General-Purpose Join Algorithms for Large Graph Triangle Listing on Heterogeneous Systems

Daniel Zinn*
LogicBlox Inc.
DanielQZinn@gmail.com

Haicheng Wu
Georgia Institute of Technology
hwu36@gatech.edu

Jin Wang
Georgia Institute of Technology
jin.wang@gatech.edu

Molham Aref
LogicBlox Inc.
molham.aref@logicblox.com

Sudhakar Yalamanchili
Georgia Institute of Technology
sudha@ece.gatech.edu

ABSTRACT

We investigate applying general-purpose join algorithms to the triangle listing problem on heterogeneous systems that feature a multi-core CPU and multiple GPUs. In particular, we consider an out-of-core context where graph data are available on secondary storage such as a solid-state disk (SSD) and may not fit in the CPU main memory or GPU device memory. We focus on Leapfrog Triejoin (LFTJ), a recently proposed, worst-case optimal algorithm and present "boxing": a novel yet conceptually simple approach for partitioning and feeding out-of-core input data to LFTJ. The "boxing" algorithm integrates well with a GPU-Optimized LFTJ algorithm for triangle listing. We achieve significant performance gains on a heterogeneous system comprised of GPUs and CPU by utilizing the massive-parallel computation capability of GPUs. Our experimental evaluations on real-world and synthetic data sets (some of which containing more than 1.2 billion edges) show that out-of-core LFTJ is competitive with specialized graph algorithms for triangle listing. By using one or two GPUs, we achieve additional speedups on the same graphs.

CCS Concepts

•Information systems → Relational parallel and distributed DBMSs; •Theory of computation → Data structures and algorithms for data management; *Massively parallel algorithms;*

Keywords

Triangle Listing, Data partitioning, GPGPU

1. INTRODUCTION

Triangle listing is the building block for many other graph algorithms and key ingredient for graph metrics such as triangular clustering, finding cohesive subgraphs etc [7, 17, 18, 16]. In addition, it attracts extensive attention in the research literature among several fields: graph theory, databases and network analysis to name a few. Here, both in-memory as well as out-of-core algorithms have been studied.

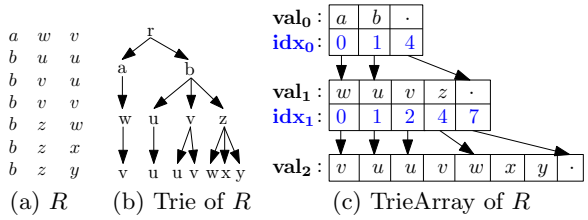
This work is motivated by the desire of building *general-purpose* systems that can empower their (domain) users to pose and run queries in a *declarative* and general language, such as SQL or Datalog , and the execution speed to be competitive with *hand-crafted* special-purpose algorithms.

In addition, the use of GPGPU has appeared as a potential vehicle for relational computations [6, 22, 24] with an order of magnitude or more performance improvement over traditional CPU-based implementations. We therefore investigate the problem on a heterogeneous system that features both the CPU and the GPUs. Discrete GPU accelerators provide massive fine-grained parallelism, higher raw computational throughput, and higher memory bandwidth compared to multi-core CPUs; however, their device memory is limited. Best-in-class card currently have only up to 32GB memory, with a relatively small CPU-GPU communication bandwidth through the PCIe bus. We thus focus on the out-of-core setting as input or intermediary data may not fit in the main CPU memory or the GPU device.

Towards this end, we select a state-of-the-art general-purpose join algorithm and apply it to triangle query in an out-of-core setting. The selected join algorithm is Leapfrog-Triejoin (LFTJ) by Veldhuizen [21] which is known for its elegance that allows efficient implementations with various optimizations, low intermediary data usage that makes it a good candidate in the out-of-core context and strong theoretical worst-case optimal guarantees. We also implement it on GPUs [24] with additional optimizations.

However, we observe excessive I/O operations when running LFTJ for triangle listing in the out-of-core setting for either CPUs or GPUs. To reduce this overhead, we then propose a partitioning and feeding strategy called "boxing" that can be used together with LFTJ on both the CPU and the GPU. The goal of this partitioning algorithm is to maintain the good computation complexity of the in-memory LFTJ in the out-of-core setting and lower the I/O cost to prevent it from becoming the main bottleneck. Note that while we discuss the specific triangle listing problem in this work, the "boxing" techniques can apply to any generic LFTJ imple-

*This work was done while Daniel was at LogicBlox; Daniel is now working at Google.



(a) R (b) Trie of R (c) TrieArray of R

Figure 1: Trie and TrieArray of a ternary relation R

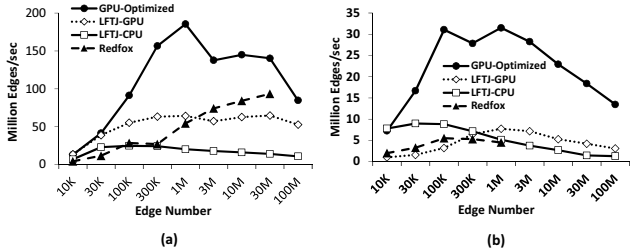


Figure 2: Performance of GPU-Optimized LFTJ: (a) Finding Triangles; (b) Finding Four-Cliques.

mentations either on the CPU or the GPU. Specifically, we make the following contributions:

1. We present and analyze a novel strategy called *boxing* for out-of-core execution of LFTJ, and apply the boxed-LFTJ to the triangle-listing problem on large scale out-of-core data sets.

2. We improve the performance of a GPU-optimized LFTJ implementation and integrate it with the boxed-LFTJ framework to utilize the computation capability of GPUs.

3. We perform an experimental evaluation of boxed-LFTJ-based triangle listing for out-of-core graphs on a heterogeneous system, which features a multi-core CPU and multiple GPUs. The results demonstrate that boxed-LFTJ has low CPU and I/O overhead and shows that the overall execution time is competitive with the specialized triangle listing algorithm MGT [7]. The integration of the GPU-optimized LFTJ further improves the performance by utilizing the massively-parallel computation capability of GPUs.

2. LEAPFROG-TRIEJOIN

LFTJ is a worst-case optimum *multi-predicate* join algorithm. We restrict our attention to full-conjunctive queries, and use a Datalog syntax and terminology to describe queries (or joins). Some notation is necessary: for a binary relation $R(x, y)$, let $R(x, \_)$ denote the set of values in the first column, i.e., R projected to its first attribute. Then, $R(a, \_)$ is the projection to the second attribute *after* only selecting tuples that have the constant a as the first attribute.

LFTJ operates by first fixing an order of the variables occurring in the rule body, e.g. x, y, z. Then, LFTJ finds all possible values a for the variable x by performing an intersection of $R(x, \_)$ and $S(x, \_)$. Now, as soon as the first of such a is found, LFTJ is looking for values b of y, the next variable in the variable-order. Here, LFTJ computes the intersection of $R(a, y)$ and $T(y, \_)$. Again, after the first b is found, LFTJ is looking for values c for z by computing the intersection of $S(a, z)$ and $T(b, z)$. If any of these c is found LFTJ reports the tuple (a, b, c) in the output. LFTJ then back-tracks its search to the variable y and looks for the next b. Back-tracking continues up to the first variable and LFTJ finishes when no new a can be found anymore.

