The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins

Oren Kalinsky Benny Kimelfeld Yoav Etsion
{okalinsk,bennyk,yetsion}@technion.ac.il
Technion–Israel Institute of Technology
Haifa, Israel

Abstract

Graph pattern matching (e.g., finding all cycles and cliques) has become an important component in domains such as social networks, biology and cyber-security. In recent years, the database community has shown that graph pattern matching problems can be mapped to an efficient new class of relational join algorithms.

In this paper, we argue that this new class of join algorithms is highly amenable to specialized hardware acceleration thanks to two fundamental properties: improved memory locality and inherent concurrency. The improved locality is a result of the bound number of intermediate results these algorithms generate, which yields smaller working sets. Coupled with custom caching mechanisms, this property can be used to dramatically reduce the number of main memory accesses invoked by the algorithm. In addition, their inherent concurrency can be leveraged for effective hardware acceleration and hiding memory latency.

We demonstrate the hardware amenability of this new class of algorithms by introducing TrieJax, a hardware accelerator for graph pattern matching that can be tightly integrated into existing manycore processors. TrieJax employs custom caching mechanisms and a massively multithreaded design to dramatically accelerate graph pattern matching. We evaluate TrieJax on a set standard graph pattern matching queries and datasets. Our evaluation shows that TrieJax outperforms recently proposed hardware accelerators for graph and database processing that do not employ the new class of algorithms by 7 – 63× on average (up to 539×), while consuming 15 – 179× less energy (up to 1750×), and systems that incorporate modern relational join algorithms by 9 – 20× on average (up to 45×), while consuming 59 – 110× less energy (up to 372×).

ASPLOS'20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378524

CCS Concepts. • **Hardware → Hardware accelerators**; • **Information systems → Join algorithms**; *Relational parallel and distributed DBMSs*; • **Mathematics of computing** → *Graph algorithms.*

Keywords. hardware acceleration; relational join; databases; graph analytics; hardware and algorithmic design

ACM Reference Format:
Oren Kalinsky Benny Kimelfeld Yoav Etsion. 2020. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In *Proceedings of Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378524

1 Introduction

Analyzing the relationships in a graph has become a key building block in many domains, including social networks [35], biology [14], and artificial intelligence [31]. A recent study [30] analyzed the common challenges in graph processing and found that pattern matching problems, namely finding all instances of a given pattern in a graph, to be a dominantly popular problem in graph application domains. Graph pattern matching problems, however, are known to be computationally intensive and thus challenge algorithm designers.

In recent years, the database community has proposed new relational join algorithms that efficiently map graph problems to query evaluation over relational databases. In particular, a new breed of *Worst-Case Optimal Join* (WCOJ) algorithms has been introduced and studied [2, 16, 22, 23, 34]. These algorithms were shown to be theoretically superior to the traditional join algorithms [4], and WCOJ-based solutions for graph matching problems have been shown to deliver superior performance compared to known graph algorithms [26].

From an architectural perspective, pattern matching via WCOJ algorithms offers two features that make it hardware friendly. First, WCOJ algorithms provably bound the number of intermediate results they generate, which greatly reduces the algorithms' working-set size, reduces data transfers to and from memory, and makes them more amenable to caching. In contrast, traditional join algorithms partition multi-way joins (a join operation on multiple relations, or tables) into a tree of binary join operations. Each binary join operation then performs a memory scan on its input

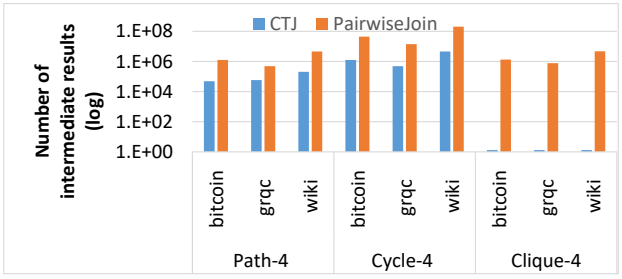

Figure 1. Number of intermediate results (log scale) being generated by CTJ, a WCOJ algorithm, compared to the pairwise join algorithm used by Q100.

relations and generates a (potentially huge) intermediate relation. Importantly, many of the intermediate results are typically filtered out by subsequent join operations. Figure 1 compares a WCOJ algorithm to a pairwise join algorithm used by Q100, a database accelerator, on three pattern matching queries over three standard datasets. For the queries in the figure, the WCOJ algorithm generates $15 - 1,000,000\times$ less intermediate results than the pairwise join algorithm.

Second, WCOJ algorithms are highly concurrent. Although the algorithms' control flow is non-trivial, which makes them unsuitable for GPGPUs, their inherent concurrency makes them amenable to hardware acceleration. In addition, this inherent concurrency allows specialized accelerators to apply multithreading techniques to hide memory latency.

In this paper, we argue that WCOJ-based algorithms for solving graph pattern matching problems are highly amenable to specialized acceleration. We describe how graph problems can be mapped to relational join operations and detail how the new breed of relational join algorithms maps to hardware. We further present TrieJax, an on-die, domain-specific accelerator that leverages the hardware-friendly properties of WCOJ algorithms to accelerate graph pattern matching problems and dramatically reduce energy consumption.

TrieJax employs a highly-concurrent WCOJ variant [16] that scans table indexes stored in a tree-based data-structures. In addition, TrieJax caches partial join results to speed up computation and minimize memory traffic. Furthermore, TrieJax is designed to be small enough to serve as a dedicated core in a standard many-core processor. This design allows TrieJax to use the same memory system as other cores in the system, a property that is crucial for operating on large data sets. This is in contrast to discrete accelerators (e.g., GPGPUs), which have limited directly-attached memory capacity and require frequent data transfers between the system memory and their local memory. We validated the performance, area, and energy consumption of TrieJax using a synthesized and placed&routed RTL implementation, whose results drove a cycle-accurate simulator.

We demonstrate the improved performance and reduced energy obtained by leveraging WCOJ-based graph pattern matching algorithms via a comparison of TrieJax to recently proposed database and graph processing accelerators [11, 36]. Unlike TrieJax, these accelerators employ traditional join algorithms and are therefore susceptible to the potential explosion of intermediate results that these algorithms generate. We also demonstrate the benefits of hardware acceleration of WCOJ algorithms by comparing TrieJax to two software database management systems [1, 16] that are based on the new breed of WCOJ algorithms. Ultimately, TrieJax outperforms the state-of-the-art hardware accelerators by $7 - 63\times$ on average, while consuming $15 - 179\times$ less energy. Compared to the software systems, TrieJax runs $9 - 20\times$ faster and consumes $59 - 110\times$ less energy.

In summary, we make the following contributions:

- We present the hardware-friendly properties of WCOJ algorithms and how they can be used for solving graph matching problems.
- We present a domain-specific hardware accelerator that leverages the hardware-friendly properties of WCOJ algorithms for graph pattern matching.
- We demonstrate the performance and energy benefits of a WCOJ-based pattern matching accelerator through a detailed experimental evaluation.

The rest of the paper is organized as follows. Section 2 explains how graph pattern matching algorithms are mapped to relational join operations and describe the new breed of WCOJ algorithms in more detail. Section 3 describes the TrieJax accelerator architecture and how it maps a WCOJ algorithm to hardware. Finally, Section 4 presents an experimental evaluation comparing the performance and energy consumption of TrieJax to database and graph analytics accelerators on standard graph pattern matching queries.

2 New relational join algorithms and graph pattern matching

Graph pattern matching problems have become paramount in different computational domains, including social networks [35], biology [14], and artificial intelligence [31]. Yet despite the focus in recent years on developing general graph analytics frameworks (including hardware accelerators and optimizations [11, 21, 27]), these frameworks mostly benefit more well-known graph problems (e.g., Breadth-First Search and PageRank) and often ignore graph pattern matching problems.

Recent advances in relational database theory have introduced methods for efficient computation of graph algorithms based on relational join operations. These methods have been found particularly effective for computing graph pattern matching problems. In this section, we describe how relational join operations can be used for graph analytics

```
SELECT *
FROM Posts as R, Likes as S, Follows as T
WHERE R.postID=S.post and S.user=T.followed
```

Figure 2. A relational join query example

and review the algorithmic advances in relational join algorithms. We then discuss how these algorithms, unlike traditional ones, avoid generating a large number of intermediate results, which makes them amenable to hardware acceleration.

2.1 Relational Join

Relational database management systems (RDBMS) are a common solution for data management. This type of database follows the relational model of data. In this model, the data is stored in relations, also known as *tables* (e.g., the tables *Posts* and *Likes* mentioned in Figure 2), the table columns are the attributes (e.g., *user* and *postID*), and the rows (tuples) are the values. Each row in the table can have an attribute that is a unique primary key. For instance, the *Posts* table has the primary key *postID*. Other tables can reference a specific table through its primary key using an attribute known as a *foreign key*. In our example, the *Likes* table can reference a post using the *post* attribute as a foreign key to *Posts*.

