TreeWalker: Partial Evaluation for Grouped Tree-Ensemble Inference

Durmus Karatay
Shopify
durmus.karatay@shopify.com

Richard Newman
Shopify
richard.newman@shopify.com

Abstract

Many prediction tasks evaluate a trained tree ensemble on groups of rows that share feature values: discrete-time survival models expand each patient into G time steps, click-through-rate models score every item in a search session, and scenario analyses vary a few inputs while holding the rest fixed. Standard inference treats each row independently and ignores this input-side correlation.

We present TreeWalker, which instantiates partial evaluation for grouped tree-ensemble inference: constant features are static, varying features dynamic. TreeWalker walks each tree once per group, partitions a row bitmask at varying splits, and skips empty subtrees via an unsplit shortcut. It reads standard LightGBM and XGBoost models; training is unchanged.

We prove a structural work decomposition: per-tree work splits into the constant-projected subtree size $|T_c|$, G leaf writes, and a predicate-mask provisioning cost Q. For the trace evaluator, per-row work approaches $(d_v+1)/(d+1)$ as $G \to \infty$, giving asymptotic speedup $1/(1-\bar{\rho}_G)$.

On Intel, TreeWalker is $\sim 3\times$ faster than a row-independent traversal at the reference model size (T=500, L=8; G=16 for SUPPORT/FLCHAIN and empirical groups for Expedia) and 7.7–9.1$\times$ faster at G=128 on the survival datasets; on Arm, 2.5–2.9$\times$ and 5.5–6.0$\times$. On a scenario-analysis benchmark it is faster in all 16 configurations on both machines. For f64 models, outputs match treelite GTIL up to tree-ordering roundoff; for f32 models, TreeWalker's f64 leaf accumulation is closer than native f32 to an exactly rounded reference on every audited row.

1 Introduction

Gradient-boosted decision tree (GBDT) ensembles dominate tabular prediction [Chen and Guestrin, 2016]. In many applications the ensemble scores a *group* of rows that share most feature values, not a single row (§2.1).

What's missing. Fast tree-inference systems optimize either per-visit cost (compilation [Boehm, 2025, Nakandala et al., 2020], vectorization [NVIDIA, 2026], data layout [Asadi et al., 2014]) or per-walk cost (early termination via leaf-bitvector intersection [Lucchese et al., 2015]). None of these exploit *input-side* correlation across the rows of a group, even though 48–70% of features are constant within a group in our three factorial-grid datasets. Each row pays the full cost of its own walk, repeating the constant-feature prefix G times.

Partial evaluation across rows. We instantiate online partial evaluation [Jones et al., 1993, Futamura, 1999, Würthinger et al., 2017] for grouped tree inference: constant features are the static

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

inputs, varying features the dynamic, and the recursion the binding-time fold. TreeWalker[1] walks each tree once per group, maintaining a row bitmask (`u16`/`u32`/`u64`, or an array of words for wider groups). The constant prefix advances in $O(1)$ regardless of G. At each varying split the algorithm partitions the bitmask, prunes empty sides via an unsplit shortcut, and continues. To our knowledge no prior work exploits constant-feature partitioning across batched rows for tree inference; the closest related work (the QuickScorer family) optimizes per-walk cost via leaf-bitvector intersection within each tree, orthogonal to our axis.

Operational contract. TreeWalker targets workloads whose request semantics supply grouped rows; it does not construct groups from unstructured batches, and it offers no advantage when rows share no feature values. In the target workloads the task itself creates the group: discrete-time expansion materializes G rows from one entity, ranking sessions arrive grouped, and scenario analysis generates the variants. TreeWalker is therefore not a drop-in replacement for unstructured batch inference.

Contributions.

1. A grouped-input partial evaluation algorithm with constant-walk and row-mask propagation (§3).
2. A structural work decomposition theorem with two corollaries: a trace-evaluator asymptote of $1/(1-\bar{\rho}_G)$ and a forest-level provisioning bound for the default implementation (§4).
3. Empirical validation of the decomposition's predicted asymptote, a factorial-grid speedup study on three datasets against LightGBM, XGBoost, tl2cgen, lleaves, and QuickScorer, and a scenario-analysis benchmark on a fourth (§5).

2 Background

2.1 Grouped Prediction Tasks

A *grouped prediction task* evaluates a model on sets of rows that share a common subset of features. Each observation i produces a group of G_i input vectors $\{(x_c^{(i)}, x_v^{(i,g)})\}_{g=1}^{G_i}$, where x_c are *constant features* (identical across rows) and x_v are *varying features* (potentially distinct per row). Three concrete instances motivate our work: discrete-time survival models [Singer and Willett, 1993, Lee et al., 2018] expand each patient into G rows over a time horizon (covariates constant, time features varying); session-level CTR models [Expedia, 2013] score each item in a search session (user/query constant, item attributes varying); scenario analyses evaluate a row under multiple parameter perturbations. The *constant fraction* $|x_c|/(|x_c|+|x_v|)$ proxies shareable work: 48% (Expedia) to 70% (SUPPORT) across our factorial-grid benchmarks. The realized work-share, captured by $(d_v+1)/L$ in Table 1, depends on the model's split-feature distribution.

Grouped data in modeling. Multilevel and nested-data models [Gelman and Hill, 2006], tree models with random effects for clustered and longitudinal data [Sela and Simonoff, 2012, Hajjem et al., 2014, Sigrist, 2022], and GBDT forecasters for large collections of time series [Makridakis et al., 2022] specify models *for* grouped data. TreeWalker is orthogonal to that choice: it accelerates exact inference for tree ensembles on grouped rows, however they were trained. These settings are sources of grouped workloads, not competing methods.

Notation. T denotes the number of trees, L the maximum tree depth, G the group width (number of rows per observation), d the average root-to-leaf depth, f a feature index, and $r \in \{0, \ldots, G-1\}$ the row index within a group.

2.2 GBDT Inference

A gradient-boosted tree ensemble $F(x) = \sum_{t=1}^{T} h_t(x)$, where h_t is the t-th tree, predicts by traversing each tree from root to leaf and summing the leaf outputs. Per-row cost is worst-case $O(TL)$,

[1] Code and benchmark harness, at the version measured here: `https://github.com/Shopify/treewalker/tree/neurips2026-final`.

with average cost tracking $T \cdot d$ for average root-to-leaf depth $d \leq L$; evaluating rows independently multiplies both by G per group.