Trie representation for relations. A Trie is a tree that stores each tuple of a relation as a path from the root node r to a child node. See Fig. 1(a) for an example of a ternary relation with its Trie in Fig. 1(b). In general, a Trie for a relation with arity h has a height of h. For a relation $R(x_1, \ldots, x_h)$, the nodes at height i store values from the ith column of R. We require that child nodes of the *same* node n are unique and ordered increasingly. For example in Fig. 1(b) at level 2, the children of b are the values u, v, and z, which are in increasing order.

TrieArrays. We use a simple array-encoding for Tries, which is inspired by the Compressed-Sparse-Row (CSR) format, a commonly used format to store graphs. As an example see Fig. 1(c) for the representation of the Trie given in Fig. 1(b). The data values are stored in flat arrays called *value*-arrays. Index arrays are used to separate children at the same tree level but from different parent nodes. An n-ary relation has n value arrays and $n-1$ index arrays. In particular, the children of a node n stored in the value array $\mathbf{val_i}$ at position j are stored in the array $\mathbf{val_{i+1}}$ starting at the index from $\mathbf{idx_i}[j]$ until the index $\mathbf{idx_i}[j+1]$ exclusively. E.g. in Fig. 1(c), the children w, x, y of z from $\mathbf{val_1}[3]$ are stored in $\mathbf{val_2}$ from $\mathbf{idx_1}[3] = 4$ to $\mathbf{idx_1}[4] = 7$.

Implementation. A basic building block of Leapfrog Triejoin (LFTJ) is Leapfrog Join (LFJ). It computes the intersection of multiple unary relations by using a linear iterator for each of its input relations to search values that are in *all* input unary relations. The searching process used in LFJ is essentially implemented as a binary search. When extending LFJ to LFTJ, the linear iterator is extended to TriIterator. LFTJ works on multiple Tries and each Trie has its TriIterator. These TrieIterators cooperate with each other to traverse the Tries in a depth-first order to find the intersections in each level of the Tries. The details of LFTJ including the computation complexity can be found in [21]. Section 4 has an example of running LFTJ on three relations to find triangles.

3. GPU-OPTIMIZED LFTJ

As described in [24], LFTJ has been efficiently ported to run on GPUs which is called GPU-Optimized LFTJ. The algorithm shares the same data structure (TrieArray) and program interface as the CPU LFTJ for easy substitution; it applies two optimizations for GPUs: (1) It turns the depth-first order into breadth-first order because breadth-first exposes more fine-grained parallelism at the cost of larger memory footprint. (2) It replaces binary search with linear search for good memory access patterns and load balance in GPUs at the cost of worse computation complexity.

This paper improves GPU-Optimized LFTJ described in [24] resulting in a faster algorithm with support of much larger input data sets. In particular, we (1) solve the problem of the increased memory footprint caused by the breadth-first order. When expanding nodes in breadth-first traversal, the new algorithm expands nodes in multiple rounds if the total size of the expanded nodes is larger than the available memory size. (2) balance the tradeoff between linear search and binary search to take advantage of the better computational complexity of binary search. For example, when intersecting two sorted arrays with a large difference in their size, we use binary search to find the first and the

Algorithm 1 Steps Performed by Leapfrog-Triejoin on the Triangles Query $T(x, y, z) \leftarrow E(x, y), E(x, z), E(y, z)$.

```
1: for a ∈ E(x, _) ∩ E(x, _) do
2:     for b ∈ E(y, _) ∩ E(a, y) do
3:         for c ∈ E(a, z) ∩ E(b, z) do
4:             yield (a, b, c)                    ▷ triangle found
```

last elements of the short array in the long array to reduce the search scope of the following linear search.

Fig. 2 reports the throughput of GPU-Optimized LFTJ with three other algorithms in finding cliques (e.g. triangles and 4-cliques) from some randomly generated graphs which still fit in the GPU main memory. All vertices of the graphs are 64-bit. Red Fox [22] runs a state-of-the-art GPU binary join algorithms and it does not support large graphs because its memory footprint is too large to fit in the GPU memory. LFTJ-CPU runs multiple vanilla LFTJ algorithms in different CPU threads in parallel. LFTJ-GPU is similar to LFTJ-CPU but runs vanilla LFTJ in GPU threads. The environment of the experiment is the same as the one used in Section 7. All GPU experiments are conducted on a K40 GPU. Fig. 2 shows that GPU-Optimized LFTJ is always the fastest algorithm and can run the largest graph supported by the other algorithms. When evaluating 100M-edge graph, GPU-Optimized is 8x faster than CPU-LFTJ in triangle listing and 11x faster than CPU-LFTJ in 4-clique listing.

4. LFTJ FOR TRIANGLE QUERY

We first explain the idea of using LFTJ for the triangle query (LFTJ-Δ) and highlight the necessary steps. Our formal setting is standard for I/O efficient algorithms: input I, intermediary and output data can exceed the size of available CPU main memory or GPU device memory M (measured in words to store one atomic value), in which case it can be read (written) from (to) secondary storage such as SSD for CPU or via PCIe bus for GPU with the granularity of a page that has size P. Reading or writing a page incurs 1 unit of I/O cost. The following example shows that LFTJ can suffer from excessive I/O operations in an external-memory setting with a page-based least-recently-used memory replacement strategy. These I/O operations happen independently of whether LFTJ runs on CPU or GPU.

Given a simple, undirected graph G and let $G^\star = (V, E)$ be its directed version, that is for each edge $\{a, b\}$ in G, E contains the pair $(\min\{a, b\}, \max\{a, b\})$. An example of G, $G^\star$ and $E(G^\star)$ is shown in Fig. 4(a) (b) (c). The query (refereed as (Δ))

$$T(x, y, z) \leftarrow E(x, y), E(x, z), E(y, z), x < y < z. \qquad (\Delta)$$

computes all triangles in $G^\star$ of the form: $a \overset{b}{\rightarrow} c$. The output T coincides with the triangles in G.

The steps that LFTJ performs for the triangle query are summarized in Algorithm 1. First, the leapfrog join at level x for the atoms $E(x, y)$ and $E(x, z)$ computes the intersection between $E(x, \_)$ and $E(x, \_)$. Then, for each found value a for x, we perform a Leapfrog Join at level y computing the intersection of $E(a, y)$ with $E(y, \_)$, because the variable y occurs in the atoms $E(x, y)$ and $E(y, z)$. In the last step, we find bindings for z by intersecting $E(a, z)$ with $E(b, z)$ because z occurs in the atoms $E(x, z)$ and $E(y, z)$.

The LFTJ-Δ may have excessive I/O operations because (1) each lookup in the intersection may load a separate page and (2) similar to cache thrashing, pages loaded by earlier lookups may be swapped out before any uses from subsequent lookups and have to be reloaded.

E :	
0,	24
1,	20
2,	16
3,	12
4,	8
5,	4
6,	24
7,	20
..	..
18,	24
..	..
23,	4
24,	24

(a) Graph G

a	b	c	$E(a, y)$	$E(b, z)$	**I/O**
0			24		
0	24		24	24	1
0	24	24	24	24	1
1			20		
1	20		20	16	1
1	20	–	20	16	1
2			16		
2	16		16	8	1
2	16	–	16	8	1
3			12		
3	12		12	24	1
3	12	–	12	24	1
..	..	..	..	..	

(b) LFTJ-Δ steps when running on G

Figure 3: Example input graph that causes LFTJ-Δ to use many I/Os. Parameters: $M = 40$, $P = 4$, $N = 24$, $T = 6$. Graph has $N + 1 = 25$ edges.

