Flexible Caching in Trie Joins

Oren Kalinsky  Yoav Etsion  Benny Kimelfeld
Technion – Israel Institute Of Technology
{okalinsk@campus, yetsion@tce, bennyk@cs}.technion.ac.il

ABSTRACT

Traditional algorithms for multiway join computation are based on rewriting the order of joins and combining results of intermediate subqueries. Recently, several approaches have been proposed for algorithms that are “worst-case optimal” wherein all relations are scanned simultaneously. An example is Veldhuizen’s Leapfrog Trie Join (LFTJ). An important advantage of LFTJ is its small memory footprint, due to the fact that intermediate results are full tuples that can be dumped immediately. However, since the algorithm does not store intermediate results, recurring joins must be reconstructed from the source relations, resulting in excessive memory traffic. In this paper, we address this problem by incorporating caches into LFTJ. We do so by adopting recent developments on join optimization, tying variable ordering to tree decomposition. While the traditional usage of tree decomposition computes the result for each bag in advance, our proposed approach incorporates caching directly into LFTJ and can dynamically adjust the size of the cache. Consequently, our solution balances memory usage and repeated computation, as confirmed by our experiments over SNAP datasets.

1. INTRODUCTION

LeapFrog Trie Join (LFTJ) is a multiway join algorithm introduced by LogicBlox and implemented within. It operates in a manner of variable elimination where there is a linear order over the variables, and query results are generated one by one by incrementally assigning values to each variable in order. Trie indices over the relations guarantee that, throughout execution, one efficiently determines whether the current prefix of assignments cannot be matched against the database (we give a detailed description of LFTJ in Section 2). Veldhuizen has shown that LFTJ is worst-case optimal. This yardstick of efficiency for join algorithms has been introduced by Ngo et al., and it states that for every join query, no algorithm can be asymptotically faster on the space of all instances; in that work they presented the first algorithm that is likewise optimal, later termed NPRR.

More traditional join optimization has been based on decomposing the multiway join into smaller join queries and combining the intermediate results. This approach has roots in Selinger’s pairwise-join enumeration, and it includes the application of Yannakakis’s algorithm over a tree decomposition of the query. The advantage of LFTJ over the traditional approach is twofold. First, LFTJ avoids the potential generation of intermediate results that may be substantially larger than the final output size (which is a key property in guaranteeing worst-case optimality). Second, LFTJ is very well suited for in-memory join evaluation, since besides the trie indices it has a close to zero memory consumption. Of course, memory is required for buffering the tuples in the final result, but these are never read and can be safely dumped to higher storage upon need. In the case of an aggregate query (e.g., count the number of tuples in the result), no such requirement arises.

But intermediate results have the advantage that their tuples can be reused, and this is especially substantial in the presence of a significant skew. In our experiments, we have found that LFTJ often loses its advantage to the built-in caching of intermediate results of the traditional approaches, and in particular, LFTJ is often required to apply many repetition of computations. The repeated traversals back and forth on the trie index generate excessive memory traffic, which has detrimental impact on the performance of database systems. For example, our analysis of the memory load induced by LFTJ found that running a single count 5-cycle query on the SNAP ca-GrQc data set generates over 45 · 10^9 memory accesses, whereas running the same query using tree decomposition and Yannakakis’s join generates less than 16 · 10^9 accesses. (The implementation of both algorithms is discussed in Section 5.)

Nevertheless, it is not clear how LFTJ can cache results, since every iteration involves a different partial assignment, and variables are interdependent by the query atoms. Our goal in this work is to accelerate LFTJ by incorporating caching in a way that (a) allows for computation reuse, and (b) does not compromise its key advantages. In particular, our goal is to incorporate caching in LFTJ so that it can utilize whatever memory it has as its disposal towards memoization.

To incorporate caching in LFTJ, we build on a recent development in the theory of join optimization, relating worst-case optimality, variable ordering and tree decomposition. Specifically, given a multijoin query, we build a tree...
decomposition (TD), find an order on the variables such that the order is compatible with the TD. But unlike existing work, we do not apply the join algorithm on each bag independently, but rather execute LFTJ as originally designed. Yet, throughout the LFTJ execution we may choose to cache partial assignments (based on some decisions that we discuss later) or reuse cached results. The manner by which caching is carried out, as explained in Section 5, is based on the fact that the variable ordering correlates with the TD. In this way, our caching is flexible (i.e., every cached item is optional), and it does not violate the inherent benefits of LFTJ, while dramatically reducing the memory load. Concretely, running the 5-cycle count query described above on the integrated algorithm generates only $1.4 \cdot 10^9$ memory accesses, which is over $30 \times$ fewer accesses than the original LFTJ algorithm (and over $10 \times$ fewer accesses than tree decomposition with Yannakakis’s join).

But where does one get a TD from? The literature contains a plethora of algorithms with different quality guarantees. The classical graph-theoretic measure refers to the maximal size of a bag, and a generalization to hypergraphs is based on the notion of a hypertree width. The optimal values of those (i.e., realizing the tree width and the hypertree width, respectively) are both NP-hard problems \cite{11, 8}, and efficient algorithms exist for special cases and different approximation guarantees \cite{6}. Other notions include decompositions that approximate the minimal fractional hypertree width \cite{14}. Joglekar et al. \cite{10} determine first the variable ordering (in order to guarantee correctness of computing an expression comprising multiple operators), and then find a tree decomposition that complies with this ordering, and has an approximation guarantee against the minimum fractional hypertree width.

In our case, a TD defines a caching scheme, and various factors are likely to determine the effectiveness of this scheme. Importantly, our caches correspond to the adhesions (parent-child intersections), and the adhesion cardinalities are the dimensions of keys of our caches; hence, small adhesions are likely to have higher hit rates. Moreover, caches are more reusable in the presence of skewed data. Hence, we prefer not to use any algorithm that generates a single tree decomposition, but rather to explore a space of such decompositions. We complement existing decomposition approaches with a heuristic algorithm for enumerating TDs, tailored primarily towards small adhesions. Once such a collection of TDs are generated, we generate compatible orders. (In fact, our approach requires a property stronger than compatibility, and we call it strong compatibility.) Given a TD and a compatible order, we can use various techniques for benefit estimation, such as the cost model of Chu et al. \cite{7}. A particular component of our heuristic is an algorithm for enumerating graph separating sets with polynomial delay, without repetitions, and by increasing size.