A relational join query is a query that analyzes the relationship between tables via shared attribute values. Figure 2 shows a simple relational join query—the natural join of the three relations *Posts*, *Likes* and *Follows*. The query computes information about posts liked by users with followers. More precisely, the query is asking for tuples where each consists of a post, a user who likes the post, and a follower of the user. While there are different types of join operations, in this work we focus on natural equi-joins where tables are joined by equality of mentioned attributes.

Relational join for graph analytics. Many graph algorithms can be translated to (SQL-like) join queries, which allows solving graph problems using RDMS solutions. A finite graph is commonly represented in an RDBMS by an adjacency list relation. Each row in the relation represents an edge between two vertices in the graph. Thus, graph patterns queries can be translated to join queries. For example, given a graph relation G, the query $Q(x, y, z) = G(x, y) \bowtie G(y, z) \bowtie G(z, x)$ returns all the triangles in the graph.

Mapping graph pattern matching algorithms to WCOJ-based systems have been shown to be highly effective, and WCOJ-based systems can provide speedups of up to two orders of magnitude compared to low-level graph analytic solutions [4]. Nevertheless, these performance benefits are not universal across all graph algorithms. For example, Aberger et al. [1] have shown that problems such as SSSP [15] and PageRank [28] do not enjoy similar benefits. In this paper, we focus on the family of graph pattern matching problems.

Database and graph analytic acceleration. Acceleration of database and graph analytics using SIMD, GPGPU, or specialized hardware accelerator has been a common interest in both the database and micro-architecture communities. Widx, an on-chip accelerator proposed by Kocberber et al. [18], focuses on hash join acceleration using specialized programmable RISC units for computing the hash and traversing index. Using the standard memory hierarchy and spanning over a small area, Widx offers 3.5× speedup on average on TPC-H and TPC-DS hash joins compared to MonetDB, a commonly used column store. Gold et al. [10] leveraged the multithreaded multi-core of the network processor for accelerating database operations, including hash-join, and achieve a speedup of up to 2.5× on the TPC-H workload compared to a CPU. The Mondrian Data Engine, a Near-Memory Processor by Drumond et al., reshapes data query algorithms to better fit the underlying near-memory processing hardware.

Q100 by Wu et al. [36] was the first hardware accelerator that fully supported relational operations. It incorporates relational operators (such as Sort, Select, and Merge-Join) as hardware components in a hardware accelerator for relational column stores. Q100 offers a solution that searches the best custom chip for a specific query from time and energy perspectives. Q100 achieves a speedup of 10× on TPC-H compared to MonetDB.

Graphicionado by Ham et al. [11] focuses on graph analytics. It implements the vertex-programming model in hardware with embedded programmable units that allow flexible support for graph algorithms such as PageRank or SSSP. This hardware accelerator achieves a speedup of $1.76 - 6.5\times$ compared to GraphMat [33], a vertex-programming framework that scales using sparse matrix representation and MPI.

New relational join algorithms for graph analytics. Recently, the database community established new theoretical and algorithmic advances in the area of relational join algorithms. Given a join query with more than two relations, standard join algorithms use the pairwise join approach. The traditional algorithms, such as hash-join [9] or sort-merge join [20], join two relations at a time to create a new intermediate relation. This intermediate relation will later be joined with another (input or intermediate) relation until the final result is computed. Recent work by Atserias et al. [4] shows that pairwise join algorithms can generate many unnecessary intermediate results that are not part of the final result. It helps define a tight bound, called the AGM bound, on the maximum number of results that can be returned from a query in the worst case.

We illustrate the AGM bound with an example. Given the relations A, B and C, consider the triangle join query: $Q(x, y, z) = A(x, y) \bowtie B(y, z) \bowtie C(z, x)$. For simplicity, we assume that each relation has N tuples. The AGM bound proves that the query result $Q(x, y, z)$ contains no more than $N^{\frac{3}{2}}$ results. However, pairwise join algorithms can generate an

intermediate result with up to N^2 tuples, while many of them will be filtered by the third relation. Any join algorithm that provides the same complexity as the AGM bound is called Worst-Case Optimal Join (WCOJ) algorithm. In contrast, the traditional pairwise join algorithms are not worst-case optimal. More formal definitions and extensions to general queries can be found in a survey by Ngo et al. [24].

Systems and acceleration of WCOJ algorithms. Wu et al. [5] used GPGPU to accelerate a WCOJ algorithm and achieved 2–6× speedups compared to CPU. EmptyHeaded by Aberger et al. [1] offered a relational query engine that maps Generic Join, a WCOJ algorithm, to parallel SIMD operations on a standard CPU. For graph algorithms, EmptyHeaded achieves comparable results to Galois [25], a low-level hand-tuned query engine comparable to GraphMat. On graph pattern matching queries, it achieves a $2 - 60\times$ speedup compared to other CPU based solutions. Section 4 compares our TrieJax to the systems above or their baselines. We omit the GPU baseline, since its performance proved inferior to those obtained on an Intel SIMD unit, presumably because of irregular data traversals in WCOJ algorithms.

2.2 Cached TrieJoin

Following the publication of the AGM bound, a plethora of Worse Case Optimal Join (WCOJ) algorithms was published [2, 16, 22, 23, 34]. Experimental analysis by Nguyen et al. [26] has shown that the WCOJ algorithms provide significant speedups on complex join queries compared to the traditional approaches such as state of the art RDBMS, graph engines and column stores.

LeapFrog TrieJoin [34], also known as LFTJ, is a commonly used WCOJ algorithm. Its main idea is to iterate over trie-based indexes of the relations in a backtracking manner to generate the join query results. LFTJ does not generate any intermediate results and thus yields a low memory footprint, but it does so at the expense of recomputing recurring intermediate partial joins. Furthermore, the recurring computations have little memory locality as they repeatedly scan irregular, tree-based tries. Cached TrieJoin (CTJ) by Kalinsky et al. [16] eliminates much of the recurring partial join computations by selectively caching partial join results using the available system memory (without violating the WCOJ property). This behavior can benefit low-memory environments and is therefore used as one of the main building blocks in our system.

CTJ operates as follows: Given a query, CTJ decomposes the query structure to detect which attributes can be valid keys and their respective cached values. Then, it uses a caching system to drive the TrieJoin, while saving partial join results in the cache and extracting them later to avoid recurrent computations. CTJ shows a 10× speedup on average compared to LFTJ, and even better speedups compared to traditional query engines.

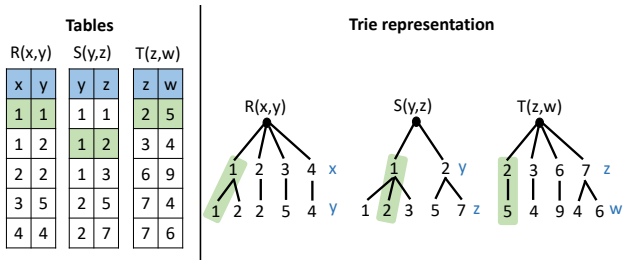

Figure 3. Example of tables from a social network (left) and their trie representation (right). Marked (green) a Path-4 join result between the three tables.

2.2.1 Indexes.

CTJ saves its relations in tries, a multi-level data structure commonly used in WCOJ systems [1], column stores and graph engines [12, 32]. Section 3 elaborates on the physical layout of our indexes. Given a relation, for example the $R(x, y)$ relation with two attributes in Figure 3, the trie representation is as follows:

- Each attribute, such as x, is a level in the trie.
- Each unique path from the root to a leaf is an entry in the relation. For example, the path $(1, 1)$ corresponds to the entry of $x = 1$ and $y = 1$ in $R(x, y)$.
- The siblings are sorted.

2.2.2 The Cached TrieJoin algorithm.

Next, we elaborate on CTJ, one of our main building blocks. Figure 4 will be used to illustrate the CTJ execution flow. The algorithm is presented in Figure 5. CTJ first orders the variables (e.g, $x \rightarrow y \rightarrow z \rightarrow w$). Then, it looks for matches for each variable in turn from first (x) to last (w). Initially, the cache is empty (line 14 in Fig. 5). Starting from x, CTJ will search all the tries that contain x (e.g., $R(x, y)$) for a match (① in Fig. 4). Each match is found using a variation of merge-join, called *leapfrog-join* [34]. Leapfrog-join uses lowest upper bound searches to leap over the variable levels until a match is found. If a match is found, it sets the x value in the result buffer (line 15 in Fig. 5).

Before continuing to the next variable (e.g., y), CTJ adjusts the tries (Line 16) such that they align on the children of the current partial join path (the nodes below the paths marked in red in Fig. 4). For example, after step ①, the R trie is set on the y child nodes of $R(x) = 1$. Finally (Line 17), the algorithm calls CTJ to look for a match for the next variable y (②). In practice, CTJ uses queues instead of recursion. Once all the variables are set (④), CTJ saves the result (line 2). If no other matches are found for the current variable, CTJ resets the adjusted tries (line 20) to focus on the previous attribute.