Axes of fast tree inference. Three cost axes characterize fast tree inference; prior work occupies the first two.

Per-visit cost. Compilation lowers interpretation overhead at each internal node: lleaves [Boehm, 2025] compiles trees to LLVM IR, tl2cgen [Cho, 2024] generates C from treelite [Cho and Li, 2018] models, and Hummingbird [Nakandala et al., 2020] reformulates trees as tensor programs; Yggdrasil-DF [Guillame-Bert et al., 2023] uses SIMD-aware ahead-of-time compilation. Vectorization fuses per-visit work across observations: RAPIDS FIL [NVIDIA, 2026] and ONNX Runtime's `TreeEnsemble` [ONNX, 2026] are representative. Cache-aware data layouts [Asadi et al., 2014] reduce per-visit memory cost.

Per-walk cost. Early-termination methods amortize predicate evaluation across the ensemble: QuickScorer [Lucchese et al., 2015], V-QuickScorer [Lucchese et al., 2016], and RapidScorer [Ye et al., 2018] use per-tree leaf-bitvectors, intersected across trees, to short-circuit traversal. The only open-source implementation [Manneschi, 2025] limits trees to ≤ 128 leaves and has neither categorical splits nor missing-value routing; it does not cover our $L \geq 8$ cells or exploit grouped inputs (comparison on an encoded variant in Appendix D).

Number-of-walks (this work). Partial evaluation against shared constants reduces the count of tree walks from GT toward T by fusing constant-feature prefixes across the rows of a group. The axis is orthogonal to per-visit and per-walk optimizations and composes with them (§6).

3 Algorithm

3.1 Recursive Partial Evaluation

Rows sharing constant features follow the same path through the tree until they encounter a varying split. TreeWalker exploits this by walking each tree once per group, not once per row. The walk maintains two pieces of state: a node index (position in the tree) and a row bitmask (which of the G rows are still active). Algorithm 1 states the conceptual recursion; Figure 1 illustrates the idea on a toy tree with a group of four rows.

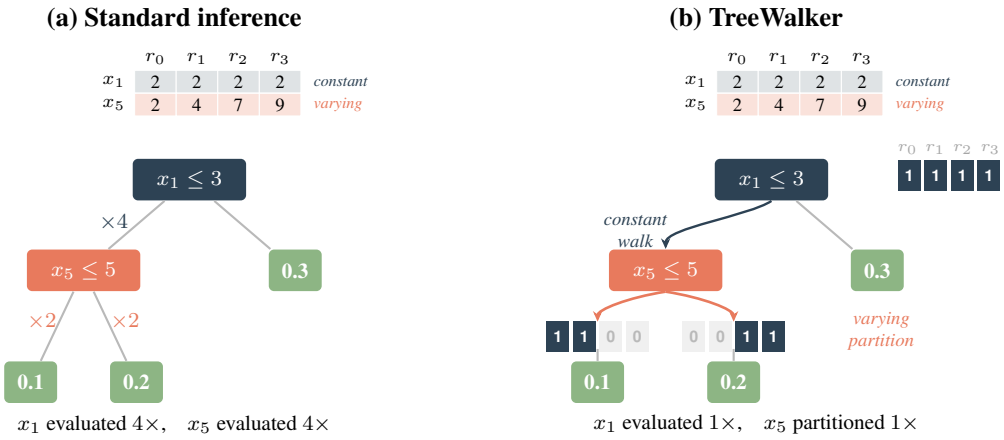

Figure 1: Worked example: a 4-row group with shared x_1=2 and varying x_5=$[2, 4, 7, 9]$. (a) Standard inference evaluates $x_1 \leq 3$ four times and $x_5 \leq 5$ four times. (b) TreeWalker evaluates the constant split once, carries the active-row mask $[1111]$, and partitions it at the varying split into $[1100]$ and $[0011]$ before leaf accumulation.

Constant walk. Starting from the root, TreeWalker follows the path determined by the constant features. At each node, if the split feature is constant across the group, the split outcome is identical for all rows and the algorithm advances to the appropriate child in $O(1)$. The walk continues until it reaches either a leaf or a varying split.

Varying split. When the walk encounters a node whose split feature varies across the group, TreeWalker partitions the active row bitmask: for each active row it evaluates the split condition and assigns the row to the left or right child's bitmask. The walk then recurses on both children, each carrying its subset of active rows.

Leaf accumulation. At a leaf, TreeWalker adds the leaf value to the output for every active row in the bitmask.

Algorithm 1 Recursive partial evaluation of one tree for a group of G rows. Analyzed in §4; the engineered variant (Algorithm 2, Appendix B) applies the constant-factor optimizations of §3.2.

```
1:  procedure PARTIALEVAL(tree, node, mask)        ▷ mask is a u16/u32/u64 or word array
2:      while node is not a leaf do
3:          if node.feature is constant for the group then
4:              node ← child matching constant value                    ▷ constant walk
5:          else
6:              (m_L, m_R) ← PARTITION(node, mask)                      ▷ varying split
7:              PARTIALEVAL(tree, node.left, m_L)
8:              PARTIALEVAL(tree, node.right, m_R)
9:              return
10:         end if
11:     end while
12:     for all bit r set in mask do                                    ▷ leaf accumulation
13:         results[r] ← results[r] + node.value
14:     end for
15: end procedure
```

3.2 Implementation Tuning

The conceptual algorithm admits four constant-factor optimizations that the implementation applies and the analysis (§4) references: an *unsplit shortcut* skips recursion into empty-mask children (Lemma 3); *heavy-path DFS* reorders nodes for cache locality, with light-side recursion bounding stack depth (Lemma 4); *monotonic early-termination* stops partitioning when the split feature is monotone along the row sequence (trace evaluator only); and *predicate precomputation* (sort + sweep per varying feature, bitmask lookup at each split) instantiates Corollary 2. The engineered variant (Algorithm 2) and data layout (16-byte node, contiguous arrays) appear in Appendix B.

