A Hardware Implementation of the Leapfrog Triejoin

by

Shruti Siva

S. B. Electrical Engineering and Computer Science and Physics,
Massachusetts Institute of Technology, 2026.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

Authored by: Shruti Siva
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Adam Chlipala
Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Hardware Implementation of the Leapfrog Triejoin

by

Shruti Siva

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Datalog has the potential to be an effective intermediate representation for a range of domain-specific languages due to its natural expression of relational computations and its amenability to parallelism. The performance of any Datalog engine, however, ultimately depends on the efficiency of its join kernel.

This thesis presents a design for a hardware accelerator for the multi-way database join that comprises the bulk of Datalog operations, based on the Leapfrog Triejoin, a worst-case optimal join algorithm. The design exploits two sources of parallelism absent from software implementations thus far: a pool of processing elements that simultaneously execute independent subtrees of the join and a distributed multi-unit architecture in which an arbiter dispatches tasks to units based on data locality. RTL simulation demonstrates a cycle efficiency substantially higher than normalized CPU baselines for the same algorithm, suggesting that the design will be competitive with and likely exceed existing software implementations once brought to hardware synthesis.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

I would like to thank Adam Chlipala for taking a chance on me last year, letting me work on what has turned out to be a dream project, challenging me when I got too into the weeds of an idea that was not going to pan out, and having faith that at some point, at least one idea was going to pan out. I would also like to thank Adam Hartz for introducing me to 6.101. His steadfastness and commitment to the class inspires me to put just as much care into everything I do. In addition, I would like to thank Joe Steinmeyer and Sam Coday for also being on my Mount Rushmore list of instructors.

I would like to thank my parents for instilling in me a passion for learning and feeding me lots of delicious food when I visit home. I would also like to thank my sister, who is the light of my life, for being a source of creativity that inspires me to think through any problem from a different angle. I'm also grateful to the broader MIT community for being my home these past four years, and I would like to thank the friends, colleagues, and South Asian A Cappella groups without whom this thesis would not be possible (and without whom the person writing this thesis would have been very miserable).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Datalog is a logic-programming language that presents a practical core for a wide range of domain-specific languages, due to its declarative style; simple, but expressive, rules-based core; and potential for parallelism. It is the substrate for analyses in topics ranging from program analysis [1] to network verification [2]. At the heart of Datalog evaluation lies a small set of graph operations, dominated by multiway set intersections. As datasets and rules grow in complexity, these joins become the performance and energy bottleneck. Recent work has shown that worst-case optimal join algorithms such as the Leapfrog Triejoin (LFTJ) [3] can significantly improve asymptotic and practical performance. Essentially, these operations involve repeatedly intersecting single-variable projections of each relation, which leaves ample opportunities for parallelism. Despite these theoretical properties there is a latency bound to a software implementation of the LFTJ. This thesis asks whether the structure of the LFTJ maps naturally onto hardware and whether doing so can close the gap between the algorithm's theoretical efficiency and its practical performance. We make the following contributions:

1. A hardware implementation of the Leapfrog Triejoin. We design and implement in Bluespec SystemVerilog an accelerator that executes the full LFTJ algorithm, with key modifications to maximize parallelism.

2. Parallel DFS execution with a pool of PEs. Rather than executing each projection-based join on a single thread, the design proceeds in a DFS-style tree and maintains a pool of processing elements that can each progress from independent subtrees.

3. A distributed multi-unit architecture and memory hierarchy. Multiple accelerator units are coordinated by an arbiter that uses memory to route computation.

4. Experimental evaluation. We evaluate the design on five benchmark queries spanning a range of complexity. RTL simulation demonstrates 25-37 cycles per result on the lowest level of configurable parallelism, or a 3–5× improvement over current CPU and GPU implementations of the algorithm.

This thesis fits into a larger effort to build a high-performance Datalog compiler and accelerator. A highly optimized hardware backend for Datalog centered around this parallelized join operation allows many DSL compilers, and users, to reap the benefits of performance.

Chapter 2

Background

The accelerator described in this thesis sits at the intersection of three distinct areas of subject matter: the Datalog language, worst-case optimal join processing, and the hardware considerations of a distributed accelerator. This chapter establishes the relevant background in each area.

2.1 Datalog

Datalog is a declarative logic-programming language for expressing recursive queries over relational data [4]. A Datalog program consists of a set of **rules** of the form:

$$H(\mathbf{x}) :- B_1(\mathbf{x}_1),\ B_2(\mathbf{x}_2),\ \ldots,\ B_k(\mathbf{x}_k) \tag{2.1}$$

where h is the **conclusion** relation and $B_1, \ldots, B_k$ are the **hypothesis** relations. Each $\mathbf{x}_i$ is a tuple of variables and constants. A conclusion tuple is derived from any binding of variables that satisfies all hypothesis relations simultaneously.

The input facts of a Datalog program are called the **extensional database** (EDB). Facts derived from applying rules are called the **intensional database** (IDB). EDB relations are fixed and provided by the user, while IDB relations are computed by the engine.

Example. The following pair of rules computes reachability over a directed graph:

$$\mathit{Reach}(x, y) :- \mathit{Edge}(x, y) \tag{2.2}$$
$$\mathit{Reach}(x, z) :- \mathit{Edge}(x, y),\ \mathit{Reach}(y, z) \tag{2.3}$$

Edge is the EDB relation, while *Reach* is the IDB relation. Rule (2.2) states that any direct edge constitutes a reachability pair. Rule (2.3) states that reachability is transitive. Together they compute all reachable pairs in the graph.

Fixpoint semantics. The meaning of a Datalog program is its **least fixpoint**, or the smallest set of IDB facts closed under all rules [5]. Starting from the EDB, rules are applied repeatedly until no new facts can be derived. This process is guaranteed to terminate because

the domain is finite and IDB facts are never retracted, so the IDB grows monotonically toward the fixpoint.

Modern Datalog engines, including Soufflé [4] and LogicBlox [2], and interpreters of the related language Prolog use **bottom-up evaluation**: the engine begins with the EDB and derives new facts forward until fixpoint. This contrasts with **top-down evaluation**, which begins from a query goal and works backward through rules to find supporting facts. Bottom-up evaluation lends itself to an implementation that can exploit the parallelism of deriving multiple tuples satisfying a rule, or evaluating multiple rules, simultaneously.

Applications. Datalog's combination of recursion, declarative syntax, and clean fixpoint semantics has made it the foundation for a wide range of practical systems. In program analysis, the Doop framework [1] expresses pointer analyses as Datalog programs. Commercial systems such as LogicBlox [2] have demonstrated the viability of Datalog as an enterprise database engine. Each of these applications share the bottleneck of Datalog: multiway joins over large relations dominate the cost of evaluation. An optimized backend for performing this join in a Datalog engine allows all upstream applications to reap the benefits of improved performance.

2.2 Fixpoint Evaluation

Naive evaluation. The simplest approach to computing the Datalog fixpoint is to apply every rule over the full current database at each iteration and add any newly derived facts, repeating until nothing changes. This approach is correct but wasteful, because every iteration re-derives all facts in addition to any new ones and performs many redundant reads to memory. For a recursive rule over a relation of size n, the cost per iteration is $O(n^k)$ for a k-body rule, most of which is rederivation.

Semi-naive evaluation. Semi-naive evaluation [4,6] avoids this waste by observing that a new fact can only be derived at iteration k if at least one of the hypothesis relations contributes a fact that was itself new at iteration $k-1$. The engine therefore maintains a **differential** $\Delta R^{(k)} = R^{(k)} - R^{(k-1)}$ for each IDB relation R, containing only the facts newly derived at the most recent iteration. Rather than evaluating the join over the full hypothesis relations, the engine substitutes the differential in place of the full relation for each hypothesis relation in turn, unions the result, and deduplicates against the current full relation to produce the next differential. The full relation grows by absorbing the differential at the end of each iteration. Since the cost per iteration is proportional to the size of the differential, rather than the full relation, and since the differential shrinks as the computation approaches a fixpoint, the total work across all iterations is substantially less than naive evaluation. We refer to the full accumulated relation as the **FULL** set and the differential at each iteration as the **DELTA** set. These terms are used throughout Chapter 7.

A correctness requirement is that deduplication must check new facts against the full accumulated relation. The mechanism by which this check is implemented depends on the underlying data structure and is discussed in Chapter 7.

The leading software implementation of Datalog with semi-naive evaluation is Soufflé [4,7], which compiles Datalog programs to parallel C++ and achieves strong performance on program-analysis workloads at scale. Soufflé represents each relation as a B-tree, which supports $O(\log n)$ amortized insertion with stable iterators, allowing new facts to be inserted directly into the full relation without invalidating any in-progress iterator.

2.3 Worst-Case Optimal Join Algorithms

A **full conjunctive query** is a join over k relations of the form considered throughout this thesis. Given relations of size at most n, the maximum possible result size is bounded by the **AGM bound** [8], which comes from the fractional edge cover of the query's hypergraph. For the triangle query $Q(a, b, c) :- R(a, b), S(b, c), T(a, c)$ with $|R| = |S| = |T| = n$, the AGM bound gives $|Q| \leq n^{3/2}$.

A join algorithm is **worst-case optimal** if its running time is proportional to the AGM bound on the output size. Several worst-case optimal algorithms have been developed, beginning with NPRR [9], which achieves $O(n^{\rho^*})$ time, where ρ^* is the fractional edge cover number. The practical algorithm used in this thesis, Leapfrog Triejoin, achieves the same bound up to a log factor while being substantially simpler to implement.

2.3.1 Leapfrog Triejoin

Leapfrog Triejoin (LFTJ) [3] is a join algorithm that evaluates full conjunctive queries by binding one variable at a time in a canonical ordering. For each variable, it computes the intersection of the sorted sets of values that satisfy all hypothesis relations containing that variable, given the values already bound at previous levels, without materializing any intermediate results. The algorithm requires relations to be stored as **tries** and achieves running time $O(Q^* \log N)$, where Q^* is the AGM bound on the result size and N is the size of the largest input relation [3], establishing worst-case optimality up to a log factor. The design and hardware implementation of the trie representation, the intersection algortihm, and the parallel execution model are the subjects of Chapters 3 and 4.

2.4 Hardware Primitives

2.4.1 FPGAs

A field-programmable gate array (FPGA) is a reconfigurable integrated circuit consisting of look-up tables (LUTs) that implement combinational logic and flip-flops that store state between clock cycles. The performance and resource cost of a design are measured primarily in LUT count, flip-flop count, and the number of clock cycles taken to complete execution.

FPGAs are well-suited to parallelizable workloads with regular memory-access patterns, because many independent modules can be instantiated and execute in parallel, and there are many memory interfaces available to experiment with. They are thus a natural platform for the join accelerator designed in this thesis, where the bottleneck is typically memory bandwidth and latency.

2.4.2 Bluespec SystemVerilog

Bluespec SystemVerilog (BSV) [10] is a hardware description language in which all behavior is expressed as a set of rules, or atomic actions. Each rule fires when its guard condition is satisfied, atomically updating the module's state. The BSV compiler is responsible for scheduling rules that detect conflicts and creating synthesizable Verilog such that the design works at the level of rule semantics rather than clock edges. BSV's type system is derived from Haskell and supports parametric polymorphism. In this thesis, modules are parameterized by numeric types that are resolved at elaboration time.

2.4.3 Block RAM

Block RAM (BRAM) is dedicated on-chip memory available in fixed-size blocks on FPGA devices. Each BRAM block provides two independent ports, each capable of one read or write per clock cycle, with a configurable data width. A read request issued in cycle t returns data in cycle $t+1$. The cost of the seek algorithm described in Chapter 3 is easily determined by the number of access to memory required. In contrast, off-chip memory has a latency of hundreds of cycles.

On-chip BRAM capacity on a modern FPGA is typically smaller than the relations that arise in large Datalog workloads. The memory hierarchy described in this thesis attempts to bridge the gap between the predictable single-cycle read latency of BRAM and the scale of off-chip memory. Because off-chip memory is expensive to access, cache-miss avoidance becomes crucial to system performance.

Chapter 3

Trie Representation and Single Iterator Interface

The most basic building block of the Leapfrog Triejoin accelerator is the representation of a relation in memory and how a single iterator traverses that representation. Each higher-level component, including the individual Leapfrog Join element, the parallel join engine, and the distributed arbiter, is built on top of the interface defined here. In this chapter, we determine (1) how to lay out a multi-variable relation in hardware-accessible memory such that the Leapfrog Triejoin algorithm's `seek` and `next` primitives can be implemented efficiently, and (2) how to connect that memory layout to the on-chip cache hierarchy so that large relations do not require all data to reside in on-chip BRAM, which necessarily limits the size of Datalog workloads that can be performed on this accelerator.