We experiment on three types of queries: paths, cycles and random graphs, in various sizes. In par with recent studies on join algorithms, we base our experiments on data sets from the SNAP \cite{13} and IMDB workloads. Our experiments compare the performance of LFTJ with and without caching, and Yannakakis’s algorithm over the TD (with each subquery computed separately, as in \cite{21, 22}), as well as other various systems. The results show consistent improvement compared to LFTJ (in orders of magnitude on large queries), as well as general improvement compared to the examined algorithms and systems. As part of our experiments we research several attributes of cached LFTJ, such as running on different TDs and using a different cache sizes.

1.1 Contributions

Our contributions can be summarized as follows.

- We extend LFTJ with caching, without compromising the key benefits. Our caching is executed alongsize LFTJ, and its size can be determined dynamically according to memory availability.
- We devise a heuristic approach to enumerating tree decompositions of a CQ; this approach favors small adhesions, and is based on enumerating graph separating sets by increasing size.
- We present a thorough experimental study that evaluates the effect of caching on LFTJ and compares the results to state-of-the-art join algorithms.

2. PRELIMINARIES

In this paper we study the evaluation of a Conjunctive Query, or CQ for short, and the problem of counting the number of tuples resulting from this evaluation. As in recent work on worst-case optimal joins \cite{17, 19, 24}, we focus here on full CQs, which are CQs without projection. Formally, a full CQ is a sequence $\varphi_1, \ldots, \varphi_m$ where each $\varphi_i$ is a subgoal of the form $R(t_1, \ldots, t_k)$ with $R$ being a k-ary relation name and each $t_i$ being either a constant or a variable. We denote by $\text{vars}(\varphi_j)$ the set of variables that occur in $\varphi_j$, and we denote by $\text{vars}(q)$ the union of the $\text{vars}(\varphi_j)$ over all atoms $\varphi_j$ in $q$ (i.e., the set of all variables appearing in $q$).

Let $q$ be a full CQ. A partial assignment for $q$ is function $\mu$ that maps every variable in $\text{vars}(q)$ to either a constant value or null (denoted $\bot$). If $\mu$ is a partial assignment for $q$, then we denote by $q_{\mu}$ the full CQ that is obtained from $q$ by replacing every variable $x$ with $\mu(x)$, if $\mu(x) \neq \bot$, and leaving $x$ intact if $\mu(x) = \bot$.

The Gaifman graph of a full CQ $q$ is the undirected graph that has $\text{vars}(q)$ as its node set and an edge between every two variables that co-occur in a subgoal of $q$.

2.3 Ordered Tree Decompositions

Let $q = \varphi_1, \ldots, \varphi_m$ be a full CQ. A tree decomposition (TD) of $q$ is a pair $(T, \chi)$ where $T$ is a tree and $\chi$ is a function that maps every node of $t$ to a subset of $\text{vars}(q)$, called a bag, such that both of the following hold.
such that for every node \( v \) of the parent adhesion form \( v \) all of the descendants of a node of \( t \) that is rooted at \( v \) and contains all of the descendants of \( v \). An adhesion of \( t \) is a set of the form \( \chi(p) \cap \chi(v) \), where \( v \) is a node of \( t \) with a parent \( p \). The parent adhesion of a non-root node \( v \) (or simply the adhesion of \( v \)) is the set \( \chi(p) \cap \chi(v) \) where \( p \) is the parent of \( v \), and is denoted by \( \text{adhesion}(v) \).

Let \( q \) be a full CQ, and let \( \langle t, \chi \rangle \) be an ordered TD of \( q \). The preorder of \( t \) is the order \( \prec \) over the nodes of \( t \) such that for every node \( v \) with a child \( c \) preceding another child \( c' \), and nodes \( u \) and \( u' \) in \( t_{c'} \) and \( t_{c'} \), respectively, we have \( v \prec u \prec u' \). We denote the preorder of \( t \) by \( \prec_{\text{pre}} \). For a variable \( x \) in \( \text{vars}(q) \), the owner bag of \( x \), denoted \( \text{owner}(x) \), is the minimal node \( v \) of \( t \), under \( \prec_{\text{pre}} \), such that \( x \in \chi(v) \). We say that \( \langle t, \chi \rangle \) is compatible with an ordering \( \langle x_1, \ldots, x_n \rangle \) of \( \text{vars}(q) \) if for every \( x_i \) and \( x_j \), if \( \text{owner}(x_i) \) is a parent of \( \text{owner}(x_j) \) then \( i < j \). We say that \( \langle t, \chi \rangle \) is strongly compatible with \( \langle x_1, \ldots, x_n \rangle \) if for every \( x_i \) and \( x_j \), if \( \text{owner}(x_i) \prec_{\text{pre}} \text{owner}(x_j) \) then \( i < j \). Observe that strong compatibility indeed implies compatibility (but not necessarily vice versa).

### 2.4 Trie Join

We now describe Veldhuizen’s Leapfrog Trie Join (LFTJ) algorithm [24]. Our description is abstract enough to apply to the tributary join of Chu et al. [7]. Let \( q = \varphi_1, \ldots, \varphi_m \) be a full CQ. The execution of LFTJ is based on a predefined ordering \( \langle x_1, \ldots, x_n \rangle \) of the \( \text{vars}(q) \). The correctness and theoretical efficiency of LFTJ are guaranteed on every order of choice, but in practice the order may have a substantial impact on the execution cost [7]. Moreover, in our instantiation of LFTJ we will use orderings with specific properties.

