# 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

While traditional algorithms for multiway join are based on reordering binary joins, more recent approaches have instantiated a new breed of "worst-case-optimal" in-memory algorithms wherein all relations are scanned simultaneously. Veldhuizen's Leapfrog Trie Join (LFTJ) is an example. 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 in join optimization, tying variable ordering to a tree decomposition of the query. While the traditional usage of tree decomposition computes the entire result for each bag, our proposed approach incorporates caching directly into LFTJ and can dynamically adjust the size of the cache. Consequently, our solution balances between memory usage and repeated computation. Our experimental study over the SNAP dataset compares between various (traditional and novel) caching policies, and shows significant speedups over state-of-the-art algorithms on both join evaluation and join counting.

## CCS Concepts

•**Information systems** → **Join algorithms;** Query optimization; *Main memory engines;*

## Keywords

Databases, trie joins, tree decomposition, caching

## 1. INTRODUCTION

Traditional optimization of multiway joins has been based on decomposing the query into smaller join queries, and combining intermediate relations. This approach has roots in Selinger's pairwise-join enumeration [26], and it includes the application of the algorithm of Yannakakis [30] over a tree decomposition of the query [13, 14]. Recent approaches have developed a new breed of in-memory algorithms wherein all relations are scanned simultaneously [1, 10, 15, 16, 21, 28, 29], featuring the complexity guarantee of *worst-case optimality*. This yardstick of efficiency has been introduced by Ngo et al. [21], and it states that for every join query, no algorithm can be asymptotically faster on the space of all databases; in that work they presented the first worst-case optimal algorithm, later termed *NPRR* [21]. Effectively, the running time is bounded by the *AGM bound* [5] that determines the maximal number of tuples in the multiway join of relations with given sizes.

Leapfrog Trie Join (LFTJ) [29] is another worst-case-optimal algorithm, introduced by LogicBlox and implemented in the company's product [3]. 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-structured indices over the relations allow to efficiently determine whether the next variable in consideration can be assigned a value that is consistent with the assignments to the previous variables. (We give a detailed description of LFTJ in Section 2). Beyond being worst-case optimal, LFTJ has two important features. First, it 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. Moreover, these tuples are not even needed in the case of common aggregate queries (e.g., count the number of tuples in the result).

Yet, 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 repetitions of computations. The repeated traverals back and fourth on the trie index generate excessive memory traffic, which has detrimental impact on the performance of database systems [2]. 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 dataset generates over $45 \cdot 10^9$ memory accesses, whereas running the same query using tree decomposition and Yannakakis's join generates less than $16 \cdot 10^9$ accesses. (The implementation of both algorithms is discussed in Section 5.)

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 at its disposal towards memoization. However, it is not clear how LFTJ can cache intermediate results (without com-

 Published in Proc. *20th International Conference on Extending Database Technology (EDBT)*, March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. 

puting and storing full results of subqueries as done in other algorithms [1, 28]). Intuitively, the challenge lies in the fact that every iteration involves a different partial assignment, and variables are interdependent through the query structure. Our solution is inspired by recent developments in the theory of join optimization, relating to worst-case optimality and tree decomposition [15, 16, 28]. But unlike existing work, we do not apply the join algorithm on each bag independently (which would result in high memory consumption due to intermediate results), but rather execute LFTJ as originally designed.

Specifically, to enable effective caching our approach applies the following steps. We first build a Tree Decomposition (TD) for the query, in a manner that we discuss later on. Intuitively, a TD transforms the query into a tree structure by grouping together several relations, where each group is called a *bag*. We then execute LFTJ as usual, but throughout the execution we use caches (deploying a caching/eviction policy) for partial assignments. More formally, each bag of the TD is assigned a cache, and the application of the cache happens when the iteration over the variables enters a new bag. The correctness of the cache usage (i.e., the fact that the intermediate assignments are consistent with the current assignment in construction) is crucially based on two properties.

1. The variable ordering is required to be *compatibile* with the TD. Intuitively, compatibility means that the variable order is consistent with the preorder of the TD. (The formal definition is in Section 2.)
2. Each cache applies to partial assignments only for the variables it contains (for evaluation) or the subtree underneath (for counting).

For TD computation, there is 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 [4, 13], and efficient algorithms exist for special cases and different approximation guarantees [8]. Other notions include decompositions that approximate the minimal *fractional hypertree width* [15, 19]. In our case, a TD defines a caching scheme, and various factors determine the effectiveness of this scheme. Caches are more reusable in the presence of skewed data, and hence, data statistics can be used to estimate the goodness of a TD. Importantly, our caches correspond to the *adhesions* (parent-child intersections); in order to better capture opportunities of a high skew (and a high hit rate), we give precedence to keys from a domain of a smaller dimension, and hence, we favor smaller adhesions. Due to these arguments, we chose not to use any specific algorithm that generates a single tree decomposition, but rather to explore *a large space of such decompositions*. We devise a heuristic algorithm for enumerating TDs, tailored primarily towards small adhesions. Once such a collection of TDs are generated, we deploy a cost function that takes various factors into account, including the skew-based cost model of Chu et al. [10].

We experiment on three types of queries: paths, cycles and random. In par with recent studies on join algorithms, we base our experiments on datasets from the SNAP [18] and IMDB workloads. We explore several attributes of our cached LFTJ, such as the cache size and the eviction policy. We also experiment with the count version of the queries. Our experiments compare among LFTJ, with and without caching, and Yannakakis's algorithm over the TD (as in DunceCap [24,28]), as well as other various systems and engines (LogicBlox [3], PostgreSQL [27] and EmptyHeaded [1]). The results show consistent improvement compared to LFTJ (in orders of magnitude on large queries), as well as general improvement

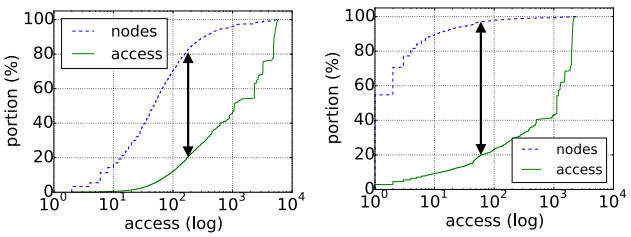


**Figure 1: Mass-count disparity plots for value accesses on the evaluation of a 5-path (left) and a 5-cycle (right) over the SNAP ca-GrQc dataset; the double-headed arrows indicate that 80% of the accesses are applied to just 20% (left) and 5% (right) of the nodes**

compared to the examined algorithms and systems. The only alternative that outperforms our implementation on a large portion of the count queries is EmptyHeaded, as it implements a parallel implementation using the *Single Instruction Multiple Data* (SIMD) parallelization model (while our implementation applies standard sequential computation). We defer hardware utilization of this sort to future research.

While retaining the inherent features of LFTJ, our caching dramatically reduces the memory accesses. For illustration, running a 5-cycle count query generates only $1.4 \cdot 10^9$ memory accesses, which is over $30\times$ fewer accesses than vanilla LFTJ (and over $10\times$ fewer accesses than TD with Yannakakis's algorithm). Figure 1 provides some intuition on why we are able to establish such a dramatic improvement with a modest memory usage. The figure depicts *mass-count disparity plots* [11] for value accesses on the evaluation of a 5-path (left) and a 5-cycle (right) over the SNAP ca-GrQc dataset, which has a graph structure. The x-axis corresponds to the number of accesses. A tick at number $n$ refers to the nodes that are accessed *at most* $n$ times by our algorithm (node popularity); the dashed curve shows the fraction of such nodes among all nodes, and the solid curve shows the fraction of accesses to such nodes among all accesses. On the left plot we can see, for example, that 80% of the accesses are directed to around 20% of the most popular nodes (as indicated by the double-headed arrow), and on the right one we can see that 80% of the accesses are applied to 5% of the most popular nodes!

To summarize, our contributions are as follows. First, we extend LFTJ with caching, without compromising the key benefits. Our caching is executed alongside LFTJ, and its size can be determined dynamically according to memory availability. This is achieved by combining LFTJ with a TD, a suitable variable ordering, and a suitable set of target variables for each cache. Second, 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. Third, we present a thorough experimental study that evaluates the effect of caching on LFTJ, on both evaluation and counting, and compares the results to state-of-the-art join algorithms.

## 2. BACKGROUND

In this section we give preliminary definitions and notation that we use throughout the paper.

### 2.1 Conjunctive Queries

We study the problem of *evaluating* a Conjunctive Query (CQ), and the problem of *counting* the number of tuples in the result of

a CQ. As in recent work on worst-case optimal joins [21, 22, 29], 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 $R(\tau_1, \ldots, \tau_k)$ with $R$ being a $k$-ary relation name and each $\tau_j$ being either a constant or a variable. In the remainder of this paper, we say simply "CQ" instead of "full CQ." We denote by $\mathsf{vars}(\varphi_j)$ the set of variables that occur in $\varphi_j$, and we denote by $\mathsf{vars}(q)$ the union of the sets $\mathsf{vars}(\varphi_j)$ over all atoms $\varphi_j$ in $q$ (i.e., the set of all variables appearing in $q$).

Let $q$ be a CQ. A *partial assignment* for $q$ is function $\mu$ that maps every variable in $\mathsf{vars}(q)$ to either a constant value or *null* (denoted $\perp$). If $\mu$ is a partial assignment for $q$, then we denote by $q_{[\mu]}$ the CQ that is obtained from $q$ by replacing every variable $x$ with $\mu(x)$, if $\mu(x) \neq \perp$, and leaving $x$ intact if $\mu(x) = \perp$. If $X$ is a subset of $\mathsf{vars}(q)$, then we denote by $\mu_{|X}$ the restriction of $\mu$ to $X$; that is, $\mu_{|X}$ is defined only over $X$, and $\mu_{|X}(x) = \mu(x)$ for all $x \in X$.