3.1 Relations as Prefix Tries

The Leapfrog Triejoin algorithm [3] requires each relation to be stored as a **trie**, or a tree whose root-to-leaf paths encode the tuples of the relation in the order in which each variable appears in the relation. For example, a binary relation $R(x, y)$ is represented by a level-0 layer of sorted, unique x values, where each x node has children corresponding to the sorted y values that appear with that x in R. The algorithm traverses this structure in two ways: (1) advancing to the next element in sorted order (`next`) and jumping to the least element greater than or equal to a target value (`seek`). Both operations must be supported in hardware with low and predictable latency.

The standard software implementation of a trie uses heap-allocated nodes traversed by pointers. However, in hardware, successive pointer traversal creates a random access pattern in BRAM, because each pointer value is only known after the previous pointer is dereferenced, making the latency of the `seek` operation dependent on the depth of the pointer chain rather than the number of BRAM reads, which cannot be ameliorated with a wider memory bus or prefetching subsequent addresses.

We instead represent each level of the trie as a sorted, contiguous array per variable level per relation. Each element of the array is a `TrieNode`, which consists of a 32-bit node `value` and a 26-bit index into the global address space pointing to the first element of the

corresponding child array at the next level (Figure 3.1). The 26-bit index was chosen to support workloads of tens of millions of nodes. Because all children of a given parent are stored contiguously, a `seek` within a child run reduces to a search within a contiguous sorted array, which can be made efficient by binary searching, and children of a specific parent value can be treated as iterators themselves (a property that will prove useful in architecting a parallelizable Leapfrog Triejoin algorithm). The child iterator can be easily traversed from the parent by calling `configureSlice` with a start address given by the `childStart` field of a node. The `next` operation is simply an increment of the current position.

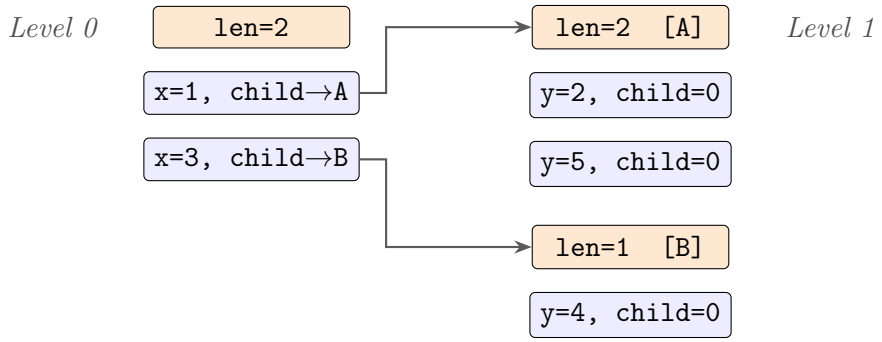


Figure 3.1: Sorted, contiguous trie layout for the relation $\{(1,2),(1,5),(3,4)\}$. Each level is a contiguous array of `TrieNodes`. Position 0 of every array is a **header** node whose `value` field stores the run length; data nodes follow at positions $1 \ldots$ `length`. The `childStart` field of each level-0 node points to the header of the corresponding level-1 subarray.

3.1.1 Physical Representation of a Node

In this current iteration of the architecture, a single `TrieNode` occupies 64 bits: 32 bits to store the `value` field and 26 bits to store the `childStart` field, with 6 bits unused. The cache-line size was chosen to be 512 bits, to fit 8 nodes per cache line. Any cache-line size that consists of two or more nodes results in a measurable speedup, due to the frequency of the `next` operation, which occurs at least once per iterator each time a joined value is found, and once each time an iterator is first traversed, to read both the header and the first piece of data at the start of the iterator.

3.1.2 Header Nodes

Position 0 of every level is reserved for a **header** node. Its `value` field stores the number of data nodes in the array, and its `childStart` field is zero. This convention allows an iterator to discover the length of any subarray by reading a single node at a predictable offset. Making memory accesses multiple nodes wide and caching on a per-line basis allows access to the first node for free when the header is read, while saving bits in the node itself for a larger width `childStart` or `value` field, if needed.

3.2 The Iterator Interface

At its core, an iterator keeps track of the current position it is being read at and the value of the iterator at that position. It increments its current position upon request from the Leapfrog Join engine. The hardware implementation of the trie iterator exposes the five-method interface required by a single Leapfrog Join:

`data()` Returns the value of the iterator at the iterator's current position.

`configureSlice(trie, start)` Points the iterator at the array beginning at global address `start` within the physical trie identified by `trie`. Resets all internal state.

`next()` Advance to the next element in the current iterator. Enqueues a `NEXT` request to an internal request queue and enqueues a response to a response queue when it is produced, which is collected by `waitResp`.

`seek(value)` Advance to the least element $\geq$ `value` in the current array. Enqueues a `SEEK` request to an internal request queue and enqueues a response to a response queue when it is produced, which is collected by `waitResp`.

`waitResp` The interface to an internal result queue that waits for a result to be returned and dequeues it. The result consists of the original operation type and flag indicating whether the result is valid. The `data()` port can be safely read at the time a valid result is dequeued.

The `next` and `seek` methods are the atomic methods that comprise the Leapfrog Join algorithm over a single iterator, and they must meet the guarantees specified in [3]. Namely, the next operation requires at most one memory access, by virtue of the contiguous memory layout. The seek operation achieves $O(\log N)$ time (where N is the cardinality of the relation) using the implementation described below. The `waitResp` method allows for a non-blocking memory interface; requests and responses are mediated by queues to decouple the internal work of the iterator from the interface that accepts new requests, allowing the join engine to issue a memory request and continue scheduling other work before polling for the result.

3.2.1 The Seek Algorithm

`seek(value)` must find the least element $\geq$ `value` in a sorted array of up to 2^{26} entries. A naive linear scan requires $O(N)$ reads, while a pure binary search requires $O(\log N)$ reads but starts from the beginning of the iterator each time. In practice, a Leapfrog Join calls the `seek` method from the middle of the iterator, and the target is known to be at or ahead of the current element. This locality motivates a two-phase algortihm.

Phase 1: Exponential search. Starting from the current position, stored in the `currPos` register, the iterator probes at addresses `currPos` $+1$, `currPos` $+2$, `currPos` $+4$ ..., doubling the stride at each step until either the probe value matches or exceeds `value` or the end of the array is reached. if the first probe already satisfies $\geq$ `value`, the seek terminates immediately.

If the end of the iterator is reached without finding a candidate value, we terminate the search with a failure. Otherwise, it is possible that we have "skipped past" the earliest value that satisfies $\geq$ `value`. If we had encountered such a value before starting the most recent probe, we would have left the exponential seek phase. Thus, the most recent stride establishes a region [`lo`, `hi`] in which the target value must lie.

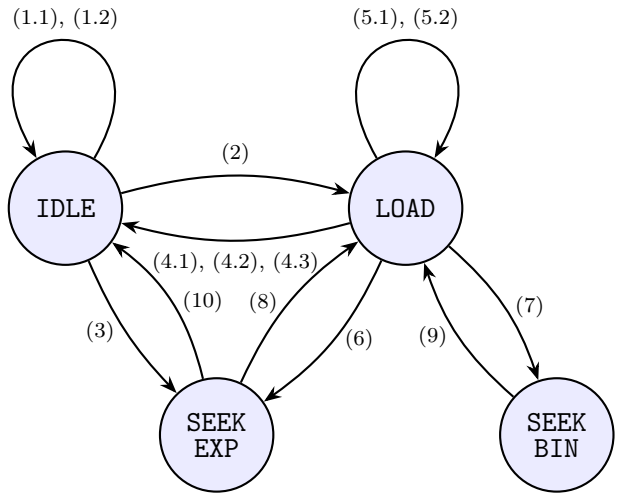
Phase 2: Binary Search. Within the bracket identified by Phase 1, a standard binary search narrows the interval to the single element that is the least upper bound of `value` by successively probing the midpoint at each step. At each probe, two outcomes are possible. If the probe value is $\geq$ `value`, the probe is a valid least upper bound candidate. The true least upper bound may still lie to the left, so the interval narrows leftward, and the probe address is saved as the best candidate seen so far. If the probe value is $<$ `value`, the true least upper bound must lie to the right, and the interval narrows rightward. If the final probe was one such rightward step, the true least upper bound is the saved candidate address rather than the current probe address, and that address must be refetched to produce the final result. The binary search resolves in $O(\log \textit{stride length})$ additional reads.

The length that must be searched per seek operation decreases with the number of seek targets in the iterator, because each iterator is of a finite size. Over a sequence of m seek operations, if the targets are presented in sorted order, as they are required to be during Leapfrog Join intersection, the total cost of all seeks is $O(m \log(N/m))$, which is worst-case optimal.

State Machine. The implementation uses four states: the `IDLE` state accepts new requests, the `LOAD` state retrieves a memory response, the `SEEK_EXP` state handles the exponential-search phase, and the `SEEK_BIN` state handles the binary-search phase. The `LOAD` state is further differentiated by the purpose of the pending read: `NEXT` (resolving a next call), `CURR` (loading the current node to check against the target before beginning exponential search or reloading a cached candidate), `EXP` (handling an exponential probe response), and `BIN` (handling a binary-search probe response). Figure 3.2 describes the transitions between these states.

3.3 Memory Virtualization

The trie iterator described in the previous sections issues memory requests using **global node addresses**: 26-bit integers that index into a logical address space of up to 2^{26} nodes shared across all tries and all variable levels. Global addresses are necessary because the `childStart` field of every `TrieNode` is a global address written into the trie at construction time. When the join engine finds a match at one variable level and descends to the next, it reads the `childStart` field of the matched node and uses it directly as the start address for the child iterator. To preserve correctness, every iterator on every unit must interpret addresses in the same uniform address space. However, depending on the size of the Datalog workload, the on-chip BRAM available to each LFTJ unit is far smaller than this space. The memory-virtualization layer sits between the iterator and the BRAM, transparently translating global addresses to local ones and fetching missing data from the global `PageMem`



(1.1) request is `seek`, current node loaded, node value $\geq$ target
(1.2) already at end of iterator
(2) current node not yet loaded
(3) request is `seek`, current node loaded, node value $<$ target
(4.1) loading `NEXT`
(4.2) loading `CURR`, loaded value $\geq$ target
(4.3) loading `EXP` or `BIN`, interval size is 1, loaded value $\geq$ target
(5.1) loading `NEXT`, header loaded and data not in two-node cache
(5.2) loading `BIN`, interval size is 1, loaded value $<$ target, reload last candidate
(6) loading `CURR` or `EXP`, loaded value $<$ target
(7) loading `EXP`, interval length > 1, loaded value $\geq$ target
(8) double interval length
(9) halve interval length, issue request for midpoint
(10) at end of iterator

Figure 3.2: State machine for `seek` and `next`.

when needed. It is composed of two modules: `SharedMem`, which manages the physical BRAM that can be read by a parameterizable number of iterators, and `PageCache`, which performs the address translation and manages a set-associative cache of recently accessed pages.

3.3.1 SharedMem: Multi-Reader On-Chip BRAM

`SharedMem` wraps a dual-port BRAM and exposes n independent reader interfaces (one per iterator participating in a given execution of a Leapfrog Join), plus a single write interface. The BRAM stores 512-bit lines of eight `TrieNode` values each, matching the node packing described in Section 3.1.

A read request specifies a local node address. The module computes the containing line address, issues a read to the dedicated read port, and extracts the node and its immediate neighbor at the requested offset, which is consumed by the two-node lookahead cache that is maintained by the iterator. There is limited benefit in passing along and caching the entire line, since the most common stride size that is serviced by the iterator is a size of

1. Requests from multiple readers are serialized through a single request queue using a round-robin arbiter.

The write interface accepts a line address and a vector of 8 `TrieNodes` and issues a write to the dedicated write port of the BRAM.

3.3.2 PageCache: Set-Associative Address Translation

`PageCache` translates the global node addresses issued by the iterators into local BRAM addresses, fetching missing pages from the arbiter when necessary. It is a set-associative cache, with a parameterizable number of sets and ways, that tracks which global pages are currently resident in the `SharedMem`'s dedicated BRAM. A **page** is a contiguous block of 2^{10} nodes, of size 8KB, in the global address space.

Address Translation. Each cache entry records the global page tag of a resident page and the set and way it occupies. On a hit, the set and way indices determine the location of the page within the truncated address space of the `SharedMem`'s memory using a simple formula: $setID \times numSets + wayID$. Adding the page offset extracted from the low bits of the global address produces the final local node address, which is forwarded to `SharedMem`.

Miss handling. On a cache miss, the page tag is enqueued to a queue for pending miss requests, along with information about the requesting reader. A reader with a pending miss request will not issue further requests until its miss is resolved, preventing out-of-order responses but allowing other parallel readers to progress. The miss tag is collected by the `LeapfrogTriejoinUnit` and forwarded to a memory arbiter, which fetches the missing page from global memory (whether it is stored in BRAM or off-chip DRAM) and streams it back as a sequence of 512-bit packets (Figure 3.3). Upon receiving a page fill, the cache might decide to evict a resident page from memory. This information is also enqueued to the `LeapfrogTriejoinUnit`, which triggers an update to the global memory arbiter's metadata.