For each subgoal \( \varphi_j \), LFTJ maintains a trie structure on the corresponding relation \( r \), where each level \( i \) in the trie corresponds to a variable \( x_i \) in \( \text{vars}(\varphi_j) \) and holds values that can be matched against \( x_i \) so that whenever \( x_i \) is in a level above \( x_j \) it holds that \( j < j' \). Moreover, every path from root to leaf corresponds to a unique tuple of \( r \) and vice versa. Sibling values in the trie are stored in a sorted manner, and so, LFTJ applies a sequence of sort-merge-joins as follows. Each trie holds an iterator, which is initialized by pointing the root. First, all the subgoals that contain \( x_i \) advance their iterators in the first level until a matching value \( a \) is found (i.e., all iterators point to \( a \)). The algorithm then proceeds recursively with the full CQ \( Q_{x_j/a} \), then proceeds to the next matching value, and so on, until no matching values are found. A balanced-tree storage of the sibling collections in the tries guarantees that alignment of the iterators on matching attributes is done efficiently (in an amortized sense), which in turn guarantees that LFTJ is worst-case optimal [17]. See [21] for more details.

In this paper, it suffices to regard LFTJ abstractly as depicted in the algorithm of Figure [1] and refer to it as trie join. The pseudocode does not compute the join, but in fact counts the tuples in the join; the translation into evaluation is straightforward, but we find the presentation more elegant for count. Moreover, we will later experiment with both evaluation and counting of joins. In the algorithm, the assignments \( x_i/a \) are represented using a global partial assignment \( \mu \) that is updated by the subroutine RJoin (Recursive Join). Note that in addition to \( \mu \), also global is the variable total (which, in the end, stores the resulting count).

#### 3. CACHING IN TRIE JOIN

We now describe our proposed incorporation of caching in LFTJ. For simplicity, we will focus on the counting query and show how we extend the algorithm TJCount of Figure [1]

#### 3.1 Intuition

The general idea is the following. Given the full CQ \( q \), we first construct an ordered TD \( \langle t, \chi \rangle \) of \( q \). Let \( v \) be a node of \( t \), let \( \alpha \) be adhesion(\( v \)), and let \( X \) be the set of all the variables \( x \) such that \( \text{owner}(x) \) is in the subtree \( t_{\chi} \). Then in the result of evaluating \( q \) we have the multivalued dependency \( \alpha \rightarrow X \). Therefore, for every assignment \( \mu \) to \( \alpha \) we can cache the assignments to \( X \) (or their number in the case of counting) and reuse them on the next time \( \mu \) is encountered.

One way of obtaining the above caching is by computing the join for every bag using the trie join, and then join the intermediate results using an algorithm for acyclic joins such as that of Yannakakis [25], as done in DunceCap [21, 23]. However, we wish to control the memory consumption of our algorithm and avoid computing full intermediate queries. So, the idea is to run the trie join ordinarily, but then cache results as the algorithm runs, conditioned on a choice of whether or not to cache using some utilization function that estimates the value of caching. For this to work, the ordered TD \( \langle t, \chi \rangle \) needs to be strongly compatible with \( \langle x_1, \ldots, x_n \rangle \), as defined in Section [2.3].

#### 3.2 Algorithm

Our algorithm is depicted in Figure [2]. Again, the algorithm is described for the counting problem. The algorithm
The algorithm, named CachedTJCount, takes as input a full CQ \( q \), an ordered TD \( (t, \chi) \) for \( q \), a variable ordering \((x_1, \ldots, x_n)\) for \( q \), and a database \( D \). The algorithm returns the count \(|q[D]|\).

The algorithm uses several global variables that are shared across procedure calls.

- As in TJCount, the global variable \( total \) counts the joined tuples and \( \mu \) stores the current partial assignment.
- The counter \( intrmd(v) \), where \( v \) is a node of \( t \), stores an intermediate count of the assignments to the variables owned by the nodes in \( t_v \), given the assignment to adhesion(\( v \)) in the current iteration. More precisely, let \( i \) be the maximal number such that \( x_i \) is in the adhesion of \( x_n \), and consider a partial assignment \( \mu \) that is nonnull on precisely \( x_1, \ldots, x_i \). Then in an iteration where \( \mu \) is constructed, \( intrmd(v) \) will eventually hold the number of assignments \( \mu' \) for the variables owned by the nodes in \( t_v \), such that some full assignment \( \mu \) for \( q \) is consistent with \( \mu \). Observe that this number is the same for assignments \( \mu \) that agree on the adhesion of \( v \). The counter \( intrmd(v) \) has the correct value once we are done with the variables owned by \( v \).
- \( cache \) stores cached values \( cache[A, \mu'] \) for adhesions \( A \) and assignments \( \mu' \) for \( A \). This value is obtained from \( intrmd(v) \) once the computation of \( intrmd(v) \) is done.

The algorithm CacheTDJCount simply initializes the global variables and call the subrouting \( RCachedJoin \), which is the caching version of \( RJoin \). Next, we explain this subroutining. The input takes not only the variable number \( d \), but also a factor \( f \) that aggregates cached intermediate counts.

The first part of the algorithm, lines 1–3, tests whether we are done with the variable scan, and if so, adds \( f \) to the total count. Now assume that \( d \leq n \), and the currently iterated variable is \( x_d \). Let \( v \) be owner(\( x_d \)) and \( \alpha \) be adhesion(\( v \)). In lines 6–12 we handle the case where we have just entered \( v \) from a different node of \( t \). This is determined by testing whether \( d > 1 \) and the previous variables, \( x_{d-1} \), has a different owner (in which case owner(\( x_{d-1} \) \( \neq \) \( \mu \) \( \alpha \) must hold)). In that case, the adhesion of \( v \) is already assigned values in \( \mu \) (since our TD is strongly compatible with the variable ordering), and we check whether we already have a cached result for this assignment. If so, then this cached result is stored in \( cache[A, \mu] \), where \( \mu \) is the restriction of \( \mu \) to \( A \). Hence, if \( cache[A, \mu] \) is defined, we skip to the next variable outside the subtree \( t_v \) with the factor multiplied by \( cache[A, \mu] \). This skipping is where strong compatibility is required, since it ensures that the nodes owned by \( t_v \) constitute a consecutive interval in \( 1, \ldots, n \). We then set \( intrmd(v) \) to the cached number and return.