For a CQ $q$, a partial assignment that maps every variable to a (nonnull) constant is called a *complete* assignment. Let $D$ be a database over the same relation names as $q$. *Evaluating* $q$ over $D$ is the task of producing the set $q(D)$, which consists of all complete assignments $\mu$ such that all the ground subgoals of $q_{[\mu]}$ are facts (tuples) of $D$; such an assignment is also called an *answer* (for $q$ over $D$). *Counting* $q$ over $D$ is the task of computing the number of answers, that is, $|q(D)|$.

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

EXAMPLE 2.1. Our running example uses the following CQ $q$ over a single binary relation $R$.

$$R(x_1, x_2), R(x_2, x_3), R(x_2, x_4), R(x_3, x_4), R(x_3, x_5), R(x_4, x_6)$$

Observe that $q$ does not have constant terms. This CQ is illustrated in the graph of Figure 2(a); in this case the graph is also the Gaifman graph of $q$ (since $q$ is binary). The graph is also the Gaifman graph of the following CQ:

$$R(x_1, x_2), S(x_2, x_3, x_4), R(x_3, x_4), R(x_3, x_5), R(x_4, x_6)$$

Let $\mu$ be the partial assignment that maps $x_1$ and $x_2$ to the constants 1 and 2, respectively, and the other variables to $\perp$. Then $q_{[\mu]}$ is

$$R(1, 2), R(2, x_3), R(2, x_4), R(x_3, x_4), R(x_3, x_5), R(x_4, x_6)\,.$$

Our example database $D$, depicted in Figure 2(b), consists of a single relation. It can verified that $q(D)$ contains the following assignments $\mu_1$ and $\mu_2$:

- $\mu_1$: $x_1 \mapsto 1$, $x_2 \mapsto 2$, $x_3 \mapsto 1$, $x_4 \mapsto 2$, $x_5 \mapsto 3$, $x_6 \mapsto 1$
- $\mu_2$: $x_1 \mapsto 1$, $x_2 \mapsto 2$, $x_3 \mapsto 2$, $x_4 \mapsto 1$, $x_5 \mapsto 1$, $x_6 \mapsto 3$

If we remove from $\mu_1$ and $\mu_2$ the assignments for $x_1$ and $x_2$, then we get answers in $q_{[\mu]}(D)$ for the above defined $\mu$. □

## 2.2 Ordered Tree Decompositions

Let $q = \varphi_1, \ldots, \varphi_m$ be a CQ. A *Tree Decomposition* (*TD*) of $q$ is a pair $\langle t, \chi \rangle$ where $t$ is a tree and $\chi$ is a function that maps every node of $t$ to a subset $\chi(v)$ of $\mathsf{vars}(q)$, called a *bag*, such that both of the following hold.

- For each $\varphi_j$ there is a node $v$ of $t$ with $\mathsf{vars}(\varphi_j) \subseteq \chi(v)$.
- For each $x$ in $\mathsf{vars}(q)$, the nodes $v$ with $x \in \chi(v)$ induce a connected subtree of $t$.

An *ordered TD* of a CQ $q$ is pair $\langle t, \chi \rangle$ defined similarly to a TD, except that $t$ is a rooted and ordered tree. We denote the root of $t$ by $\mathrm{root}(t)$. Let $v$ be a node of $t$. We denote by $t_{|v}$ the subtree of $t$ that is rooted at $v$ and contains all of the descendants of $v$. If

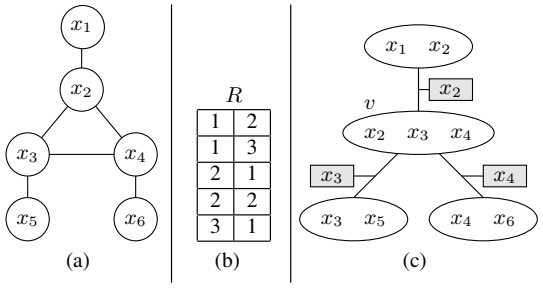


**Figure 2: (a) Example of a CQ $q$; (b) A database $D$ with a single relation; (c) An ordered tree decomposition of $q$**

$v$ is a non-root node, then the *parent adhesion* of $v$ (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 $\mathrm{adhesion}(v)$. Every set $\mathrm{adhesion}(u)$, where $u$ is a non-root node of $t$, is called an *adhesion* of $\langle t, \chi \rangle$.

EXAMPLE 2.2. We continue with our running example. Figure 2(c) depicts an ordered tree decomposition $\langle t, \chi \rangle$ of the query $q$ of Figure 2(a). The tree $t$ has four nodes, and the order is top down, left to right. The root is the top node with the bag $\{x_1, x_2\}$. To verify that it is indeed a tree decomposition of $q$, the reader needs to check that every edge in Figure 2(a) is contained in some bag of $\langle t, \chi \rangle$. The adhesions of $\langle t, \chi \rangle$ are shown in the gray boxes. Let $v$ be the node of $t$ with $\chi(v) = \{x_2, x_3, x_4\}$. The parent adhesion of $v$, which we denote by $\mathrm{adhesion}(v)$, is the singleton $\{x_2\}$. □

Let $q$ be a 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 $\prec_{\mathsf{pre}}$. For a variable $x$ in $\mathsf{vars}(q)$, the *owner bag* of $x$, denoted $\mathrm{owner}(x)$, is the minimal node $v$ of $t$, under $\prec_{\mathsf{pre}}$, such that $x \in \chi(v)$. For a node $v$ of $t$, we denote by $\mathrm{owned}(v)$ the set of variables $x$ that have $v$ as the owner. We say that $\langle t, \chi \rangle$ is *compatible* with an ordering $\langle x_1, \ldots, x_n \rangle$ if $i < j$ whenever $\mathrm{owner}(x_i) \prec_{\mathsf{pre}} \mathrm{owner}(x_j)$. We may also say that the ordering $\langle x_1, \ldots, x_n \rangle$ is compatible with $\langle t, \chi \rangle$ if the latter is compatible with the former.

EXAMPLE 2.3. Consider the given ordering $\langle x_1, \ldots, x_6 \rangle$ of the variables in our running example (Figure 2), and the TD $\langle t, \chi \rangle$ of Figure 2(c). The preorder of $t$ is given by $\{x_1, x_2\}$, $\{x_2, x_3, x_4\}$, $\{x_3, x_5\}$, $\{x_4, x_6\}$. We have $\mathrm{owner}(x_3) = \mathrm{owner}(x_4) = v$, and $\mathrm{owned}(v) = \{x_3, x_4\}$. Note that $\mathrm{owner}(x_2) \neq v$ since $x_2$ occurs already in the root of $t$ (and therefore $\mathrm{owner}(x_2) = \mathrm{root}(t)$). □

## 2.3 Trie Join

We now describe the Leapfrog Trie Join (LFTJ) algorithm [29]. Our description is abstract enough to apply to the *tributary join* of Chu et al. [10]. Let $q = \varphi_1, \ldots, \varphi_m$ be a CQ. The execution of LFTJ is based on a predefined ordering $\langle x_1, \ldots, x_n \rangle$ of $\mathsf{vars}(q)$.

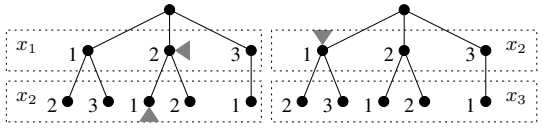


**Figure 3: The trie structures for subgoals $R(x_1, x_2)$ and $R(x_2, x_3)$, respectively, in the running example**

**Algorithm** TrieJoin$(q, \langle x_1, \ldots, x_n \rangle, \mathcal{T})$

1: **for** $d = 1, \ldots, n$ **do**
2: $\quad \mu(x_d) := \perp$
3: RJoin$(1)$

**Subroutine** RJoin$(d)$

1: **if** $d = n + 1$ **then**
2: $\quad$ **print** $\mu$
3: $\quad$ **return**
4: **for all** matching values $a$ for $x_d$ in $\mathcal{T}$ **do**
5: $\quad$ position $\mathcal{T}$ on $x_d \mapsto a$
6: $\quad \mu(x_d) := a$
7: $\quad$ RJoin$(d + 1)$
8: reset $x_d$ pointers in $\mathcal{T}$

**Figure 4: Trie join**

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 [10]. Moreover, in our instantiation of LFTJ we will use orderings with specific properties.

For every subgoal $\varphi_k$, LFTJ maintains a trie structure on the corresponding relation $r$. Each level $i$ of the trie corresponds to a variable $x_j$ in $\mathsf{vars}(\varphi_k)$, and holds values that can be matched against $x_j$. Whenever $x_j$ 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.

EXAMPLE 2.4. Figure 3 depicts two of the tries used for evaluating the CQ $q$ of Example (2.1), of our running example. The left trie is for $R(x_1, x_2)$ and $R(x_2, x_3)$. (The reader should ignore the gray triangles for now; we discuss them in the next example.) In this case, the tries are identical (as they index the same relation), but they are used differently during query evaluation. Each level of the trie corresponds to a variable and a corresponding attribute. The path root$\rightarrow$1$\rightarrow$2 corresponds to the tuple $R(1, 2)$, and root$\rightarrow$2$\rightarrow$1 corresponds to $R(2, 1)$. □

LFTJ applies a sequence of unary joins, called *leapfrog joins*, as follows. Each trie holds an iterator, initialized by pointing to the root. A mapping $\mu$, which is initialized with nulls, is maintained throughout the execution. First, all the subgoals that contain $x_1$ advance their iterators in the first level until a matching value $a$ is found (i.e., all iterators point to $a$), and $\mu(x_1)$ is set to $a$. The matching value is found efficiently in a technique referred to as *leapfrogging* [29]. The algorithm then proceeds recursively[1] with the CQ $q_{[\mu]}$ by proceeding to the next matching value, and so on, until all variables are assigned values (and then $\mu$ is printed) or no matching values are found; then backtracking takes place by advancing the previous iterator. 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* [21].

