GB

B. Space Consumptions

As shown in Table 1, the full Join Core sizes, without applying any space reduction methods [18], are 4, 13.8, and 39.7GB for the 1, 4, and 10GB TPC-H datasets, respectively. "Reduced 1" is obtained by removing the smallest relations Region, Nation, and Supplier, which have 5, 25, and 10,000 tuples from the graph, respectively. "Reduced 2", is obtained by further removing the Customer relation from "Reduced 1".

While removing relations can certainly reduce the space consumption, joins would have to be performed when removed relations are referenced in the queries. Fortunately, removed relations are generally small and joins with them are relatively quick.

1) Query Processing Time

We measure the response and elapsed time of the test queries that come with the TPC-H 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. Response time measures the time up until the first result tuple is written to the disk, while elapsed time measures the time from beginning to end, after writing all result tuples to the disks.

TABLE II. TIME CONSUMPTIONS

Query	Join Core		MySQL		Result Tuples
	Response 1/4/10GB	Elapsed 1/4/10GB	Response 1/4/10GB	Elapsed 1/4/10GB	
12: ⋈ {R_2, R_3}	0.008	5.456	360	367	38,928
	0.008	22.409	701	725	155,585
	0.008	56.023	2,084	2,107	388,058
14: ⋈ {R_3, R_4}	0.008	0.502	411	411	1,717
	0.008	1.865	1,307	1,310	6,718
	0.008	3.865	2,014	2,018	16,943
19: ⋈ {R_3, R_4}	0.007	0.012	516	516	200
	0.007	0.041	1,485	1,485	864
	0.007	0.103	2,386	2,387	2,096
4: ⋈ {R_2, R_3}	0.009	0.397	284	285	3,040
	0.009	1.518	656	660	11,889
	0.009	3.625	1,963	1,969	29,447
16: ⋈ {R_4, R_5}	0.008	0.812	79	81	3,795
	0.008	3.005	300	306	15,208
	0.008	9.686	856	867	38,195
3: ⋈ {R_1, R_2, R_3}	0.008	1.579	6,782	6,785	11,620
	0.008	8.016	-	-	45,395
	0.008	17.455	-	-	114,003
18: ⋈ {R_1, R_2, R_3}	0.007	0.010	61	61	6
	0.007	0.012	91	91	11
	0.007	0.013	291	291	22
10: ⋈ {R_1, R_2, R_3, R_7}	0.009	1.706	5,060	5,063	3,773
	0.009	5.667	7,562	7,573	14,800
	0.009	14.560	-	-	36,975
2: ⋈ {R_4, R_5, R_6, R_7, R_8}	0.010	1.890	322	325	3,162
	0.010	7.005	838	845	12,723
	0.010	18.609	2,112	2,131	31,871
5: ⋈ {R_1, R_2, R_3, R_6, R_7, R_8}	0.010	1.760	-	-	15,196
	0.010	6.809	-	-	60,798
	0.010	16.355	-	-	152,102

Table 2 shows the query processing time with a full-sized Join Core. In the first column, the ID of the TPC-H 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. 5. For each query, we measured the time spent on all three datasets. Queries were aborted if they took more than 4 hours (= 14,400 sec), as indicated by –'s in the table.

With Join Core, 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. On the other hand, MySQL took minutes to hours to output its first result tuples.

As explained, the result size, not the complexity, of the query determines the query processing time because the join result is readily available in the Join Core. Queries 12 and 18 best illustrate this characteristic. Query 12 has only one join but generates large numbers of result tuples. On the other hand, Query 18 has two joins, including the join of Query 12, but generates much smaller numbers of result tuples. Therefore, it took much longer to process Query 12 than Query 18. As shown in Table 2, it took 5.456, 22.409, and 56.023 seconds to process Query 12 for 1, 4, and 10GB datasets, respectively, but it took only 0.010, 0.012, and 0.013 seconds, respectively, to process Query 18. Note that all these times were mainly spent on the disk accesses, namely, reading Join Cores and writing the result tuples. Since there were no joins to perform in the proposed method, many queries completed instantly too. On the other hand, many queries took hours to complete on MySQL.

The response time remained similar for all cases. The elapsed time was, however, longer for larger datasets than for smaller datasets because the former generated larger Join Cores and larger join results, which required more time to read and write.

Join Core is used to answer queries with anti-joins and outer-joins. Table 3 shows the processing time. The response and elapsed times of Query 22 are less than 10 millisecond for both 1, 4, and 10GB datasets while the same query consumes more time to be processed with MySQL. As shown in Table 3, Query 13 saw its first responses instantly and took 4.5, 18.4, and 40.0 to be completed for 1, 4, and 10GB datasets, respectively. On the other hand, the response and elapsed times were longer on MySQL for the same query.

TABLE III. TIME CONSUMPTIONS FOR QUERIES WITH ANTI-JOINS AND OUTER-JOINS

Query	Join Core		MySQL		Result Tuples
	Response 1/4/10GB	Elapsed 1/4/10GB	Response 1/4/10GB	Elapsed 1/4/10GB	
22: ▷ {R_1, R_2}	0.007	0.009	7.8	8	3
	0.007	0.009	32.9	33	6
	0.007	0.009	122.4	123	9
13: ⋈ {R_1, R_2}	0.008	4.5	360	367	15504
	0.008	18.4	701	725	71013
	0.008	40.0	2,084	2,107	155018

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.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an innovative way to process queries without having to perform expensive joins and set operations. We proposed to store the equi-join relationships in the form of maximally extended match tuples to facilitate query processing. We have designed an innovative way to group the join relationships into tables, called the Join Core, so that queries can be answered quickly, if not instantly, by merely merging subsets of these tables. The Join Core is applicable to queries involving arbitrary sequences of equi-joins, semi-joins, outer-joins, anti-joins, unions, differences, and intersections. Preliminary experimental results have confirmed that with Join Core, join queries can be responded to instantly and the total elapsed time can also be dramatically reduced. We will discuss concurrency control in the face of updates, and perform extensive experiments in different environments in the future.

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] 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.

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

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

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

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

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

[9] 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.

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] 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.

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

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