Lines 13–18 are executed in the case where we have not entered a new node of \( t \) or we so did but did not get a cache hit. In this case, we continue as in \( RJoin \). In the case where \( x_d \) is the last variable owned by \( v \) (i.e., \( d = n \) or \( v \) is not the owner of the next variables \( x_{d+1} \)), we update the intermediate count by adding the product of the intermediate results of the children of \( v \). (Note that this product is 1 when \( v \) is a leaf.)

![Figure 2: Cached count over trie join](image-url)

Finally, lines 20–22 consider again the case where we have entered a new node of \( v \) from a previous node. At this point, we are about to go back to the previous node, and so, we check whether we should cache for \( A \) and \( \mu \). We will consider this choice later, and for now treat it as a decision obtained from a black box. If we indeed choose to cache, then the cached result \( cache[A, \mu] \) is set to \( intrmd(v) \). (This explains why we need to maintain \( intrmd(v) \) to begin with.) Next, we give an example of the execution.

**Example 3.1.** The graph on the left side of Figure 3 denotes a query \( q \) where every edge corresponds to an atom over a binary relation \( R \) with the adjacent variables (that is, \( R(x_1, x_2), R(x_2, x_3), R(x_2, x_4) \) and so on). The right side depicts a tree decomposition \((t, \chi)\) of \( q \). The bags are denoted by the ellipses and adhesions are written in the grey boxes. The tree is directed top down and ordered left to right. Observe that \((t, \chi)\) is strongly compatible with \((x_1, \ldots, x_n)\), which is our order of choice in this example. Finally, our example database \( D \) consists of four facts:

\[
R(1, 1) \quad R(1, 2) \quad R(2, 1) \quad R(2, 2)
\]
The following theorem states the correctness of our algorithm. The proof has two steps. In the first step we prove, by induction on time, that whenever we complete with a node \( v \), the number \( \text{intrmd}(v) \) is correct, that is, it stores the number of intermediate results for the subtree \( t_u \) given the assignment for adhesion(\( v \)). In the second step we prove that every unit added to total accounts for a unique tuple in \( q(D) \) and vice versa.

**Theorem 3.2.** Let \( q \) be a full CQ, \( \{ t, \chi \} \) a tree decomposition for \( q \), and \( \{ x_1, \ldots, x_n \} \) an ordering of \( \text{vars}(Q) \) such that \( (t, \chi) \) is strongly compatible with \( \prec_{\text{pos}} \). The algorithm \( \text{CachedTJCount}(q, \{ t, \chi \}, \{ x_1, \ldots, x_n \}, D) \) returns \( q(D) \).

### 3.4 Discussion

We now discuss some additional aspects of the algorithm. The decision of line 21 of whether or not to cache may entail arbitrary arguments. In our implementation we adopt a fairly naive approach: we cache only if each assignment has a support (i.e., number of occurrences) larger than a threshold. As we show in Section 5 this already gives us a great benefit; in future work we plan to investigate caching policies in depth. Also, note that the algorithm allows for arbitrary replacements or deletions from the cache.

\( \text{CachedTJCount} \) has been described for the task of counting, which is simpler than actually evaluating \( q(D) \). Nevertheless, the counting variant of the algorithm entails all of the important aspects, and evaluation would mainly differ in additional details. We discuss those now. First, in evaluation \( \text{intrmd}(v) \) will contain (representations of) tuple sets. We maintain \( \text{intrmd}(v) \) only it is actually needed, which means that we decide to cache for either \( v \) or an ancestor of \( v \). Second, instead of forwarding \( f \) in the recursive calls of lines 10 and 15, we forward a sequence of pointers to the intermediate results. Effectively, this means that in the result (which is currently \( \text{total} \)) will constitute a factorized representation [4, 20] that may be decomposed upon need (as we do in the comparisons of our experimental evaluations). Similarly, the product of line 18 is replaced with a factorized representation.

### 4. Enumerating decompositions

An important factor in the effectiveness of the caches in the algorithm \( \text{CachedTJCount} \) is their dimensionality, which is determined by the size of the adhesions. Small adhesions imply that are caches have a low dimension, and hence, the chance of a cache hit (i.e., the assignment for the variables in the adhesion has occurred in the past) is higher. There are, however, additional criteria one may wish to apply in the choice of a TD towards beneficial caching. For example, we would like to use adhesions such that their corresponding subqueries have high skew in the data, and then caching a small number of intermediate results can save a lot of repeated computation. Moreover, we would like to have a TD that is strongly compatible with an order that is estimated as good to begin with. Finally, we would like to get decompositions with a large number of bags, so that we can manipulate many caches. Therefore, instead of applying an algorithm that selects a single TD (aiming at optimizing some specific cost function), we take the approach of generating multiple TDs, estimating a cost on each, and selecting the one with the best estimate. In this section, we describe a heuristic algorithm that we use for enumerating a set of “good” TDs where goodness is tailored towards small adhesions. We are not aware of any nontrivial algorithm for enumerating TDs, except for special cases (e.g., chordal graphs [15]) that do not apply here.

---

1This statement applies to cases where the input relations have a small arity (e.g., a graph has binary relations), and less to the case of wide relations where the hypertree decomposition better captures cache effectiveness. Indeed, this section focuses on the former case, and we leave the latter to future work.
4.1 Generic Decomposition