EXAMPLE 2.5. Continuing Example 2.4, the gray triangles in Figure 3 show a possible positioning of the pointers on the tries during the execution. Here, the pointer for $x_1$ is set on 2, the pointers of $x_2$ in both tries is set on 1 (which is a matching value found), and next a matching value for $x_3$ will be sought in under the pointed node in the right trie (and the other tries). □

[1]The actual algorithm of [29] is not recursive, but rather applies a single procedure call. Recursion simplifies our presentation.

We refer the reader to the original publication [29] for more details on LFTJ. In this paper, it suffices to regard LFTJ abstractly as depicted in Figure 4. We call the algorithm of Figure 4 *trie join* and denote it by TrieJoin. This algorithm updates the global partial assignment $\mu$ using the subroutine RJoin (Recursive Join).

# 3. CACHING IN TRIE JOIN

In this section we devise an algorithm that incorporates caching within TrieJoin (Figure 4). We first discuss the intuition.

## 3.1 Intuition

The general idea is as follows. Let $q$ be the evaluated CQ, and let $\langle x_1, \ldots, x_n \rangle$ be $\mathsf{vars}(q)$ in the order of iteration. Consider a point in the iteration where we complete the assignment for $x_1, \ldots, x_j$ ($j < n$), and suppose that we have already encountered the assignment for $x_i, \ldots, x_j$ in the past for some $i$ such that $1 < i < j$. We would like to be able to reuse the past assignments, at least for a few of the next variables, say $x_{j+1}, \ldots, x_k$, instead of searching again for matches. Integrating simple memoization in the algorithm will not suffice. The problem is that the assignments for $x_{j+1}, \ldots, x_k$ may depend not just on those for $x_i, \ldots, x_j$, but rather on the assignments for variables in $x_1, \ldots, x_{i-1}$, and so reusing past assignments may lead to incorrect results (false assignments).

The above problem is avoided as follows. First, we deploy an ordered TD $\langle t, \chi \rangle$, and use an ordering $\langle x_1, \ldots, x_n \rangle$ that is compatible with $\langle t, \chi \rangle$ (as defined in Section 2.2). Second, cache keys are assignments to sequences $x_i, \ldots, x_j$ of variables *only* if the set $\{x_i, \ldots, x_j\}$ is an adhesion of some node $v$ of $t$. Finally, we cache assignments *only* for the variables $x_{j+1}, \ldots, x_k$ that are owned by $v$. Due to the nature of the TD, we can rest assured that the assignments to $x_{j+1}, \ldots, x_k$ are independent of the assignments to $x_1, \ldots, x_{i-1}$ (once we know the assignments for $x_i, \ldots, x_j$).

EXAMPLE 3.1. Consider again our running example around Figure 2. At some point in the execution of TrieJoin we construct the assignment $\mu$ with $\mu(x_1) = 1$ and $\mu(x_2) = 2$, and then continue to the rest of the variables in order. The next assignments we construct are $x_3 \mapsto 1$ and $x_4 \mapsto 2$. Once we are done with the complete assignments for the extended $\mu$, we construct the assignments $x_3 \mapsto 2$ and $x_4 \mapsto 1$, and later on $x_3 \mapsto 2$ and $x_4 \mapsto 2$. Later in the execution, we encounter the assignment $\mu'$ with $\mu(x_1) = 2$ and $\mu(x_2) = 2$. Since $\mathrm{adhesion}(x) = \{x_2\}$, we check to see whether there is a cache for $x_2 \mapsto 1$, and if so, then it tells us exactly where to position the pointers for $x_3$ and $x_4$ (which are the variables owned by $v$) in each of the possibilities (which are $(1, 2)$, $(2, 1)$, $(2, 2)$ and nothing else). We may similarly have a cache for $\{x_3\}$ (lower left adhesion) and for $\{x_4\}$ (lower right adhesion). □

Caching could be obtained by computing the complete join for every bag (using TrieJoin), and then joining the intermediate results using an algorithm for acyclic joins such as Yannakakis [30], as done in DunceCap [24, 28]. However, we wish to control the memory consumption and avoid storing the complete joins of subqueries. Our algorithm executes TrieJoin ordinarily, yet caches results during the execution based on a deployed caching policy.

COMMENT 3.2. Compatibility of the variable ordering with the TD has implications on the trie structures, which need to be consistent with the variable ordering [29]. Therefore, similarly to EmptyHeaded [15], the design of our tries depends on the TD. As building

**Algorithm** CacheTrieJoin$(q, \langle x_1, \ldots, x_n\rangle, \langle t, \chi\rangle, \mathcal{T})$

```
1: for d = 1, ..., n do
2:     μ(x_d) := ⊥
3: for all nodes v of t do
4:     cache_v := ∅
5: CacheRJoin(1)
```

**Subroutine** CacheRJoin$(d)$

```
1: if d = n + 1 then
2:     print μ
3:     return
4: v := owner(x_d)
5: {x_l, x_{l+1}, ..., x_k} := owned(v)
6: α := adhesion(v)
7: if v ≠ root(t) and d = l then
8:     if μ_|α is a cache hit in cache_v then
9:         for all cached entries μ' in cache_v(μ_|α) do
10:            for i = l, ..., k do
11:                μ(x_i) := μ'(x_i)
12:            AdjustTries(T, μ')
13:            CacheRJoin(k + 1)
14:        reset x_l, ..., x_k pointers in T
15:        return
16: for all matching values a for x_d in T do
17:    position T on x_d ↦ a
18:    μ(x_d) := a
19:    CacheRJoin(d + 1)
20:    if v ≠ root(t) and d = k then
21:        ApplyCachePolicy(cache_v, μ_|α, μ_|owned(v))
22: reset x_d pointers in T
```

**Figure 5:** TrieJoin **with caching**

the trie may take considerable time, our approach matches the scenario where the join is known in advance, but not the data (which is common in Web applications where queries arise due to user interaction with the UI). Another matching scenario is where the relations are narrow (e.g., graphs), and then we can compute in advance multiple trie structures (which is the design choice of EmptyHeaded [15]) and load the proper ones upon need. □

## 3.2 Algorithm

We now turn to a more formal description of our algorithm, which we call CacheTrieJoin, and is depicted in Figure 5. The algorithm extends upon the algorithm of Figure 4 in the sense that when no caching takes place, the two algorithms coincide. The algorithm takes as input a CQ $q$, a variable ordering $\langle x_1, \ldots, x_n\rangle$, an ordered TD $\langle t, \chi\rangle$ that is compatible with $\langle x_1, \ldots, x_n\rangle$, and a trie structure $\mathcal{T}$ for a database $D$. The algorithm prints all tuples in $q(D)$. The algorithm uses a cache, denoted $cache_v$, for every node $v$ of $t$, for caching computed assignments for the variables owned by $v$. The algorithm CacheTrieJoin simply initializes a global partial assignment $\mu$ and each $cache_v$, and calls the subroutine CacheRJoin (the caching version of RJoin of Figure 4), which we describe next.

The first part of the algorithm, lines 1–3, tests whether we are done with the variable scan (that is, the algorithm is called with the index $n + 1$) and, if so, prints $\mu$. Now assume that $d \leq n$. So the currently iterated variable is $x_d$. We denote by $v$ the owner of $x_d$, and by $\alpha$ the adhesion of $v$ (as defined in Section 2). Moreover, we assume that the nodes owned by $v$ are $x_l, \ldots, x_k$ in ascending indices. Observe that $\text{owned}(v)$ is indeed a consecutive set of variables, since the order is compatible with $t$.

In lines 8–15 we handle the case where we have just entered $v$ from a different node of $t$, which means that $x_d$ is the first node $x_l$ owned by $v$, and $v$ is not the root (that is, $v > 1$). From our construction, the adhesion of $v$ is already assigned values in $\mu$ (again due to compatibility), and we check whether there is a cache hit for $\mu_{|\alpha}$ (the restriction of $\mu$ to $\alpha$) in $cache_v$. If indeed there is a cache hit, then in lines 9–15 we scan the cache that contains all assignments $\mu'$ that we have already computed for $\mu_{|\alpha}$. For each such $\mu'$, we extend $\mu$ with $\mu'$ and adjust the trie structure $\mathcal{T}$ according to $\mu'$. By *adjusting* $\mathcal{T}$ we consider every variable $x_i$ in $x_l, \ldots, x_k$ and if $x_i$ is later used for a join, then we position the pointer precisely where it should have been if we scanned the trie and got to $\mu'(x_i)$;[2] and if $x_i$ is not used for a future join, then we do nothing. As an example, in our running example (Figure 2), we ignore $x_5$ if we have a cached assignment for it.

Lines 16–22 are executed in the case where we have not just entered $v$, or we do but had a cache miss on line 8. In this case, we continue exactly as in RJoin (Figure 4), but we also test whether $x_d$ is the last variable owned by $v$ (i.e., $x_d$ is $x_k$). If so, we either cache or do not cache the assignment $\mu_{|\text{owned}(v)}$ based on the underlying caching policy for $\mu_{|\alpha}$. Observe that this action may lead to an eviction of a previously stored entry for some $\mu'_{|\alpha}$.