Take Fig. 3(a) for an example. The graph has $N + 1 = 25$ edges. With $M = 40$ and $P = 4$, the assumed memory is able to store 10 pages. The two columns of E are stored separately using the TrieArray data representation, where one page contains four consecutive atomics from one column (e.g. (0,1,2,3) from the first column is in one page and (24,20,16,12) from the second column is in another page). We place values in the second column of E by P apart which will cause LFTJ to perform an I/O for every lookup of b in $E(y, \_)$ for line 2 in Algorithm 1. For example, the first lookup of b in line 2 corresponds to $a = 0$ and $E(a, y) = 24$, resulting in $b = 24$ and loading the page containing '24' of the first column of E. The second lookup of b corresponds to $a = 1$ and $E(a, y) = 20$, requiring loading $b = 20$ which is a different page containing '20' of the first column of E. Furthermore, values in the second column repeat in groups large enough that loading all pages in a group will preempt the first page from memory effectively prohibiting the algorithm to reuse the earlier loaded pages. The example uses a group size $T = 6$. The first 5 lookups of b ('24', '20', ..., '8') will load 5 pages. Each lookup of $E(b, z)$ to compute c using the step of line 3 in Algorithm 1 requires another I/O to load from the second column of E. For example, $E(b, z) = 24$ corresponding to $b = 24$ requires loading the page containing '24' from the second column of E. Together the triangle searches for the first 5 nodes in E using Algorithm 1 load 10 pages which fill up the memory so that the 6th lookup of b ('4') corresponding to $a = 5$ has to evict the least-recently-used page '24' of the first column of E. However, this page will be immediately required and loaded by the 7th lookup of b corresponding to $a = 6$, requiring another I/O. The complete I/O cost for the example data are shown in Fig. 3(b).

In general cases, for a graph $G_N = (V, E)$, we set edges $E = \{(x, y) \mid x = 0, \ldots, N \text{ and } y = N - P \times (x \bmod (T))\}$, $2N \geq M + 2P$ where $N + 1$ is the number of edges, and group size $T = M/(2P) + 1$. Each node in E causes two page I/Os in LFTJ-Δ: the first one comes from looking up each b in $E(y, \_)$ corresponding to $E(a, y)$ and the second one comes from loading $E(b, z)$ for each found b to perform intersection $E(a, z) \cap E(b, z)$. Therefore LFTJ-Δ incurs at least $2|E(G_N)|$ I/Os for the above defined graph G_N with a TrieArray data representation and a LRU memory replacement strategy.

To reduce the number of I/Os incurred in LFTJ-Δ and thereby increase I/O efficiency, we propose "boxing" which is an innovative and effective approach to deal with out-of-core data sets by sectioning them into boxes that can fit into memory. We adapt LFTJ to utilize the boxing strategy and then apply it to the triangle query.

5. BOXING LFTJ

5.1 High-Level Idea

LFTJ with a variable order $x_1, \ldots, x_n$ computes the join by essentially searching over an n dimensional space in which each dimension i spans over the domain of the variable x_i. In our approach, we partition the n-dimensional search space into "hyper-cubes" or *boxes* such that the required data for an individual box fits into memory. LFTJ is then run over each box individually—finding all input data ready in memory, so that no extra I/Os are necessary for loading data as described in Section 4.

Fig. 4 illustrates this strategy for LFTJ-Δ. The join uses three variables x, y, z – resulting into a 3-dimensional search-space. If the input graph G represented via a TrieArray does not fit into the available memory, then we partition the search space into boxes, for example as in Fig. 4(e). The partitioning is chosen such that the input data restricted to an individual box fits into memory. LFTJ-Δ is then executed for each box individually one after another while join results are written append-only in a streaming fashion. Note that in the triangle listing problem, the search space can be triangular if constraining the space with $x < y < z$. However, the framework we built can be used to compute any relational joins. In that sense, we hope illustrating "boxes" at this higher level exposes the generality. We first introduce the concept of TrieArray Slices which are subsets of the input trie and can be obtained through the two processes called *provisioning* and *probing*. Then we explain the boxing procedure using TrieAray Slices.

5.2 TrieArray Slices

As defined in Section 2, the input data are given on external storage in a TrieArray representation and are directly processed to load a subset into the main memory.

As an example, consider the binary relation E from Fig. 4(c) and its TrieArray T in Fig. 4(d). We are interested in the subset S of T that restricts the first attribute to the interval $[3, 5]$, i.e., $S = \{(x, y) \in T \mid x \in [3, 5]\}$. We call this a slice of S at level 0 from 3 to 5. A TrieArray for this slice is shown at the top of Fig. 4(f). To build this slice, we can simply copy the values in $\mathbf{val_0}$ for the interval $[3, 5]$; then look up where the corresponding y values are in $\mathbf{idx_0}$ and copy them. We then add a wrapper during accesses dynamically to the index arrays that can subtract the offset which is the first index in $\mathbf{idx_0}$ of the slice (wrapper shown as $\mathbf{idx_0}[-5]$ at the top of Fig. 4(f)). In this way all data used in the arrays of the slice are simply sub-arrays of the original data. Similarly, restricting the second attribute to the interval $[6, \infty]$ results in a slice shown at the bottom of Fig. 4(f).

In general, for an n-ary relation R, we are interested in creating slices at a level k, $0 \leq k < n$. At level k the values are restricted to an interval given by a low-bound l and a high-bound h; at levels $0, \ldots, k-1$, the slice contains only a single element each, all of which together combine to a k-ary prefix. Formally: Let R be an n-ary relation, $0 \leq k < n$

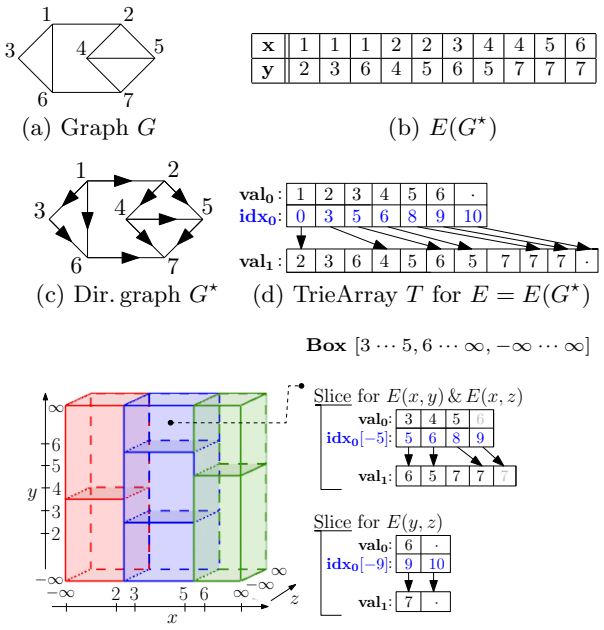


Figure 4: Example for out-of-core technique for LFTJ-Δ, i.e. $T(x, y, z) \leftarrow E(x, y), E(x, z), E(y, z)$ on $E(G^\star)$

an integer, s be a k-ary tuple prefix, and l and h be two domain values. The *Slice S of R at level k for s from l to h* (in symbols $S = R^s_{l \to h}$) is the defined as:

$$S = \{(x_0, ..., x_{n-1}) \in R \mid (x_0, ..., x_{k-1}) = s \text{ and } l \leq x_k \leq h\}$$

We create and store Slices in the TrieArraySlice data structure, which is a conventional TrieArray—except that the index arrays can be parameterized with an offset to perform dynamic index-adaptation as explained in the example above. As with TrieArrays, we identify the Slice (set of tuples) with the TrieArraySlice data structure and vice versa in the rest of the paper.