4 Analysis

The analysis has three steps. Theorem 1 writes TreeWalker's per-tree work as an identity on every group: structural terms (visited splits, leaf writes) plus a single *predicate-mask provisioning cost* Q for supplying varying-feature partition masks. Corollary 1 instantiates Q for the trace evaluator and yields the asymptote $1/(1-\bar{\rho}_G)$. Corollary 2 instantiates Q for the default precompute evaluator and bounds forest-level work. §4.4 checks both against emitted counters.

4.1 Setup and Definitions

Fix a single binary tree τ of depth at most L. A *group* is a set $X = \{x_1, \ldots, x_G\}$ of rows sharing a constant-feature set $F_c \subseteq \{1, \ldots, n\}$; the remaining features F_v vary across rows. For each node u of τ, write $A(u) \subseteq [G]$ for the *active row set*: the rows that reach u under TreeWalker's partial evaluation of τ on X.

Definition 1 (Constant projection)**.** $T_c(\tau, X)$ is the directed subgraph of τ obtained by deleting the child not selected by the group's constant value at each constant-feature split, while retaining both children at each varying-feature split. By construction, every node visited under partial evaluation lies in $T_c(\tau, X)$.

Definition 2 (Trace counters)**.**

$$C(\tau, X) = \big|\{u \in \tau : u \text{ constant split}, A(u) \neq \emptyset\}\big|$$
$$V(\tau, X) = \big|\{u \in \tau : u \text{ varying split}, A(u) \neq \emptyset\}\big|$$

Definition 3 (Path statistics)**.** For a row $x \in X$, let $d_\tau(x)$ be its root-to-leaf depth in τ (number of split nodes on its path), and $d_{v,\tau}(x)$ the number of varying splits on that path. Define group averages $d(\tau, X) = \frac{1}{G}\sum_x d_\tau(x)$ and $d_v(\tau, X) = \frac{1}{G}\sum_x d_{v,\tau}(x)$.

Cost model. We work in an *abstract unit-cost model*: $W(\tau, X)$ counts split-node visits, leaf writes, and predicate-mask provisioning operations, each weighted as one unit. Wall-clock time is bounded above by W up to machine-dependent constants for each component (one comparison per split visit, one G-bit mask op per leaf write, etc.); we suppress these constants in the abstract bound. Charging one unit per G-bit mask operation holds for groups of up to 64 rows, whose mask is one 16- to 64-bit word. Wider groups use masks of K 64-bit words, which make each mask operation $\Theta(K)$. The engine takes groups of any width this way, in pieces of at most $1{,}024$ rows (Appendix G).

Predicate-mask provisioning. When queried at a varying split u with active mask $A(u)$, a *predicate-mask provisioning step* computes the partition $(A_L(u), A_R(u))$. Let $Q(\tau, X)$ be the aggregate cost of computing the queried predicate masks; Corollaries 1–2 instantiate two strategies.

4.2 Theorem

Theorem 1 (Structural work decomposition)**.** *For any tree τ and group X of size G, in the unit-cost model above,*

$$W(\tau, X) \;=\; C(\tau, X) + V(\tau, X) + G + Q(\tau, X) \;\le\; |T_c(\tau, X)| + G + Q(\tau, X). \tag{1}$$

Proof sketch. Every visited split node contributes one unit of control work (Lemma 1: the unsplit-enabled evaluator visits each node with non-empty active mask exactly once), giving $C + V$ in total. Every row reaches exactly one leaf per tree, contributing G leaf writes. Provisioning queries at varying splits aggregate to Q. Equality at unit cost follows; the structural inequality $C + V \le |T_c|$ is Lemma 2. Full proofs of Lemmas 1–4 are in Appendix C.

4.3 Corollaries

Corollary 1 (Trace evaluator)**.** *Under* trace provisioning*, where each varying predicate is evaluated per active row at the visit, $Q_{\text{trace}}(\tau, X) = \sum_{x \in X} d_{v,\tau}(x) = G \cdot d_v(\tau, X)$. Let the row-independent baseline be $W_{\text{full}}(\tau, X) = G \cdot (d(\tau, X) + 1)$. Then*

$$\frac{W_{\text{trace}}(\tau, X)}{G(d+1)} \;\le\; \frac{|T_c(\tau, X)|}{G(d+1)} \;+\; \frac{d_v + 1}{d+1}. \tag{2}$$

For group distributions $\{\mathcal{D}_G\}$ with $|T_c|, d, d_v$ uniformly bounded, define $1 - \bar{\rho}_G = \mathbb{E}[d_v{+}1]/\mathbb{E}[d{+}1]$. As $G \to \infty$ the structural term vanishes and the per-row trace ratio is asymptotically bounded by $1 - \bar{\rho}_G$, with asymptotic speedup $1/(1 - \bar{\rho}_G) = \mathbb{E}[d + 1]/\mathbb{E}[d_v + 1]$.

Corollary 2 (Default implementation)**.** *Under* precompute provisioning*, the forest-level strategy used in TreeWalker's default path (sort the G group values per varying feature, sweep against the deduplicated predicate set Π_f, emit a bitmask per predicate), for an ensemble $\mathcal{F} = \{\tau_1, \ldots, \tau_M\}$,*

$$Q_{\text{pre}}(\mathcal{F}, X) \;=\; O\!\left(\sum_{f \in F_v} \big(G \log G + |\Pi_f^{\text{num}}| + G \cdot |\Pi_f^{\text{cat}}|\big)\right), \tag{3}$$

paid once per group (not per tree). Total ensemble work is bounded by $\sum_{i=1}^{M}\big(|T_c(\tau_i, X)| + G\big) + Q_{\text{pre}}(\mathcal{F}, X)$.

Note on cross-tree sharing. With prefix grouping disabled and predicate sharing disabled, the ensemble bound reduces to the sum of per-tree statements of Theorem 1. The default forest path strictly improves on that sum: predicate sharing collapses $|\Pi_f|$ across trees, and prefix grouping reduces $\sum_i (C(\tau_i, X) + V(\tau_i, X))$ via shared visits at the prefix.