CTJ uses a cache for partial join results to avoid the computation of recurrent partial joins. Given a join variable (e.g., z), CTJ extracts the variables that are used as caching keys (e.g., y) and the variables that are cached by these keys (lines 4–5). During the standard execution, CTJ searches the cache

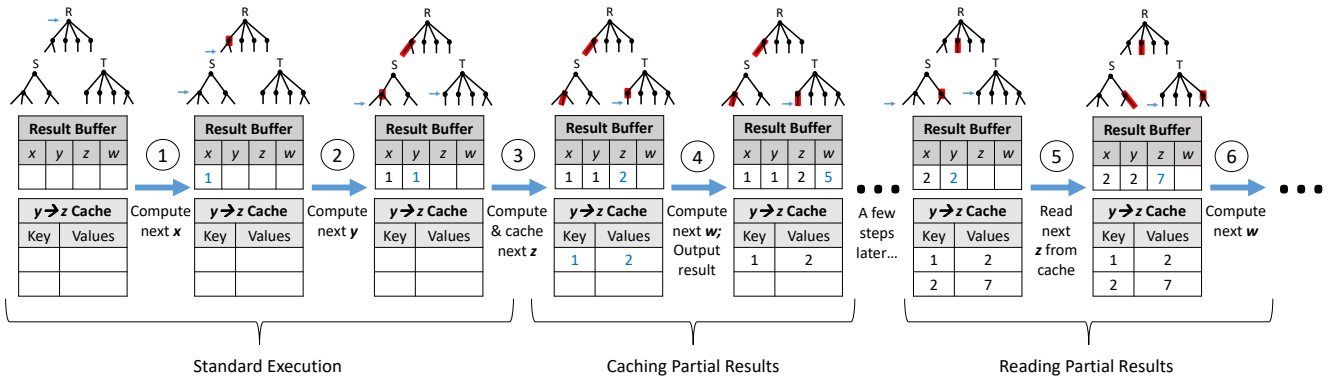

Figure 4. Cached TrieJoin execution and caching flow of an example Path-4 query. Each step marks the current partial join path on the tries (top)

Algorithm CachedTrieJoin(d, inds, cache, res)

```
1:  if d = n + 1 then
2:      save res
3:      return
4:  keyIDs := cacheKeysOf(d)
5:  valIDs := cacheValsOf(keys)
6:  if res[keyIDs] is a cache hit in cache then
7:      for all cached in cache(res[keyIDs]) do
8:          res[valIDs] := cached
9:          AdjustTries(inds, res)
10:         next := max(valIDs) + 1
11:         CachedTrieJoin(next, inds, cache, res)
12:     ResetTries(inds, min(valIDs))
13:     return
14: for all matching values a for attribute d in inds do
15:     res[d] := a
16:     AdjustTries(inds, res)
17:     CachedTrieJoin(d + 1, inds, cache, res)
18:     if d = max(valIDs) then
19:         ApplyCaching(cache, res[keyIDs], res[valIDs])
20: ResetTries(inds, d)
```

Figure 5. The CachedTrieJoin algorithm

for the value of the current variable. If found, it uses the results from the cache and adjusts the tries accordingly (lines 9–11). For example, ⑤ finds $y = 2$ in the cache and reads the value of z from the cache instead of recomputing the join for z. Writing to the cache is done during the standard execution (③), after finding a valid match for a cached entry (line 19).

3 TrieJax Architecture

3.1 System-level overview

TrieJax is designed as a small, on-die co-processor that integrates to the main processor like an additional core, as depicted in Figure 6. This design allows TrieJax to use the main processor's memory system and avoids the coherence and data transfer overheads that plague discrete accelerators.

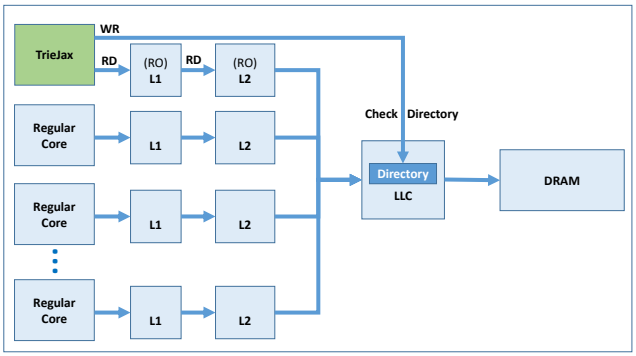

Figure 6. System architecture with TrieJax as another core and the communication between TrieJax and the memory

TrieJax uses local read-only L1 and L2 caches that cache table data and indexes (Tries). Since TrieJax stores its intermediate results in a private scratchpad, as we describe later on, the final results of the join operations are written directly to memory. The data written in these operations bypass the private L1 and L2 caches and are streamed directly to memory to avoid cache thrashing and congestion in the cache queues (only the directory in the shared L3 cache is checked, in order to invalidate stale copies).

For example, on some of the benchmarks we evaluate (e.g., *path-4* query), where the size of the resulting join table is extremely large, bypassing the private caches improves performance by up to 2.2×. The coherence of the input indexes and the output data is managed by the caching system. Moreover, TrieJax uses standard virtual memory translation between the TrieJax core and the memory system.

The regular processor cores communicate with TrieJax using a co-processor interface similar to that of ARM [3] or RISC-V [29]. For instance, we use a parallel of the ARM *LDC* command to load the query to the TrieJax internal memory.

Name	Query
Path-3	$path3(x, y, z) = R(x, y), S(y, z).$
Path-4	$path4(x, y, z, w) = R(x, y), S(y, z), T(z, w).$
Cycle-3	$cycle3(x, y, z) = R(x, y), S(y, z), T(z, x).$
Cycle-4	$cycle4(x, y, z, w) = R(x, y), S(y, z), T(z, w), U(w, x).$
Clique-4	$clique4(x, y, z, w) = R(x, y), S(y, z), T(z, w), U(w, x), V(z, x), W(w, y).$

Table 1. Graph pattern matching operations used to evaluate TrieJax and their mapping to join queries (shown in datalog format, for brevity).

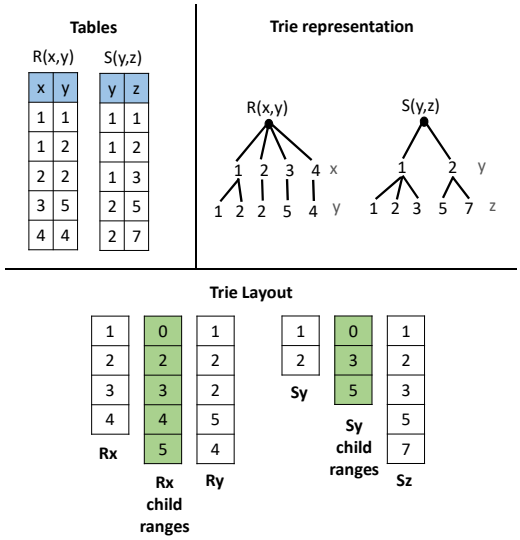

Figure 7. Tables for an example query $path3(x, y, z) = R(x, y), S(y, z)$. (top left), their trie representations (top right), and their memory layout (bottom)

3.2 Query language and index memory layout

We use the CTJ compiler [16] to compile SQL join queries for TrieJax. Table 1 lists the graph pattern matching queries used in our evaluation (Section 4) and their mapping to join queries (for brevity, the table uses the compact datalog format rather than SQL).

Figure 7 illustrates a trie layout in TrieJax. TrieJax uses in-memory trie indexes (described in Section 2.2) in a physical layout similar to that of EmptyHeaded [1]. Specifically, this layout stores the unique values of the first join attribute in the relation as a sequential array. The next join attribute is then stored by concatenating the values that match the previous attribute to a continuous array. To identify which values of the second join attribute belong to which elements in the previous attribute, the child ranges array lists the corresponding ranges of the second attribute. For example, Figure 7 shows that the values $\{(1, 1), (1, 2)\}$ in R can be extracted by focusing on $R(x = 1)$, extracting the ranges of its children from the respective indexes $[0, 2)$ in the *Rx child ranges* array, and accessing the extracted index range in the *Ry* array.

3.3 From algorithm to architectural components

The architectural design of TrieJax is dictated by the operational flow of the CTJ algorithm. The CTJ algorithm can be divided into two major flows, the TrieJoin flow and the flow of reusing recurrent join results via caching. We design our architecture similarly, as presented in Figure 8. Specifically, the TrieJoin is decoupled to four components: *LUB*, *MatchMaker*, *Midwife* and *Cupid*. The flow of reusing recurrent join results is managed by the Partial Join Results Cache (PJR cache) and Cupid. Table 2 summarizes the components and their roles in the architecture. This section describes how the algorithmic flow is mapped to architectural components, and the operational flow of the architecture is depicted in the following section.