Provisioning. Given a relation R on secondary storage, we can create slices of R efficiently. This process is called *provisioning*. The provisioning process of slice $S = R^s_{l \to h}$ is as follows: using k binary searches on the value arrays $\mathbf{val_0}, \ldots, \mathbf{val_{k-1}}$, we locate the prefix s in R; the slice is empty if the prefix does not exist. Then, using two more binary searches we locate the smallest element $l' \geq l$ and the largest $h' \leq h$ in $\mathbf{val_k}$ of R. Their positions are the boundaries in $\mathbf{val_k}$ and $\mathbf{idx_k}$ for the interval we copy into the slice. For the remaining $n-k$ value arrays and $n-k-1$ index arrays, we iteratively follow the pointers within the $\mathbf{idx}$ arrays and copy the appropriate ranges. As a last step we adjust the index-array's offset parameter: for each $j = k, \ldots, n-2$, we set the offset parameter of $\mathbf{idx_j}$ to $-\mathbf{idx_j}[0]$.

Probing. As the last building block, we are interested in provisioning slices that will fill up a certain budgeted amount of memory. This process is referred as *probing*. In particular, for a TrieArray T on secondary storage, we specify the prefix-tuple s and lower bound l as before. But instead of providing an upper bound h, we give a memory budget m in pages as shown in Fig. 5. We are then interested in a maximal upper bound $h \geq l$ such that the slice at s from l to h requires no more than m pages of memory. The process is similar to slice provisioning, except that we do a binary

function PROBE(T, s, l, m) **returns** h
in: n-ary TrieArray T ▷ on secondary storage
k-Tuple s ▷ start tuple for attributes $0,..,k-1$
value l ▷ Lower bound for attribute k
int m ▷ memory budget in pages
out: Maximal $h \geq l$ such that the slice $T^s_{l \to h}$ occupies $\leq m$ pages of memory, or SPILL if no such h exist.

Figure 5: Interface for Single Slice Probing

1: $l \leftarrow -\infty$ ▷ Value at the start of the search space
2: **repeat**
3: probe R, S, T from l for upper bounds h_R, h_S, h_T
4: $h \leftarrow \min(h_R, h_S, h_T)$
5: provision R, S, T from l to h
6: run LFTJ on the provisioned slices
7: $l \leftarrow \mathsf{succ}(h)$ ▷ lower bound is successor of old upper
8: **until** $\infty = h$ ▷ until we have searched all space

Figure 6: Example: Boxing for $R(x), S(x), T(x)$

search for the upper bound and check for each guess how many pages the TrieSlice would occupy. This can be done by following the **idx** pointers. Note that for skewed data, it is possible that the slice $T^s_{l \to h}$ requires more than m pages of memory, even when $h = l$. Should this case occur, we report via the sentinel value SPILL instead of returning an upper bound h.

5.3 Boxing Procedure

To help exposition, we first describe aspects of the boxing approach via examples, before we cover the general case.

Joins with one variable. Consider a join over multiple unary relations such as

$$Q(x) \leftarrow R(x), S(x), T(x).$$

Imagine each of the body relations is larger than the available internal memory M. We can divide the internal memory into four parts, one for the output data and one for each of the input relations. Since the output is written append-only, a relatively small portion of memory, which is written to disk once it fills up, is sufficient. We thus divide up the bulk of the memory for the three input relations. We can use the simple strategy to *evenly* divide the space. A boxed LFTJ execution would then simply alternate probing, provisioning, and calling LFTJ as described in Fig. 6.

Unary cross-products. Consider the cross-product of m unary relations, with each relation larger than M:

$$Q(x_1,...,x_m) \leftarrow R_1(x_1), \ldots, R_m(x_m).$$

We again split the bulk of the available memory across the m input relations. The boxing procedure is recursive where each *dimension* i of the recursion corresponds to a variable x_i (See Fig. 7). The procedure starts with $i = 1$. In general, at a dimension i, we loop over the predicate R_i via the probe-provisioning loop. Then, for each slice at dimension i, we do the same recursively for the next higher dimension. At the bottom of the recursion—when we reached the $i = m$, we call LFTJ on the created slices. Then, the slices provide data for the box [low···high], i.e., in which the variable x_i can range from low[i] to high[i]. Note that (like above) we can run the original query over the slice data since the slices are guaranteed to not have data outside their range and thus the boxes partition the search-space without overlap.

General joins. The general approach combines the two previous algorithms while also considering corner cases. Let

variables: m-Tuple low, high ▷ Box boundaries
1: **procedure** MAIN
2: BOXUP(1)
3: **procedure** BOXUP(int i) ▷ i corresponds to x_i
4: low[i] $\leftarrow -\infty$
5: **repeat**
6: probe inputs R_i from low[i] for upper bound h_i
7: high[i] $\leftarrow h_i$
8: provision R_i from low[i] to high[i]
9: **if** $i < m$ **then** : BOXUP(i+1)
10: **else**: run LFTJ on slices ▷ Box: [low···high]
11: low[i] $\leftarrow$ succ(high[i])
12: **until** ∞ = high[i]

Figure 7: Example: Boxing for $R_1(x_1)$,..., $R_m(x_m)$

Q be a general full-conjunctive join of m atoms, and variable order $\pi = x_1, \ldots, x_n$ with no atom containing the same variable twice, and all atoms in Q mentioning variables consistent with π. We first group the atoms based on their first variable x_j: we place all atoms that have as first variable x_j into the array atoms[1..n] at position j. To follow the exposition, consider the join

$$Q(x_1, x_2, x_3) \leftarrow R(x_1, x_2), S(x_1, x_3), T(x_2, x_3), U(x_1)$$

where we put R, S, and U into atoms[1] and T into atoms[2]. Like for cross-products, we recursively provision for the dimension i ranging from 1 to n. For each i, we use the method for joining unary relations for the atoms in `atoms`[i]. In particular, for each $A_j \in$ atoms[i] we probe and create slices for A_j *at level 0* regardless of i or the arity of A_j. Thus, at dimension i, we iteratively provision atoms with x_i as their first attribute restricting the range of x_i but not any of the other variables x_k, $k > i$. This ensures that we can freely choose any partitions we might perform on these variables x_k for $k > i$. Like with cross-products, we call LFTJ at the lowest level when $i = n$.

The above works well unless any of the probes reports a SPILL, which can occur if a relation exhibits significant skew. For example, imagine there is a value a for which $|S_a(x_3)|$ exceeds the allocated storage. Then, at dimension $i = 1$, probing S at level 0 with a lower bound a will return SPILL. We handle these situations by setting the upper bound at level $i = 1$ to a, and essentially marking S_a as a relation that needs to be provisioned at the dimension of its second attribute (eg, 3) alongside the atoms in `atoms`[3]. Note that a relation of arity α can spill $\alpha - 1$ times in worst case.