EXAMPLE 3.3. We will now show how the scenario of Example 3.1 is realized in the algorithm CacheTrieJoin, where we consider again our running example (Figure 2). The algorithm first calls CacheRJoin$(1)$, and the execution is the same as in RJoin, all the way until we reach the call to CacheRJoin$(3)$ where we have $\mu(x_1) = 1$ and $\mu(x_2) = 2$. Observe that $\text{owner}(x_3) = v$, which is a non-root node, $\text{owned}(v) = \{x_3, x_4\}$ (hence, $l = 3$ and $k = 4$). Also note that $\text{adhesion}(v) = \{x_2\}$. The test of line 7 is true, but that of line 8 is false since $cache_v$ is empty at that point. So, we continue to lines 16–19 and apply the different assignments for $x_3$, starting with $x_3 \mapsto 1$. We then call CacheRJoin$(4)$, where we find the assignment $x_4 \mapsto 2$. The test of line 20 is true, since $x_4$ is the last owned by $v$. Therefore, we may decide (based on the applied caching policy) to cache the entry $x_2 \mapsto 2$ in $cache_v$, and then we store there the assignment $(x_3, x_4) \mapsto (1, 2)$. We later store in that entry the assignment $(x_3, x_4) \mapsto (2, 2)$.

Later in the execution, we call CacheRJoin$(3)$ when we have $\mu(x_1) = 2$ and $\mu(x_2) = 2$. We may then find out that in $cache_v$ we have cached the entry of $x_2 \mapsto 2$, and we simply use the two tuples $\mu'$ that maps $(x_3, x_4)$ to $(1, 2)$ and to $(2, 2)$, as in lines 9–13. However, if there is a cache miss then we repeat the above first execution of CacheRJoin$(3)$. □

The following theorem states the correctness of the algorithm CacheTrieJoin. The proof is by a fairly straightforward application of the basic separation properties of a tree decomposition.

THEOREM 3.4. *Let $q$ be a CQ, $\langle x_1, \ldots, x_n\rangle$ an ordering of* vars$(Q)$ *and $\langle t, \chi\rangle$ a TD compatible with $\langle x_1, \ldots, x_n\rangle$. Let $D$ be a database, and $\mathcal{T}$ a trie structure for* TrieJoin. *Algorithm* CacheTrieJoin$(q, \langle x_1, \ldots, x_n\rangle, \langle t, \chi\rangle, \mathcal{T})$ *prints $q(D)$.*

## 3.3 Counting

We now describe a variation of CacheTrieJoin (Figure 5) for *counting* the number of tuples in $q(D)$. The counting algorithm,

[2] Technically, this is done by storing the position with $\mu'$ in $cache_v$.

which we refer to as CacheTJCount, is depicted in Figure 5. The input is the same as that of CacheTrieJoin, and the flow is very similar. There are, however, a few key differences, and our explanation (next) will focus on these.

The algorithm CacheTJCount uses some new global variables and data structures. The variable $total$ counts the joined tuples throughout the execution, and in the end stores the required number. For every non-root node $v$ of $t$ we have a counter $intrmd(v)$ that stores the intermediate count of the assignments to the variables owned by the nodes in $t_{|v}$ (i.e., the subtree of $t$ that consists of $v$ and all of its descendants), given the assignment to $\text{adhesion}(v)$ in the current iteration. More precisely, let $i$ be the maximal number such that $x_i$ is in the adhesion of $v$, and consider a partial assignment $\mu$ that is nonnull on precisely $x_1, \ldots, x_i$. In an iteration where $\mu$ is constructed, $intrmd(v)$ will eventually hold the number of assignments $\mu'$ that TrieJoin can assign to the variables owned by the nodes in $t_{|v}$. As $\langle t, \chi \rangle$ is compatible with the ordering $\langle x_1, \ldots, x_n \rangle$, this number is the same for all assignments $\mu$ that agree on the adhesion $\alpha$ of $v$. The counter $intrmd(v)$ holds the correct value once we are done with the variables owned by $v$. Another fundamental difference from CacheTJCount is that now $cache_v$ stores a natural number (rather than a collection of assignments) for each assignment $\mu_\alpha$; this number is precisely the value of $intrmd(v)$ once we are done with the variables in $t_{|v}$.

Following the initialization, the algorithm calls the subroutine CacheRJoinCount, which is the counting version of CacheRJoin. The input takes not only the variable index $d$, but also a factor $f$ that aggregates cached intermediate counts. When we are done scanning all of the variables (i.e., we reach line 2), the factor $f$ is added to $total$. When we are at the first node owned by the current non-root owner $v$ (lines 7–13), we reset the counter $intrmd(v)$. If we have a cache hit for $\mu_{|\alpha}$ in $cache_v$, then we copy the number $cache_v(\alpha)$ into $intrmd(v)$, multiply $f$ by this number, and jump directly to the first index outside of $t_{|v}$ (line 12). This skipping is where compatibility is required, since it ensures that the nodes owned by $t_{|v}$ constitute a consecutive interval in $\langle 1, \ldots, n \rangle$.

As previously, lines 14–20 are executed in the case where we have not just entered $v$, or experience a cache miss. We then continue as in RJoin. However, if $x_d$ is the last variable owned by $v$, then we update the intermediate count by adding the product of the intermediate results $intrmd(c)$ of the children $c$ of $v$. (Note that this product is 1 when $v$ is a leaf.) Finally, in lines 22–23 we consider again the case where we have just entered a node $v$. Then, we are about to go back to the previous node, and so we apply the caching policy to possibly cache the number $intrmd(v)$ for $\mu_{|\alpha}$ in $cache_v$. (This is why we maintain $intrmd(v)$ to begin with.)

EXAMPLE 3.5. We illustrate CacheTJCount on our running example (Figure 2). On CacheRJoinCount$(1,1)$ we set (in lines 16–17) $\mu(x_1) = 1$ and call CacheRJoinCount$(2,1)$, where we set $\mu(x_2) = 2$ and call CacheRJoinCount$(3,1)$. We reach line 8 and initialize $intrmd(v)$ to 0. We have a cache miss (as the cache is empty), and we reach line 14, where CacheRJoinCount$(4,1)$ is called with $\mu(x_3) = 1$. From there we call CacheRJoinCount$(5,1)$ with $\mu(x_4) = 2$. Let $c_l$ and $c_r$ be the left and right children of $v$, respectively. When the call returns, we have $intrmd(c_l) = 2$ and $intrmd(c_r) = 2$, as $x_5$ can be mapped to 2 and 3 and $x_6$ can be mapped to 1 and 2. At this point $total$ is equal to 4, since the scan has ended four times. We then reach line 18 (since $x_4$ is the last owned by $v$) and set $intrmd(v) = 0 + 2 \times 2 = 4$. Similarly, after the call to CacheRJoinCount$(4,1)$ with $\mu(x_3) = 2$ there will be a call with $\mu(x_4) = 1$ and $intrmd(v)$ will be incremented by another 4, and so will be the case with $\mu(x_4) = 2$. So, when we reach line 23 for $d = 3$, we may cache the number 12 as $cache_v(\mu_\alpha)$.

**Algorithm** CacheTJCount$(q, \langle x_1, \ldots, x_n \rangle, \langle t, \chi \rangle, \mathcal{T})$

```
1: for d = 1, ..., n do
2:     μ(x_d) := ⊥
3: for all nodes v of t do
4:     cache_v := ∅
5:     intrmd(v) := 0
6: total := 0
7: CacheRJoinCount(1, 1)
8: return total
```

**Subroutine** CacheRJoinCount$(d, f)$

```
1: if d = n + 1 then
2:     total := total + f
3:     return
4: v := owner(x_d)
5: {x_l, x_{l+1}, ..., x_k} := owned(v)
6: α := adhesion(v)
7: if v ≠ root(t) and d = l then
8:     intrmd(v) := 0
9:     if μ_|α is a cache hit in cache_v then
10:        intrmd(v) := cache_v(μ_|α)
11:        m := max{i | owner(x_i) is in t_|v}
12:        CacheRJoinCount(m + 1, f · cache_v(μ_|α))
13:        return
14: for all matching values a for x_d in T do
15:     position T on x_d ↦ a
16:     μ(x_d) := a
17:     CacheRJoinCount(d + 1, f)
18:     if v ≠ root(t) and d = k then
19:         let c_1, ..., c_k be the children of v in t
20:         intrmd(v) := intrmd(v) + ∏_{i=1}^{k} intrmd(c_i)
21: reset x_d pointers in T
22: if v ≠ root(t) and d = l then
23:     ApplyCachePolicy(cache_v, μ_|α, intrmd(v))
```

**Figure 6: Cached count over trie join**

The next time CacheRJoinCount$(3,1)$ is called with $\mu(x_2) = 2$ (i.e., when $\mu(x_1) = 2$), we check $cache_v$ and may find that $cache_v(\mu_\alpha)$ exists (i.e., cache hit) with $cache_v(\mu_\alpha) = 12$. If so, we reach line 12 and call CacheRJoinCount$(7, 1 \times 12)$. As $7 > n$, we skip to line 2 and add 12 to $total$. If there is a cache miss for $\mu_\alpha$ in $cache_v$, there might still be a cache hit when we call CacheRJoinCount$(5,1)$ with $\mu(x_3) = 2$, and then we immediately call CacheRJoinCount$(6, 1 \times 2)$ on line 12, as 6 is the minimal index outside the subtree of $c_l$ (which contains only $c_l$). □

The following theorem states the correctness of CacheTJCount.