The most basic operation in TrieJoin, as described in Section 2.2, is a variant of merge-join called Leapfrog-join. It is leveraged for finding matching values of two trie levels that share a join parameter. Leapfrog-join uses lowest-upper-bound searches to look for the first matching value of two arrays. The TrieJax design decouples the Leapfrog-join from the lower-upper-bound search, which allows the former to drive many concurrent search operations and thereby hide memory latency. The binary search logic of the lowest-upper-bound is executed by the multi-threaded *LUB* component, and the Leapfrog-join itself is managed by the *MatchMaker*.

The TrieJoin flow, namely finding matches for all variables over all the tries, is managed by the *Cupid* component. This component communicates with the *MatchMaker* to compute a match for a join parameter. Once a match is found on two nodes in the trie, *Cupid* drives the *Midwife* component to extract the children array of the matching nodes. These arrays, which correspond to the values of the next join parameter in the trie, are then sent to the *Matchmaker* component for finding a match.

Finally, *Cupid* also manages reads from and write to the PJR cache. Section 3.6 describes the PJR cache in detail.

3.4 The TrieJax operational flow

Figure 8 presents the high-level design of TrieJax and the communication between the different components it utilizes to answer graph pattern matching queries. The five major building blocks are lowest upper bound (*LUB*), *MatchMaker*, *Midwife*, *Cupid* and the partial join results cache (PJR cache). We illustrate the operational flow of TrieJax on the *Path-3* example query in Figure 7. We begin by describing the operational flow as single threaded. Section 3.5 then describes how TrieJax incorporates multi-threading internally to hide

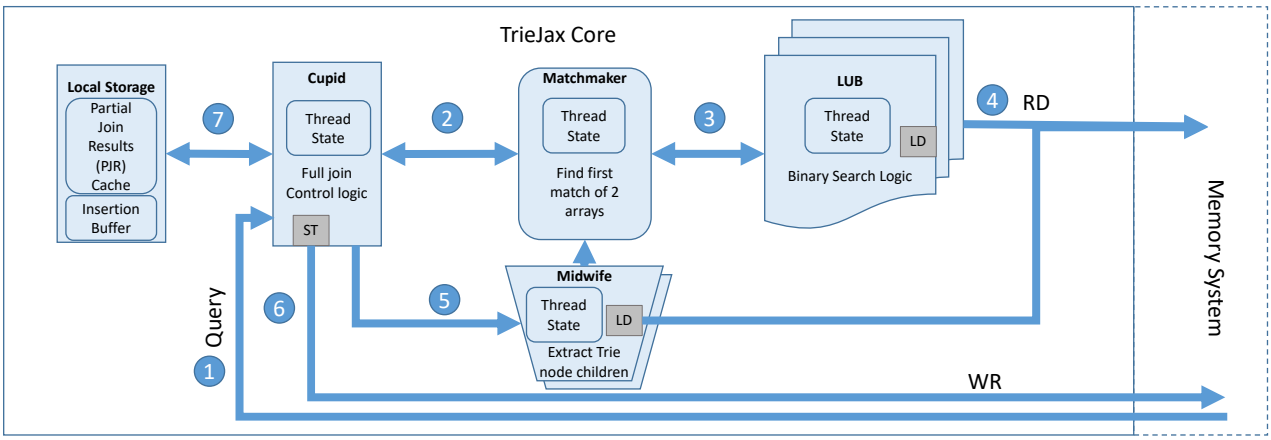

Figure 8. TrieJax core components and its high-level operational flow.

memory latency. The micro-architecture and the internal flow of each component are described in Section 3.7.

The query execution begins by loading the compiled query to a local read-only store in *Cupid*, which controls the execution of the full join query (marked ❶ in Figure 8). *Cupid* starts with the first join variable x, by extracting the pointers to the trie arrays of x embedded in the compiled query structure (e.g., Rx), and sending them to *MatchMaker* (marked ❷ in Figure 8). Until a response is returned, *Cupid* saves the current state (e.g., result buffer, parameter id) in its local State Store.

MatchMaker is in charge of finding the first matched value of x in all the tries (based on the Leapfrog Join algorithm described in Section 2.2). Our running example (Figure 7) only has one array (Rx), so *MatchMaker* will request the *LUB* unit to find the first value for x in Rx (❸). *LUB* performs a binary search to look up the lowest upper bound of a value in a given array. The unit will search for the index in the trie array in which the value is stored (❹), e.g., $Rx_{index} = 0$, and send the result back to *MatchMaker*. *MatchMaker* will send the read value and index (e.g., $x = 0$, $Rx_{index} = 0$) back to *Cupid*, which will save the index in the current state of variable x. Similarly to *Cupid*, *MatchMaker* and *LUB* save their local state in their State Store when sending a request and reads the data when receiving the response.

Next, *Cupid* will continue to the next variable y and look for a match for y. It will read the Ry and Sy array pointers from the compiled query. As described above, the Ry array stores *all* the values of the child nodes of Rx. To extract the Ry range that belongs to the current x value, *Cupid* will send the x index, the Ry pointer, and the Rx *child ranges* pointer to *Midwife* (❺). The *Midwife* component is in charge of extracting the children of a node in the trie. For example, to extract the Ry values for $Rx_{index} = 0$, *Midwife* will read from the child ranges array the start and end ranges from indexes 0 and 1, respectively. The final Ry range (e.g., $Ry[0:2]$) will be sent to *MatchMaker*.

Component	Role
LUB	Uses binary search to find the Lowest Upper Bound of a value in an array
Matchmaker	Given two sorted arrays, looks for the first matching value in the arrays
Midwife	Given a parent node in the trie, extracts its children
Cupid	Manages the search for matches for all the levels of all the tries
PJR Cache	Stores intermediate join results

Table 2. The role of each component in TrieJax.

Once *MatchMaker* receives both Sy and Ry array ranges sent from *Cupid* and *Midwife*, it will look for the first match of the two array ranges. It will send the pointer of the first value in Ry and the Sy array pointer to *LUB* (❸). *LUB* loads the value of Ry from memory and uses binary search to look for its lowest upper bound in Sy (e.g., $Sy = 1$). The result is returned to *MatchMaker*, which extracts its state from the State Store and checks if a match $LdVal$ was found. If so, it will return the data to *Cupid* that, in turn, will set the state for variable y and continue to the next variable z. If not, it will use *LUB* to look for $LdVal$ in the Ry range, using iterative LUB searches on Sy and Ry until a match is found or it reaches the end of an array. If no match is found, *MatchMaker* will return a failed response to *Cupid*, which will restore the previous variable and look for its next match. Once a match is found for x, y and z, *Cupid* will write the result to memory (❻). TrieJax uses a small write buffer and sends results to main memory when the buffered results exceed the size of a cache line.

3.5 Multi-threading in TrieJax

TrieJax's use of multi-threading (MT) achieves two performance benefits. First, MT parallelizes the trie join and thereby

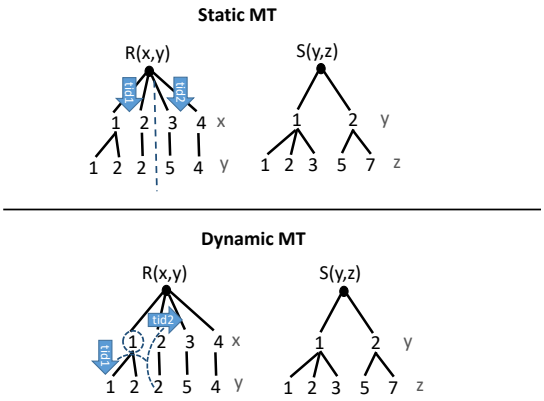

Figure 9. TrieJax MT schemes for the path-3 query. Static MT (top) divides the query search space statically based on the first join attribute. Dynamic MT (bottom) can spawn a new thread on every match to continue with the query while the current thread focuses on the sub-query given the match.

extracts memory-level parallelism (MLP), which allows TrieJax to hide memory latency. Second, MT pipelines the operations on the partial join results cache, and enables *Cupid* to lookup pre-computed partial results while the rest of the accelerator is looking for new results.

Figure 9 presents two different MT schemes, static and dynamic. The static MT scheme divides the arrays of the first attribute between the different threads. This scheme is similar to software MT solution in DBMS systems such as EmptyHeaded [1] and GPU query engines [5]. The disadvantage of this scheme is an unbalanced workload distribution between the different TrieJax threads. For example, in Figure 9, *tid1* will generate many results while *tid2* will finish quickly without any results.