Fetching at page granularity amortizes the cost of this round trip to global memory over many subsequent node accesses. Since child runs are stored contiguously in the trie, a page fetched on the first access to a child run will service the entire run at local BRAM latency. With 1024 nodes per page and an average child run length on the order of tens to hundreds of nodes, the miss penalty is amortized across enough accesses that the effective cost per node access approaches that of a local BRAM read.

3.3.3 Summary

Table 3.1 summarizes the asymptotic cost of each operation provided by the iterator interface in terms of BRAM reads, which are simulated as spanning a single clock cycle.

The bottleneck of each operation is the time taken to service cache-miss requests, which stalls the requesting iterator for the duration of a full page transfer from global memory. The architecture minimizes the impact of the single-iterator stall in three ways. First, the per-reader backpressure mechanism in `PageCache` ensures that other iterators may progress while a single iterator is stalled due to a miss. Second, when a unit performing the join algorithm

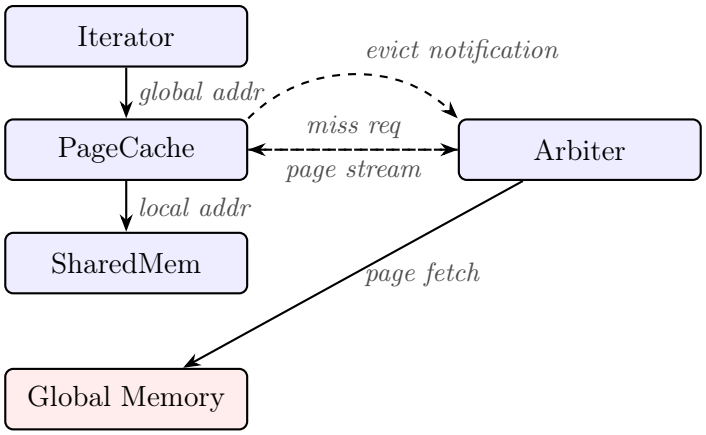


Figure 3.3: Memory-virtualization hierarchy. Processing elements issue global node addresses to the `PageCache`, which translates them to local BRAM addresses on a hit or forwards a miss request to the `Arbiter`. The arbiter feshes missing pages from global memory and streams them into the local `SharedMem`. Eviction notifications are forwarded to the arbiter to update its routing information.

detects that enough pages needed for a particular join are not resident in its local cache, it returns the task to the arbiter rather than executing it locally. The arbiter can then route the task to a unit whose cache already holds the relevant pages. This spill mechanism and the routing heuristics that support it are described in Chapter 4 and Chapter 5 respectively. Third, other units with a different local cache can continue to progress while another unit is stalled.

Taken together, the design choices in this chapter form a coherent contract between the iterator and the rest of the accelerator. The sorted-array layout enables efficient `seek` and `next` operations. The seek algorithm exploits the sorted-order access pattern to achieve $O(m \log(N/m))$ total seek cost across m operations. The two-node lookahead cache halves BRAM traffic for the sequential traversals that commonly appear during intersection. The memory-virtualization layer allows the system to reason about tries uniformly without knowing whether they are on- or off-chip.

The iterator interface itself, including `configureSlice`, `next`, `seek`, `data`, and `waitResp`, is the kernel of the only contract that the parallel join engine depends on. Everything described in this chapter is hidden behind it, and the chapters that follow treat the iterator as the atomic unit of the join.

Operation	Lookahead Cache Hit	PageCache hit	Cost
`next`	Yes	—	0 BRAM reads
`next`	No	Yes	1 BRAM read
`next`	No	No	1 BRAM read + 128 BRAM line writes
`seek` (current value $\geq$ target)	—	—	0 BRAM reads
`seek` (δ ahead)	—	Yes	$O(\log \delta)$ BRAM reads
`seek` (δ ahead)	—	No	$O(\log \delta)$ BRAM reads + 128 BRAM line writes

Table 3.1: Cost of iterator operations across the full memory hierarchy. A lookahead hit serves the request from registers with no BRAM access. A PageCache hit translates the global address to a local BRAM address with no fetch from global memory. A PageCache miss triggers a page fetch from the global `PageMem` via the arbiter, stalling the requesting iterator until the full page is streamed into `SharedMem`.

Chapter 4

Parallel Join Execution

Chapter 3 described how a single iterator navigates one level of one relation. This chapter describes how multiple iterators are orchestrated to compute a multiway join, how multiple single-iterator joins are combined to intersect tries with arbitrary numbers of levels, and how that computation is parallelized across multiple hardware processing elements (PEs). The two modules responsible are `LeapfrogJoin` and `LeapfrogTriejoinCore`, which organizes a pool of PEs to perform a depth-first traversal of the full join space across all variable levels.

4.1 The Leapfrog Join

The Leapfrog Join (LFJ) algorithm [3] computes the intersection of k sorted sets, each represented by a `TrieRepr` iterator. Because an iterator traverses a contiguous run of `TrieNodes` in memory, the operation represents computation of a join over k single-variable levels in a trie.

Algorithm overview. The algorithm maintains one iterator per participating clause and operates as follows. Every iterator is initialized to its first element via a `next` call. Then the main loop executes:

1. Find the maximum value m across all active iterators.
2. If all iterators are at m, a join result has been found; emit the current nodes and stall.
3. Otherwise `seek`(m) on every iterator whose current value is less than m. Every lagging iterator jumps forward to the next value that is $\geq m$ in its sorted set.
4. Repeat from step 1.

The hardware implementation deviates from the original algorithm of [3], which advances iterators one-at-a-time in round-robin order, seeking each to the current maximum before moving to the next. Instead, all execution is parallelized: the Join engine reads all k key registers in a single pass, computes the maximum, and issues seeks to every iterator whose value falls below that maximum. The two approaches are equivalent because seek targets are monotonically non-decreasing. Once the maximum m is established from a consistent

snapshot of all key registers, any iterator below m must seek to at least m regardless of the order in which seeks are issued, since no iterator can move backwards.

The invariant preserved by this algorithm is that the maximum-valued iterator serves as the current leader that all other iterators `seek` to catch up to. If any iterator becomes exhausted, the intersection is empty and the algorithm terminates. Thus, each iterator visits at most N_{min} values, where N_{min} is the size of the smallest set, before the join completes. As discussed in Section 3.2, the amortized cost per visit for an iterator that visits N_{min} values out of N_{max} in sorted order is $O(N_{min} \log(N_{max}/N_{min}))$, which preserves the bound in [3]. Figure 4.1 illustrates a trace for three iterators.

Step	R	S	T	Action
	{1, 4, 6, 8, 10}	{3, 6, 8, 10, 12}	{2, 4, 6, 9, 10}	
0	1	3	2	max=3, seek(3) on R, T
1	4	3	3	max=4, seek(4) on S
2	4	6	4	max=6, seek(6) on R, T
3	6	6	6	match at 6
4	8	6	6	advance R, max=8, seek(8) on S, T
5	8	8	9	max=9, seek(9) on R, S
6	10	10	9	max=10, seek(10) on T
7	10	10	10	match at 10
8	∅	12	∅	advance S, max=12, seek(12) on R, T

Figure 4.1: Leapfrog Join trace for three sets $R = \{1, 4, 6, 8, 10\}$, $S = \{3, 6, 8, 10, 12\}$, $T = \{2, 4, 6, 9, 10\}$. Matches at values 6 and 10. At each non-matching step, all iterators below the current maximum issue `seeks` to the maximum value, skipping over nonintersecting regions of each sorted set. The join terminates when any iterator participating in it terminates.

Interface. The module `LeapfrogJoin` exposes three primary methods:

`configureSlice(iter, trie, start)` configures the iterator at position `iter` in the join's iterator array, pointing it at trie memory beginning at global address `start` within the physical trie identified by `trie` by calling `configureSlice` for each active iterator. Must be done before `lfj()` is invoked.

`lfj(numIterators)` initiates a new intersection over `numIterators` iterators and loads initial values for any fresh iterators.

`waitResp` dequeues and returns the next result, consisting of the matched node tuple and a validity flag. An invalid response means at least one iterator was exhausted and the intersection is complete.

`lfj(numIterators)` initiates a new intersection over `numIterators` iterators and loads initial values for any fresh iterators. `waitResp` dequeues matched values from an internal queue and returns the next result, which consists of the matched node and a flag indicating its validity.

Hardware state machine. The `LeapfrogJoin` module implements the algorithm as a three-state state machine: `IDLE`, `SEARCH`, and `ADVANCE`. The `SEARCH` state is reached when all iterators have completed any pending memory requests related to their `seek` operations and computes the maximum over the sought values. When all iterators have returned the same value, a match response is enqueued, and the join progresses to the `ADVANCE` state, which issues a `next` request on the iterator at the current rotation pointer before seeking all lagging iterators to the new value and returning to `SEARCH`. If a match is not found in `SEARCH`, seeks are issued simultaneously to all lagging iterators and the module waits for responses before reentering `SEARCH`.

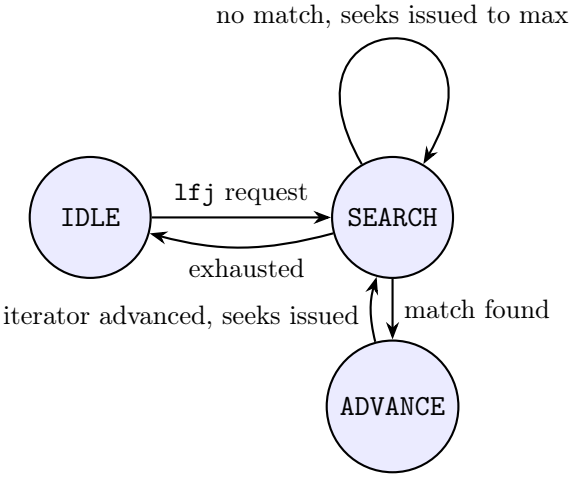


Figure 4.2: Leapfrog Join state machine. `SEARCH` finds the maximum value across all active iterators and either emits a match or issues `seek` calls to all lagging iterators. After a match, `ADVANCE` issues a `next` call to an iterator determined by a round-robin pointer before seeking all lagging iterators to the new value and returning to `SEARCH`. The join terminates when any iterator is exhuasted.

4.2 The Leapfrog Triejoin

A single Leapfrog Join invocation computes the intersection over one variable at one level. A full join over V variables requires V such intersections, one per variable, in a specific order determined by the **canonical variable ordering**.

Consider the query $Q(a, b, c) \coloneqq R(a, b),\ S(b, c),\ , T(a, c)$. The Leapfrog Triejoin algorithm evaluates this query by running one Leapfrog Join per variable in the chosen variable ordering $[a, b, c]$, which is determined by a compiler pass that reads the query left-to-right. At the first level, iterators over the level-0 arrays of R and T intersect to find values of a present in

both relations. For each matched a, the `childStart` addresses from the matched nodes are used to configure iterators over the level-1 subarrays of R, which are contiguous iterators themselves, and the level-0 array of S, which are then intersected to find values of b. For each matched b, a final intersection over the remaining sub-arrays of S and T finds values of c, completing the tuple (a, b, c).

The above example joined three **clauses**: clause 0 is $R(a, b)$, clause 1 is $S(b, c)$, and clause 2 is $T(a, c)$. Each clause has an associated trie in global memory and participates in a Leapfrog Join for each variable it contains. At every level of the join, the engine needs to know which clauses are active for the current variable and where in memory to start the corresponding iterator. The data structure that carries the state of a particular join across variable levels is the `Task`.

4.2.1 Clause, Trie, and Relation Identifiers

This section establishes the convention for identifying entities that are to be joined at three levels of indirection in the system: as logical names tracked by the evaluator, as physical sorted arrays read by the join engine, and as positions within a specific rule body. Conflating any two levels produces incorrect behavior in cases that naturally arise in real Datalog programs.

Tries as a representation of canonical variable orderings. Consider the query $Q(a, b, c) :- R(a, b), S(b, c), T(c, a)$ with variable ordering $[a, b, c]$. The Leapfrog Triejoin algorithm evaluates one variable at a time, so the first step is to find all values of a by intersecting the iterators of every clause that contains a. $R(a, b)$ stores a at level 0, so its level-0 array is immediately usable. $T(c, a)$, however, stores a at level 1: in T's trie, a is a child of c. We might then consider intersecting the iterators of every clause that contains b. $S(b, c)$ is immediately usable, but $R(a, b)$ requires a joined subset of a values before it can run joins over their child b iterators. Similarly, joining c values is blocked by knowing their parent b values in S. The way to "break" this deadlock is to require that each clause's argument list be a subsequence of some variable ordering.