The $1/L$ floor. Under the per-row normalization $W/(G \cdot T \cdot L)$ of §4.4, the G leaf writes per tree alone fix a structural floor of $1/L$. The provisioning share does not vanish as $G \to \infty$: per row, the sort term grows as $\log G$ and the categorical term stays constant. We claim only that this share is small at measured G, because Q_{pre} is paid once per group across all T trees: in the widest measured groups the precompute ratio is 12% (FLCHAIN, 128 rows) to 14% (SUPPORT, 104 rows) above the $1/L$ floor (Table 1). That ratio uses the row-evaluation proxy for Q_{pre}, which omits the sort and predicate-sweep terms (Table 8), so it understates the true provisioning share.

4.4 Empirical Decomposition Validation

Table 1 reports the per-row work ratio $W(\mathcal{F}, X)/(G \cdot T \cdot L)$ at the reference configuration ($T = 500$, $L = 8$, LightGBM, Intel) on the fixed-G datasets. The sweep covers $G \in \{4, 16, 64, 128\}$ for both the trace and precompute provisioning strategies; the rightmost column gives the Corollary 1 asymptote. The ratio decreases monotonically in G for both strategies on both datasets and approaches its predicted limit: $(d_v + 1)/L$ for trace (dataset-dependent, d_v varies) and $1/L$ for precompute (dataset-independent). At $G = 128$ the trace ratio is within 5% of the predicted limit on both datasets; precompute is within 12% (FLCHAIN) and 14% (SUPPORT) of its $1/L$ floor. The table divides by $G \cdot T \cdot L$ rather than Corollary 1's $G(d + 1)$, so the trace limit becomes $(d_v + 1)/L$. Figure 5 (Appendix E) plots the same data on a log-G axis with horizontal asymptote lines.

		Group size G				Asymptote
Dataset	Evaluator	4	16	64	128	$G \to \infty$
SUPPORT	trace	0.674	0.432	0.361	0.351	*0.334*
SUPPORT	precompute (default)	0.507	0.226	0.154	0.143	*0.125*
FLCHAIN	trace	0.544	0.451	0.439	0.431	*0.417*
FLCHAIN	precompute (default)	0.420	0.213	0.153	0.140	*0.125*

Table 1: Per-row work ratio $W(\mathcal{F}, X)/(G \cdot T \cdot L)$ at the reference configuration ($T = 500$, $L = 8$, LightGBM, Intel) on the fixed-G datasets, swept across $G \in \{4, 16, 64, 128\}$. The final column gives the Corollary 1 asymptote under this baseline: $(d_v + 1)/L$ for trace (with d_v estimated from `partition_row_evals` at $G = 128$) and $1/L$ for precompute (the leaf-write floor). Both evaluators decrease monotonically in G and approach their predicted limits. XGBoost numbers (not shown) are within 6% of LightGBM except SUPPORT trace at G=64 (13.5% higher). SUPPORT's horizons of 64 and 128 hold 58 and 104 rows (§5.2). Expedia is excluded from this sweep because it has variable group sizes; it appears alongside the speedup grid in §5.

5 Evaluation

5.1 Experimental Setup

We evaluate TreeWalker on two survival datasets (SUPPORT, FLCHAIN) and one CTR dataset (Expedia) across a factorial grid of model parameters, and on a scenario-analysis workload (§5.6). The reference configuration (T=500, L=8, G=16) anchors the grid at the mid-sweep; G=16 is the median survival horizon. We compare TreeWalker against five external baselines (LightGBM [Ke et al., 2017] and XGBoost native predict, tl2cgen, lleaves, QuickScorer) plus an internal full-walk baseline. The full-walk baseline isolates the partial-evaluation contribution from data-layout effects, distinguishing *algorithmic speedup* (TreeWalker vs. full walk) from *system speedup* (TreeWalker vs. external baselines). Hardware (x86-64 Intel Xeon and Arm Neoverse V2 GCE VMs), software versions, compilation flags, the method roster (Table 6), and the blocked paired measurement protocol are in Appendix A; numerical precision is in Appendix A.3. Latency is the median of per-block medians in μs per observation (group); speedups are ratios of latencies with paired bootstrap 95% intervals, and p5–p95 over blocks shows dispersion.

5.2 Datasets

Table 2 summarizes the three factorial-grid datasets.

Table 2: Datasets. The constant fraction $|F_c|/(|F_c| + |F_v|)$ is a feature-level proxy for shareable work; the realized work-share is captured by the path statistic $(d_v + 1)/L$ in Table 1. The scenario-analysis dataset is described in §5.6.

Dataset	Type	Groups	Rows/Group	Const Feat.	Vary Feat.	Const%
SUPPORT	Survival	1,774	Swept (1–104)	14	6	70%
FLCHAIN	Survival	1,304	Swept (1–1,008)	8	6	57%
Expedia	CTR	10,000	Variable (5–37)	10	11	48%

SUPPORT [Knaus et al., 1995] and FLCHAIN [Dispenzieri et al., 2012] are standard survival datasets discretized into panel format. Each observation expands into G rows over a time horizon; six varying features encode the time step, the remaining steps, the squared elapsed and remaining fractions, one covariate–time interaction, and a smooth hazard-shape term; SUPPORT's discretization yields fewer bins than requested, so its horizons of 64 and 128 hold 58 and 104 rows. FLCHAIN also runs horizons of 256 to 1,024 (up to 1,008 rows). Expedia [Expedia, 2013] contains hotel search sessions where each session (group) consists of search results (rows). Constant features describe the user and search context; varying features describe each hotel result (price, star rating, location score). We subsample 50,000 sessions with 2–128 results (seed 42); an $80/20$ train/test split yields 10,000 test groups of 5–37 rows.

5.3 Algorithmic Speedup

We sweep a factorial grid: datasets $D \in \{\text{SUPPORT}, \text{FLCHAIN}, \text{Expedia}\}$, frameworks $F \in \{\text{LightGBM}, \text{XGBoost}\}$, tree counts $T \in \{50, 500, 1000, 2000\}$, max depths $L \in \{2, 4, 8, 16\}$, and group widths $G \in \{1, 2, 4, 8, 16, 32, 64, 128\}$ (survival only; Expedia uses empirical group sizes), yielding 2,449 measurement rows per architecture over 544 cells. Expedia cohorts, the encoded Expedia variant (Appendix D) and FLCHAIN's wide groups (§5.5) bring the grid to 712 cells.