THEOREM 3.6. *Let $q$ be a CQ, $\langle x_1, \ldots, x_n \rangle$ an ordering of $\text{vars}(Q)$ and $\langle t, \chi \rangle$ a TD that is compatible with $\langle x_1, \ldots, x_n \rangle$. Let $D$ be a database, and $\mathcal{T}$ a trie structure for* TrieJoin. *Algorithm* CacheTJCount$(q, \langle x_1, \ldots, x_n \rangle, \langle t, \chi \rangle, \mathcal{T})$ *computes $|q(D)|$.*

The proof is more involved than Theorem 3.4, and has two steps. We first prove, by induction on time, that whenever we complete iterating the variables of $v$, the number $intrmd(v)$ is correct (i.e., it is the number of intermediate results for $t_{|v}$ given the assignment

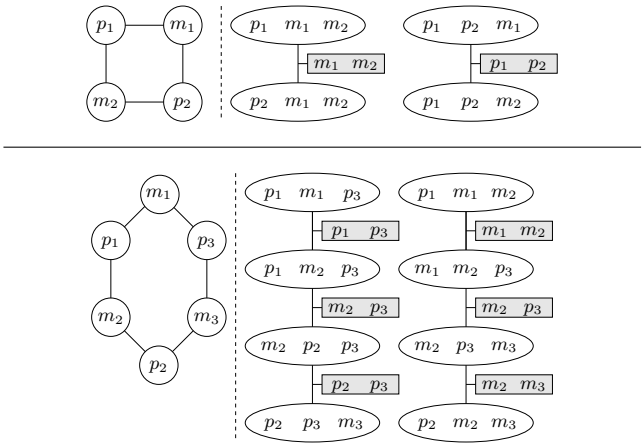


**Figure 7: 4-cycle (top) and 6-cycle (bottom) queries on IMDB, each with two isomorphic TDs**

for $\mathrm{adhesion}(v)$). In the second step we show that every unit added to $total$ accounts for a unique tuple in $q(D)$ and vice versa.

# 4. DECOMPOSITION

We now discuss the challenge of finding a TD $\langle t, \chi \rangle$ and a compatible variable ordering. A typical TD algorithm aims at optimising some specific cost function such as *generalized/fractional hypertree width* [12, 15]. In our case, an important factor in the effectiveness of the caches in our algorithms is their dimensionality, which is determined by the size of the adhesions. To better capture opportunities of a high skew and hit rate, we give precedence to keys from a domain of a smaller dimension, and hence, we favor smaller adhesions. There are, however, additional criteria beyond the topological properties of the TD. For example, we would like to use adhesions such that their corresponding subqueries have high *skews* 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 compatible with an order that is estimated as good to begin with. For a (rather extreme) illustration, Figure 7 depicts two TDs of two queries, 4-cycle and 6-cycle, over the IMDB dataset (see Section 5), where $m$ and $p$ denote movie and person identifiers, respectively. The left TD favors persons for caching and the right favors movies for caching. While the decompositions are isomorphic, their performance of counting varies greatly: 4-cycle took around 40 seconds with the left TD, and around 4,000 with the right one; and 6-cycle took around 600 seconds and 27,000 on the left and right TDs, respectively.

We take the approach of generating many TDs, estimating a cost on each, and selecting the one with the best estimate. In our implementation (described in the next section), we deploy a heuristic cost function that ranks TDs based on three criteria, in a lexicographic manner: the maximal size over the adhesions (lower is better), the number of bags (higher is better), the sum of adhesions (lower is better), and the cost function of Chu et al. [10] for some variable ordering that is compatible with the TD.

## 4.1 Enumerating TDs

We now describe our technique for enumerating ordered TDs. In future work we plan to compare our enumeration to a recent one by Carmeli et al. [9]. We begin with a simple method for generating a single TD. The two common heuristics to generating TDs are *graph separation* and *elimination ordering* [7]. We adopt the former, as it will later allow us to plug in an algorithm for enumerating separating sets of a graph. 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 $g - S$ (obtained by removing from $g$ every node of $S$) 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 $S$ a *$C$-constrained* separating set. We denote a call for a solver of this problem by $\mathsf{ConstrainedSep}(g, C)$. We later discuss an actual solver. For convenience, we assume that a solver returns the pair $\langle S, U \rangle$, where $U$ is the set of all nodes in the connected components of $g - S$ that intersect with $C$.

**Subroutine** $\mathsf{GenericTD}(g, C)$

1: $\langle S, U \rangle \leftarrow \mathsf{ConstrainedSep}(g, C)$
2: **if** $S = \perp$ **then**
3: **return** the singleton decomposition of $g$
4: $\langle t_0, \chi_0 \rangle := \mathsf{RecursiveTD}(g[S \cup U], C \cup S)$
5: let $V_1, \ldots, V_k$ be the connected comps. of $g - (S \cup U)$
6: **for** $i = 1, \ldots, k$ **do**
7: $\langle t_i, \chi_i \rangle := \mathsf{RecursiveTD}(g[S \cup V_i], S)$
8: let $t$ be obtained from $t_0, t_1, \ldots, t_k$ by connecting the root of $t_0$ to the root of $t_i$ for all $i > 1$
9: $\chi := \cup_{i=0}^{k} \chi_i$
10: **return** $(t, \chi)$

**Figure 8: Tree decomposition via adhesion selection**

The algorithm, called $\mathsf{GenericTD}(g, C)$, is depicted in Figure 8. It takes as input a graph $g$ and a set $C$ of nodes that is empty on the first call. The algorithm returns an ordered TD of $g$ with the property that the root bag contains all nodes in $C$. So, the algorithm first calls $\mathsf{ConstrainedSep}(g, C)$. Let $\langle S, U \rangle$ be the result. It may be the case that the subroutine decides that no (good) $C$-constrained separating set exists, and then the returned $S$ is null ($\perp$). In this case, the algorithm returns the TD that has only the nodes of $g$ as the single bag. This case is handled in lines 1–3. Suppose now that the returned $\langle S, U \rangle$ is such that $S$ is a $C$-constrained separating set. Denote by $V_1, \ldots, V_k$ the connected components of $g - (S \cup U)$. The algorithm is then applied recursively to construct several ordered TDs. First, an ordered tree decomposition $\langle t_U, \chi_U \rangle$ of the induced subgraph of $S \cup U$, which we denote by $g[S \cup U]$, such that the root contains $C \cup S$ (line 4). Second, for $i = 1, \ldots, k$, an ordered tree decomposition $\langle t_i, \chi_i \rangle$ of $g[S \cup V_i]$ (the induced graph of $S \cup V_i$) such that the root bag contains $S$ (lines 5–7). Finally, in lines 8–10 the algorithm combines all of the tree decompositions into a single tree decomposition (returned as the result), by connecting the root of each $\langle t_i, \chi_i \rangle$ to $\langle t_U, \chi_U \rangle$ as a child of the root.

The algorithm $\mathsf{GenericTD}(g, C)$ of Figure 8 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 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 of the generated TDs), *it is guaranteed that we have seen the $k$ smallest $C$-constrained separating sets*.

We are then left with the task of enumerating the $C$-constrained separating sets by increasing size. For that, we have devised an algorithm that establishes the following complexity result.

THEOREM 4.1. *The $S$-constrained separating sets of a graph $g$ can be generated by increasing size with polynomial delay.*

Our algorithm uses the well known technique for ranked enumeration with polynomial delay, namely the Lawler-Murty's procedure [17, 20][3] that 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 a *constrained* optimization problem. Due to lack of space, we omit the details and defer them to the long version of the paper.

# 5. EXPERIMENTAL STUDY

Our experimental study examines the performance benefits of our approach and algorithms. We compare our implemented algorithms to state-of-the-art solutions, and explore the effect of a number of key parameters and design choices.

## 5.1 Algorithms and Systems Evaluated

Our evaluation compares between implementations of several join algorithms, as listed below. All implementations were compiled using g++ 4.9.3 with the -O3 flag.

Our algorithms are CacheTrieJoin for CQ evaluation (Figure 5) and CacheTJCount for CQ counting (Figure 6). These implementations extend the vanilla implementation of LFTJ [29], which we describe below. We refer to the implementations by the acronyms CTJ-E and CTJ-C, respectively. The caches are implemented using STL's unordered_map. The computation of a TD is as described in Section 4. If no bound is mentioned for the cache size, then no eviction takes place (and every partial assignment is cached). We compare against the following alternatives.

**LFTJ**: We use a vanilla implementation of LFTJ [29]. Our implementation uses C++ STL *map* as the underlying Trie data structure. Notably, this implementation adheres to the complexity requirements of LFTJ.

**YTD**: This algorithm combines Yannakakis's acyclic join algorithm [30] with a TD, as described by Gottlob et al. [14]. The implementation is based on DunceCap [24]. For each intermediate join (bag) a worst-case optimal algorithm is used. The complexity requirement for the indices seekLowerBound is provided by a binary search, enabled through the use of the cascading vectors for the Trie. We use the query compiler from EmptyHeaded [1] (which uses a YTD-like algorithm) to generate the TD and variable ordering. For queries with only two bags we use a regular join since, in this case, the Yannakakis reduction stage generates an unnecessary overhead. Moreover, for count queries whose TDs yield more than two bags, we save the relevant result for the matching join attributes (rather than storing full intermediate results). Notably, we have experimented with alternative YTD implementations, but they all proved inferior to the one described above.

**YTD-Par**: EmptyHeaded [1] is a state-of-the-art graph query engine that operates as a parallel implementation of DunceCap [24], with optimizations for graph databases. We view it as a *query engine* rather than a pure algorithm, since the implementation is tied to the hardware: it parallelizes the execution through the Single-Instruction Multiple-Data (SIMD) model. Parallel operations are executed using the *vector unit* available on modern Intel processor cores. Specifically, *each core* on our test platform (Intel Xeon E5-2630 v3) includes a 256-bit vector unit that executes 8 integer (4-byte) operations in parallel.