Dynamic MT, on the other hand, balances the load by dynamically allocating new threads on each match. On a match, the *Cupid* unit splits the search space to two subspaces. The original thread is then bound to the search space containing the current partial result, and the new thread will be bound to the search space after the current match. For example, in Figure 9, a match is found for $x = 1$ while using dynamic MT. After the match, *tid1* will compute the results for $x = 1$ and *tid2* will compute the results for $x > 1$.

On its own, however, dynamic MT might incur a slow initialization time for queries with infrequent matches due to fewer matching opportunities. TrieJax thus combines both static and dynamic MT.

Each component supports multi-threading by replicating its internal execution state. As shown in Figure 8, each component in the system maintains a small local memory to store the execution state when waiting for a response from another component (e.g. when *Cupid* requests *MatchMaker* to find the next match for a variable). Replicating this state allows each component to maintain multiple operations in-flight.

For duplicated components, such as *Midwife*, we use banked stores to support parallel accesses. Section 4.2 examines the performance impact of MT and identifies the number of threads required to achieve performance/storage balance.

3.6 Caching with threads

Locality across partial join results enables TrieJax to break query execution into two parallel flows using the TrieJax partial join results cache (PJR cache). In the main flow, *Cupid*, *Midwife*, *MatchMaker* and *LUB* construct new partial results. In the other flow, *Cupid* maintains partial results in the PJR cache and checks if the stored partial results can be used instead of executing the main flow and recompute them. Decoupling the flows and using MT in both allows the units to execute concurrently.

The PJR cache is a locally managed scratchpad that stores intermediate partial join results. The TrieJax compiler directs the cache's logical structure by generating the relational attributes that will serve as keys to the cache and the those that will be stored as values. In the example depicted in Section 2.2 (Figure 4), the cache entry of PJR cache stores the partial join values and indexes of z (e.g., the value 2 and its indexes) and it is keyed by a hash of the corresponding y value (e.g., 1). The indexes in the trie array are stored to allow the expansion of the children nodes by *Midwife*.

When *Cupid* finds a match on a key attribute (e.g., y), it searches the PJR cache for its partial join results. For instance, the values of z for the given y (similarly to step ⑤ in Figure 4). In the case of a cache hit, it uses the cached results instead of recomputing them. Otherwise, PJR cache will allocate a new entry for the array of intermediate results, and *Cupid* will use the main flow to compute the values and set them (and their indexes) in the entry.

Since partial join results are continuously allocated, are of variable lengths, and their final size is unknown until they are computed, the PJR cache must allow values to be pushed to existing entries. At the same time, it must prevent large partial join results from overflowing their allocated storage. The PJR cache thus allocates 1KB for new entries and maintains a free space counter in conjunction with each entry. The counter is updated whenever new values are pushed to the entry. In the event that an entry overflows, it is deallocated to avoid storing incomplete results. The PJR cache is also specialized for managing intermediate result reads and writes from several threads.

A major challenge when using MT to fill the PJR cache is managing read/write race conditions. We solve this issue by adding an insertion buffer that stores entries that were not fully analyzed, meaning that TrieJax did not finish analyzing all the paths under the entry key. Once an entry is fully analyzed it is copied atomically to the PJR cache. Another MT challenge is write/write races. A keen reader will notice that

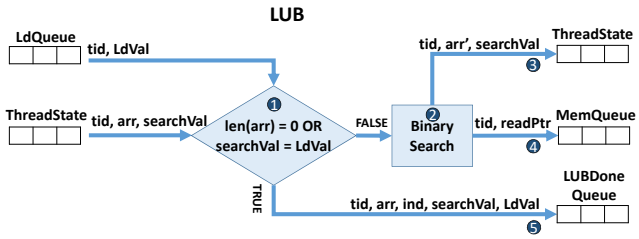

Figure 10. The logical flow of *LUB*: ❶ If array is empty or match found, ❺ return current result. ❷ Otherwise, Update array range (namely *arr'*), read middle of array from memory (❹) and store current state in local Thread Store (❸).

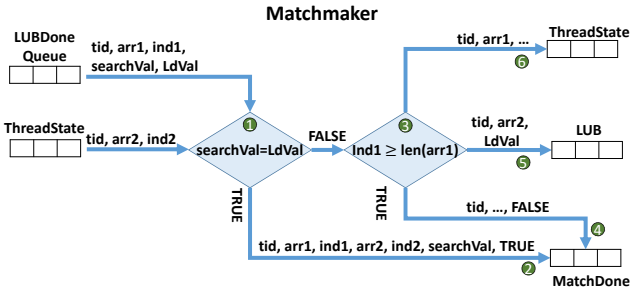

Figure 11. The logical flow of *MatchMaker*: ❶ If *match* found return True (❷). ❸ Otherwise, check if *index1* in range. If not, return False (❹). Otherwise, look for lowest upper bound in *arr2* (❺) and store thread state in the local Thread Store (❻).

the same partial join result can be accessed from different paths. If a partial join result is available in the cache, it will be used by the querying thread. However, if the entry is still in the insertion buffer, two threads from different paths might try to append values to the same entry. To avoid this race, the insertion cache uses all the values leading to the key to validate that the values are stored from just one path. For example, when looking on the dynamic MT in Figure 9, we can see that *tid1* computes the results for $(x = 1, y = 2)$, while *tid2* looks for the results of $(x = 2, y = 2)$. If the results for $y = 2$ are not in the PJR cache, *tid2* will compute the results and store them in the insertion buffer. if $y = 2$ is in the insertion buffer, and the value of x for *tid2* is different, then the results of *tid2* will not be stored in the PJR.

The last caching challenge is how to determine that an entry is fully analyzed. With dynamic MT, multiple threads can work on the same cache entry in parallel. New threads can even be created on the fly to help analyze the cache entry. To avoid race conditions we add a thread counter to each entry that maintains the number of threads currently working on the entry. Each thread that is involved with a cached entry notifies the cache of its allocation or deallocation to update the count. Once the count reaches zero, the cached entry analysis is done and the entry is copied to the PJR cache.

3.7 The TrieJax components

We now turn to describe the microarchitecture and internal flow in each TrieJax component.

LUB. This component searches for a value in a sorted array using binary search. If the value does not exist, it returns the lowest upper bound. This component has a load (LD) unit to communicate with the memory. Figure 10 presents the logical flow of *LUB*. Since most of the read operations issued by TrieJax are part of binary search operations, encapsulating the binary search logic in the specialized *LUB* component allows us to duplicate the component and generate multiple memory accesses that look for matches on different sub-arrays concurrently.

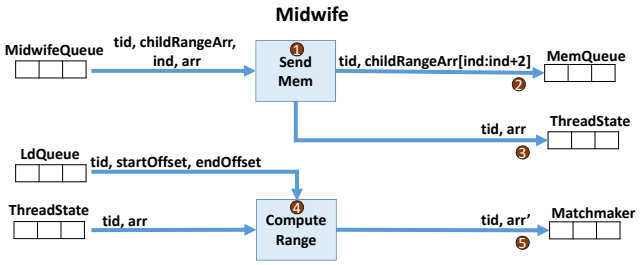

Figure 12. The logical flow of *Midwife*: ❶ Midwife receives the child range array of a parameter in the trie and a (matched value) index. ❷ It reads from memory the start and end offsets of the next parameter array that fits the (matched) value. Next, ❹ it computes the next parameter array range and sends it to *MatchMaker* (❺).

MatchMaker. This component implements the leapfrog join algorithm described in Section 2.2 for one join variable. Figure 11 presents the logical flow of *MatchMaker* and the queues used to communicate with *LUB* and *Cupid*. Given two array ranges, this unit communicates with *LUB* to find the first value that is contained in both arrays. If no such value is found, the component returns false.

Midwife. This component is used to extract the child nodes of a value node in the trie data structure, following the logic presented in Figure 12. The component computes the memory range in which the child nodes of a given parent node are stored. The inputs to the unit are a pointer to a child node array and a parent index. For example, given the *Rx child ranges* array in Figure 7 and the index 0 of *Rx*, *Midwife* extracts the range $[0, 2)$ and returns the $Ry[0:2]$ range (start and end pointers) to *MatchMaker*. The LD unit is used to access the children array. This component is duplicated to allow the extraction of two child node ranges in parallel.