The general algorithm is given in Algorithm 2. We evenly divide the available storage among the n dimensions, and assign the atoms A to `atoms`[i] accordingly (lines 3-4). We also use a variable `leftoverMem` to let lower dimensions utilize memory that was not fully used by higher dimensions. In line 11, we union the spills from the previous level to the atoms we need to provision. The method `probe` in line 12, probes atoms in `atms` to find an upper bound such that all atoms can be provisioned. We here, evenly divide `mem` by the size of `atms`. The lower bound for probing are taken from `low`, which is also used to determine the starting tuples for possible spills. The method sets the upper bound at the current dimension and fills the spills predicate if necessary. The method `provision` provisions the predicate A with bounds from `low` and `high` adapted to the variables occurring in A. It returns the slice and the size of used memory.

I/O Complexity. Using the analysis and method described in [26], the I/O complexity of boxed LFTJ with n variables, input I and output of size K is $O(|I|^n/(M^{n-1}P) + K/P)$.

Algorithm 2 Boxing Leapfrog Triejoin

in: mem_{max} ▷ available memory in pages
$A_1, \ldots, A_m$ ▷ body atoms and TrieArrays
$x_1, \ldots, x_n$ ▷ key order, n variables
variables:
n-Tuple low, high ▷ Box boundaries
Array of AtomSet atoms[1..n] ▷ atoms per level
Array of SliceSet S[1..n] ▷ provisioned slices
Array of AtomSet spill[0..n] ▷ spilled-over atoms
Array of int budget[1..n] ▷ of memory in pages

1: **procedure** MAIN
2: **for** $i \in \{1, \ldots, n\}$ **do**
3: budget[i] $\leftarrow \text{mem}_{max}/n$
4: atoms[i] $\leftarrow \{A_i \mid x_i$ is first variable in $A_i\}$
5: BOXUP(1, 0) ▷ 1st variable, no leftover memory

6: **procedure** BOXUP(i, leftoverMem)
7: mem $\leftarrow$ budget[i] + leftoverMem
8: low $\leftarrow [-\infty, ..., -\infty]$; high $\leftarrow [\infty, ..., \infty]$
9: **repeat**
10: S[i] $\leftarrow \emptyset$; usedMem $\leftarrow 0$
11: atms $\leftarrow$ atoms[i] $\cup$ spill[$i-1$]
12: spill[i], high[i] $\leftarrow$ probe(i, atms, mem, low)
13: **for** $A \in$ atms $\setminus$ spill[i] **do**
14: slice, m $\leftarrow$ provision(A, low, high)
15: usedMem $\leftarrow$ usedMem + m
16: S[i] $\leftarrow$ S[i] $\cup$ slice
17: **if** $i < n$ **then**
18: leftoverMem $\leftarrow$ mem $-$ usedMem
19: BOXUP($i+1$, leftoverMem)
20: **else**
21: **run** LFTJ on $\bigcup_{k=1..n}$ S[k] on Box[low$\cdots$high]
22: low[i] $\leftarrow$ succ(high[i])
23: **until** ∞ = high[i]

Applying boxed LFTJ to the triangle query generates an I/O complexity for LFTJ-Δ of $O(|E|^3/(M^2P) + K/P)$. If there are no spills the complexity is $O(|E|^2/(MP) + K/P)$ which matches the I/O complexity of MGT [7] and is optimal if $M \geq |V|$ as shown in [7].

6. IMPLEMENTATION

We implement a general-purpose join-processing framework using the proposed boxed-LFTJ on a heterogeneous system. Predicates (stored as TrieArrays) can have variable arities and we support marking a prefix of the attributes as key (the TrieArray then needs fewer index arrays). For CPU, the system uses secondary storage which is SSD (via memory-mapped files) to allow processing of data that exceeds the main memory. For GPU, if the data size exceeds the device memory, they can be retrieved first from the CPU main memory and then from the SSD.

6.1 Optimizations on CPU

We are also deploying a parallelization scheme for LFTJ to utilize multiple CPU cores by evenly mapping the first variable to the threads to perform independent LFTJ, i.e. each thread has a fixed range for the first variable. In the boxed LFTJ version, boxes are worked on one after another, yet LFTJ utilizes available cores while processing a box.

In the boxing procedure, using more memory at smaller dimensions reduces the number of boxes created. We pick a ratio of 4:1 for dividing up the memory between x:y in the triangle query. We also do not allocate budget to dimensions j that do not have an atom using x_j as first variable (e.g., z). In case there is a spill the budget for the spilling relation will be moved over to the next dimension.

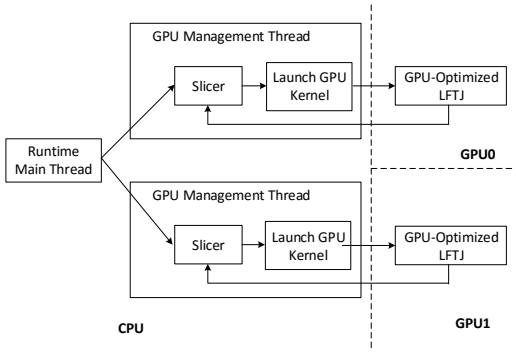


Figure 8: Integration of GPU-Optimized LFTJ

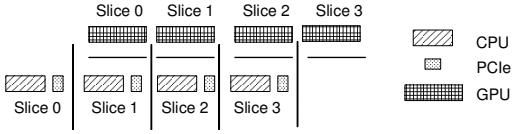


Figure 9: Use double buffer to hide CPU boxing and PCIe data movement.

We envision that for some queries, an optimizer, aided by constraints provided by the user, can avoid provisioning certain boxes because it can infer that there cannot possibly be a query result within that box. For example, in our case, we know that $x < y < z$. This can easily be inferred from the constraint $a < b$ for any $(a, b) \in E$. Based on this, we do not need to provision at dimension y if the high bound for y is smaller than the low bound for x. We have put a hook into the boxing mechanism to bypass provisioning if *after* probing this condition is met.

6.2 GPU-Optimized LFTJ Integration

We integrate the GPU-Optimized LFTJ algorithm with the boxed LFTJ framework as shown in Fig. 8. The integration is seamless because the CPU and GPU LFTJ use the same data structure and program interface. Multi-GPU implementation is also supported by the system. The main thread in the CPU spawns several new threads, each for one GPU in the system, to perform boxing by the slicer, send the slices to the GPUs and invoke the GPU-Optimized LFTJ kernels on the GPUs. Therefore, CPU is in charge of partitioning data when GPUs take control of the computation. The boxing algorithm is multi-thread safe. The boxes generated by different threads are not overlapped and can be processed independently by different GPUs. The GPU management threads complete after all the data are processed.

We further optimize the performance by concurrently executing different stages on the GPU by using double buffering [23]. Three stages can run concurrently: CPU boxing, PCIe data transferring, and GPU LFTJ computation. Considering the fact that GPU LFTJ computation dominates the overall time, the integrated system overlaps the first two stages with GPU LFTJ execution as shown in Figure 9. Thus, the time spent in preparing data for GPU is hidden by the computation.