We adopt a simple method for generating tree decompositions. The importance of this method is in the ability to plug it into an algorithm for enumerating graph separating sets, as we explain in the next section. More particularly, the algorithm calls a method for solving the side-constrained graph separation problem, or just the constrained separation problem for short, which is defined as follows. The input consists of an undirected graph \( g \) and a set \( C \) of nodes of \( g \). The goal is to find a separating set \( S \) of \( g \) (that is, a set \( S \) of nodes such that \( g - S \) is disconnected). In addition, \( S \) is required to have the property that at least one connected component in \( g - S \) is disjoint from \( C \). Hence, \( S \) is required to separate \( C \) from some nonempty set of nodes. We call \( g \) a \( C \)-constrained separating set. We denote a call for a solver of this problem by \( \text{ConstrainedSep}(g,C) \). In the next section we will discuss an actual solver. For convenience of presentation, we assume that a solver returns the pair \( (S,U) \). We then construct the Gaifman graph \( q \) of \( g \) depicted in Figure 3 and described in Example 4.2. The Gaifman graph \( g \) of \( q \) is the same as \( g \), so we refer to \( q \) as \( g \). We first call \( \text{RecursiveTD}(g,\emptyset) \). So, suppose that the pair constructed in line 1 is \( (S,U) \) where \( S = \{x_2\} \) and \( U = \{x_1\} \). In line 4 we call the algorithm recursively with \( q[S\cup V] \). Note that \( q[S\cup V] \) is simply an edge, and so it returns as the singleton decomposition. Moreover, here \( k = 1 \) and \( V = \{x_3, \ldots, x_6\} \). In line 7 we call the algorithm with \( q[S\cup V] \) and \( S = \{x_2\} \). Note that \( q[S\cup V] \) is the graph \( g \) with \( x_1 \) removed. Let \( T \) be the TD on the right of Figure 3. If the TD returned from the recursive call is \( T \) with the root removed, then \( T \) is the returned TD. In the execution with the input \( q[S\cup V] \) and \( S = \{x_2\} \), returned values of \( \text{ConstrainedSep} \) can be \( \{(x_2,x_3,x_4), \{x_3\}, \{(x_3,x_4), \{x_2\}\} \). and so on.

We note that, as defined, \( \text{RecursiveTD} \) may return a TD that contains redundancy in the form of a bag that is contained in another. In this case we can eliminate redundancy by eliminating the smaller bag and connecting its children to the larger set.

4.2 Enumerating Constrained Cuts

The algorithm \( \text{GenericDecompose}(g) \) of Figure 4 generates a single ordered TD. We transform it into an enumeration algorithm by replacing line 1 with a procedure that efficiently enumerates \( C \)-constrained separating sets, and then executing the algorithm on every such a set. A key feature of the enumeration is that it is done by increasing size of the separating sets, and hence, if we stop the enumeration of separating sets after \( k \) sets have been generated (to bound the number the generated TDs), it is guaranteed that we have seen the \( k \) smallest \( C \)-constrained separating sets. So, we are left with the task of enumerating the \( C \)-constrained separating sets by increasing size. For that, we are using a well known technique for ranked enumeration with polynomial delay.

Lawler-Murty’s procedure reduces a general ranked (or sorted) enumeration problem to an optimization problem with simple constraints. Roughly speaking, to apply the
procedure to a specific setting, one needs just to design an
efficient solution to the constrained optimization problem.
Lawler-Murty’s procedure is a generalization of Yen’s algo-
rithm [23] for finding the \( k \) shortest simple paths of a graph.
Applying the algorithm gives us the reduction described by
the following lemma. In this lemma, a “membership con-
straint” means a constraint of the form “the result contains
a node \( v \)” or “the result excludes a node \( v \).”

**Lemma 4.3.** Suppose that, given \( g \) and \( S \), a minimal \( S\) -
constrained separating set under membership constraints can
be found in polynomial time. Then the \( S\)-constrained sepa-
rating sets can be enumerated by increasing size with poly-
nomial delay.

So, to get our enumeration it suffices to devise an algo-
rithm for finding a minimal \( S\)-constrained separating set un-
der membership constraints. This can be done by a reduc-
tion to an ordinary minimum-edge-cut problem. The proof
is omitted, and will be given in the full version of the paper.
Consequently, we get the following theorem.

**Theorem 4.4.** Given \( g \) and \( S \), the \( S\)-constrained sepa-
rating sets can be enumerated by increasing size with poly-
nomial delay.

4.3 Discussion

In our implementation (described in the next section), we
enumerate TDs by bounding the maximal size of the ad-
hesions (separating sets) in the enumeration that replaces
line 1 of \texttt{GenericDecompose}. To select the actual TD, we
employ heuristic cost functions that involve the size of the
adhesion, the number bags (higher is better), and the tree’s
depth (lower is better). Moreover, we produce a variable
ordering from each ordered TD (so that the TD is strongly
compatible with the ordering) and apply the cost function
of Chu et al. [7]. We leave for future work the investigation
for optimizing the TD selection.

5. Experimental Study

Our experimental study examines the performance bene-
fits of our caching in LFTJ, which we call here CLFTJ for
short. The counting version of CLFTJ is depicted in Fig-
ure 3. To demonstrate that its performance is comparable
(and often superior) to common high-performance join al-
gorithms, we also compare it to YTD (Yannakakis and Tree
Decomposition). The study explores both count aggrega-
tions (denoted as \texttt{count}) and query evaluation. The former
computes \( |q(D)| \) and the latter computes \( q(D) \). Finally, we
explore the effect of a number of key parameters of CLFTJ.

5.1 Implementations

Our experiments are based on a vanilla implementation
of LFTJ [24]. The implementation of CLFTJ extends
the vanilla LFTJ by integrating caches, as described in Sec-
tion 3 and depicted in Figure 3. STL’s \texttt{unordered\_map} is
used for the caches, which support indices that consist of
up to two dimensions (attributes). The selection of a TD is
done as described in Section 4. We first consider caches that
store every intermediate result, and later study the impact
of bounding the cache.

YTD is implemented by combining Yannakakis’s acyclic
join algorithm [25] with TD, as described by Gottlob et al.
[9]. Each bag uses \texttt{GenericJoin\_\texttt{abbrev}} (GJ), a worst-case
optimal algorithm [15]. Furthermore, the complexity re-
quirement for the indices \texttt{seek\_\texttt{Lower\_\texttt{Bound}}} is provided by a
binary search, which is enabled through the use of cascading
vectors for the Trie. We order the attributes in a manner
where the Yannakakis’s join attributes will be higher in
the Trie, similarly to DunceCap [21]. We use the query
compiler of EmptyHeaded [1] (which applies an algorithm
similar to YTD) to generate the TD. For queries with
only two bags we use a regular join since, in this case, the
Yannakakis reduction stage generates an unnecessary over-
head. Moreover, for count queries whose tree decomposi-
tions yields more than two bags, we save the relevant result
for the matching join attributes (rather than storing full in-
termediate results). Notably, we have experimented with
alternative YTD implementations, but they all proved infe-
rior to the one described above.