Cupid. This unit manages the execution of the full join query. Figure 13 shows the single threaded logical flow of

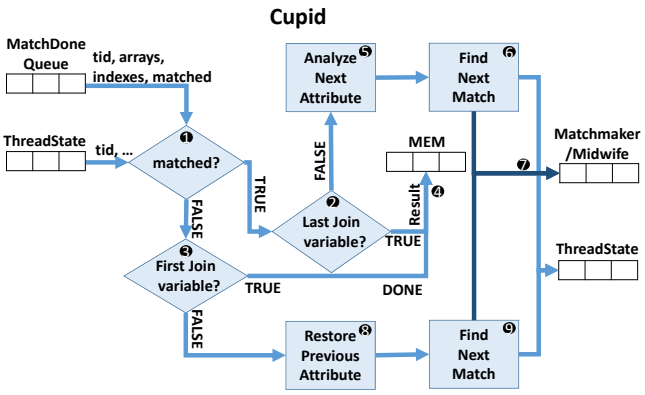

Figure 13. The logical flow of *Cupid* (single thread, no cache): ❶ If a match is found for the current join variable, ❷ check if it is the last join variable. If so, save the result to memory (❹). If not, save the arrays, matched indexes and value of the current join variable, and extract the indexes of the next join attribute from the query structure (❺). If the join variable is in the first trie level, send it directly to *MatchMaker*. Otherwise, use *Midwife* to extract the parameter values (❻). If a match was not found, check if it is the first join variable (❸). If so, ❹ write a *DONE* token to memory. Otherwise, ❽ restore the previous join variable data and ❾ look for the next match similarly to ❻

Cupid. For each join attribute k, *Cupid* utilizes the *MatchMaker* to look for the next match in all the indexes that contain k. If a match is found, it updates the result buffer and moves to the next join attribute. Otherwise, *Cupid* backtracks and looks for the next match for the previous join attribute. Finally, the unit uses its store (ST) unit to save the final results to memory. This component is also in charge of the thread management and partial join results caching. More information can be found later in this section.

3.8 TrieJax local memory system

There are three local memory stores in TrieJax. The first is the PJR cache, which is accessed by *Cupid*. The second is a constant read-only memory that stores the query structure and cache structure described in Section 3. Finally, each component holds a small local memory for maintaining the thread state. For example, *LUB* units store the searched value and array range before sending the request to memory. Once the request is returned, the thread information is read from the *LUB* thread store and continues according to the result. The local memory stores temporary partial join results or internal data that do not reach the main memory. Thus, no coherence is required for the local memory.

We use SRAM for the local memory stores. The PJR cache is the biggest store, amounting to 4 MB (for brevity, we do not present the full design space exploration for the cache size). This store uses 4 banks to allow fast concurrent accesses. The *Cupid* thread store is the second biggest amounting to 16 KB, while the remaining stores containing less than 512 B. It supports 32 threads, a configuration that offers the best performance/storage as examined in Section 4.2.

4 TrieJax Evaluation

In this section, we present our evaluation of TrieJax when executing graph pattern queries and compare its performance and power benefits to four state-of-the-art baselines: Cached Trie Join (CTJ) [16] and EmptyHeaded [1] are software systems, and Graphicionado [11] and Q100 [36] are hardware accelerators. Our evaluation focuses on the following core questions:

- What is the effect of the multi-threading on the performance of TrieJax?
- What is the performance of TrieJax compared to the baselines?
- Where does the speedup come from?
- What is the power consumption of TrieJax compared to the baselines?

4.1 Methodology

TrieJax. We implemented all the TrieJax building blocks using PyRTL [7]. We then used Cadence Innovus and Design Vision in tandem with the OpenCell 45nm design library to synthesize and place&route the Verilog code generated by PyRTL. The design achieves a fixed frequency of 2.38GHz (critical path of 0.42ns), and the results are for dynamic multi-threading with 32 threads (unless otherwise noted).

The timing and power figures obtained by the physical design were used to drive a cycle-accurate simulator of TrieJax. The simulator models all micro-architectural components described in Section 3. We integrated the simulator with Ramulator [17] to obtain accurate performance of the memory system. The DRAM energy is simulated with DRAMPower [6] using the memory traces from Ramulator. The performance and power of the on-chip SRAM memory were simulated using Cacti 6.5 [13]. The TrieJax uses a default PJR cache size of 4 MB, which includes the insertion buffer. The off-chip memory is simulated as 64GB of DDR3_1600 DRAM with two channels. The total TrieJax core area is estimated to be $5.31mm^2$ in a 14nm process (based on standard scaling factors from the 45nm cell library we used in our flow). It includes the L1/L2 caches and the 4MB SRAM scratchpad used for storing partial join results. The scratchpad is responsible for 72% of the accelerator core area. For comparison, based on die shots, the area of an Intel SkyLake server core (14nm) is close to $15mm^2$.

Baselines. We compare TrieJax to the Q100 and Graphicionado hardware accelerators and to CTJ and Emptyheaded software systems (these baselines are discussed in detail in Section 2.1). Since we do not have the original code for the hardware accelerators, we estimate the performance of Q100

	TrieJax	Software frameworks
Processing Unit	TrieJax core @ 2.38GHz PRJ 4MB SRAM	16 × Xeon E5-2630 v3 cores @ 2.4GHz
On-chip memory	L1D ReadOnly 32KB 8-way L2 ReadOnly 32KB 8-way L3 20MB	L1I/L1D 32KB/core 8-way L2 512KB/core 8-way L3 40MB
Off-chip memory	4 × DDR3-1600 2 × 12.8GB/s channels	4 × DDR3-2133 2 × 17GB/s channels

Table 3. Experimental configuration for TrieJax and the software baselines

Dataset	#Nodes	#Edges	Category
ca-GrQc (grqc)	5,242	14,496	Collabor.
soc-sign-bitcoin-alpha (bitcoin)	3,783	24,186	Bitcoin
p2p-Gnutella04 (gnu04)	10,876	39,994	P2P
ego-Facebook (facebook)	4,039	88,234	Social
wiki-Vote (wiki)	7,115	103,689	Social
p2p-Gnutella31 (gnu31)	62,586	147,892	P2P

Table 4. Dataset statistics

and Graphicionado (marked with a star) based on their original software baselines, MonetDB and GraphMat, respectively. We configured the original software baselines as described in the original Q100 and Graphicionado papers and scale the baselines' results given the best speedup and energy improvement reported in the papers. We believe this methodology provides a comparison that is favorable to both Q100 and Graphicionado.

In addition, we use CTJ and EmptyHeaded as highly tuned WCOJ software baselines. To achieve a comparison that is favorable to the baselines, each system is run 3 times and the minimum value is reported.

To evaluate the power consumption of all software systems, we measured the power of the Package and DRAM using the Intel Running Average Power Limit (RAPL) meters [8]. We sample the energy consumption during the benchmark and deduct idle energy consumption measured on the same machine for the same amount of time. For Q100 and Graphicionado, whose papers did not report the DRAM energy, we estimate the memory energy consumption by dividing the DRAM energy consumption of the respective baseline by the accelerator speedup. Because the accelerators use similar algorithms to their baselines, our scaling reduces the idle energy consumption of the DRAM avoided by the speedup.

Finally, our experimental platform is a Supermicro 2028R-E1CR24N server with two Intel Xeon E5-2630 v3 processors running at 2.4 GHz, 64GB of DDR3 DRAM with two channels, and is running a stock Ubuntu 16.04 Linux.

Datasets and queries. We test five query patterns (listed in Table 1) on 6 different datasets. We focus on common graph pattern matching queries. Testing on linear algorithms returned comparable results to the baselines, which follows the worst-case complexity of the underlying algorithm. We leave the integration of other operators, such as SORT, to future work which can be done similarly to Q100. Our datasets are real-world graphs taken from Stanford SNAP [19]. Table 4 presents the different datasets. Due to the polynomial complexity of our queries, the runtime of the join query drastically increases on bigger datasets. Therefore, we only use datasets with simulation time smaller than five days.

4.2 Performance

Main memory accesses. Figure 14 shows the number of main memory accesses for each baseline on a logarithmic scale. The figure clearly shows that the WCOJ algorithms, namely EmptyHeaded and CTJ, generate fewer accesses to main memory than the traditional approaches, namely MonetDB and GraphMat. On average, CTJ generates 2.8× fewer memory accesses than EmptyHeaded, 47× fewer than GraphMat, and 105× fewer than MonetDB. This matches our motivation for using a WCOJ-based solution for our system.

Impact of multithreading on performance. Figure 16 shows how using a different number of internal threads affects its performance. By using only 8 internal threads, TrieJax's performance is improved by 5.8×, on average, compared to single-threaded implementation. Similarly, running 32 internal threads improves the average performance speedup by 10.8× over the single thread version. Using 64 threads, however, has a minor effect on the performance. We, therefore, choose to use 32 internal threads in our benchmarks.

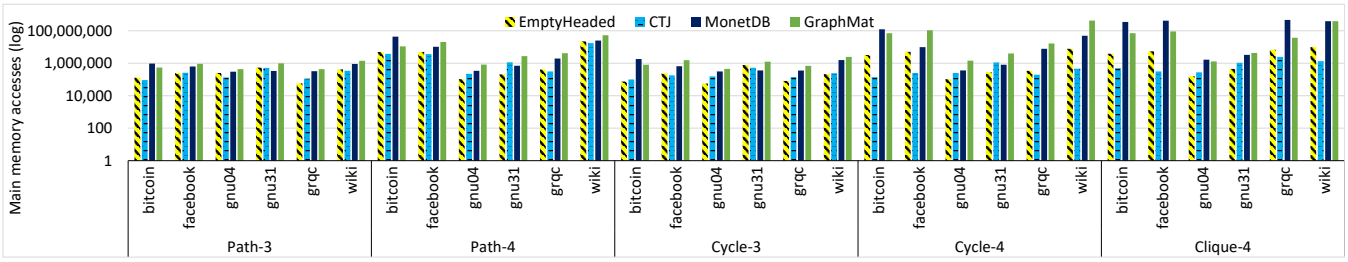

Figure 14. Number of memory accesses (log scale) for each baseline.