7. EXPERIMENTAL EVALUATION

In our experimental evaluation, we focus on the triangle listing problem on a heterogeneous system that has a multi-core CPU and multiple GPUs. we investigate and an-

	Mem	Measured Mem BW	Core #	Max Core Freq
K40c	12GB	203GB/s	2880	745 MHZ
Titan	6GB	220GB/s	2688	876 MHz

Figure 10: Comparison of GPUs used in the experiments

	LiveJournal(LJ)	Orkut	RAND16	RMAT16	RAND80	RMAT80	Twitter
csv	500MB	1.8GB	4.1GB	4.0GB	22GB	22GB	25GB
TA	315MB	1.2GB	2.3GB	2.2GB	11GB	11.2GB	10GB
$\|V\|$	4 Mio	3 Mio	16 Mio	16 Mio	80 Mio	80 Mio	42 Mio
$\|E\|$	35 Mio	117 Mio	256 Mio	256 Mio	1.28 Bill	1.28 Bill	1.2 Bill
$\frac{\|E\|}{\|V\|}$	8.7	38.1	16	16	16	16	28.9
$\#\Delta$	178 Mio	628 Mio	5457	2.2 Mio	5491	884,555	35 Bill

Figure 11: Characteristics of the data sets. Self and duplicate edges are removed. The CSV sizes refer to the CSV data where each undirected edge $\{a, b\}$ is mentioned only once. TA stands for the TrieArray representation as described in the earlier sections.

alyze the overhead of boxed LFTJ, its behavior with limited available main memory and its performance compared to best-in-class competitors.

Evaluation environment. We use a desktop machine with an Intel i7-4771 core, that has 4 cores (8 hyper-threaded), each clocked at 3.5GHz. The machine has 32GB of physical memory and a single SSD disk. It is running Ubuntu 14.04.1 with a stock 3.13 Linux kernel. Two GPUs are attached to the system, one Tesla K40c and one Geforce GTX Titan. Fig. 10 compares the key parameters of them. In general, K40c has larger memory but lower bandwidth, more cores but lower frequency. If the data fit in GPU memory, K40c and Titan have similar throughput for GPU-Optimized LFTJ. The CPU and two GPUs appeared in the market about in the same time and they are the most advanced devices at that time.

Data. We use both real-world and synthetic input data of varying sizes. The data statistics are shown in Fig. 11. The real-world data include graphs from online community: "LiveJournal (LJ)" [25], "Orkut" [12] and "Twitter" [9]. We also consider synthetically generated data due to its better understood characteristics. We focus on two datasets: "RAND" and "RMAT". Each comes in a medium-sized version (RAND16 and RMAT16) and a large version (RAND80 and RMAT80). In the RAND dataset, we create edges by uniformly randomly selecting two endpoints from the graph's nodes. The RMAT data contains graphs created by the Recursive Matrix approach [3] that closely match real-world graphs such as computer networks, or web graphs. The LiveJournal and the synthetic graphs were also used by the MGT work in [7] and earlier work [4] to evaluate out-of-core performance for the triangle listing problem.

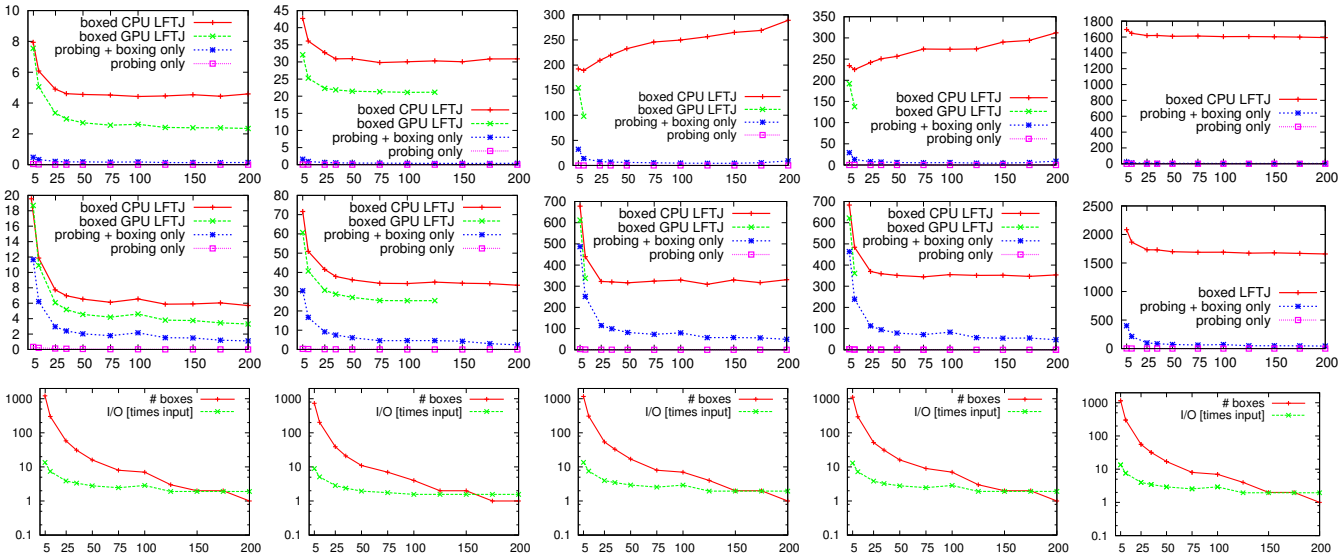
Methodology. We measure and present the time for running our TrieArray-based implementation of LFTJ and two competing algorithms on the mentioned data sets with various configurations and memory restrictions. The TrieArray-based experiments are conducted on both CPU and GPUs. Input data for LFTJ are given in TrieArray format.

CPU Overhead of Boxed-LFTJ. We advise boxed LFTJ to only use memory the size equaling a fraction of the input during execution to test the impact of the box size on the CPU overhead. Recall that the input workloads in SSD are memory mapped. To further (almost completely) remove I/O, we prefix the execution by `cat`-ting all input data to `/dev/null`, which essentially pre-loads the Linux file-system cache. The boxing is performed in one CPU thread and CPU LFTJ runs with eight threads to fully exploit the four cores of the CPU. We now consider the three questions (i) What is the CPU overhead for probing and copying? (ii) What is the overhead introduced by running LFTJ on individual boxes in comparison to running LFTJ on the whole input data? and (iii) What is the performance of running LFTJ on one or multiple GPUs?

To answer these questions, we run four variants: (a) the full CPU LFTJ, (b) the full GPU LFTJ, (c) probing and copying data into TrieArraySlices without running LFTJ, and (d) only probing without copying input data nor running LFTJ. Results are shown in the first row of Fig. 12. On the X-Axis, we vary the space available for boxing. The individual points range from $5, 10, \ldots$ up to 200% of the input data size in TrieArray representation. We choose to range up to 200% since the input is essentially read twice by LFTJ-Δ: once for each of the dimensions x and y.

Results. Answering question (i): We can see that the CPU work performed for probing and copying is very low in comparison to the work done by the CPU join evaluation, even when the box sizes are limited to as little as 5% of the size of the input. Answering (ii), we look at the top lines for LFTJ and compare the curve with the value at the far right as this one is achieved by using a single box which is essentially the same as the vanilla LFTJ. The real-world data sets behave as expected: starting at around 25%, they level out demonstrating that the CPU overhead is low if the available memory is not too much smaller than the input data size. Now, for the synthetic data sets, we see that unexpectedly, using *more* boxes *reduces* the CPU work (memory range 10%–200%). We speculate that this is because the boxed version might reduce the work done in binary searches since the space that needs to be searched is smaller. Only at 5%, does this trend reverse and using more smaller boxes takes longer. Answering question (iii): The second top lines of the five figures represents the performance of executing GPU-Optimized LFTJ in K40c. The largest box size supported by GPU is much smaller than CPU because GPU has much smaller memory size and GPU-Optimized LFTJ uses much larger memory footprint. The GPU number is not shown in the TWITTER graph because GPU performance is more than 10x slower than its CPU counterpart for this power-law graph. The linear search used by GPU-Optimized LFTJ is not efficient for this graph. Replacing linear search by binary search in GPU is even worse which can cause another 10x slow down. For the rest of graphs, GPU-Optimized LFTJ is 1.46-2.95x faster compared with the CPU approach even when CPU uses 200% boxes. The CPU overhead of probing and copying is still less than 15% when LFTJ is performed in GPU.