5.2 Methodology

5.2.1 Workloads

In par with other join algorithms, our evaluation is based,
for the most part, on datasets from the SNAP collection [13].
The datasets consist of wiki-Vote, p2p-Gnutella04, ca-GrQc,
ego-Facebook and ego-Twitter. Since the distribution of val-
ues in SNAP dataset is highly skewed, we also use IMDB to
explore the effect of datasets that are less skewed and whose
data skew is not uniform across attributes. To this end,
we partition IMDB’s \texttt{cast\_\texttt{info}} table into a \texttt{male\_\texttt{cast}}
and a \texttt{female\_\texttt{cast}} tables, each with attributes (person\_\texttt{id}\nand movie\_\texttt{id}).

5.2.2 Queries

We experiment using 3 types of queries:

- \( (3-7)\)-path: find paths of lengths 3 to 7 for all pos-
sible nodes \( a \) and \( b \). For example, a valid 4-path can
  comprise \( E(a, b), E(b, c), E(c, d) \).
- \( (3-6)\)-cycle: find cycles of length 3 to 6 (e.g., a 4-cycle
  is \( E(a, b), E(b, c), E(c, d), E(a, d) \)).
- Random graphs: we generate random graphs using the
  Erdős-Reyni generator. The generator sets the num-
  ber of nodes to \( N \) and uses a probability \( P \) to
  generate an edge between two nodes. The graph is un-
directed, contains no self edges, and has at most one edge
  between two nodes. We use only connected graphs with
  \( N = \{5, 6\} \) and \( P = \{0.4, 0.6\} \). Random graph queries
  are denoted as \texttt{N\_\texttt{random}} \( P \). For example, \texttt{5\_\texttt{random}} \( 0.4 \)
  is a random graph where \( N = 5 \) and \( P = 0.4 \). For each
  set of parameters we generate six different graphs.

Note that we do not examine \texttt{clique} queries since they
cannot be decomposed and, therefore, CLFTJ will not offer
any advantage over LFTJ on this type of a query.

5.2.3 Algorithms

The main algorithms we compare against are LFTJ and
YTD. In addition to pure algorithms, we also experiment
with full systems:

- We thank the EmptyHeaded team [1] for sharing the code
  and helping us with the setup.
Figure 5: The runtimes observed when executing count queries using the different algorithms. YTD failed to execute 7-rand queries because its requirements exceeded the machines’ memory capacity, and its results are omitted. Bars that represent executions that times out are marked with a crisscross pattern.

- **System 1 (SYS1):** A DBMS using a worst case-optimal join algorithm as its join engine.
- **System 2 (SYS2):** Another DBMS using a worst case-optimal join algorithm which uses aggressive vector parallelism as its join engine.
- **PostgreSQL 9.3.4 (PGSQL):** An open source relational DBMS. For optimal results, the optimizer is configured to avoid merge joins and materialization.

Of course, a system has a necessarily overhead that pure algorithms do not have. We make this comparison simply to provide a context for the recorded running times. We further emphasize that our experiments are restricted to a single core, which means that we needed to restrict the above system from utilizing our cores.

We omit other DBMSs and graph engines from our experimental study, as they were already compared to these systems in a previous study [19].

5.2.4 Hardware and System setup

We use Supermicro 2028R-E1CR24N servers as our experimental platform. Each server is configured with two Intel Xeon E5-2630 v3 processors running at 2.4 GHz, 64GB of DDR3 DRAM, and is running a stock Ubuntu 14.0.4 Linux.

5.2.5 Testing Protocol

Each experiment was run three times, and the average runtime is reported. We set an execution timeout of 10 hours. Executions that timed out are highlighted, and their related speedup/slowdown is conservatively computed as if they completed the run at the timeout mark.

5.3 Experimental Results

We experimented with both counting and evaluation for full CQs. We present the results for each type separately.

5.3.1 Count queries

We first examine the performance of count queries. Figure 6 presents the runtime of 5-path, 5-cycle, and 5-rand queries on different datasets. The figure shows that CLFTJ is faster than the alternatives on 5-path and 5-cycle.

The results demonstrate the effectiveness of CLFTJ when running on datasets that are large and whose value distribution is skewed—two properties that make them highly amenable to caching. For example, the ego-Twitter dataset exhibits these properties and is therefore amenable to caching.

When comparing the performance of the different algorithms, we see that CLFTJ is consistently 2–5× faster than YTD and orders of magnitude faster than LFTJ. On the other extreme, we have the p2p-Gnutella04 dataset that is relatively small and whose value distribution is fairly balanced. For this dataset, the performance benefits of CLFTJ are moderate (for 5-rand queries, both YTD and LFTJ even marginally outperform CLFTJ).

Notably, the results are consistent across different query sizes. Figure 6 presents the runtimes observed when running {3–7}-path queries. (The figure also examines the performance of full systems, which is discussed later in Section 5.3.5 below.) For brevity, we show the results for only two of the datasets. The figure shows that the speedups delivered by CLFTJ over LFTJ even grow with the size of the query. Furthermore, it shows that CLFTJ is even faster than YTD by more than 3×.

Figure 7 examines the performance of CLFTJ for {3–7}-cycle queries (again with systems discussed in Section 5.3.5). Again, the figure shows that CLFTJ outperforms LFTJ and YTD, especially on larger cycle queries. Interestingly, we see little difference in the algorithms’ running times for small, 3-cycle queries. The reason for that is there is no tree decomposition for triangles, and CLFTJ is effectively LFTJ. Similarly, the performance of CLFTJ and YTD is comparable, as YTD uses GJ for {3–4}-cycle queries.