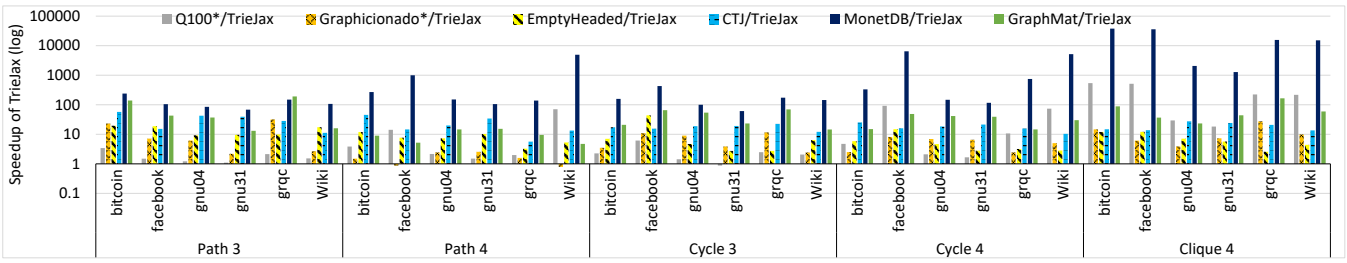

Figure 15. TrieJax performance speedup compared to the baselines.

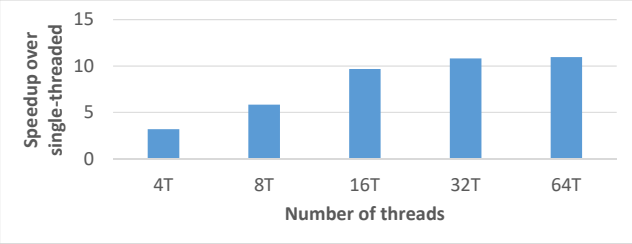

Figure 16. Performance speedup of TrieJax limited to number of dynamic threads compared to single-threaded TrieJax

Performance comparison. Figure 15 shows the speedup of TrieJax compared to the four baselines (log scaled) for the queries listed in Table 1 and datasets listed in Table 4.

TrieJax consistently outperforms the software baselines. Compared to CTJ, TrieJax achieves a speedup of 5.5 – 45× and 20× on average. In comparison to EmptyHeaded, which uses a highly parallel WCOJ algorithm with SIMD operations, TrieJax reaches a 2.5 – 44× speedup and 9× on average.

Notably, TrieJax also delivers substantial speedups compared to the hardware accelerators. TrieJax is 7× faster than Graphicionado*, on average, ranging between 0.8 – 32×. The speedup is because TrieJax avoids a large number of unnecessary intermediate results. These intermediate results take the form of messages being passed between the different graph nodes in Graphicionado*.

While TrieJax offers a considerable speedup on average, Graphicionado* was able to perform faster on the *Path-4* wiki and *Path-4* Facebook queries. In these queries, Graphicionado* outperforms TrieJax by up to 1.25×. The reason for this minor slowdown is that these queries generate a large number of results, and the TrieJax memory system becomes a bottleneck. Nevertheless, these slowdowns may be artifacts of our optimistic estimation of Graphicionado, which does not limit its memory bandwidth.

Finally, TrieJax outperforms Q100* by 63×, on average, ranging from 0.9 – 539×. This is mostly due to the inherent inefficiency of the join algorithm of Q100 that generates a large number of intermediate results. While the Q100* performance on the *Path-3* query is comparable to TrieJax for most datasets, TrieJax outperforms Q100* by up to 539× on complex queries such as *Clique4*. Q100* is also outperformed by Graphicionado*, which is aimed for graph operations and offer better parallelism and sharing of data than Q100* for large queries such as *Cycle-4* and *Clique-4*.

To summarize, TrieJax delivers dramatic speedups over both software and hardware baselines. Thanks to its internal design that reduces the number of intermediate results, TrieJax is able to serve most of its data from its fast SRAM caches and minimizes its DRAM accesses. Finally, TrieJax's aggressive use of multithreading allows it to hide the latency incurred by accessing both internal and external memories.

Where does the speedup come from? The performance of different queries benefits from different architectural and algorithmic elements of TrieJax architecture. Cyclic and clique queries primarily benefit from the reduced number of memory writes guaranteed by the worst-case optimal property of the algorithm, which does not benefit path queries. For example, Section 2 illustrates the AGM bound on a Cycle-3 query. The illustration shows that the number of final results is bound by $N^{\frac{3}{2}}$, while traditional join algorithms might generate additional N^2 intermediate results. The algorithmic component of TrieJax thus dramatically reduces the number of intermediate results. Notably, TrieJax benefits clique queries only because of this bound on intermediate results, since cliques cannot be decomposed by the CTJ compiler and do not benefit from caching partial queries.

For the non-clique queries, we explore the performance benefits of the architectural elements described in Section 3, namely the PJR cache and bypassing the caches when writing the final results (denoted $-bypass). We evaluate the architectural optimizations over a baseline system in which the TrieJax accelerator is connected to a standard 3-level (x86-like) cache system. The performance impact of the PJR cache is evaluated by comparing the baseline system to a system in which the algorithmic cache is stored locally in the PJR cache and not cached in the standard caches (PJR mode). We then compare the baseline system to one in which the final results are written directly to memory ($-bypass mode). Finally, we compare the baseline to a system in which both optimizations are enabled ($-bypass & PJR mode).

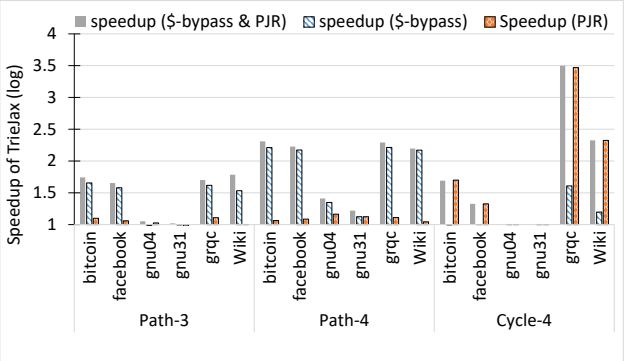

Figure 17. The effect of the architectural solutions of TrieJax on the speedup. cycle-3 and clique-4 queries are not shown as they are not affected by the custom cache system.

Figure 17 presents the performance improvements obtained by each of the architectural optimizations. The figure shows that the combination of PJR and $-bypass delivers an average speedup of 1.75× (up to 3.5×). The smallest effect is observed in the GNU datasets and is attributed to the sparseness of these datasets' graphs and the small number of final results. The $-bypass mechanism mostly affects path queries. This is due to the huge amount of the final results these queries generate, which thrash the caches if these are not bypassed. Specifically, cache bypassing delivers an average speedup of 1.47× (up to 2.2×) for these workloads. In contrast, the PJR cache is mostly beneficial for cyclic queries. In these queries, the algorithmic caching of partial joins is highly effective. Mapping partial results to the dedicated PJR cache yields fast access to more cached interim results. The PJR cache delivers an average speedup of 1.32× (up to 3.5×).

In summary, we see that the different components benefit different types of queries and complement each other. While acyclic queries profit mostly from the bypass of the memory caches, the cyclic queries leverage the PJR cache to efficiently cache partial join results. Furthermore, the clique queries benefit from the algorithmic bound to the number of intermediate results. Notably, the remainder of the speedup is attributed to the efficient memory usage of the underlying algorithmic approach and its inherent parallelism. Thus, by combining the algorithmic and architecture design TrieJax benefits different types of queries.

4.3 Energy efficiency

In order to understand the potential energy efficiency of TrieJax, we first explore how the accelerator's energy consumption is distributed among its components.

Figure 18 presents the energy distribution in TrieJax for the examined queries, averaged over the different datasets. The figure clearly shows that the lion's share of the energy consumed by TrieJax goes to the memory system and only a fraction of the energy is consumed by the TrieJax core logic.