The solution is to materialize a reverse-ordered copy $T'(a, c)$, whose level-0 array contains a values directly and allows the join to proceed over R nad T''s a values. T and T' are logically the same relations but are physically distinct tries in global memory. they must therefore carry distinct **trie IDs** so that the page cache can distinguish their pages and catalog misses correctly. A **relation ID** identifies the logical relation, and the set of trie IDs that correspond to a single relation ID is required to populate all corresponding tries with various permutations of any new tuples generated for a given relation.

Clauses as indirection for relations joined with themselves. Consider the query $\mathit{Reachable}(a, c) :- \mathit{Edge}(a, b), \mathit{Edge}(b, c)$ with variable ordering $[a, b, c]$. Both clauses correspond to the same logical relationship, and both variable orderings (a, b) and (b, c) are subsequences of $[a, b, c]$, which means these clauses are compatible with the same trie. However, the location in memory that corresponds to the variable b in the second clause is level 0 of the trie, while the location in memory that corresponds to the variable b in the first

clause is the level-1 child address of a previously joined a value, two independent positions in memory that must be tracked separately. Thus, task metadata cannot rely on trie ID alone to specify the necessary start locations of the iterators that comprise a join. Keying on **clause ID**, or the relative position of each rule-body element, gives each occurrence a distinct entry in the task mapping, correctly maintaining independent iterator positions even when two clauses share a trie. A separate mapping must be kept from clause ID to trie ID to interact with the correct trie in the cache.

4.2.2 The Task: The Unit of a Join

The identifier conventions established in the previous section determine how the join engine refers to clauses, tries, and relations. The `Task` data structure is what carries these identifiers through execution.

The algorithm as described so far presents many opportunities for parallelism: at each variable level, the Leapfrog Triejoin finds all matching values at a single level before the next level can begin. It feels tempting to run individual joins over all levels in parallel, but the serialized execution of the naive Leapfrog Triejoin comes from a real data dependency. In the above example, the values of b that are candidates for the final join are children of any joined a values, the locations in memory of which are only known after the full set of joined a values has been computed.

However, the dependency is shallower than it initially appears. To start the level-b join, the engine does not need all matched values of a — it needs exactly one. A single matched a value provides the `childStart` addresses for the b-level iterators that comprise its children. The child subarrays unlocked by a matched a node are sorted, contiguous arrays with exactly the same iterator interface as any top-level variable in a relation. Those iterators can begin intersection immediately, while the remaining a values can continue being found in parallel. The Leapfrog Join at the b level does not know, and does not need to know, that its iterators were derived from a specific a value rather than from the root of the relation, since its interface accepts only a series of configured iterators and finds their intersection. Every variable level, at any depth in the query, presents the join engine with the same interface, such that a join processing values deep in the middle of a large trie is structurally indistinguishable from a join processing the first variable of a rule for the first time.

The `Task` is the data structure that encapsulates this idea. It keeps track of a context of bindings and iterator addresses determined so far in order to configure the next level of the join and spawn new tasks when new joins are required. For example, a `Task` with no bindings and only root-level start addresses represents the entry point of a rule. A `Task` with partial bindings and some child-level start addresses represents a continuation partway through that same rule. From the join engine's perspective, both tasks are treated identically.

The two components of a `Task` are the `BindingCtx`, which tracks the state of the join as it progresses through variable levels, and a set of fixed metadata that describes the structure of the rule being evaluated.

Binding Context. The inner component `BindingCtx#(pMaxClauses, pMaxVars)` stores two elements:

- The concrete value assigned to each variable once it has been joined, which is populated in-order throughout execution of the Triejoin and used to construct the output tuple when all variables are bound.
- For each clause and each variable that clause contains, the global node address at which to start the iterator for that variable given the values already bound at higher levels. When a variable is joined to a value, the `childStart` address from each matching node is written into this table for the corresponding clause's next variable, so that the iterator for the next level begins at the correct position in the trie.

Task content. The fixed metadata carried alongside the `BindingCtx` describes the structure of the rule and how to route its results:

- The index of the variable currently being joined, which determines which level of each clause's trie the iterators should be configured at.
- For each clause, a bitmask indicating which variables that clause participates in, consulted at each level to determine which iterators should participate in the join.
- For each clause, the physical trie ID of the sorted array it reads from.
- The total number of variables in the rule.
- The identifier of the rule that produced this task, which will be used to route output facts to the correct relation.

A `Task` is therefore a complete, self-contained description of a single variable level's worth of join work. Given a `Task`, the join engine can configure its iterators, run the leapfrog join, and either emit a result or produce a child `Task` for the next variable level, without any knowledge of how it was created or where it sits in the overall query evaluation. The following section describes how the join engine manages multiple `Tasks` simultaneously in a parallel execution model.

4.2.3 Multi-PE DFS

As the previous section establishes, a match at variable level l immediately produces a new join task involving the level $l + 1$ children of the matched variable. Since the child task is ready the moment a match is found, and since it is independent of any further matches at level l, it can be executed in parallel with the completion of the level l join if another PE is available to perform the join. Suppose the join has V variables, and each Leapfrog Join at each level produces an average of b matches before exhausting its iterators. The first join at level l_0 produces b matches, and each match spawns a child task at level l_1. Each of those b child tasks produces b more matches, spawning b^2 tasks at level l_2. In general, the number of simultaneously live tasks at level l_k is b^k. For a query with V variables and a branching factor of even $b = 2$, the number of simultaneously live tasks at the deepest level is 2^V. In addition, PE's spawn new tasks at frequency b times the rate that they finish a full join sequence and become idle. Thus, a hardware system with a fixed number of processing elements cannot

allocate a new PE for every task in this frontier, and any fixed-size queue that buffers tasks that are waiting for a PE will quickly overflow.

For a query with V variables, any single path from the root of the join tree to a leaf passes through exactly V levels. A depth-first traversal of the join tree requires at most V simultaneously live tasks at a given time, regardless of the branching factor. The total work is the same, but both the peak concurrency and the hardware utilization is bounded by V, rather than b^V.

Multiple processing elements can still provide parallelism within this bound. When a match generates a child task and a free PE is available, the child task can be handed off immediately and proceed concurrently with the parent. The parent continues to search for the next match at level $l-1$, while the child explores the subtree rooted at the matched value. When no free PE is available, the parent executes the child itself.

Though this traversal is necessary to avoid exponential utilization blowup or deadlock, it requires a given PE to halt execution of a join at level $l-1$ to begin a child join at level l. Recall from Section 4.1 that the Leapfrog Join's complexity bound of $O(N_{min} \log(N_{max}/N_{min}))$ comes from the fact that the iterator is consumed as joined values are successively produced, and the length of the iterator left to search decreases with every join found. If, upon completion of the level l join, the iterators corresponding to the level $l-1$ join were reloaded with the old join's start addresses and restarted from the beginning, the accumulated progress would be lost, and the search space would widen to the full length N_{max} of the iterator. The join would start its search from the beginning of the iterator every time the parent join returns from execution of a child task, so the bound would degrade to $O(N_{min} \log N_{max})$. Thus, we introduce a mechanism to "save" iterator positions at the moment a parent task is suspended and "restore" them at the moment of resumption.

Save and Restore

Consider the moment at which a PE is about to leave a level $l-1$ join to execute a level l child task. Before the PE can switch its attention to the child task, it must record the state of the level $l-1$ join that will be needed to resume it correctly later.

The first and most obvious piece of state to record is the description of the task itself, which will be used to construct future child tasks once the join at the parent level is resumed.

The second piece of state to record is the position of each active iterator. We define a **snapshot** of an iterator as the tuple of values needed to reconstruct its state exactly: the start address and length of the current array, which defines the bounds of the search; the current position within that array; the current node value at that position, so that the join can immediately read the current value on resumption without issuing a memory request; and the physical trie identifier, so that resumed memory requests are routed to the appropriate pages in the cache hierarchy. Two methods are added to the iterator interface to support these operations:

`getSnapshot()` Returns the current snapshot as a single register read. This is valid to call at any point when no memory request is in-flight, since all snapshot fields live in registers.

`restorePosition(snapshot)` Writes the snapshot fields back into the iterator's registers, repositioning the iterator exactly as it was when the snapshot is taken. Since no memory

request is issued, the restored node value is immediately available via a call to `data()`.

Both operations are single-cycle and involve no memory traffic, since they read and write only the iterator's internal state.

The third piece of state to record is state associated with the progress of the Leapfrog Join. Because a child task replaces a parent in a PE after it has been created from a newly discovered join value, one iterator, chosen by a round-robin pointer, will advance past the match point and set a new value for lagging iterators to `seek` to. Two methods are added to the Leapfrog Join interface to support saving and restoring this pointer:

`getIterPtr()` Returns the current rotation pointer as a single register read, identifying which iterator position in the join was most recently advanced.

`restore(nActive, iterPtr, nodes)` Resumes a previously suspended join by reinstalling the saved rotation pointer; the saved number of active iterators, which can differ between levels, depending on the number of clauses participating in the join at a given level; and the node values from each iterator's saved snapshot into the join's internal state. The join then transitions to the advance state, issuing a `next` call on the saved rotation pointer iterator.

The inclusion of the iterator position in the saved snapshot reveals a constraint on when the save must happen. The snapshot must be taken after the match response has been received and before the join has been asked to advance past the matched value. At that moment, all pending requests for the current join have resolved, so the snapshots reflect stable register values. If the join were allowed to advance immediately after emitting a match, the iterator positions would move forward and no longer accurately reflect the snapshot taken, and new memory requests will be issued for the `ADVANCE` stage of the join and any subsequent calls to `seek`. Since these requests are sent to nonblocking memory, inflight memory requests for a join at level $l - 1$ might be resolved after the system transitions to a level l join. The join must therefore be held at the matched value until the save is complete and the child task has been handed off.

Stall and Unstall

To that end, we introduce a new `STALL` state in the Leapfrog Join state machine. When a match is found, the join transitions into the `STALL` state, which can only consume any orphaned memory requests, and stops advancing. It remains stalled until it receives an explicit signal to resume from the upstream Triejoin, at which point it transitions to the advance state and proceeds normally. The stall guarantees that iterator positions are stable at the moment the snapshot is taken.

An individual join can resume execution when either a match is found and no child task needs to be dispatched, because the match has terminated, or when another PE is available to execute the child task in parallel, the join is resumed immediately via an `unstall()` call. When a child task is dispatched, the join remains stalled until the dispatching logic has finished constructing the child task, determined whether it needs to be re-enqueued to the parent, and retrieved and pushed parent state to the stack. Only then is the join signalled to

resume, either via `unstall()` if the parent continues on the same PE, or via `restore()` if the PE is switching to execute the child task directly. The updated state machine is shown in Figure 4.3.

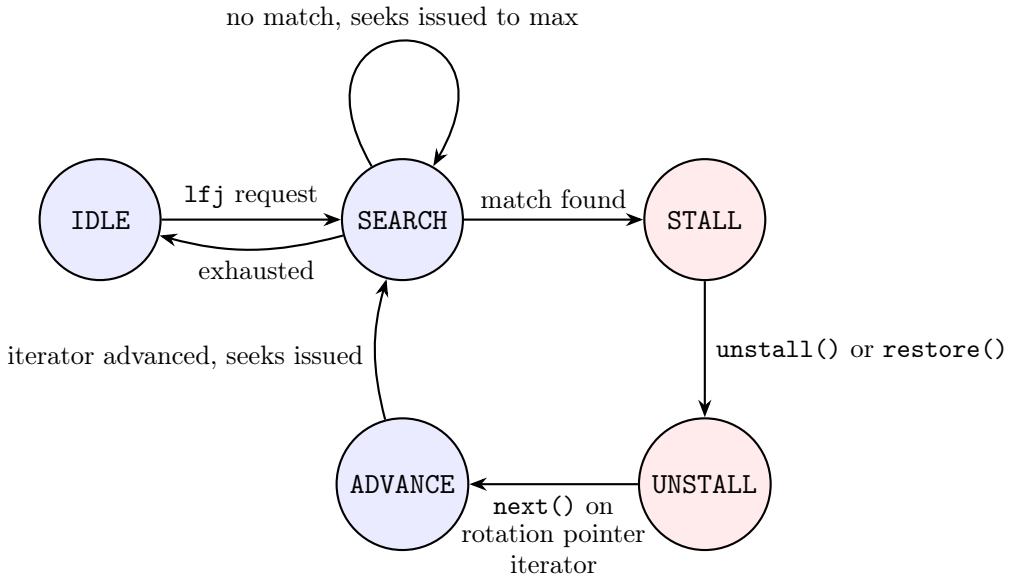


Figure 4.3: Full Leapfrog Join state machine, with new states labeled in red. The `STALL` state is entered after every valid match and persists until the join is explicitly resumed. `unstall()` is called when the parent join continues on the same PE, while `restore()` is called when the PE is resuming a previously suspended join after popping its DFS stack. Both pathways pass through the same `UNSTALL` and `ADVANCE` states to resume a join following a match.