[3]Lawler-Murty's procedure is a generalization of Yen's algorithm [31] for finding the $k$ shortest simple paths of a graph.

| Dataset | #Nodes | #Edges | Category |
|---|---|---|---|
| ca-GrQc | 5,242 | 14,496 | Collaboration net |
| p2p-Gnutella04 | 10,876 | 39,994 | P2P net |
| ego-Facebook | 4,039 | 88,234 | Social net |
| wiki-Vote | 7,115 | 103,689 | Social net |
| ego-Twitter | 81,306 | 1,768,149 | Social net |
| imdb-Actresses | 2,714,695 | 4,700,000 | Movies |
| imdb-Actors | 3,539,013 | 7,000,000 | Movies |

**Table 1: Dataset (SNAP) statistics**

In addition to pure algorithms, we also experiment with full systems. Pure algorithm implementations avoid the overhead associated with a full DBMS. We make this comparison simply to provide a context for the recorded running times.

**LB-LFTJ**: LogicBlox (LB) 4.3.18 [3]: A commercial DBMS configured to use LFTJ as an its join engine.

**LB-FAQ**: LogicBlox (LB) 4.3.18 configured to use InsideOut [16] as its join engine.

**PGSQL**: *PostgreSQL* [27] is an open-source relational DBMS (version 9.3.4). For query evaluation (as opposed to *count*), we use the *curser* API of PGSQL to avoid storage of join results in memory.

Other popular DBMSs and graph engines were compared to the above systems in a previous study [22], and were shown to be inferior in performance. Hence, we omit the other DBMSs from our experimental study. We further emphasize that our experiments explicitly restricted all algorithms and systems to utilize only a single core on the test machine, which does not affect YTD-Par SIMD parallelization.

## 5.2 Methodology

The setup and methodology we adopted in our experimental study are as follows.

**Workloads.** In par with other studies on join algorithms our evaluation is based, for the most part, on datasets from the SNAP collection [18], similarly to Nguyen et al. [22]. The datasets consist of wiki-Vote, p2p-Gnutella04, ca-GrQc, ego-Facebook and ego-Twitter. Table 1 gives some basic statistics on the datasets. As the distribution of values 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 *cast_info* table into a *male_cast* and a *female_cast* tables, each with attributes *(person_id* and *movie_id)*. We exclude the TPC-C and TPC-H benchmarks as the join queries in these benchmarks are small.

**Queries.** Our datasets can be viewed as graphs, and so, we experiment using 3 types of CQs (again, consistently with Nguyen et al. [22]). The first type, denoted $n$-path for $n = 3, \ldots, 7$, finds all paths of length $n$. For example, the 4-path CQ is

$$E(x, y), E(y, z), E(z, w)\,.$$

The second type is $n$-cycle, where $n = 3, \ldots, 6$, and the query finds cycles of length 3 to 6. For example, the 4-cycle CQ is

$$E(x, y), E(y, z), E(z, w), E(w, x)\,.$$

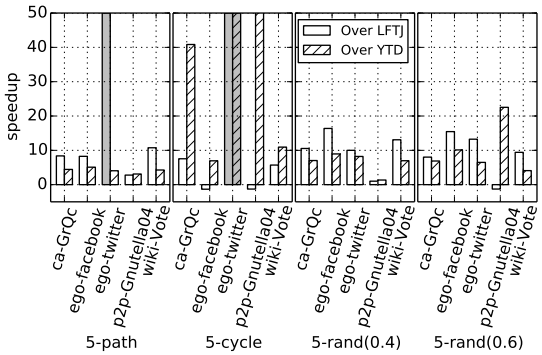


**Figure 9: The speedups obtained with CTJ-E over LFTJ and YTD for full query evaluation. Bars that represent executions that timed out are marked as gray.**

The third type consists of *random* CQs. We generate such CQs by forming a graph pattern using the Erdös-Reyni generator. The generator takes $n$ nodes and adds an edge between every two nodes, independently, with a specified probability $p$. The graph is undirected and without self loops. We use only connected graphs with $n = 5$ and $n = 6$, and with $p = 0.4$ and $p = 0.6$. Random graph queries are denoted as $n$-rand$(p)$. For each set of parameters we generate four different graphs. We do not examine *clique* queries as these cannot be decomposed, and hence our algorithms are the same as LFTJ in this case.

**Hardware and system setup.** Our experimental platform uses Supermicro 2028R-E1CR24N servers. 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.

**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.

## 5.3 Experimental Results

We start by experimenting with unlimited caches on query evaluation of CQs. Next, we compare different cache sizes, caching policies and other cache attributes.

Query evaluation produces all the tuples in the result of the query (as opposed to counting thereof). We focus our exploration of query evaluation on *computing* the materialized result rather than *storing* it. With the help of the related parties, the algorithms and systems were configured to ignore the final result and not store it. The only exception is YTD-Par, for which we could not disable the materialization of the final result. It is therefore not shown in our examination of query evaluation.

Figure 9 presents the results for running query evaluation of 5-path and 5-cycles queries. The figure shows that for 5-path queries, CTJ-E outperforms YTD by 4× and LFTJ by over 9×. The performance gap is attributed to CTJ-E's caching, which captures frequently used intermediate results. CTJ-E's caching eliminates redundant scans of the Trie structure that occur in LFTJ. CTJ-E also outperforms YTD by up to 4.6× (3.2× on average), because the computation of YTD becomes memory bound in the final join stages due to the memory complexity of the Yannakakis joins.

For 5-cycle queries, Figure 9 shows that CTJ-E is faster than LFTJ by an average of 8× for ca-GrQc, twitter and wiki datasets. For the facebook and p2p-Gnutella04 datasets, however, CTJ-E experiences a small slowdown. Finally, CTJ-E outperforms YTD by 26× on average. The reason is that YTD's Yannakakis and the worst-case optimal join algorithm used by YTD, favor the opposite attributes order, which dramatically affects its performance.

| Query | Algorithms | | | Systems | | |
|---|---|---|---|---|---|---|
| | CTJ-E | YTD | LFTJ | LB-FAQ | LB-LFTJ | PGSQL |
| 3-path | 23 | 2× | 1.3× | 46× | 34× | 10116× |
| 4-path | 133 | 4× | 5× | 17× | 26× | 27679× |
| 5-path | 2222 | 4× | 8× | 23× | 19× | t/o |
| 6-path | 78528 | 4× | 10× | 23× | 23× | t/o |
| 7-path | 3265542 | 4× | 10× | t/o | t/o | t/o |
| 4-cycle | 558 | 1.1× | 1.9× | 9× | 5× | 4409× |
| 5-cycle | 4125 | 41× | 8× | 11× | 19× | 11526× |
| 6-cycle | 84248 | 9× | 14× | 40× | 37× | 564× |

| Query | Algorithms | | | Systems | | |
|---|---|---|---|---|---|---|
| | CTJ-E | YTD | LFTJ | LB-FAQ | LB-LFTJ | PGSQL |
| 3-path | 72 | 1.5× | 3× | 14× | 17× | 18355× |
| 4-path | 791 | 4× | 9× | 23× | 21× | 59157× |
| 5-path | 29458 | 4× | 11× | 26× | 24× | t/o |
| 6-path | 1.33e+06 | 4× | 11× | 26× | t/o | t/o |
| 7-path | t/o | t/o | t/o | t/o | t/o | t/o |
| 4-cycle | 2192 | 0.7× | 1.7× | 5× | 4× | 2312× |
| 5-cycle | 27855 | 11× | 6× | 3× | 14× | t/o |
| 6-cycle | 415783 | 4× | 17× | 5× | 44× | t/o |

**Figure 10: CTJ-E runtimes (in msecs) for {3–7}-path and {4–6}-cycle queries and relative runtimes for compared solutions (i.e., $m\times$ means $m$ times slower than CTJ-E), for ca-GrQc (top) and Wiki (bottom) datasets. Timeout (t/o) means over 10 hours. YTD-Par is omitted from the comparison as it always stores the materialized result.**

CTJ-E also delivers performance benefits for sparse random pattern queries. Figure 9 shows the results for representative graphs (which are consistent with the results for the other graphs). Specifically, for 5-rand$(0.4)$ queries, CTJ-E outperforms LFTJ by 5× on average. CTJ-E is also consistently 3–4× faster than YTD, with the exception of p2p-Gnutella04 for which the results are comparable. These trends are consistent for denser 5-rand$(0.6)$ random graphs. Here too, the results demonstrate the effectiveness of CTJ-E, whose runtime is, on average, 10× faster than LFTJ and 7× than YTD (CTJ-E and LFTJ runtimes are comparable for p2p-Gnutella04).

Figure 10 presents the results of query evaluation for {3–7}-path and {4–6}-cycle queries over the ca-GrQc (top) and Wiki (bottom) datasets for different algorithms and systems. For brevity, we show the results for only two datasets: ca-GrQc and Wiki. These results are consistent with the results obtained for the other SNAP datasets.

The figure shows that the performance benefits of CTJ-E over LFTJ increase with the size of the query. CTJ-E is 10× faster than LFTJ for 7-path queries and 14× for 6-cycle queries. Compared to YTD, CTJ-E speedup is 4× for 7-path queries and 4–9× for 6-cycle queries. The only case where CTJ-E is slower than another algorithm is the small 4-cycle query, for which YTD is faster by 30% on the Wiki dataset. Importantly, these results are consistent across the other datasets, excluding p2p-Gnutella04 for which the algorithms are comparable.

Figure 10 also compares the performance of our algorithms to that of full DBMSs (PGSQL, LB-LFTJ, and LB-FAQ). We observe that the speedups are even larger (as expected, due to system overhead). Notably, the ratio between the performance of LFTJ and LB-LFTJ is more or less constant, showing that the system overhead here accounts for around 2× slowdown.