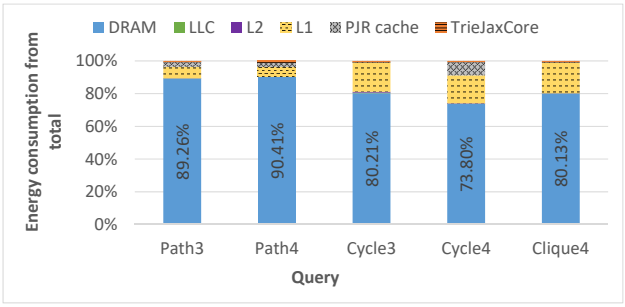

Figure 18. Average energy consumption distribution of TrieJax for each query. Note that the energy consumption is completely dominated by the memory system.

The dominant memory component is DRAM, which consumes 74–90% of the overall energy. DRAM energy consumption is attributed to explicit read/write operations (mostly writing the final results to memory) and to implicit DRAM idle power (predominantly DRAM refresh operations). The second dominant energy consumer is the L1 cache. Although the cache is read-only and is not involved in writing the final results to memory, it serves the trie traversals triggered by misses in the partial join results cache. Finally, the PJR cache is responsible for up to 7.8% of the total energy consumption for the *cycle4* query. However, for *Cycle3* and *Clique4* queries, there are no valid intermediate result caches and it does no use any energy.

We now turn to Figure 19 which compares the energy consumption of TrieJax versus that of the baseline systems. Importantly, understanding that DRAM is by far the main energy consumer in TrieJax enables us to explain its reduced energy consumption compared to the baselines.

When comparing TrieJax to the software baselines, it is not surprising to find that TrieJax much more energy efficient. TrieJax is 110× more energy efficient on average than CTJ and 59× more efficient than EmptyHeaded. This is attributed to three factors. The first factor contributing the TrieJax's energy efficiency is the reduced energy consumption of its core logic, which is specifically designed to execute join operations and is thus much more energy efficient than a general-purpose core. This factor is slightly less dominant in the case of EmptyHeaded (which is almost 2× more efficient than CTJ), since EmptyHeaded uses efficient CPU SIMD operations. The second contributing factor is the efficient TrieJax's on-die caching of intermediate results and reduced thrashing of the memory system caches, which dramatically reduces the energy consumption of its memory system. The final contributing factor is the speedups obtained by TrieJax, which dramatically reduce the DRAM's idle energy consumption.

TrieJax is also much more energy efficient than other hardware accelerators. Specifically, TrieJax consumes, on average, 179× and 15× less energy than Q100* and Graphicionado*,

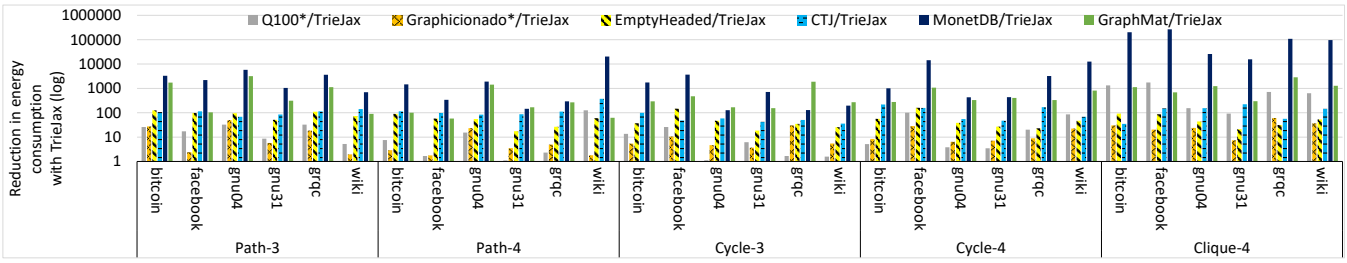

Figure 19. Reduction in energy consumption obtained with TrieJax compared to the baselines.

respectively. This improved energy efficiency is attributed to two factors. First, the use of a WCOJ algorithm that reduces the number of intermediate results and eliminates most of the expensive DRAM accesses executed by Q100* and Graphicionado*. Second, the faster run time of TrieJax dramatically reduces the idle energy consumption of the DRAM, which is mostly the result of periodic DRAM refresh operations.

To summarize, we see that the energy efficiency of graph and database hardware accelerators is bounded by their memory system. Primarily, the number of explicit read/write operations affects both energy consumption and performance. But the performance impact of the memory system also increases DRAM's idle energy consumption. TrieJax's caching of internal results thus minimizes its reliance on the memory system, thereby reducing both its active memory energy (read/write) and idle DRAM energy (implicit refresh).

5 Conclusions

We presented TrieJax, an on-chip domain-specific accelerator for graph operations specializing in graph pattern matching. It is driven by new advances in the database community and a plethora of new join algorithms that outperform traditional approaches. TrieJax leverages the inherent concurrency of the algorithm for parallel execution and latency hiding of irregular memory accesses. Furthermore, it integrates a specialized store for intermediate results that can drastically reduce recurrent computations.

We showed that TrieJax outperforms state-of-the-art graph analytics and database accelerators by 7 – 63× on average, while consuming 15 – 179× less energy. TrieJax further outperforms RDBMS that use the modern join algorithms by 9 – 20× and consumes 59 – 110× less energy.

We plan to extend our accelerator to other important graph operations such as aggregations (e.g., triangle counting), and use novel algorithmic approaches to offer approximate estimations in a fraction of the time.

Acknowledgments. Oren Kalinsky is supported by the Hasso Plattner Institute. Yoav Etsion is supported by the Israel Science Foundation (ISF grant 979/17).

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Intl. Conf. on Management of Data (SIGMOD)*. https://doi.org/10.1145/2882903.2915213

[2] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *ACM Symposium on Principles of Database Systems (PODS)*. https://doi.org/10.1145/2902251.2902280

[3] ARM. [n. d.]. ARM ISA. https://developer.arm.com/architectures/instruction-sets.

[4] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440

[5] Rajesh Bordawekar, Tirthankar Lahiri, Bugra Gedik, and Christian A. Lang (Eds.). 2014. *ADMS*. http://www.adms-conf.org/adms_2014.html

[6] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. DRAMPower: Open-source DRAM power & energy estimation tool. http://www.drampower.info.

[7] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *Intl. Conf. on Field Programmable Logic and Applications (FPL)*. https://doi.org/10.23919/FPL.2017.8056860

[8] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *Intl. Symp. on Low power Electronics and Design (ISLPED)*. https://doi.org/10.1145/1840845.1840883

[9] David J DeWitt and Robert Gerber. 1985. *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department.

[10] Brian Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. 2005. Accelerating Database Operators Using a Network Processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DaMoN '05)*. ACM, New York, NY, USA, Article 1. https://doi.org/10.1145/1114252.1114260

[11] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics. In *Intl. Symp. on Microarchitecture (MICRO)*. http://dl.acm.org/citation.cfm?id=3195638.3195707

[12] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*. https://doi.org/10.1145/2150976.2151013

[13] HP Labs. [n. d.]. CACTI. https://www.hpl.hp.com/research/cacti/.

[14] Trey Ideker, Owen Ozier, Benno Schwikowski, and Andrew F. Siegel. 2002. Discovering regulatory and signalling circuits in molecular interaction networks. In *Intl. Conf. on Intelligent Systems for Molecular Biology*.

[15] Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24, 1 (Jan. 1977), 1–13. https://doi.org/10.1145/321992.321993

[16] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible Caching in Trie Joins. In *Intl. Conf. on Extending Database Technology (EDBT)*. 282–293. https://doi.org/10.5441/002/edbt.2017.26

[17] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *Computer Architecture Letters* 15, 1 (2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[18] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[20] Priti Mishra and Margaret H Eich. 1992. Join processing in relational databases. *ACM Computing Surveys (CSUR)* 24, 1 (1992), 63–113.

[21] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sánchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Intl. Symp. on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2018.00010

[22] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *PODS*. 234–245. https://doi.org/10.1145/2594538.2594547

[23] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *ACM Symposium on Principles of Database Systems (PODS)*. https://doi.org/10.1145/2213556.2213565

[24] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[25] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM Symp. on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/2517349.2522739

[26] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2015. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *3rd. Intl. Workshop on Graph Data Management Experiences and Systems (GRADES)*. https://doi.org/10.1145/2764947.2764948

[27] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. https://doi.org/10.1109/FCCM.2014.15

[28] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[29] RISC-V. [n. d.]. RISC-V ISA. https://riscv.org/risc-v-isa/.

[30] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 420–431. https://doi.org/10.1145/3186728.3164139

[31] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. 2006. The semantic web revisited. *IEEE intelligent systems* 21, 3 (2006), 96–101.

[32] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *Intl. Conf. on Very Large Data Bases (VLDB)*. http://dl.acm.org/citation.cfm?id=1083592.1083658

[33] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[34] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106. https://doi.org/10.5441/002/icdt.2014.13

[35] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. 2009. User Interactions in Social Networks and Their Implications. In *EuroSys*. https://doi.org/10.1145/1519065.1519089

[36] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*. https://doi.org/10.1145/2541940.2541961