With the save, restore, and stall mechanisms established, the pieces needed for parallel depth-first join execution are in place. The following section describes how these pieces interact with each other, from task arrival to result emission.

4.2.4 The Execution Sequence

The core of the Leapfrog Join execution proceeds in three phases. A task arrives and is assigned to a free processing element, which configures its iterators and launches the join. The join then runs until it produces a response: either a valid match, which triggers child-task construction and dispatch, or an invalid response indicating that the join has terminated, which may trigger stack unwinding. Each phase is described in turn below.

Dispatch. When a task arrives, the scheduler scans the PE pool in round-robin order to find a free PE. Child tasks generated by active joins are given priority over externally submitted tasks, since they are generated with a higher frequency and come from PEs that are stalled waiting for them to be dispatched. Once a free PE is found, the scheduler determines which clauses participate in the current variable level and configures one iterator per participating clause, pointing each iterator at the start address recorded in the task's binding context for that clause and variable level. The join is then launched, and the PE is marked busy.

If no free PE is found, the current PE saves its join state to the stack and executes the child task directly. The PE takes a snapshot of each active iterator, records the current rotation pointer and number of active iterators, and saves the current task alongside the newly recorded values. This stack entry is pushed in a single cycle. The PE then configures its iterators for the child task and launches the child join. The parent join remains suspended on the stack until the child join has terminated.

Handling a valid response. When the join finds a match, it transitions to the stall state and waits. The PE reads the matched node from each active iterator, records the matched value in the binding-context, and writes the `childStart` address from each matched node into the binding context entry for that clause's next variable. The system then determines the next variable to join by finding the lowest-indexed unbound variable across all participating clauses.

If no unbound variable remains, all variables are bound, and a complete result tuple is emitted to an output queue. The join is then `unstalled` and continues searching for the next match at the current level.

If an unbound variable remains, a child task is constructed for that variable level with the updated binding context. The system then makes a routing decision: if the pages needed by the child task are unlikely to be resident in the local cache, the task is placed in a **spill** queue for the global arbiter to redispatch elsewhere, and the current join is unstalled. If the pages are likely resident, the task is enqueued to a child queue and waits to be dispatched as described above.

Handling an invalid response. When the join is exhausted, the PE checks its stack. If the stack is non-empty, the PE pops the top entry, restores the saved iterator snapshots via `restorePosition` on each active iterator, and resumes the parent join via `restore`, providing the saved rotation pointer and node values. The parent join transitions to the `ADVANCE` state, issues a `next` call to the rotation-pointer iterator, and continues from exactly the point at which it was suspended. If the stack is empty, the PE becomes idle and is available to receive the next task.

4.2.5 Interface and Architecture

The `LeapfrogTriejoinCore` module exposes a narrow interface to the unit above it. Tasks are submitted via `enqueueTask`, and complete results are collected asynchronously via `waitResp`. Spilled tasks are exposed via a `Get` interface on the spill queue, which will eventually be picked up by the arbiter for redispatch.

The core of the Leapfrog Triejoin manages all join execution internally and only exposes task inputs and outputs. The memory interface and interaction with a global task router is handled by a wrapper module described in Chapter 5.

Figure 4.4 shows the full architecture of the core, including the PE pool, the queues connecting its components, and the signals that flow between them.

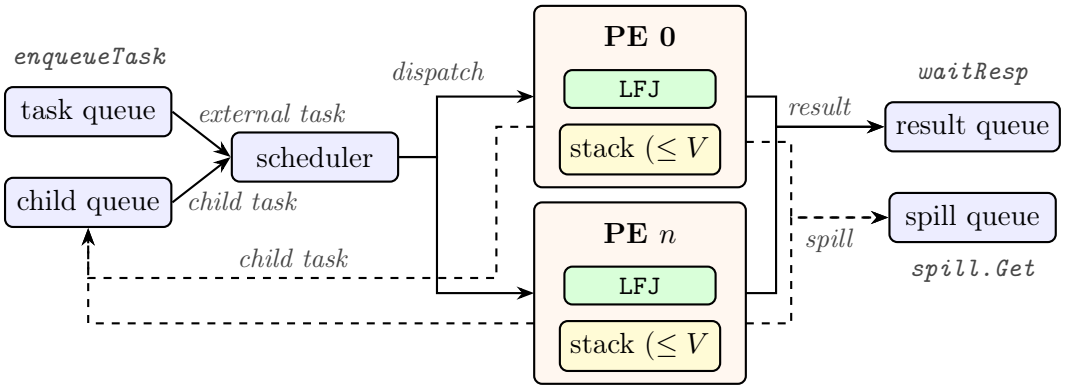


Figure 4.4: Architecture of `LeapfrogTriejoinCore`. External tasks enter via `enqueueTask` and are placed in the task queue. Child tasks generated by matches feed back into the child queue, which is given dispatch priority. Each PE contains a Leapfrog Join instance and a stack of depth at most V, where V is the number of variables in the join. Results are collected via `waitResp`, and spilled tasks are forwarded to an arbiter via the spill interface.

Chapter 5

Distributed Execution

Chapter 4 described the parallel execution of a join within a single pool of processing elements sharing one page cache. Scaling beyond a single unit requires multiple independent instances of this pool, each with its own page cache and processing elements, executing different tasks simultaneously. This chapter describes how multiple independent units are coordinated to share work across a larger compute fabric, how memory is managed globally, and how routing decisions exploit data locality to minimize the number of cache misses that tasks encounter when they are dispatched.

5.1 The Leapfrog Triejoin Unit

The `LeapfrogTriejoinCore` of the previous chapter exposes a narrow interface that has no knowledge of where its memory comes from or whether its cache is well-populated for a given task. The `LeapfrogTriejoinUnit` is the module that wraps the core and connects it to the memory hierarchy.

5.1.1 Likely Pages and the Distributed Task

The point where memory first intersects task execution is routing. When a task is ready to be dispatched, the system must decide which unit to send it to. Sending a task to a unit whose cache already contains the pages the task will access means the task runs at local BRAM latency throughout. Sending it to a unit that does not have those pages means the task generates cache misses from its first iterator access, stalling until each missing page is fetched from global memory. To make this decision, the system needs to know, before dispatching a task, which pages that task is likely to access first.

This information is captured in the **likely pages** of a task, which includes one page tag per active iterator at the current variable level that represents the page containing the start address of each iterator's trie slice. The task will access these pages on its very first `seek` operations, making them a reasonable proxy for predicting which unit has the most relevant data already cached. Likely pages are computed from the binding context: for each clause whose participation bitmask includes the current variable level, the page tag of the corresponding start address is recorded.

Likely pages are bundled with the task into a **distributed task**, which is the unit of work that flows between Leapfrog Triejoin units and the arbiter, and represented by the `DistTask` type. A `DistTask` wraps a `Task` with the list of likely page tags and count of active iterators. Every task that enters the arbiter, whether submitted externally or spilled from a unit, carries its likely pages or can construct them from binding information, allowing the arbiter to make an informed routing decision without inspecting any unit's cache directly.

As alluded to in Section 4.2.4, the likely pages are used by the `LeapfrogTriejoinUnit` wrapper to determine whether a child task has enough pages resident in local memory to be performed by the unit. The unit checks the child's likely pages against the local cache of pages resident in memory to decide whether the task should be spilled. If at least half the likely pages are already resident, the task is kept local. Otherwise, the unit turns the spilled task into a `DistTask` and spills it to the arbiter.

5.1.2 The Leapfrog Triejoin Unit Interface

Beyond routing, the core is insulated from higher-level execution and memory in two major ways that the unit handles: memory-event forwarding and busy signaling.

When a join running inside the unit crosses a page boundary into a page that is not resident in the local cache, a miss request is exposed by the `PageCache`, as described in Section 3.3.2. The unit is the immediate consumer of this miss request, which it forwards to the arbiter to service. The arbiter fetches the missing page from global memory and streams it back through the unit into the local cache, at which point the stalled iterator can resume. Similarly, when the local cache evicts a page to make room for a new one, the unit forwards the evicted page tag to the arbiter, which updates its routing information based on the new event. From the perspective of the core, its iterators are either serviced by the cache locally or stalled until the unit resolves the miss in the background.

The second concern is busy signaling. An eventual Datalog evaluator that submits tasks to the arbiter needs to know when a round of join execution is complete, i.e. when all submitted tasks have been fully executed and all results have been produced. Without this signal, the evaluator cannot safely move on to the next round of computation. The unit exposes an idle signal that the evaluator uses to make this determination. A unit is idle only when it has no pending work of any kind: no currently executing joins, no buffered results awaiting collection, and no enqueued tasks waiting to be spilled or dispatched within the unit.

The full interface is described as follows:

`dispatch` Accepts a `DistTask` from the arbiter for execution.

`spillUp` Exposes child tasks that the unit has determined should be redispatched by the arbiter.

`missReq` Exposes page tags for pages missing from the local cache during execution, for the arbiter to fetch from global memory.

`missResp` Accepts the stream of page packets returned by the arbiter in response to a miss or prefetch request, forwarding them into the local cache.

`evictReq` Exposes page tags for pages evicted from the local cache, for the arbiter to invalidate in its routing table.

`lftjResult` Exposes completed join results for a Datalog evaluator to handle.

`isIdle` Asserted when there is no pending join work within the unit.

5.2 The Arbiter

Throughout the previous chapter and description above, we have assumed the existence of an entity that dispatches tasks, redispatches spills, fetches missing pages, and collects results from units. We can now give this entity a name and shape: the `Arbiter` is the top level coordinator of the distributed execution model. It is the single point through which tasks enter the system, through which global memory is accessed, and through which results leave. Every interaction between an entity that is involved in join calculation and the outside world is mediated by the arbiter.

5.2.1 Task Dispatch

Task dispatch is the arbiter's primary responsibility and the mechanism through which the likely page routing described in Section 5.1.1 is implemented. The arbiter maintains a task queue that accepts both externally submitted tasks and tasks spilled up from units. Because both types of tasks are treated identically, it is possible that a spilled task is reenqueued back to the unit that spilled it, e.g. when the originating unit, despite not meeting the local threshold for keeping the task, still has better cache residency for the task's pages than any other unit. However, now that it is being enqueued externally, it will only resume execution once other tasks local to the unit have completed, which prevents the requisite page fetches from evicting other useful pages in the originating unit.

The arbiter also maintains a set-associative cache that maps page tags to unit IDs, recording which unit most recently fetched each page. When a task reaches the front of the queue, the arbiter queries this cache with each of the task's likely pages and "scores" each unit by the number of hits. The unit with the highest score is selected as the destination, on the basis that it most recently held those pages and is therefore most likely to still have them cached. If no unit scores any hits, the arbiter falls back to round-robin selection to distribute work evenly. Because the cache of likely pages is small enough to fit in BRAM, the cost of accessing this memory to make the dispatching (at most one cycle per page in the join) is low compared to the cache-miss cost (at most one page fill per page) that is prevented by memory-based routing.

Before dispatching the task, the arbiter prefetches any likely pages that are not already owned by the selected unit. For each such page, the arbiter fetches it from global memory and streams it into the destination unit's cache. Thus, the first iterator accesses are serviced from the local cache rather than generating immediate misses.

Finally, the dispatcher forwards the task to the unit. The arbiter does not move to the next taks in the queue until unit selection, prefetching, and forwarding are complete for the current task. Since prefetching requires fetching full pages from global memory, the dispatch

pipeline is the primary source of contention on the global memory port, discussed further in the miss-handling section below.

5.2.2 Cache-Miss Handling

During execution, a unit may descend into a child iterator whose start address falls on a page that was not among the likely pages prefetched at dispatch time and is thus not in its local cache. In this case, the unit's page cache raises a miss request, which the unit forwards to the arbiter as described in Section 5.1.2.

The arbiter services miss requests by polling all units' miss-request queues in round-robin order. When a miss is found and the global memory port is not already in use, the arbiter fetches the missing page and streams it to the requesting unit, using the same streaming mechanism as the prefetch step. The cache of likely pages is updated when the stream completes, recording the requesting unit as the new owner of the page. Similarly, on receipt of an eviction notice, the evicting unit is discarded as the owner of a likely page.

Each page fetch transfers a full page of 2^{10} nodes as a sequence of 512-bit or 8-node packets, regardless of how many nodes the requesting iterator actually needs from that page. The global memory round-trip time is amortized over many subsequent accesses, either from the requesting iterator or joins involving siblings of that iterator.