On average, CTJ-E is over 20× faster than LB-FAQ and LB-LFTJ for all path queries, and 3-44× faster for all cycle queries. An even more extreme speedup is evident when comparing to PGSQL, where CTJ-E is consistently 3–5 orders of magnitude faster.

| dataset | 5-path 25% | 5-path 10% | 5-path 1% | 5-cycle 25% | 5-cycle 10% | 5-cycle 1% |
|---|---|---|---|---|---|---|
| ca-GrQc | 1.7× | 3× | 18× | 1.9× | 3× | 5× |
| p2p-Gnutella04 | 4× | 5× | 6× | 1.3× | 1.3× | 1.3× |
| ego-twitter | 6× | 9× | 18× | 2× | 2× | 2× |
| wiki-Vote | 1.2× | 1.3× | 3× | 4× | 4× | 4× |

**Table 2: Slowdown due to cache sizes with LRU, over unlimited cache size for 5-path and 5-cycle query evaluation**

To conclude, we have shown that CTJ-E is substantially faster than the alternatives. Furthermore, the performance benefits of CTJ-E increase with the size of the query.

### *5.3.1 Cache Parameters*

Tuning the parameters of the CTJ-E cache (e.g., cache size, eviction policy, cache partitioning) do not affect the correctness of the CTJ-E. Instead, these parameters only affect the caching efficiency of CTJ-E and, by proxy, the performance of the algorithm. The caching of partial results in CTJ-E thus presents a tradeoff between memory consumption and performance. Interestingly, LFTJ and YTD represent the two extremes of this classic tradeoff. On one hand, LFTJ caches no partial or intermediate results but rather repeatedly scans the Trie to regenerate partial results. On the other hand, YTD must maintain all intermediate results generated by the individual joins on each bag. As a result, its memory consumption is even higher than CTJ-E with an unbounded cache.

In this section we explore the memory-performance tradeoff by examining the impact of the different cache parameters on the performance of LFTJ. Unless stated otherwise, the memory allocated for the cache is evenly partitioned across the individual caches.

**Cache size.** The size of the CTJ-E cache is, naturally, the primary parameter that affects caching performance. We explore this parameter's impact on performance by bounding the cache size to 1%, 10%, and 25% of the size needed to store all partial results. For example, a 5-cycle query running on the twitter dataset requires 476MB to cache all partial results. We thus examine CTJ-E performance when bounding the total cache size to 4.76MB (1%), 47.6MB (10%) and 119MB (25%). In this experiment, all bounded caches use the least-recently-used (LRU) eviction policy.

Table 2 presents the performance of CTJ-E with bounded cache size for representative queries and datasets. The performance is presented as the slowdown over a run with an unbounded cache.

As expected, the table shows that performance degrades when reducing the cache size. For example, the performance of a 5-path query on the ca-GrQc dataset (0.4MB unbounded cache) slows down by 1.7× when bounding the cache to 25% of full capacity, by 2.5× with 10% of full capacity, and by 18.4× with 1% of full capacity. The performance degradation is less acute in other cases. A 5-cycle query running on ca-GrQc (8.8MB unbounded cache) slows down by 1.9×, 3×, and 5× for caches bounded at 25%, 10%, and 1%, respectively, of full capacity. In other cases, the performance impact of a bounded cache is fairly constant regardless of the bound. For example, running a 5-cycle query on the twitter dataset (476MB unbounded cache) results in a slowdown of 2.4–2.5× for caches bounded at 1–25% of full capacity.

In summary, Table 2 shows that bounding the cache size to 25% of its full capacity only yields an average slowdown of ∼3× over an unbounded CTJ-E run. Bounding the cache size even further to 10% of an unbounded cache results in an average slowdown of ∼2.7× over the performance obtained with an unbounded cache. Notably, the performance obtained with a 10% bound is still superior to LFTJ, as well as to YTD 5-cycle queries, and comparable to YTD for 5-path queries.

| dataset | 5-path |
|---|---|
| ego-twitter | 1.8× |
| ca-GrQc | 1.3× |
| p2p-Gnutella04 | 1.3× |
| wiki-Vote | -1.1× |

| dataset | 5-cycle |
|---|---|
| ego-twitter | 11× |
| ca-GrQc | 2× |
| p2p-Gnutella04 | -1.3× |
| wiki-Vote | 4× |

**Table 3: Speedup for LRU over RANDOM on cache bounded to 10% for 5-path and 5-cycle query evaluation**

**Eviction policy.** We now turn to examine the impact of the cache eviction policy on overall CTJ-E performance. Specifically, we compare the performance obtained with both *LRU* and *Random* eviction policies (the Random policy, as its name suggests, randomly selects a cache entry to evict with uniform distribution). We note that we have experimented with other eviction and insertion policies, some based on statistical analysis of the datasets, but none provided much better results than the classic LRU policy.

Table 3 presents the LRU performance as speedup over Random. For brevity, we only show results for a 10% cache bound. The table shows that for path queries LRU outperforms Random by 1.4× on average. This is because most cached values will be effective in path queries on the datasets we tested, and due to the overhead of the LRU bookkeeping. On cycle queries, LRU outperforms Random by 4.5× on average. We therefore choose to use the LRU eviction policy with bounded CTJ-E caches.

**Cache partitioning.** The final parameter we explore is the allocation of memory among caches. We test the LRU performance speedup for bounded cache size, which is partitioned between the caches in three different configurations. The first configuration (1×) divides the allocated memory equally between the caches. The second (2×), divides the allocated memory between the caches, such that each level is bounded to 2× of the size of the level above it. Here, a cache *level* means the position in the pre-order of the TD. As an example, for the 5-cycle query on the twitter dataset, we allocate a total of 476MB. In the second configuration, the first cache will be bounded to 1/3 (158MB) and the second cache will be bounded to 2/3 (317MB). The last configuration (5×) is similar to the previous, but with a scale of 5× instead of 2×.

Table 4 shows the results for CTJ-E with LRU eviction, bounded to 5%, over the different partitioning configurations. The results show that for small caches of equal size, the caches can become ineffective due to thrashing. The results also show that a different cache partitioning that allocates more memory to greater level caches, such as 2× and 5×, can improve the performance by 10%-3×. With these configurations CTJ-E outperforms LFTJ even with

| dataset | 5-path 1x | 5-path 2x | 5-path 5x | 5-cycle 1x | 5-cycle 2x | 5-cycle 5x |
|---|---|---|---|---|---|---|
| ca-GrQc | 11.6× | 3.4× | 3× | 10.4× | 9.2× | 8.3× |
| wiki-Vote | 2.3× | 1.5× | 1.6× | 5.5× | 5× | 5× |
| p2p-Gnutella04 | 5× | 5× | 5× | 1.3× | 1.3× | 1.3× |
| ego-twitter | 14× | 8× | 5.5× | 2.4× | 2.4× | 2.4× |

**Table 4: Slowdown due to different cache partitioning with LRU bounded to 5% of the full cache capacity, over unlimited cache size for 5-path and 5-cycle query evaluation)**

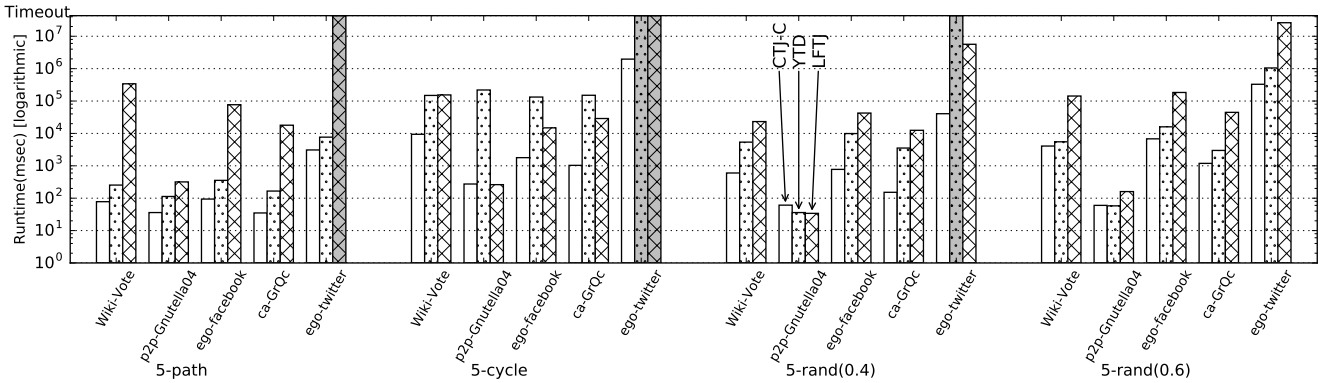


**Figure 11: Runtimes for count queries using the different algorithms. Gray bars represent executions that timed out.**

very small memory allocation. The reason we observed is that a cache in a greater level is accessed more often, and therefore accounts for a larger portion of the recurring joins. Note that different cache partitions do not affect queries on p2p-Gnutella04, since CTJ-E caches are less effective for this dataset. This crude allocation depicts the importance of dynamically allocating the memory between the caches, which we plan to pursue in future research.

**Summary.** We conclude that bounded caches enable CTJ-E to benefit from both worlds. On one 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.

## 5.4 Results on Count Queries

We now examine the performance benefits of CTJ-C for count queries. Figure 11 presents the runtime of 5-path, 5-cycle, and 5-rand queries on different datasets. It shows that CTJ-C executes the queries substantially faster than the alternatives for all datasets except p2p-Gnutella04. CTJ-C is faster than LFTJ by over an order of magnitude. When compared to YTD, CTJ-C is typically 2–5× faster, with the exception of 5-rand(0.4) over p2p-Gnutella04, where CTJ-C results in a marginal slowdown.