When comparing the benefits of CLFTJ over large cycle queries (Figure 7) and path queries (Figure 6), we see that CLFTJ delivers better speedups for paths. This is attributed to the cache dimension property (the size of adhesions). Therefore, the cache dimension for paths is set to one, and for cycles it is set to two. Notably, a cache whose dimension is one shows as much more effective. Another interesting result in the case of 5-cycle is that YTD performs worse than LFTJ (and underperforms CLFTJ). This is because YTD and GJ favor the opposite attributes order, which dramatically affects its performance.

Finally, Figure 5 presents the running times for random graph queries. The figure presents the results of two representative 5-rand queries. Over all of the 5-rand(0.4) and 5-rand(0.6) queries, CLFTJ is consistently faster by orders of
magnitude than LFTJ on average across datasets. The only exception is the p2p-Gnutella04, which CLFTJ is slightly slower by $1.7 \times$ on two queries and faster by $200 \times$ for the others. Compared to YTD there is a consistent speedup of an $\sim 8 \times$, with exception of one query where YTD is faster by $2 \times$. The results for 6-rand are consistent with 5-rand, with one exception of similar runtime on 6-clique query.

5.3.2 Query Evaluation

Query evaluation produces all the tuples in the result of the query (as opposed to counting thereof). Since our experiments measure the total query execution runtime, including the time required to generate the materialized result, the performance benefits of CLFTJ are expected to be less pronounced than for count queries. In contradistinction, the generation of intermediate results during query evaluations may affect the runtime of YTD. Specifically, YTD generates the intermediate results for all bags, even if they will not be used in the final materialized result. In contrast, a key property of LFTJ (and CLFTJ) is that the algorithm generates only intermediate assignment that can be matched along with the entire prefix assignment (according to the variable order). The performance of YTD may thus be affected by the generation of excessive intermediate results.

Importantly, we focus our exploration of query evaluation on computing the materialized result rather than storing it, and ignore queries for which the materialized result does not fit in our machines’ 64GB RAM. For this reason, we only show results for $\{3–4\}$-path and $\{3–5\}$-cycle queries, and do not discuss the ego-Twitter data set.

The results for running $\{3–4\}$-path query evaluations are depicted in Figure 8. The figure shows that, while gains over LFTJ are marginal for the smaller 3-path queries, CLFTJ outperforms LFTJ on the larger 4-path queries by up to $4.6 \times$ ($3.5 \times$ on average). The performance gap is attributed to CLFTJ’s caching, which captures frequently used inter-
mediate results. In turn, this eliminates many redundant memory operations executed by LFTJ. The CLFTJ also outperforms YTD by up to $4.6 \times$ ($3.2 \times$ on average), since the computation of YTD, which uses Yannakakis joins, becomes memory bound in the final join stages.

Figure 8 also presents the execution time of $\{3-5\}$-cycle query evaluations. The figure shows that CLFTJ is faster than LFTJ by $2000 \times$ on average for the larger 5-cycle queries. Interestingly, CLFTJ also proves faster than YTD by up to $800 \times$ ($280 \times$ on average) for 5-cycle queries. Similar to path queries, this performance gap is attributed to the excessive number of memory operations issued by the Yannakakis join algorithm in the final stages of the join.

Finally, CLFTJ also delivers performance benefits for random graphs queries. Figure 8 shows the results for representative graphs (which are consistent with the results for the other graphs). Specifically, for 5-rand(0.4) queries, CLFTJ outperforms LFTJ by 4-30 $\times$. CLFTJ is also consistently 3-4 $\times$ faster than YTD, with the exception of p2p-Gnutella04 for which the results are comparable. These trends are also consistent for denser 5-rand(0.6) random graphs. Here too, the results demonstrate the effectiveness of CLFTJ, whose runtime is, on average, $\sim 10 \times$ faster than LFTJ and $\sim 4 \times$ than YTD (CLFTJ and LFTJ runtimes are comparable for p2p-Gnutella04).

5.3.3 Dynamic Cache Size

A key benefit of LFTJ is that its memory footprint is proportional to the original dataset and does not depend on any intermediate results. This key property is preserved in CLFTJ through the ability to dynamically bound its cache sizes. Consequently, CLFTJ offers substantial speedups when executing large queries or, alternatively, when running in environments with limited memory resources. Moreover, dynamic cache bounds allow CLFTJ to support multi-tenancy of queries while preserving quality of service.

Figure 10 presents the runtime required to execute a 4-cycle and 6-cycle count aggregation queries (shown in Figure 14) over the IMDB dataset using different overall cache sizes. Importantly, due to the triangle in the lollipop graph, the treewidth is 2. We conclude that bounded caches enables CLFTJ to benefit from both worlds. One on hand, it delivers substantial speedups over LFTJ while preserving the bounded memory footprint property. On the other hand, it can execute in settings where traditional join algorithms, which store all intermediate results, either cannot execute or suffer substantial slowdowns due to disk I/O.

5.3.4 Tree Decomposition

The next experiment considers the impact of orderings and strongly-compatible TDs on the running time. The results are in Figure 11. The figure presents the runtime of CLFTJ on a $\{3,2\}$-lollipop query with different cache structures. Importantly, due to the triangle in the lollipop graph, the treewidth is 2. We compare the CLFTJ runtime on three cache structures that provide the same treewidth: a single 1-
dimension cache (CS1), two 1-dimension caches (CS2), and a cache structure with a single 1-dimension and a single 2-dimension caches (CS3). The figure shows that CS1 provides a speedup of 70–80× over LFTJ, CS2 provides a speedup of 180–190×, and CS3 only provides a speedup of 10×. These results demonstrate that the CLFTJ decomposition should not target (only) small treewidth, but rather its adhesions.

The data skew in cached attributes is another important factor that impacts CLFTJ’s performance, yet common tree decomposition algorithms do not take data properties into account. We demonstrate the effect of data skew on CLFTJ performance using the IMDB dataset, whose different attributes manifest different degrees of data skew.