Boxing with limited memory. We are also interested in the performance of the boxing technique when disk I/O needs to be performed. Here, we run the same experiments as above but we clear all Linux system caches before we start a run. We further use Linux's *cgroup* feature to limit the total amount of RAM used for the program (data+instructions) and any caches used by the operating system to buffer I/O on behalf of the program. As actual limit we use the value given to the boxing and shown on the X-Axis plus a fixed 100MB (that accounts for the output buffer and the size of the executable).

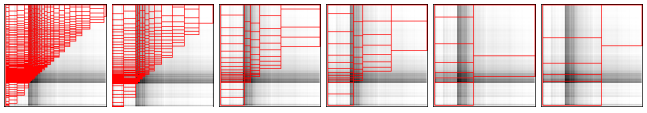


(a) LJ (b) ORKUT (c) RAND80 (d) RMAT80 (e) TWITTER

Figure 12: Boxed LFTJ Analysis. On the X-Axis, we vary the total memory available for boxing shown as percent of the size of the input relation. We vary from 5%, 10%, 25%,... to 200%; we choose 200% to hold the input twice: once for $E(x, y)$ and once for $E(y, z)$. First row shows total runtime on the Y-Axes in seconds without OS-level (cgroup) main-memory-restrictions and warm caches to evaluate the additional CPU work necessary for boxing. For performance in an out-of-core scenario, we enforce OS-level memory restrictions and have all caches cleared before execution in the second row; which also plots wall-clock execution time on the Y-Axis. The third row shows the number of boxes and the amount of provisioned memory in multiples of the size of the input data. Omitted graphs for {RAND|RMAT}16 look like the "80" variants.

Results are shown in the second row of Fig. 12. These results are end-to-end which include all the overhead that happens in real world when running out-of-core data set. In this scenario, we see that probing is still very cheap even for the 5% memory setting; Provisioning the data now has noticeable costs for low-memory settings (25% and below). However, even then, it is mostly dominated by the time to actually perform the in-memory joins. This is even more so for the real-world data sets. Overall, with around 25% or more memory, boxed LFTJ's performance stays constant indicating that I/O is not the bottleneck. For example, we can count all 37 billion triangles in the TWITTER dataset in around 29 minutes without I/O and only need up to 35 minutes with disk I/O. For this power law graph, the I/O overhead is negligible. If executing LFTJ in GPU, the total execution time is still 1.30-1.73x faster than using the same box size to run CPU LFTJ for the first four graph. The ratio of probing and provision increases since GPU is faster in joining. Note PCIe overhead is not included in the provision, but included in the boxed GPU LFTJ. In two real-world graphs, LJ and Orkut, the overhead is still less than 1/3. But in two synthetic graphs, the overhead is between 66%-75% which indicating that the I/O is the bottleneck. In these cases, we can perform two level of boxing: first find a large box from the secondary storage and then find small boxes to send to GPU from this large box. This should significantly reduce the I/O cost because the second row of Fig. 12 proves finding large box has much smaller overhead than finding small box from SSD and the first row of Fig. 12 proves boxing from memory has negligible cost.

When using GPU to do triangle listing, either the GPU computation or disk I/O are the main bottlenecks depending on the characteristics of the graph. Here, PCIe overhead can be hidden by overlapping computation. However, previous research shows that when only standard binary joins are processed [2], the PCIe can become the bottleneck.

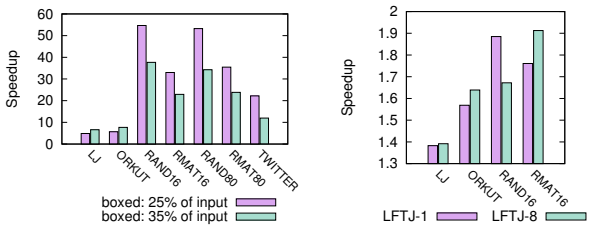


(a) 5% (b) 10% (c) 25% (d) 35% (e) 75% (f) 100%

Figure 13: Selected boxes for TWITTER dataset

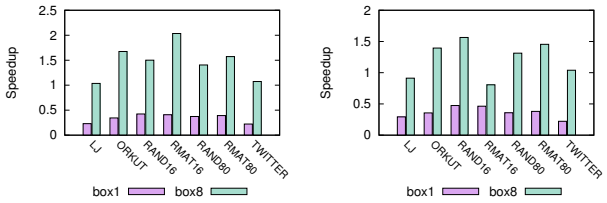
In the third row of Fig. 12, we show number of boxes used as well as the total amount of memory copied for provisioning as a multiple of the TrieArray input size from Fig. 11. We see that the number of boxes is generally below 100 unless the memory is restricted to below 25%; similarly, we never copy more than 15x of the input data even for a 5% memory restriction. An example for how the boxes were chosen for the TWITTER data set is shown in Fig. 13. The figures show how these boxes are created by projecting the 3-D input space onto the x-y plane. Darker pixels indicate areas where there is more data. In particular, the image was created as follows: For $E(x, y)$ of the directed graph for the TWITTER dataset which can be viewed as a point-set in 2D space, create a 2D histogram H with 150x150 bins. Then, because we slice along the first dimension and collect the nodes plus their neighbors, we aggregate over H's second dimension (eg, y) values to obtain a 1D histogram D showing the total number of neighbors the nodes in a certain bin have. We then spread this 1D histogram into a 2D space by setting the value at position x, y to $D(x) + D(y)$. This "image" is indicative of the total amount of data for a rectangular box.

Spill is a corner case that never occurred for any of the tested graphs as long as the available memory fraction is larger than or equal to 5% of the total input size. As sug-



(a) (b)

Figure 14: (a) Speedup of Boxed LFTJ over Vanilla LFTJ. Memory restrictions are 25% and 35% of input size respectively. (b) Speedup of LFTJ over Graphlab for single and 8 threaded configurations without resource limitations.



(a) Limit: 10% of input (b) Limit: 25% of input

Figure 15: Speedup of Boxed LFTJ (1,8 threads) over MGT (1 thread) with limited memory.

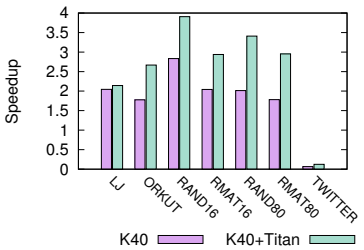


Figure 16: Speedup of one or two GPUs over CPU

gested in Fig. 12, it is not beneficial to use even smaller fractions for boxing because the number of boxes grows exponentially when the memory shrinks. We include spill in the algorithm for correctness.