The global memory port is a shared resource used by both prefetch streams (initiated during dispatch) and miss streams (initiated during execution). At most one stream can be in-progress at a time. Miss handling is therefore blocked while a prefetch is in-progress and vice versa. This serialization is the primary latency cost of a cache miss; a unit that misses on a page must wait not only for its own stream to complete but potentially for a prefetch stream already in-progress to finish first. The natural way to reduce this contention is to minimize the duration of each stream, which is determined by the bandwidth of the global memory. For on-chip global memory (BRAM), bandwidth is relatively high, while latency is low, but chip utilization scales with the expected size of a Datalog workload. At scale, a high-bandwidth memory interface such as HBM would reduce stream duration proportionally and allow the arbiter to serve miss and prefetch requests in fewer cycles, reducing the window during which other streams are blocked. Conversely, a narrow DDR interface would lengthen each stream and increase contention.

5.2.3 Result Collection

As units complete joins, they enqueue results to their local result queues. The arbiter collects these by polling all units' result interfaces in round-robin order and forwarding each result to its own output queue. Result collection runs concurrently with task dispatch and miss handling since it does not use the global memory port. A result carries a rule index identifying which rule produced it, allowing the caller to attribute each result to its origin and handle output facts accordingly.

Once all results have been produced, each Leapfrog Triejoin unit exports an idle signal. The arbiter aggregates the idle signals of all units into a single system-level idle signal. It is asserted only when every unit is idle, the input task queue is empty, no memory operations are

ongoing, and the output result queue is empty, i.e. when the entire distributed computation has quiesced and no further results are expected until a new task is submitted.

5.2.4 Arbiter Interface

The arbiter exposes a narrow interface to the caller above it:

`submit(task)` Enqueues `task` for dispatch. The caller constructs the initial `DistTask` with the addresses for the first variable level of the rule and submits it here.

`getResult` Dequeues a completed join result, consisting of the bound variable values and the rule index identifying which rule produced it.

`isIdle` Asserted when there is no pending work: all units are idle, no page stream is in progress, and any input and output queues are empty.

The complexity of task routing and cache operations is hidden behind this interface. Everything exposed to the caller concerns the submission of an initial join and observing the ultimate result of the join.

Throughout the execution of the `Arbiter` as described above, pages are fetched from and streamed out of a global memory. We now turn to our discussion of the interface that this memory exposes.

5.3 Global Memory: PageMem

`PageMem` is a dual-port BRAM that holds the full global address space of trie data. Beyond serving as the backing store for every page stream the arbiter initiates, it must also support writes: as new facts are derived during a Datalog computation, new trie nodes need to be written into the global address space so that subsequent joins can read them. These two access patterns have different requirements and thus claim separate ports so that one does not block the other.

Port A: Page Streaming Port A is dedicated to the `Arbiter` for page streaming. When the arbiter requests a page, `PageMem` reads one 512-bit packet per-cycle and forwards each packet to the destination unit with a packet index and a flag indicating whether it is the last in the stream. Port A is optimized for this sequential access pattern and is never used for random single-node access.

Port B: Single-Node Access and Trie Construction Port B supports both reading and writing individual trie nodes into the global address space. Writing a single node requires a read-modify-write sequence: read the full 512-bit line containing the target node, update the relevant slot, and write the full line back. Port B maintains its own state machine to sequence these operations. When writing to fresh memory regions, as is the common case during trie construction, where new facts are accumulated and written sequentially, a full line of eight nodes can be written directly without a prior read.

Because the two ports are physically independent channels of the underlying dual-port BRAM, streaming on Port A and writing on Port B can proceed simultaneously without reference. Thus, pages can be streamed into unit caches at the same time that results are being written to new trie memory, allowing database updates and join execution to overlap.

5.4 Full System Architecture

Figure 5.1 shows the complete architecture of the Leapfrog Triejoin engine, from the unit level to the global memory interface, summarizing the interactions described throughout this chapter.

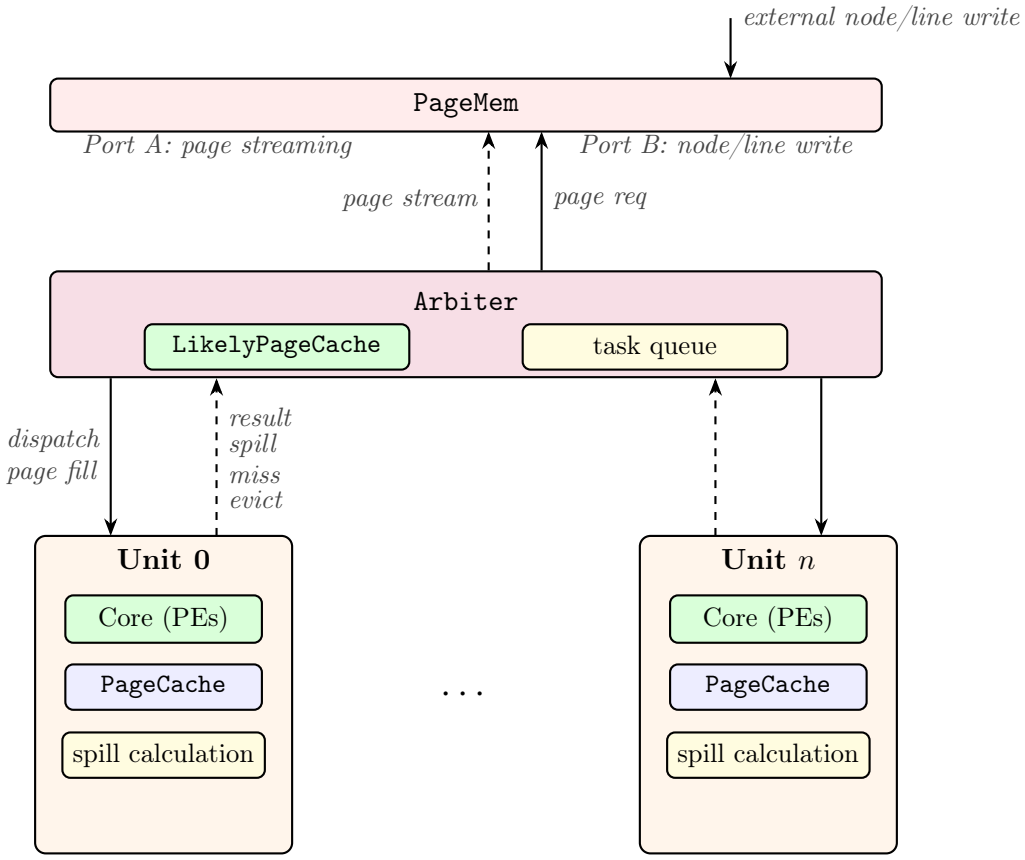


Figure 5.1: Top-level system architecture for Leapfrog Triejoin engine. PageMem serves as the global backing store, accessed exclusively by the arbiter via two independent ports: Port A for sequential streaming and Port B for node-level reads and writes. The arbiter coordinates task routing using the `LikelyPageCache`, prefetches page before dispatch, handles cache-miss requests from units, and collects results. Each unit wraps a `LeapfrogTriejoinCore` and a `PageCache`, using the spill calculation to decide whether child tasks are executed locally or returned to the arbiter for redispatch.

The system described in this chapter is a complete, self-contained join accelerator. Given a set of trie data loaded into `PageMem` and a set of initial tasks submitted to the arbiter,

it will find all tuples satisfying the join, exploiting parallelism across processing elements and join units while managing memory locality automatically. What it does not yet do is decide which tasks to submit, interpret the results it produces to determine when a fixed point in computation is reached, or update the tries in `PageMem` as new facts are derived. These are the responsibilities of the semi-naive evaluator and memory updater, whose design is described as future work in Chapter 7.

Chapter 6

Results and Discussion

This chapter evaluates the claim that a hardware implementation of the Leapfrog Triejoin can reduce the per-result cycle cost by an order of magnitude relative to software. We describe the evaluation setup, confirm that the design produces correct output across all benchmark queries, and measure how performance scales with problem size and parallelism. The results are discussed in the context of existing CPU and GPU implementations of the same algorithm.

6.1 Evaluation

The full design is compiled with the Bluespec SystemVerilog (BSV) compiler and simulated using the generated C++ model. A Python test harness, given a target number of output nodes and number of PEs to generate, generates synthetic Datalog datasets for five benchmark queries that represent five possible rules that could be run on a Datalog engine built over this join:

$$
\begin{aligned}
\textit{Triangle}(a,b,c) &:- R(a,b),\ S(b,c),\ T(a,c)\\
\textit{4-Chain}(a,b,c,d) &:- R(a,b),\ S(b,c),\ T(c,d)\\
\textit{4-Cycle}(a,b,c,d) &:- R(a,b),\ S(b,c),\ T(c,d),\ U(a,d)\\
\textit{5-Star}(a,b,c,d,e) &:- R(a,b),\ S(a,c),\ T(a,d),\ U(a,e)\\
\textit{5-Cycle}(a,b,c,d,e) &:- R(a,b),\ S(b,c),\ T(c,d),\ U(d,e),\ V(a,e)
\end{aligned}
$$

These queries were chosen to match the benchmarks explored in [3] and [11], and each either extends them to larger queries or inverts them to stress-test a different extreme of query.

Synthetic edge sets are generated by inverting the output-size parameter N through the AGM bound for each query shape, giving a target degree d and per-variable-level size n such that the expected result cardinality is $\Theta(N)$ [8]. Each trie is then populated by independently sampling d children per root value from a Gaussian distribution with mean d and standard deviation $\sqrt{d}$.

The test harness writes these values to the flat trie representation described in Chapter 3 and generates a top-level test module in Bluespec that submits the initial task to the

arbiter and consumes results from its output queue. All single-unit experiments use a 2-Way associative local cache with 4 cache sets. When all results are collected, they are compared against the expected join values computed in software. Performance counters, such as the cycle count, page-miss count, or BRAM-read count, are read via DPI after the simulation completes.

The RTL simulation produces the correct set of output tuples for all five benchmark queries at arbitrary scaling of problem size and parallelism. Results are verified by exhaustive comparison against many randomly-generated reference tries as described in the test harness.

6.2 Performance at Scale

Figure 6.1 shows amortized cycles per-result as a function of result count for the triangle query across five PE configurations (1, 2, 4, 8, and 16 PEs). Results are similarly distributed for other topologies, though performance varies per-topology when PE count is fixed. Figure 6.2 shows amortized cycles per-result for all five benchmark queries operating with a single PE.

In all cases, join behavior appears to flatten to a near-constant asymptote relatively quickly as the dataset grow. At small N, the cost is dominated by the fixed overhead of each join invocation: dispatching, prefetching, and eventually terminating the join. Page-miss costs can also be amortized over a larger number of results that can subsequently read from a warm cache.

Adding PEs shifts the asymptote down consistently but modestly. At $N = 10^6$, increasing from 1 to 16 PEs reduces the cost of performing the triangle query from 34.0 to 19.9 cycles per result, for a 1.71× speedup (Figure 6.3). This sublinear return is an inherent property of DFS execution. The level-0 join, which scans the root arrays of all participating tries to find valid bindings for the first variable, runs on a single PE. Only the work below that first match, or the subsequent subtrees within that join, can be distributed across PEs. As discussed in Section 4.2.3, the depth of the DFS stack bounds the number of concurrently active frames to at most V per PE group, so the triangle query tightly constrains this parallelism. Queries with deeper DFS trees benefit slightly more from PE scaling: The 5-Star achieves 1.81× at 16 PEs, while the 4-Chain achieves only 1.40×. Running tasks from multiple rules on a single join engine will better be able to take advantage of the PE pool from the outset.

At $N = 10^6$, the single-PE cost converges to between 25 and 37 cycles per result depending on the topology (Table 6.1).