The distinction between the datasets is rooted in their value distribution. Skewed value distributions are more amenable to caching. Specifically, when some values appear frequently in multiple tuples, caching partial walks through the LFTJ Trie will likely prevent redundant walks over the Trie. For example, the ego-Twitter dataset exhibits such skew. For this dataset, CTJ-C is consistently 2–5× faster than YTD and orders of magnitude faster than LFTJ.

On the other hand, when the distribution of values across the dataset is not skewed, as is the case with p2p-Gnutella04, caching partial values have little benefit. Indeed, for this dataset the performance benefits of CTJ-C are moderate (for 5-rand queries, both YTD and LFTJ even marginally outperform CTJ-C). The results demonstrate the effectiveness of CTJ-C when running on datasets whose value distribution is skewed.

Figure 11 compares the algorithms when running two representative 5-rand random graph queries. Comparing CTJ-C with the LFTJ algorithm, we see that CTJ-C is consistently faster by orders of magnitude. The only exception is the p2p-Gnutella04 that, as discussed above, exhibits a balanced value distribution. When comparing CTJ-C to YTD, we observe an average speedup of ∼8×. Again, the only exception is the p2p-Gnutella04 dataset. Notably, the results for 6-rand (not shown) are consistent with 5-rand.

The performance benefits of CTJ-C are consistent across different query sizes. Figure 12 presents the runtimes for {3–7}-path and {3–6}-cycle queries. For brevity, we show the results for only two of the datasets. (The figure also shows the performance of DBMSs, which is discussed below.) The figure shows that for path queries CTJ-C is consistently 3× faster than YTD. Moreover, CTJ-C is orders of magnitude faster than LFTJ, and the performance benefits only increase with the size of the query.

For {3–7}-cycle queries, Figure 12 shows that CTJ-C outperforms LFTJ and YTD, especially on larger cycle queries. Interestingly, we see little difference in the running times for 3-cycle queries. The reason for that is there is no tree decomposition for triangles, and CTJ-C effectively behaves like LFTJ. Similarly, the performance of CTJ-C and YTD is comparable for 3-cycle queries.

When comparing the benefits of CTJ-C over large cycle and path queries (Figure 12), we see that CTJ-C 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 is shown to be much more effective. 5-cycle queries present another interesting result. For these queries YTD performs worse than LFTJ (and CTJ-C). The reason is that YTD's Yannakakis and the worst-case optimal join algorithm used by YTD, favor the opposite attributes order, which dramatically affects its performance.

Figure 12 shows that the performance benefit of CTJ-C and YTD over LFTJ increase with the query size at an exponential rate. Moreover, while CTJ-C and YTD have similar scaling trends for path queries, CTJ-C is an order of magnitude faster for {5–6}-cycle.

**Comparison to systems and engines.** To explore the scaling trends of the pure algorithms compared to those of DBMSs, we ran the queries on PGSQL (using pairwise join), LB-LFTJ, LB-FAQ (worst-case optimal join algorithms) and YTD-Par (parallel implementation of YTD). 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 12 shows the results for {3–7}-path count queries. The first thing to note in the table is that the scaling of vanilla LFTJ and LB-LFTJ are correlated. We attribute the 4–10× ratio in performance between the two to overheads associated with running a full DBMS vs. a pure algorithm. A comparison between YTD-Par and YTD shows that YTD-Par is much faster than YTD. This to be expected, as YTD-Par engine is a parallel implementation of YTD pure algorithm, using the processor's wide vector unit. Due to the parallel implementation, YTD-Par is also faster than CTJ-C and LFTJ on path queries. Nevertheless, the sequential CTJ-C implementation is comparable to YTD-Par for {5–6}-cycles queries (and is even faster on some datasets).

| | Algorithms | | | Systems and engines | | | |
|---|---|---|---|---|---|---|---|
| query | CTJ-C | YTD | LFTJ | LB-FAQ | LB-LFTJ | PGSQL | YTD-Par |
| 3-path | 36 | 3× | 5× | 9× | 54× | 19× | 0.08× |
| 4-path | 58 | 3× | 133× | 5× | 615× | 364× | 0.05× |
| 5-path | 78 | 3× | 4362× | 8× | 21113× | 11161× | 0.09× |
| 6-path | 97 | 3× | 157691× | 7× | t/o | 402735× | 0.10× |
| 7-path | 119 | 4× | t/o | 7× | t/o | t/o | 0.13× |
| 3-cycle | 24 | 1× | 1× | 13× | 13× | 41× | 0.21× |
| 4-cycle | 1474 | 0.85× | 3× | 7× | 7× | 3× | 0.16× |
| 5-cycle | 9401 | 16× | 16× | 1.66× | 43× | 13× | 2× |
| 6-cycle | 28615 | 11× | 235× | 1× | 617× | 242× | 0.90× |

| | Algorithms | | | Systems and engines | | | |
|---|---|---|---|---|---|---|---|
| query | CTJ-C | YTD | LFTJ | LB-FAQ | LB-LFTJ | PGSQL | YTD-Par |
| 3-path | 26 | 5× | 4× | 14× | 39× | 22× | 0.12× |
| 4-path | 48 | 4× | 62× | 10× | 308× | 174× | 0.06× |
| 5-path | 94 | 4× | 818× | 7× | 7973× | 2116× | 0.07× |
| 6-path | 119 | 3× | 15086× | 6× | t/o | 56534× | 0.08× |
| 7-path | 150 | 3× | t/o | 6× | t/o | t/o | 0.09× |
| 3-cycle | 48 | 1.1× | 1× | 12× | 12× | 42× | 0.13× |
| 4-cycle | 569 | 1.31× | 1× | 5× | 5× | 4× | 0.12× |
| 5-cycle | 1785 | 74× | 8× | 3× | 37× | 26× | 12× |
| 6-cycle | 4639 | 54× | 81× | 3× | 366× | 208× | 5× |

**Figure 12: CTJ-C runtimes (in msecs) for {3–7}-path and {3–6}-cycle count queries and relative runtimes for compared solutions (i.e., $m\times$ means $m$ times slower than CTJ-C), shown for Wiki (top) and Facebook (bottom) datasets. *t/o* indicates runtime over 10 hours (timeout).**

On average, CTJ-C is over 39× faster than LB-LFTJ for all path queries, and 5-208× faster for all cycle queries. CTJ-C speedup over LB-FAQ is 7× and 4× on average for path and cycle queries, respectively. Compared to PGSQL, CTJ-C is consistently 3–5 orders of magnitude faster for big cycle and path queries.

# 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 further exploration of different caching strategies, different TD enumerations and cost functions, extension to general aggregate operators (e.g., based on the work of Joglekar et al. [15] and Khamis et al. [16]), and generalizing beyond joins [29]. A highly relevant work is that on *factorized representations* [6,23,25], which we can incorporate in two manners. First, our caches can hold factorized representations instead of flat tuples. Second, the final result can be factorized by itself, and in that case our caching is likely to become even more effective, since it will save the cycles then we effectively spend on de-factorizing our cached results.

**Acknowledgments.** We are very grateful to LogicBlox for sharing their code, and especially to Hung Ngo and Martin Bravenboer from LogicBlox for helpful suggestions and help with their system. We are also grateful to Christopher Aberger and Chris Ré from Stanford University for insightful discussions and assistance in setting up EmptyHeaded. This research is supported by the Israeli Ministry of Science, Technology, and Space. Yoav Etsion is supported by the Center for Computer Engineering at Technion. Benny Kimelfeld is a Taub Fellow supported by the Taub Foundation, and supported by the Israeli Science Foundation, Grants #1295/15 and #1308/15.

# 7. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446, 2016.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *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 logicblox system. In *SIGMOD*, 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] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[6] N. Bakibayev, T. Kociský, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[7] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations i. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[8] V. Bouchitté, D. Kratsch, H. Müller, and I. Todinca. On treewidth approximations. *Discrete Applied Mathematics*, 136(2-3):183–196, 2004.

[9] N. Carmeli, B. Kenig, and B. Kimelfeld. On the enumeration of all minimal triangulations. *CoRR*, abs/1604.02833, 2016.

[10] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[11] D. G. Feitelson. Metrics for mass-count disparity. In *MASCOTS*, pages 61–68, 2006.

[12] G. Gottlob, G. Greco, and F. Scarcello. Pure nash equilibria: Hard and easy games. *J. Artif. Intell. Res. (JAIR)*, 24:357–406, 2005.

[13] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, pages 1–15, 2005.

[14] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *PODS*, pages 21–32, 1999.

[15] M. R. Joglekar, R. Puttagunta, and C. Ré. AJAR: aggregations and joins over annotated relations. In *PODS*, pages 91–106, 2016.

[16] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016.

[17] 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.

[18] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, 2014.

[19] D. Marx. Approximating fractional hypertree width. *ACM Trans. on Algorithms*, 6(2), 2010.

[20] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[21] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[22] 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 *GRADES*, pages 2:1–2:8, 2015.

[23] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. on Database Systems (TODS)*, 40(1):2, 2015.

[24] A. Perelman and C. Ré. DunceCap: Compiling worst-case optimal query plans. In *SIGMOD*, pages 2075–2076, 2015.

[25] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.

[26] 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*, pages 23–34, 1979.

[27] The PostgreSQL Global Development Group. PostgreSQL. www.postgresql.org.

[28] S. Tu and C. Ré. DunceCap: Query plans using generalized hypertree decompositions. In *SIGMOD*, pages 2077–2078, 2015.

[29] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[30] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

[31] J. Y. Yen. Finding the $k$ shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.