Last, we are interested in how the boxed LFTJ compares to a variant without our extension. Since LFTJ as presented in [21] is a family of algorithms that needs to be parameterized by how data is physically stored and how the TrieIterator operations are implemented, answering this question is hard since conclusions for one specific implementation of the data back-end might not hold for another. In particular, our approach of storing data in huge arrays and performing mostly binary searches over them might be particularly bad from an I/O perspective. However, having these considerations in mind, we also ran our version of LFTJ with the *cgroup* memory restrictions and a provisioning mode that does not copy the data but leaves it in memory-mapped files. The data is thus paged in (from the input file) by the Linux virtual memory system that using a standard replacement strategy. Results for this experiment are shown in Fig. 14(a): The average speed ratios of vanilla over boxed for the memory levels of 10%, 25%, and 35% are 65x, 30x, and 20x, respectively.

Comparison to competitors. We compare to (1) the triangle counting algorithm presented in [19] and implemented in Graphlab [10]. We chose this algorithm as our in-memory competitor since it supports multiple threads and was used in other comparisons [24] before. We also (2) compare to the MGT algorithm [7] as the (to the best of our knowledge) currently best triangle listing algorithm in the out-of-core setting. Our results are shown in Fig. 14(b) and Fig. 15. The boxed LFTJ is on average 65% slower than Graphlab, both when run in single-threaded mode as well as in multi-threaded mode with 8 threads. Graphlab, being optimized for an in-memory setting with optional distribution, was not able to run any of our large data sets getting "stuck" once all of the 32GB of main memory and 32GB of swap space had been consumed.

Comparing to MGT (Fig. 15): We used the cgroup-memory restrictions and cleaned caches for running MGT and boxed-LFTJ. When we run in single-threaded mode, then MGT outperforms boxed LFTJ by a factor of 3.1, 2.9, and 2.9 in the configurations with 10%, 25%, and 35% of the memory, respectively. When we allow LFTJ to utilize all of the 4 available cores, we are on average 47%, 22%, and 28%, respectively, faster than the single-threaded MGT. We have not investigated how well MGT parallelizes. Note that MGT internally uses only 32 bits as node identifiers (vs. our 64bit identifiers). Nevertheless, we used the same values to configure and limit the amount of memory for both MGT and LFTJ. Note that boxed-LFTJ scales well with the number of threads because the workload is often CPU bound.

Running multiple GPUs. The last experiment demonstrates the capability of using multi-GPUs with the boxing algorithm. Fig. 16 shows the speedup of using one K40 and using both GPUs. The baseline is executing LFTJ in CPU by using 1GB box. K40 and Titan both use 512MB boxes so that the smallest graph, LJ, needs two boxes. Memory is not restricted in this set of experiment. Recall that two CPU threads execute the boxing algorithm as introduced in Section 6 when two GPUs are used. One K40 can bring around 2x speedup because of its high computation throughput. When concurrently executing LFTJ in two GPUs, the speedup increases to 2-4x compared with CPU LFTJ. Using two GPUs is always faster than using one GPU. The scalability differs from graph to graph. The extra GPU only brings 5% additional speedup for LJ because there is not enough data to saturate both GPUs. For all other benchmarks, extra GPU can bring at least 40% additional speedup including the largest TWITTER graph which has 90% additional speedup because of the low cost of boxing and full utilization of two GPUs executing the join algorithm.

8. RELATED WORK

The SociaLite effort [20] at Stanford also proposes to use systems based on relational joins (in this case Datalog) for graph analysis. They show that declarative methods not only allow for more succinct programs but are also competitive, if not outperform typical other implementations.

A worst-case optimal join algorithm has first been presented by Ngo et al. in [14] following the AGM bound [1] that bounds the maximum number of tuples that can be produced by a conjunctive join. Most recently, Khamis et al. [13] propose so-called *beyond-worst-case-optimal* join algorithms. Here, the performed work is not measured against a worst-case within a set family of inputs—but instead must be proportional to the size of a shortest proof of the results correctness. Furthermore, [8] combines ideas from geome-

try and resolution transforming the algorithmic problem of computing joins to a geometric one.

Studies have been shown in the out-of-core context for triangle listing. Following up on the MGT work [7], Rasmus et al. [15] improve the I/O complexity of MGT from $O(|E|^2/(MP))$ to an expected $O(E^{3/2}/(\sqrt{M}P))$. They also give lower bounds and show that their algorithm is worst-case optimal by proving that any algorithm that enumerates K triangles needs to use at least $\Omega(K/(\sqrt{M}P))$ I/Os. Menegola [11] proposes an algorithm with an I/O complexity of $O(|E| + |E|^{1.5}/P)$; furthermore [5] proposes an algorithm with an I/O complexity of $O(|E|^{1.5}/P \cdot \log_{M/P}(|E|/P))$.

9. CONCLUSION

For the well-studied problem of triangle listing, we have investigated how a *general-purpose* & worst-case optimal join algorithm compares against *specialized* approaches in the out-of-core context. By using Leapfrog Triejoin, we are able to devise a framework with the boxing strategy that reduces I/O cost and demonstrate competitive performance in a heterogeneous environment with the CPU and the GPUs. Our positive results can be interpreted as a confirmation for the database community's theme of creating systems to empower users via declarative query interfaces while providing very good performance. Moreover, the proposed framework is a complete query engine that can run any relational join. It can be combined with a query optimizer to determine the variable ordering to get better performance.

10. ACKNOWLEDGMENTS

We gratefully acknowledge the support of National Science Foundation under grant CCF-1337177 and the Intel Science and Technology Center on Cloud Computing.

11. REFERENCES

[1] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS'08*, pages 739–748. IEEE, 2008.

[2] S. Baxter. Modern gpu. http://nvlabs.github.io/moderngpu/, 2013.

[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[4] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *SIGKDD*, pages 672–680. ACM, 2011.

[5] R. Dementiev. *Algorithm engineering for large data sets.* PhD thesis, Saarland University, 2006.

[6] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. PPoPP '13, pages 301–302, 2013.

[7] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *SIGMOD*, pages 325–336. ACM, 2013.

[8] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. A resolution-based framework for joins: Worst-case and beyond. *CoRR*, abs/1404.0703, 2014.

[9] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10*, pages 591–600, New York, NY, USA, 2010. ACM.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[11] B. Menegola. An external memory algorithm for listing triangles. Technical report, Universidade Federal do Rio Grande do Sul, 2010.

[12] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC'07*, San Diego, CA, October 2007.

[13] H. Q. Ngo, D. T. Nguyen, C. Ré, and A. Rudra. Beyond worst-case analysis for joins with minesweeper. *CoRR, abs/1302.0914*, 2014.

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

[15] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *PODS'14*, pages 224–233, 2014.

[16] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[17] J. W. Raymond, E. J. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6):631–644, 2002.

[18] N. Rhodes, P. Willett, A. Calvet, J. B. Dunbar, and C. Humblet. Clip: similarity searching of 3d databases using clique detection. *Journal of chemical information and computer sciences*, 43(2):443–448, 2003.

[19] T. Schank. Algorithmic aspects of triangle-based network analysis. *Phd in computer science, University Karlsruhe*, 2007.

[20] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

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

[22] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. CGO '14, pages 44:44–44:54, 2014.

[23] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *PLC*, IPDPSW '12, pages 2433–2442, 2012.

[24] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili. Multipredicate join algorithms for accelerating relational graph processing on GPUs. *ADMS 2014*, September 2014.

[25] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[26] D. Zinn. General-purpose join algorithms for listing triangles in large graphs. *CoRR*, abs/1501.06689, 2015.