Table 6.1: Amortized cycles per-result at $N = 10^6$, 1 PE, 1 unit

Query	Results	cycles/result
4-Chain	1,004,368	25.2
4-Cycle	955,210	26.9
5-Cycle	899,969	28.3
Triangle	986,409	34.0
5-Star	702,637	37.4

The FSM and dispatch overhead accounts for roughly 70% of this cycle count and as

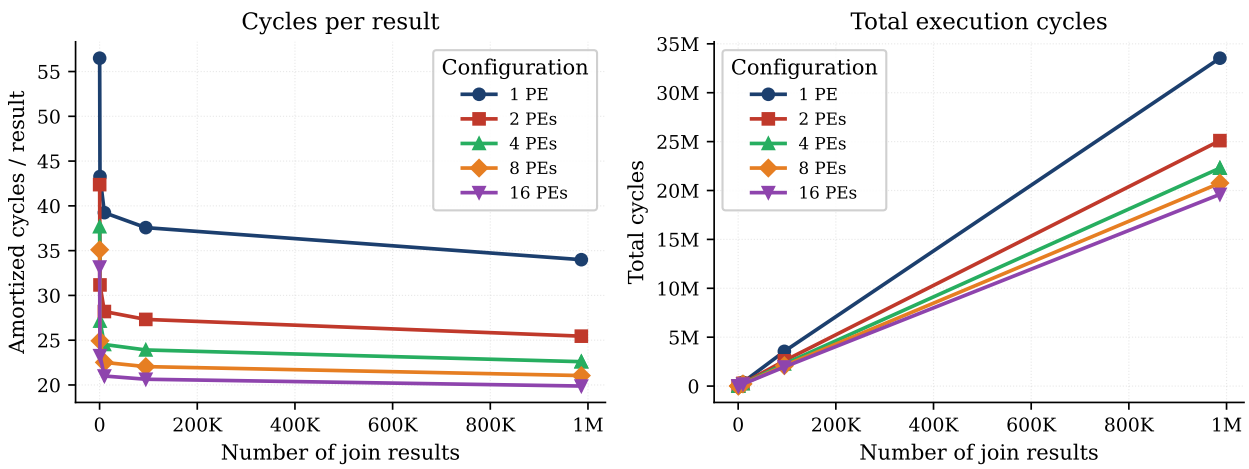


Figure 6.1: Output scaling for the Triangle query.
Left: amortized cycles per-result as a function of result count, for PE counts 1–16. *Right*: total cycles elapsed per-problem size, for PE counts 1–16.

much as 97% for the 5-Star. This component comprises the fixed sequence of rule firings for every matched tuple: detecting equality, entering `STALL`, collecting and dequeueing the join response, updating the binding context and enqueueing a child task, configuring iterators, entering `UNSTALL`, and resuming the join account for a 6-cycle minimum latency between results before any subsequent memory accesses. This overhead fires once per result per DFS level, so queries with more variables incur a proportionately higher FSM cost. Adding PEs reduces this component approximately linearly, since independent subtrees can process their per-result overhead in parallel. The remaining cost is split between BRAM reads (∼21% for the Triangle) and stalls due to page misses (∼9%); see Figure 6.4. The BRAM fraction per result *falls* with increasing N because longer tries allow more parent bindings to share the same child subarray. A single subarray for a frequently occurring join value is loaded once and traversed many times, amortizing its read cost over all future results in the DFS chain that share that value.

The variation in asymptotic cost reflects the interplay between query structure and DFS overhead as previously described. The 4-Chain is the cheapest because it is acyclic, while the 5-Star is the most expensive despite being dominated by the cheaper `next` operation (1.27 calls to `next` per result) and performing almost no `seeks`. With $V = 5$, each result triggers the full 8-cycle FSM sequence four times as the join descends from the hub to each spoke variable in turn, though the higher chain allows more room for parallelism to close the gap. The larger size of the intersection also incurs a penalty due to memory serialization; future work may explore whether banking the on-chip BRAM to allow multiple simultaneous reads would reduce this serialization cost.

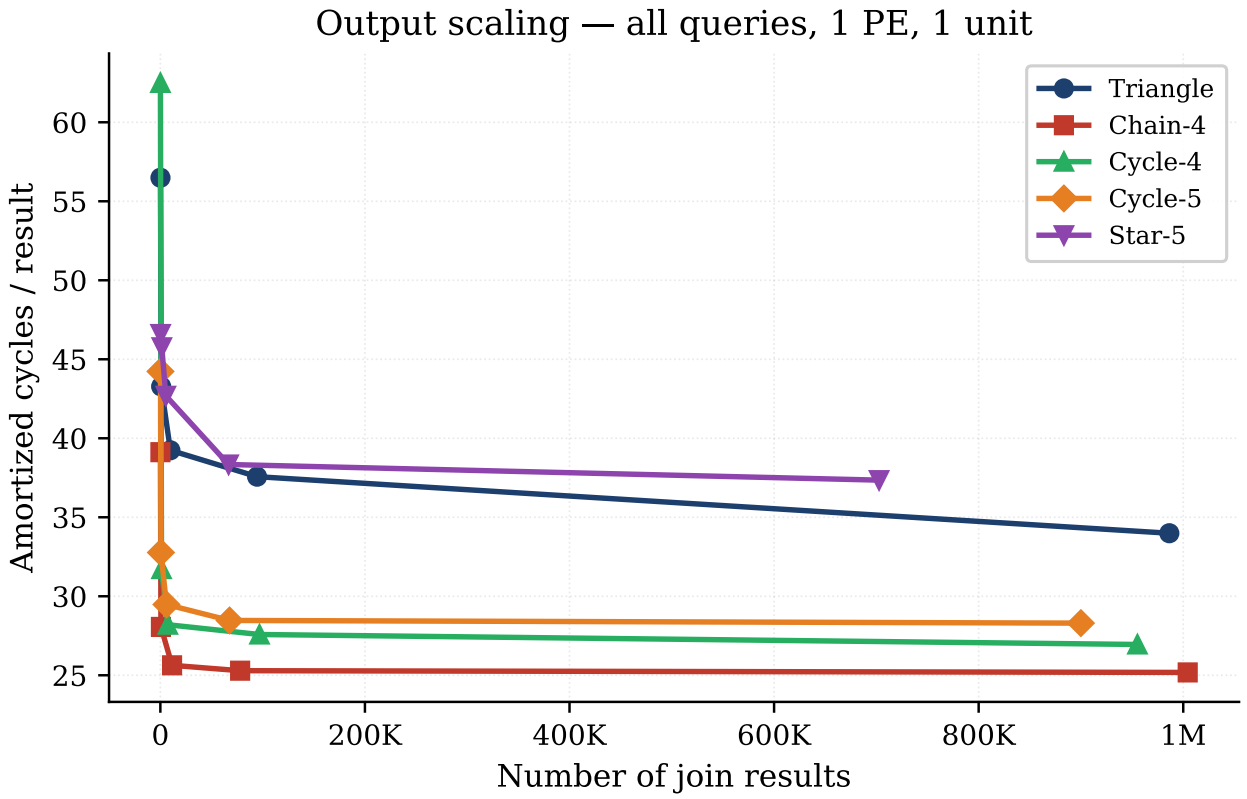


Figure 6.2: Amortized cycles per-result for all five benchmark queries at $P = 1$, as a function of result count.

6.3 Comparison to Prior Work

The closest related work is Wu et al. [11], who implement both a CPU-parallel and GPU-optimized Leapfrog Triejoin on an NVIDIA GeForce GTX Titan, evaluating triangle and 4-clique queries on randomly generated sparse graphs. Their CPU implementation processes 100M edges in 51.9 seconds on a single thread of a 3.5 GHz Intel i7-4771, and 30M edges in 12.8 seconds; the single-threaded figures are used to most accurately compare performance on a single unit.

Because this work evaluates performance in RTL simulation rather than on physical FPGA hardware, a direct wall-clock comparison is not yet possible. Instead, we normalize both sides to cycles per input edge traversed, using the 3.5 GHz clock frequency of the baseline in [11]. The single-PE hardware Leapfrog Triejoin design processes each output edge in 34 cycles, and the 16-PE design processes each edge in 19.9 cycles. By the above estimate, the single-threaded CPU baseline requires approximately 1,800 cycles per edge, constituting a $\sim$50$\times$ gap in cycle efficiency at 1 PE and $\sim$90$\times$ at 16 PEs.

The GPU variants in [11] close some of this gap. The LFTJ-GPU variant is 3.12$\times$ faster than LFTJ-CPU on the triangle benchmark, and the GPU-optimized variant is 5.23$\times$ faster on the same query. Our design's $\sim 50 - 90\times$ improvement in cycle efficiency over the CPU baseline exceeds both GPU variants by a significant margin on a per-cycle basis. Whether these margins are reflected post-hardware synthesis remains as future work. Translating our

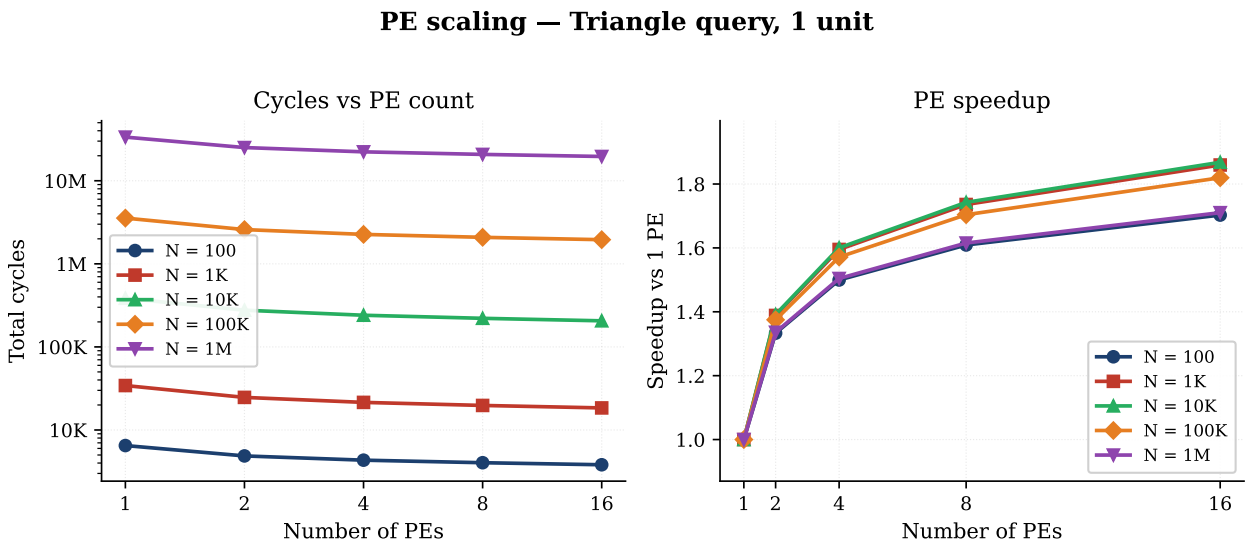


Figure 6.3: PE scaling for the Triangle query. *Left*: total cycles on log–log axes for each problem size N. *Right*: speedup relative to 1 PE.

cycle counts into a wall-clock throughput requires FPGA synthesis and timing closure to determine the achievable clock frequency. Furthermore, GPU variants operate at high clock frequencies with thousands of parallel cores, while our cycle counts reflect how efficiently the algorithm is processed on a bounded pool of parallel workers with a single unit, rather than multiple units scaled across a fabric of compute nodes. However, the per-result overhead on the proposed architecture is far lower than CPU baselines, and the gap is large enough that even a conservative FPGA clock frequency would be competitive with the GPU implementations on a throughput basis, even at the lowest levels of parallelism, i.e. a single unit performing a join.

It is also worth noting that the GPU-Optimized variant is not a faithful implementation of the iterator-based Leapfrog Triejoin. Rather than advancing iterators one `seek` at a time, it replaces the seek loop with a bulk-synchronous set intersection across all threads simultaneously, which requires materializing significantly more intermediate state. The iterator-based design presented here has no such intermediate materialization: memory consumption is proportional to the trie representation of the input relations and the depth of the DFS stack, both of which are small and fixed.

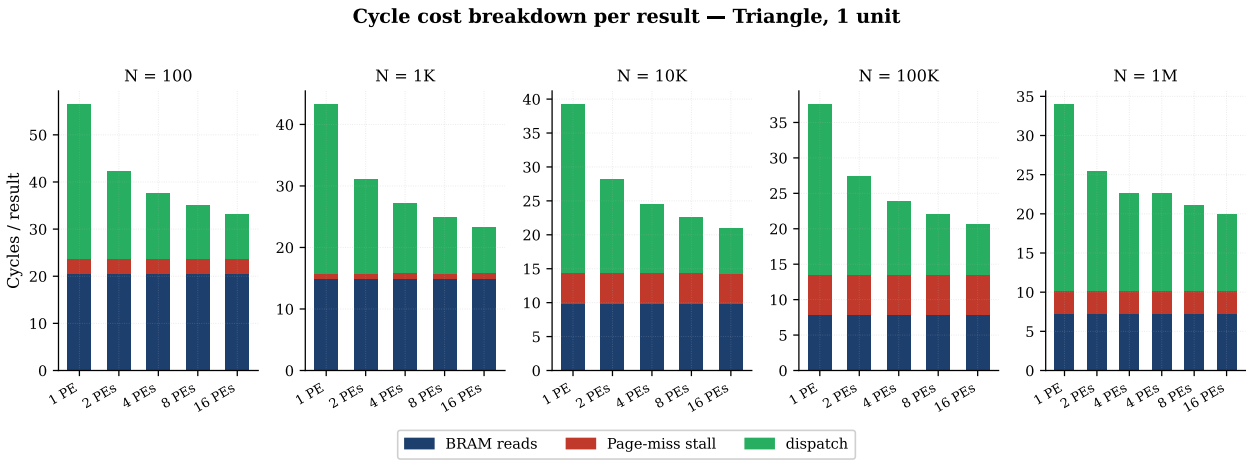


Figure 6.4: Cycle cost per result attributed to BRAM reads, page-miss stalls, and FSM overhead for the Triangle query across five problem sizes and PE configurations.

Chapter 7

Semi-Naive Evaluation: Design and Future Work

The accelerator described in Chapters 3 through 5 is a high-performance join engine, but it does not evaluate a Datalog program. Datalog evaluation requires repeatedly applying a set of rules until no new facts can be derived, a process known as fixpoint computation, and it requires efficiently identifying which joins need to be reevaluated after each round of new derivations.