Figure 2 shows the speedup of TreeWalker over the full-walk baseline at G=16 (survival) and empirical groups (Expedia), aggregated across frameworks. Speedup exceeds $1.0\times$ in every (T, L) heatmap cell, reaching $4.3\times$ (FLCHAIN, T=500, L=2); across the 544 cells (all $G \leq 128$, Intel), the largest algorithmic speedup is $9.1\times$ at G=128 on FLCHAIN with XGBoost; Arm speedups are $0.84\times$ Intel's at the median of these cells (§5.5). Outside the heatmap, speedup falls below $1\times$ mainly at G=1 (no amortization) and at G=2 where workspace overhead dominates — the boundary conditions predicted by Corollary 1.

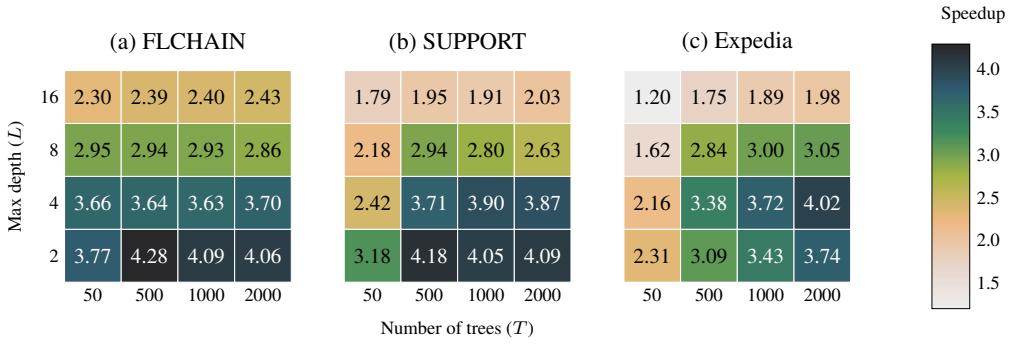

Figure 2: Algorithmic speedup across all (T, L) combinations on Intel, for (a) FLCHAIN (G=16), (b) SUPPORT (G=16), and (c) Expedia (empirical groups). Each cell: arithmetic mean of the two framework-specific speedups. Darker = higher speedup.

5.4 System Comparison

Figure 3 compares all methods at the reference. TreeWalker is $7.1\times$ faster than LightGBM native and $9.0\times$ faster than XGBoost native on SUPPORT, decomposing into a $2.5\times$ data-layout effect (full-walk vs. LightGBM native) plus a $2.9\times$ partial-evaluation effect on top. Against tl2cgen and

lleaves (native flags) TreeWalker is 2.1–2.3× and 1.1–1.2× faster at G=16 on LightGBM survival; the lleaves gap grows to 2.1–2.6× at G=128 as group structure amortizes. Among the 544 cells, TreeWalker's largest LightGBM-native system speedup is 54× (T=2000, L=8, G=128 on SUPPORT, 33 ms vs. 0.61 ms), combining algorithmic speedup with the data-layout/API gap to native LightGBM. On Expedia/XGBoost at the reference, TreeWalker is 6.0× faster (168 vs. 1,015 μs). QuickScorer computes a different function on nearly all of these models (no categorical splits or missing-value routing); the methods are on orthogonal axes (per-walk vs. number-of-walks), and Appendix D compares them on an encoded Expedia variant.

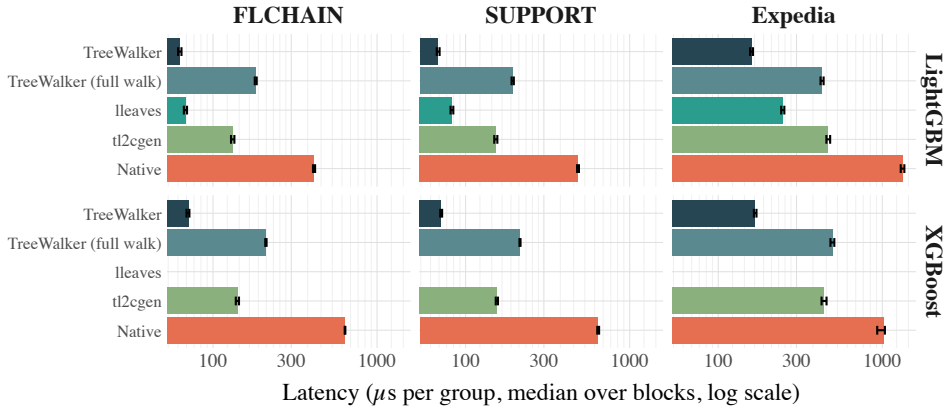

Figure 3: Per-observation latency (μs) for each inference method at the reference configuration (T=500, L=8, G=16), on Intel. Error bars: p5–p95 over the cell's 48 measurement blocks.

5.5 Group-Width Sensitivity

Figure 4 shows the central sensitivity: algorithmic speedup grows monotonically with G, from $\sim 1\times$ at G=1 to 7.7–9.1× at G=128 on both survival datasets. This is the wall-clock counterpart of the work-ratio convergence visualized in Figure 5 (Appendix E): the same x-axis (G). On the survival datasets, shallower trees give higher speedup (3.9–4.7× at L=2 vs. 1.9–2.6× at L=16); speedup declines from T=500 (2.9–3.0×) to T=2000 (2.5–2.9×). Arm Neoverse V2 speedups are 0.84× Intel Xeon's at the median cell. Figure 4(c) shows Expedia cohorts of 4–32 rows; speedup increases monotonically with cohort size. Detailed plots are in Appendix E.

On FLCHAIN's horizons of 256 to 1,024 (1,008 rows), whole groups reach up to 17.1× algorithmic speedup on Intel (10.1× on Arm), and 128-row chunks take 1.17–1.22× the latency of whole groups on Intel and 1.12–1.20× on Arm (medians over models and frameworks; Appendix G).