Figure 14 depicts two TDs, TD1 and TD2, of two queries, 4-cycle and 6-cycle, and Figure 13 presents their respective runtimes. TD1 favors person_id for caching and TD2 favors movie_id for caching. While the decompositions are isomorphic (similar from a graph perspective), we see that their performance vary greatly. The reason for the performance variation is that the person_id attribute exhibits greater data skew than the movie_id. It is therefore more effective to apply caches to the person_id attribute.

Another interesting result is the performance impact of the order of attributes. For each TD, we selected an ordering such that the TD is strongly compatible with the ordering. Simply using LFTJ with the imposed attribute order offers a 10× speedup over the original LFTJ order. Notably, a recent study by Chu et al. [7] proposed a method to estimate the cost of attributes order in LFTJ. The method estimates the cost of TD2_order to be ~2× higher than TD1_order. The runtimes of the different attributes orders is shown in Figure 13. Hence, in these queries the cost function of Chu et al. [7] turns out to be very beneficial as parameter of choosing the TD for caching.

5.3.5 Comparison to Systems

To explore the scaling trends of the pure algorithms, compared to those of the DBMSs, we ran the queries on PGSQL (using pair-wise join), SYS1 and SYS2 (which are based on worst-case optimal join algorithms). For brevity, we show the results for only two datasets: Wiki-Vote and ego-Facebook. Notably, these are consistent with the results obtained for the other SNAP datasets.

Figure 6 shows the results for {3–7}-path count queries. The first thing to note in the figure is that the scaling of vanilla LFTJ and SYS1 are correlated. We attribute the 10× performance difference between the two to the overheads associated with running a full DBMS vs. a pure algorithm. Importantly, the figure demonstrates that the performance benefit of CLFTJ and YTD over LFTJ increases with the query size at an exponential rate. Moreover, it also shows that even though CLFTJ and YTD have similar scaling trends for path queries, CLFTJ runs almost an order of magnitude faster.
Figure 3 depicts a similar comparison for the queries \{3–6\}-cycles. Again, the figure shows consistent scaling trends for the vanilla algorithms and the DBMSs that utilize them. A comparison between SYS2 and YTD shows that SYS2 is much faster than YTD. In this case, the DBMS is faster than a pure algorithm since its implementation is massively parallelized using the processor’s wide vector unit. Due to the parallel implementation, SYS2 is much faster than LFTJ on path queries. Nevertheless, the sequential CLFTJ implementation is still comparable to SYS2 for \{5–6\}-cycles queries (and is even faster on some datasets).

6. CONCLUDING REMARKS

We have studied the incorporation of caching in LFTJ by tying an ordered tree decomposition to the variable ordering. The resulting scheme retains the inherent advantages of LFTJ (worst case optimality, low memory footprint), but allows it to accelerate performance based on whatever memory it decides to (dynamically) allocate. Our experimental study shows that the result is consistently faster than LFTJ, by orders of magnitude on large queries, and usually faster than other state of the art join algorithms.

This work gives rise to several directions for future work. These include the exploration of caching strategies, finding decompositions with beneficial caching, extension to general aggregate operators (e.g., based on the work of Joglekar et al. [10] and Khamis et al. [11]), utilizing factorized representations [5,20], and generalizing beyond joins [24].

7. REFERENCES

[1] C. R. Aberger, A. Nötzi, K. Olukotun, and C. Ré. EmptyHeaded: Boolean algebra based graph processing. CoRR, abs/1503.02368, 2015.
[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In Intl. Conf. on Very Large Data Bases (VLDB), pages 266–277, 1999.
[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logixbox system. In SIGMOD Conf., pages 1371–1382, 2015.
[4] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. SIAM J. Alg. Disc. Meth., 8(2):277–284, 1987.
[5] N. Bakshyav, T. Kocińsky, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorized databases. Proc. of the VLDB Endowment (PVLDB), 6(14):1990–2001, 2013.
[6] V. Bouchitte, D. Kratsch, H. Müller, and I. Todinca. On treewidth approximations. Discrete Applied Mathematics, 136(2-3):183–196, 2004.
[7] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In SIGMOD Conf., pages 63–78, 2015.
[8] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In Intl. Workshop on Graph-Theoretic Concepts in Computer Science (WG), pages 1–15, 2005.
[9] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In Symp. on Principles of Database Systems (PODS), pages 21–32, 1999.
[10] M. Joglekar, R. Puttagunta, and C. Ré. Aggregations over generalized hypertree decompositions. CoRR, abs/1508.07532, 2015.
[11] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. FAQ: questions asked frequently. CoRR, abs/1504.04044, 2015.
[12] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. Management Science, 18(7):401–405, 1972.
[13] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, 2014.
[14] D. Marx. Approximating fractional hypertree width. ACM Trans. on Algorithms, 6(2), 2010.
[15] Y. Matsui, R. Uehara, and T. Uno. Enumeration of perfect sequences of chordal graph. In Intl. Symp. on Algorithms and Computation (ISAAC), 2008.
[16] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. Operations Research, 16(3):682–687, 1968.
[17] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In Symp. on Principles of Database Systems (PODS), pages 37–48, 2012.
[18] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. CoRR, abs/1310.3314, 2013.
[19] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In Graph Data-management Experiences & Systems Workshop (GRADES), pages 2:1–2:8, 2015.
[20] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. ACM Trans. on Database Systems (TODS), 40(1):2, 2015.
[21] A. Perelman and C. Ré. DunceCap: Compiling worst-case optimal query plans. In SIGMOD Conf., pages 2075–2076, 2015.
[22] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In SIGMOD Conf., pages 23–34, 1979.
[23] S. Tu and C. Ré. DunceCap: Query plans using generalized hypertree decompositions. In SIGMOD Conf., pages 2077–2078, 2015.
[24] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Intl. Conf. on Database Theory (ICDT), pages 96–106, 2014.
[25] M. Yamakakis. Algorithms for acyclic database schemes. In Intl. Conf. on Very Large Data Bases (VLDB), pages 82–94, 1981.
[26] J. Y. Yen. Finding the k shortest loopless paths in a network. Management Science, 17:712–716, 1971.