As discussed in Chapter 2, semi-naive evaluation implements fixpoint computation by maintaining a delta for each relation and running delta joins at each iteration. The evaluator interacts with the join accelerator by constructing `DistTask`s and submitting them to the `Arbiter`. The initial task submitted loads the start address of the FULL set associated with each clause. Upon completion of an iteration that produces new facts and writes them to DELTA sets, the evaluator creates a set of tasks, where each task replaces the FULL memory of a single variable with the DELTA set, if the delta is nonempty. All other clause positions use their normal FULL start addresses.

In software systems such as Soufflé [4], the B-tree relation representation supports in-place insertion into an iterator. The sorted array used by this accelerator does not support in-place insertion: inserting a new node requires either shifting all subsequent nodes or rebuilding the array entirely. This difference motivates the three-tier memory architecture described in Section 7.2, which preserves correctness guarantees with a mechanism more suited to the BRAM access model.

7.1 Semi-Naive Evaluation

The semi-naive evaluator would sit above the arbiter and orchestrate the fixpoint computation. It is worth examining a concrete correctness problem that arises when mapping semi-naive evaluation onto this architecture, since the solution to the problem informs the design.

A correctness problem. Consider the rule $\mathit{Related}(b, a) :- \mathit{Related}(a, b)$ and suppose that in iteration k, the join derives the new fact $(1, 2)$ and places it in DELTA. The next join for iteration $k + 1$ uses this DELTA and produces $(2, 1)$, which is placed in the next DELTA.

The join for iteration $k+2$ uses $(2,1)$ and produces $(1,2)$ again. For the deduplication stage to identify $(1,2)$ as already known, it must be present in the FULL set, and it must have been added to the full set in between iteration k, when it is first found, and iteration $k+2$, when it is rediscovered. Merging DELTA into FULL after each iteration requries invalidating and refetching all cached copies of FULL across every unit, an operation whose cost grows to dominate the cost of the join. If we instead attempt to deduplicate against a stale FULL set, the $(1,2)$ tuple is treated as new, and a new join is executed to find $(2,1)$, which is treated as new for the same reason. The computation will never reach a fixed point as it cycles between these two values.

The NEW tier as a solution. The problem arises because facts derived during the current round exist only in DELTA, and DELTA is replaced at each iteration. Deduplication requires a record of everything derived **so far**, at the time of the current round, which grows monotonically across iterations without requiring FULL to be modified.

We introduce a third tier, **NEW**, that accumulates all facts derived during the current computation round. After each iteration, the current DELTA is merged into NEW before the next iteration begins. Deduplication then checks against NEW∪FULL, rather than FULL alone.

In this particular example, after iteration k, NEW contains $(1,2)$. After iteration $k+1$, NEW contains $\{(1,2),(2,1)\}$. When iteration $k+2$ derives $(1,2)$ and checks it against NEW∪FULL, it finds $(1,2)$ in NEW and correctly discards it, ending the computation. When no new DELTA set is produced, NEW is merged into FULL as a single, low-frequency operation, and unit caches are then invalidated for any pages that changed. The cost of the various memory-update operations that comprise this scheme is explored in the next section.

7.2 Memory-Update Operations

When the join engine produces results, those results must be sorted, deduplicated, and written back to the trie representation in `PageMem`, so that subsequent iterations can read them. This pipeline runs between join rounds and constitutes the memory-update side of the evaluator.

Sorting. Results arrive from the arbiter in arbitrary order and must be sorted to adhere to the trie representation requirements outlined in Chapter 3. Traversing a sorted array to find duplicates also makes deduplication easier than otherwise. The sort costs $O(D \log D)$ cycles, where D is the number of new facts in the current DELTA set.

Deduplication. The sorted facts are scanned against NEW∪FULL to remove any facts already known. Since both NEW and FULL are themselves sorted tries, we can check for membership by traversing the sorted new facts and existing tries in parallel, discarding any fact present in either. The cost is in $O(D + N + F_{\text{range}})$, where N is the size of the relevant portion in NEW and F_{range} is the corresponding range as a subset of FULL.

Trie construction and merging. Surviving facts are assembled into a new DELTA trie by a similar joint-sorting approach. At the end of each iteration, DELTA is merged into NEW

at cost $O(D + N)$, where N is the current size of NEW and D is the size of DELTA. At the end of a round, NEW is merged into FULL at cost $O(N + F)$, where F is the size of FULL, which represents a much larger merge cost than the B-tree-based merge in Soufflé; future work remains to replace the sorted array structure with a B-tree interface that can support $O(\log n)$ incremental merging while maintaining the seek performance of the flat array.

Implementing and validating this design is the natural next step toward a full hardware Datalog backend that evaluates a set of rules to completion by iteratively running these accelerated joins.

Chapter 8

Conclusion

The central claim that this thesis attempts to make is that the Leapfrog Triejoin is already a hardware algorithm. The sequential seek loop is a finite state machine with predictable memory-access patterns, while its decomposition into individual joins maps naturally to a PE-style architecture. Software implementations on general-purpose processors are unable to exploit fully these opportunities for parallelism, but an LFTJ engine built on configurable hardware can.

Evaluation confirms that the amortized cost of evaluating a single rule is lower than software counterparts, even at the lowest levels of configurable paralellism, and scales well as parallelism increases, especially for larger queries with more-variable depth. At scale, roughly 70% of cycles are spent in the cycle-to-cycle work of processing each matched node, rather than memory accesses. As the number of parallel workers increases, this processing overhead can be executed concurrently in a way that memory could not if it were the bottleneck.

Simulation results are encouraging for the transition to physical hardware. Future work will 1) test the design on FPGAs and establish performance metrics for workloads larger than what is feasible in software simulation and 2) build out a Datalog evaluator around this accelerated join engine that can take full advantage of the natural parallelism baked into the language.

Appendix A

Parameters and Datastructures

This appendix documents the central data structures used in the hardware implementation of the Leapfrog Triejoin accelerator.

A.1 Parameters

The following numeric parameters appear throughout the module interfaces. Each is a BSV `numeric type` passed at elaboration time.

Parameter	Meaning	Constraints
`pMaxTries`	Maximum number of distinct orderings of any relations present in the rule	
`pMaxClauses`	Maximum number of hypothesis clauses per rule	$\geq$ `pMaxTries`
`pMaxVars`	Maximum variables per rule	
`pMaxArgs`	Maximum iterators per leapfrog join level	$\leq$ `pMaxClauses`
`pMaxRules`	Maximum number of rules evaluated concurrently	1 for these single-rule programs
`pNumPEs`	Processing elements per unit	
`pNumSets`	Sets in the local cache	$\leq$ the number of page bits
`pNumWays`	Ways per set in the local cache	$\leq$ the number of page bits
`pNumLFTJUnits`	Number of accelerator units behind the arbiter	

A.2 Data Structures

TrieNode

The fundamental unit of storage in the flat trie representation. Every node occupies one 64-bit word in `PageMem`.

Field	Width	Description
`value`	32 bits	Sorted key stored at this node
`childStart`	26 bits	Global address of this node's child-subarray header
(padding)	6 bits	Reserved

TrieReprSnapshot

A complete snapshot of one `TrieRepr` iterator, used to save and restore iterator state across DFS stack push/pop operations.

Field	Type	Description
`iter.startIdx`	`GlobalNodeAddr`	Base address of current subarray
`iter.currPos`	`GlobalNodeAddr`	Current position within the subarray
`iter.len`	`GlobalNodeAddr`	Length of current subarray (header value + 1)
`node`	`TrieNode`	Cached value of the node at `currPos`
`trie`	`UInt#(TLog#(pMaxTries))`	trie identifier for this iterator

BindingCtx

The binding context carried by a `Task` as the DFS descends through variable levels.

Field	Type	Description
`valsPerVar`	`Vector#(pMaxVars, Bit#(32))`	Contretized value for each joined variable
`varStartPositions`	`Vector#(pMaxClauses, Vector#(pMaxVars, GlobalNodeAddr))`	Start address of the child subarray for each (clause, variable) pair

Task

The unit of work dispatched to a Leapfrog Triejoin Core. A task fully describes a single level of the DFS together with the binding context accumulated above it.

Field	Type	Description
`varLevelIdx`	`UInt#(TLog#(pMaxVars))`	Variable level currently being joined
`numVars`	`UInt#(TLog#(pMaxVars))`	Total number of variables in this rule
`ctx`	`BindingCtx`	Accumulated bindings and child addresses
`clauseVarsMap`	`Vector#(pMaxClauses, Bit#(pMaxVars))`	Bitmask of which variables each clause participates in
`clauseToTrie`	`Vector#(pMaxClauses, UInt#(TLog#(pMaxVars)))`	Trie index for each clause
`ruleIdx`	`UInt#(TLog#(pMaxRules))`	Rule identifier, returned with each result

DistTask

A `Task` augmented with prefetch hints for the arbiter.

Field	Type	Description
`distTask`	`Task`	Underlying task
`likelyPages`	`Vector#(pMaxArgs, PageTag)`	Pages the arbiter should prefetch before dispatch
`numIterators`	`UInt#(TLog#(pMaxArgs))`	Number of valid entries in `likelyPages`

JoinResp

The result tuple returned to the caller when the join finds a complete binding for all variables.

Field	Type	Description
`values`	`Vector#(pMaxVars, Bit#(32))`	Bound value for each variable
`numVars`	`UInt#(TLog#(pMaxVars))`	Number of valid entries in `values`
`ruleIdx`	`UInt#(TLog#(pMaxRules))`	Rule that produced this result

DFSEntry

One frame of the per-PE DFS stack in the Leapfrog Triejoin core. A frame is pushed when a child task is dispatched in-place on the same PE and popped when that child's join exhausts.

Field	Type	Description
`savedTask`	`Task`	Parent task to restore on pop
`savedIterPtr`	`UInt#(TLog#(pMaxArgs))`	Leapfrog Join rotation pointer at time of push
`savedNumArgs`	`UInt#(TLog#(pMaxArgs))`	Number of active iterators in parent join
`savedSnapshot`	`Vector#(pMaxArgs, TrieReprSnapshot))`	Iterator state for each active argument
`savedArgToClause`	`Vector#(pMaxArgs, Maybe#(UInt#( TLog#(pMaxClauses))))`	Clause index for each iterator slot

PageTag

Identifies a single page with some specific tire memory. Used as the key for both the per-unit `PageCache` and the arbiter-side `LikelyPageCache`.

Field	Type	Description
`trie`	`UInt#(TLog#(pMaxTries))`	Trie identifier
`page`	`PageID` (`UInt#(16)`	Page within that trie's global address space

PageTag

One streaming packet delivered to fill a cahce miss. A full page is delivered as `PktsPerPage` = 128 consecutive packets, each carrying 512 bits of node data.

Field	Type	Description
`tag`	`PageTag`	Page that current packet belongs to
`pktIdx`	`UInt#(TLog#(PktsPerPage))`	Packet index within that page
`first`	`Bool`	True on the first packet of a page transfer
`last`	`Bool`	True on the last packet of a page transfer
`data`	`Bit#(512)`	Eight packed `TrieNode` values (64 bits each)

References

[1] M. Bravenboer and Y. Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses." In: *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Oct. 2009, pp. 243–262.

[2] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. "Design and Implementation of the LogicBlox System." In: *Proceedings of the 2015 ACM SIGMOD Conference on Management of Data*. May 2015, pp. 1371–1382.

[3] T. L. Veldhuizen. "Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm." In: Mar. 2014, pp. 96–106.

[4] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. "On Fast Large-Scale Program Analysis in Datalog." In: *Proceedings of the 25th ACM International Conference on Compiler Construction*. Mar. 2016, pp. 196–206.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] S. Ceri, G. Gottlob, and L. Tanca. "What you always wanted to know about Datalog (and never dared to ask)." *IEEE Transactions on Knowledge and Data Engineering*, **1**(1), 1989, pp. 146–166. DOI: 10.1109/69.43410.

[7] H. Jordan, P. Subotić, D. Zhao, and B. Scholz. "A Specialized B-tree for Concurrent Datalog Evaluation." In: *Proceedings of the 24th ACM Symposium on Principles and Practice of Parallel Programming*. Feb. 2019, pp. 327–329.

[8] A. Atserias, M. Grohe, and D. Marx. "Size Bounds and Query Plans for Relational Joins." In: *Proceedings of the 49th IEEE Symposium on Foundations of Computer Science*. 2008, pp. 739–748.

[9] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. "Worst-Case Optimal Join Algorithms." In: *Proceedings of the 31st ACM Symposium on Principles of Database Systems*. Mar. 2012.

[10] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications." In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[11] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili. "Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs." In: *Proceedings of the VLDB Endowment*. Vol. 7. 4. 2014.