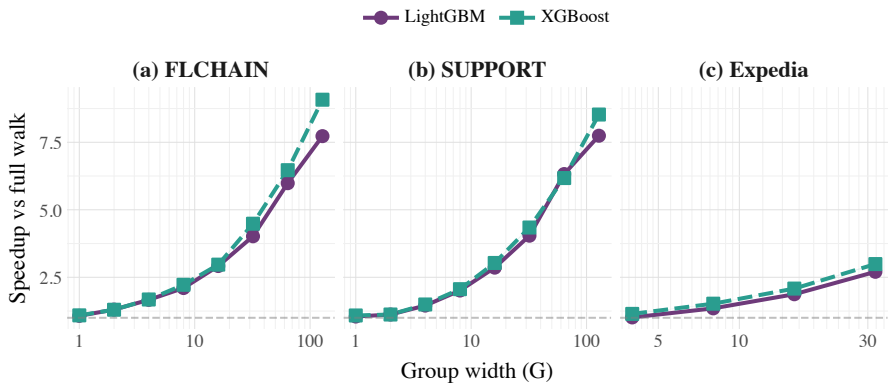

Figure 4: Algorithmic speedup vs. group width G (T=500, L=8, Intel). (a) FLCHAIN, (b) SUPPORT, (c) Expedia cohorts by size. Ribbons: 95% intervals.

5.6 Scenario Analysis

Scenario analysis generalizes the survival workloads: discrete-time expansion varies the time index, while a what-if query varies an arbitrary feature subset. We benchmark it on the UCI Default of Credit Card Clients dataset [Yeh, 2009] (30,000 rows, 23 numeric features; LightGBM T=500, L=8, seed 42). Each of 2,000 test rows expands into G variants that scale k selected features by log-uniform multipliers in $[0.5, 2]$ and keep the other $23-k$ features identical. Multiplicative shocks leave zero-valued entries constant, so k counts selected features, not realized varying ones: at k=8 the mean realized varying dimensionality is 6.80 of 8, and we measure d_v on the realized groups. We sweep $k \in \{1, 2, 4, 8\}$ and $G \in \{4, 16, 64, 128\}$ on both architectures (Appendix F).

Table 3: Scenario analysis at G=128: measured trace-evaluator work ratio $W/(G \cdot T \cdot L)$ against the Corollary 1 prediction under this baseline, $(d_v + 1)/L$, and algorithmic speedup over the same-layout full walk. Work counters are identical on both architectures. k counts selected perturbed features.

k	d_v	Measured ratio	Predicted $(d_v + 1)/L$	Speedup (Arm)	Speedup (Intel)
1	0.50	0.195	0.188	9.2×	15.2×
2	1.15	0.276	0.268	8.4×	12.8×
4	2.09	0.395	0.386	7.4×	10.3×
8	3.81	0.611	0.601	6.2×	8.1×

TreeWalker is faster than the same-layout full walk in all 16 cells on each architecture, and its speedup rises with G for every k. Speedup ranges from 1.96–15.2× at k=1 to 1.08–8.1× at k=8 on Intel, and from 1.77–9.2× to 1.04–6.2× on Arm. At G=128 the measured work ratio exceeds the Corollary 1 prediction by 1.7–3.7% across k (Table 3).

5.7 Ablation Study

We ablate eight binary flags (unsplit shortcut, monotonic early-termination, precomputation, tree ordering, prefix grouping, bitset interning, predicate sweep, predicate deduplication), each disabled in turn at one anchor per dataset (the reference for SUPPORT and Expedia, G=4 for FLCHAIN). The unsplit-shortcut ablation is the direct empirical analogue of Lemma 3: disabling it lets the algorithm descend into empty-mask subtrees, so the change in TreeWalker's emitted call count is at least $U(\tau, X) = V - B$, where B counts varying splits with both children non-empty. We do not claim a wall-clock multiplier from this lemma; the call-count effect is structural. Precomputation and predicate deduplication have the largest wall-clock effects (SUPPORT +21–39% and +28–34%, FLCHAIN +11% and +23–27%, Expedia +93–139% and +77–82%); tree ordering adds +3–8% on the survival anchors and at most +1% on Expedia, and monotonic early-termination, active only without precomputation, changes sign (−3% to +7%) (Figure 9, Appendix E).

6 Discussion

What the theorem buys. Theorem 1's exact equality lets the unsplit-shortcut ablation (§5.7) bound Lemma 3's call-count delta from below, and lets the precompute ablation swap Q_{trace} for Q_{pre} as the corollaries name.

Composability. Partial evaluation lives on a third axis (number of walks) and composes with the per-visit and per-walk axes from §2.2. Two compositions remain unimplemented: a QuickScorer-style inner kernel for the varying subtrees, and a compiled or vectorized inner loop layered under the per-group partition. The decomposition predicts that such compositions reduce $|T_c|$, the per-visit cost, and Q by independent factors.

Relation to compilation-based approaches. lleaves [Boehm, 2025] and tl2cgen [Cho, 2024] specialize a tree to compiled code at parse time; partial evaluation specializes a tree to an input *group* at predict time, with constant features as the binding-time signal; the two compose as the inner loop and the outer recursion of Algorithm 1. In the Jones, Gomard, and Sestoft [Jones et al., 1993] taxonomy this is the same *specialization-followed-by-residualization* pattern that Futamura's first

projection [Futamura, 1999] described for staged interpreters; we residualize against group-level constants rather than program-level ones.

Scope and limitations. The implementation is single-threaded CPU and reads scalar models with an identity or sigmoid output and sum or average aggregation; we measure binary-logistic GBDTs only. Evaluation covers two survival, one CTR/ranking, and one scenario-analysis workload. We do not measure the cost of constructing groups from unstructured batches; the target workloads supply groups by construction (§1). The f32 audit (Appendix A.3) shows a roundoff improvement over native f32, not a model-quality property. The per-tree theorem is the headline statement; cross-tree improvements (prefix grouping, predicate sharing) only strengthen the bound and are not separately theorized. The decomposition validates the work identity, not wall-clock constant factors. The binding-time partition must be known at predict time; mixed groups (partially overlapping constants) need finer-grained bookkeeping. The asymptote is reached only at $G \gtrsim 64$ in our experiments; small-group regimes pay workspace overhead.

Boundary cases. TreeWalker is slower than its own full walk in $33/712$ Intel cells ($78/712$ Arm), all in small groups: survival panels of $G \leq 2$ (Arm also $G{=}4$) and Expedia cohorts of 4–8 rows (Arm one of 16); 13 (28) are within $\pm 2\%$, and the worst is $1.29\times$ on Intel and $1.52\times$ on Arm. TreeWalker beats both native libraries in all 356 cells of each framework: narrowest $1.01\times$ (Intel) and $1.06\times$ (Arm) against LightGBM, on Expedia cohorts of 4, and $1.18\times$ and $1.50\times$ against XGBoost. Compiled baselines win mostly at small G: tl2cgen is faster in $122/712$ Intel cells ($70/712$ Arm), 92 of them at $G \leq 4$; lleaves in $94/356$ ($56/356$), 83 at $G \leq 8$.

7 Conclusion

We presented TreeWalker, an inference engine that exploits input-side correlation across the rows of a grouped prediction request through recursive partial evaluation, paired with a structural work decomposition (Theorem 1) that splits per-tree work into a control term, a leaf-write term, and a predicate-mask provisioning term. On Intel, across SUPPORT, FLCHAIN, and Expedia, TreeWalker delivers 2.7–$9.1\times$ algorithmic speedup over a row-independent baseline (2.5–$6.0\times$ on Arm), and it is faster than that baseline in all 16 scenario-analysis cells on each architecture; the trace-evaluator asymptote from Corollary 1 is within 5% of measurement at $G{=}128$ on both survival datasets.

Four extensions follow directly: (i) measured random-forest timings, since traversal and mask partitioning do not depend on the training procedure; (ii) hierarchical subgrouping for groups with partially overlapping constants; (iii) composing with QuickScorer could speed up shallow L, where QuickScorer is faster; (iv) predicate sharing across requests would amortize Q_{pre} across a serving session, a natural fit for survival-curve and ranking pipelines. GPU execution is out of scope: we target latency-critical CPU serving.

Acknowledgments and Disclosure of Funding

This work was funded by Shopify, the employer of both authors. The authors have no competing interests beyond their employment at Shopify.

References

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016. doi: 10.1145/2939672.2939785.

Simon Boehm. lleaves: Compiler for LightGBM gradient-boosted trees, based on LLVM (v1.4.1). `https://github.com/siboehm/lleaves/tree/v1.4.1`, 2025.

Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving.

In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 899–917, 2020. URL https://www.usenix.org/conference/osdi20/presentation/nakandala.

NVIDIA. RAPIDS Forest Inference Library (FIL), cuML 26.04. https://github.com/NVIDIA/cuml/tree/v26.04.00/python/cuml/cuml/fil, 2026.

Nima Asadi, Jimmy Lin, and Arjen P. de Vries. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2281–2292, 2014. doi: 10.1109/TKDE.2013.73.

Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. QuickScorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 73–82, 2015. doi: 10.1145/2766462.2767733.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice Hall, 1993. Free PDF: https://www.itu.dk/people/sestoft/pebook/.

Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. doi: 10.1023/A:1010095604496. Reprint of Futamura 1971; canonical reference for the Futamura projections.

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 662–676. ACM, 2017. doi: 10.1145/3062341.3062381.

Judith D. Singer and John B. Willett. It's about time: Using discrete-time survival analysis to study duration and the timing of events. *Journal of Educational Statistics*, 18(2):155–195, 1993. doi: 10.2307/1165085.

Changhee Lee, William R. Zame, Jinsung Yoon, and Mihaela van der Schaar. DeepHit: A deep learning approach to survival analysis with competing risks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pages 2314–2321, 2018. doi: 10.1609/aaai.v32i1.11842.

Expedia. Personalize expedia hotel searches — ICDM 2013. https://www.kaggle.com/c/expedia-personalized-sort, 2013. Kaggle / ICDM 2013 learning-to-rank competition dataset.

Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, 2006. doi: 10.1017/CBO9780511790942.

Rebecca J. Sela and Jeffrey S. Simonoff. RE-EM trees: A data mining approach for longitudinal and clustered data. *Machine Learning*, 86(2):169–207, 2012. doi: 10.1007/s10994-011-5258-3.

Ahlem Hajjem, François Bellavance, and Denis Larocque. Mixed-effects random forest for clustered data. *Journal of Statistical Computation and Simulation*, 84(6):1313–1328, 2014. doi: 10.1080/00949655.2012.741599.

Fabio Sigrist. Gaussian process boosting. *Journal of Machine Learning Research*, 23(232):1–46, 2022.

Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. M5 accuracy competition: Results, findings, and conclusions. *International Journal of Forecasting*, 38(4):1346–1364, 2022. doi: 10.1016/j.ijforecast.2021.11.013.

Hyunsu Cho. TL2cgen: Model compiler for decision tree ensembles (v1.0.0). https://github.com/dmlc/tl2cgen/releases/tag/1.0.0, 2024.

Hyunsu Cho and Mu Li. Treelite: Toolbox for decision tree deployment. In *Proceedings of Machine Learning and Systems (MLSys)*, 2018. URL https://mlsys.org/Conferences/doc/2018/196.pdf. Software used at v4.7.0: https://github.com/dmlc/treelite/releases/tag/4.7.0.

Mathieu Guillame-Bert, Sebastian Bruch, Richard Stotz, and Jan Pfeifer. Yggdrasil decision forests: A fast and extensible decision forests library. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 4068–4077, 2023. doi: 10.1145/3580305.3599933.

ONNX. ONNX: ai.onnx.ml.TreeEnsemble operator (opset v1.21.0). https://github.com/onnx/onnx/blob/v1.21.0/docs/Operators-ml.md#aionnxmltreeensemble, 2026. Unifies the deprecated TreeEnsembleRegressor and TreeEnsembleClassifier operators.

Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 833–836, 2016. doi: 10.1145/2911451.2914758.

Ting Ye, Hucheng Zhou, Will Y. Zou, Bin Gao, and Ruofei Zhang. RapidScorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 941–950, 2018. doi: 10.1145/3219819.3219857.

Leonardo Manneschi. quickscorer-rs: Rust implementation of QuickScorer, 2025. URL https://github.com/leomanne/quickscorer-rs. Commit 8314d03; accessed 2026-03-26.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

William A. Knaus, Frank E. Harrell, Joanne Lynn, Lee Goldman, Russell S. Phillips, Alfred F. Connors, Neal V. Dawson, William J. Fulkerson, Robert M. Califf, Norman Desbiens, et al. The SUPPORT prognostic model: Objective estimates of survival for seriously ill hospitalized adults. *Annals of Internal Medicine*, 122(3):191–203, 1995. doi: 10.7326/0003-4819-122-3-199502010-00007.

Angela Dispenzieri, Jerry A. Katzmann, Robert A. Kyle, Dirk R. Larson, Terry M. Therneau, Colin L. Colby, Raynell J. Clark, Graham P. Mead, Shaji Kumar, L. Joseph Melton III, and S. Vincent Rajkumar. Use of nonclonal serum immunoglobulin free light chains to predict overall survival in the general population. *Mayo Clinic Proceedings*, 87(6):517–523, 2012. doi: 10.1016/j.mayocp.2012.03.009.

I-Cheng Yeh. Default of credit card clients. UCI Machine Learning Repository, 2009. CC BY 4.0.

A Experimental Details

A.1 Hardware

All experiments run on Google Compute Engine VMs in `us-east4-a`, with SMT disabled (`threads_per_core=1`), the benchmark pinned to core 0, and both CPUs at their all-core turbo frequency. Intel's is set by `turbo_mode=ALL_CORE_MAX` (3.1 GHz); without it GCE runs C4 at unrestricted frequency. GCE always runs its Arm processors at their all-core turbo frequency and offers no setting on C4A.

Table 4: Hardware specifications. Per-core cache sizes; L3 shared. Both VMs in the same GCE zone, SMT disabled, at the all-core turbo frequency.

	Intel	Arm
GCE machine type	c4-standard-32	c4a-highmem-16
CPU	Xeon Platinum 8581C	Google Axion (Neoverse V2)
Frequency	2.3 GHz base, 3.1 GHz all-core turbo	3.0 GHz
Cores	16 (SMT disabled)	16
L1d / L1i (per core)	48 KiB / 32 KiB	64 KiB / 64 KiB
L2 (per core)	2 MiB	2 MiB
L3 (shared)	260 MiB	80 MiB
RAM	120 GB	128 GB
OS	Ubuntu 26.04.1 LTS	Ubuntu 26.04.1 LTS
Kernel	7.0.0-1011-gcp	7.0.0-1011-gcp

A.2 Software Versions

Table 5: Software versions (identical on both VMs).

Package	Version	Role
Rust	1.97.1	TreeWalker + sweep_bench
GCC	15.2.0	Native libraries, tl2cgen, lleaves linking
LightGBM	4.7.0	Model training + native predict
XGBoost	3.4.1	Model training + native predict
treelite	4.7.2	Model export (binary) + GTIL reference
tl2cgen	1.0.0	Compiled predict baseline
lleaves	1.4.1	LLVM-compiled predict baseline
llvmlite	0.50.0	LLVM 22.1 for lleaves
quickscorer-rs	`8314d03`	QuickScorer baseline

Table 5 lists all package versions. TreeWalker is compiled with `RUSTFLAGS="-C target-cpu=native"`. tl2cgen libraries use GCC at `-O3` with `-march=native`; the flags go into tl2cgen's compiler command, which ignores `CFLAGS`. lleaves compiles through the LLVM that llvmlite bundles, with `fblocksize=34` and code generation at `-O3` for the host CPU. lleaves's own pipeline also runs LLVM's IR optimizer, which inlines every tree into the forest loop; that optimizer does not finish on models of 500 trees or more, so for every model we compile the trees in four separate modules without it, and each tree remains a function call per row. LightGBM and XGBoost native libraries are built from pinned sources with CMake Release and `-march=native`, and predict on one thread. Every method runs on one thread, receives f64 row-major input, and converts it inside its timed call.

A.3 Numerical Precision

Every precision measurement in this section ran on the benchmark machine types (Table 4). TreeWalker's and native XGBoost's values were computed on each machine; the references (GTIL's outputs and the per-row stage oracle) were computed once, on the Intel machine that prepared the

Table 6: Inference methods compared in §5. TreeWalker (full walk) uses TreeWalker's data layout but evaluates rows independently, isolating partial evaluation from implementation differences. Each baseline with two interfaces is timed through the faster one per cell, chosen by an untimed probe.

Method	Description
LightGBM native	C API: `PredictForMat` or a loop of `PredictForMatSingleRowFast`; CMake Release with `-march=native`
XGBoost native	C API: one `PredictFromDense` call or a one-row loop; CMake Release with `-march=native`
tl2cgen (LightGBM+XGBoost)	Compiled C trees via treelite (GCC `-O3 -march=native`); its runtime or a loop of the generated `predict`
lleaves (LightGBM)	LLVM-compiled trees (`-O3` code generation for the host CPU; trees not inlined into the forest loop)
QuickScorer (LightGBM+XGBoost)	Bitvector-based predictor (`quickscorer-rs`), on the encoded Expedia variant and the five original-Expedia cells of LightGBM's T=50, L=2 model (Appendix D)
TreeWalker	Partial evaluation (this work)
TreeWalker (full walk)	Row-independent baseline, same data layout
TreeWalker (128-row chunks)	TreeWalker on 128-row chunks of wider groups (Appendix G)

cells. TreeWalker adds each row's leaf values in f64, in its loaded tree order, and applies the model's base score once after the sum.

For f64 models (LightGBM), TreeWalker is bit-exact against treelite's General Tree Inference Library (GTIL) when tree ordering is disabled: 0 ULP on all 30.2M rows of the 356 LightGBM cells, on both machines. Tree ordering, on by default, permutes the trees and so the order of the additions. It moves raw margins by at most 1.9×10^{-13} on Expedia and on survival panels of up to 256 rows, 2.0×10^{-10} on FLCHAIN's 512-row panels and 6.3×10^{-9} on its 1,008-row panels (at T=2000, L=16); outputs move by at most 7.4×10^{-15}, 9.3×10^{-15} and 2.1×10^{-13} respectively. The shifts are identical on Intel and Arm, as f64 additions in a fixed order must be.

f32 accumulation roundoff. For f32 models (XGBoost), TreeWalker compares inputs and thresholds in f32, so it takes the native split decisions, and accumulates the f32 leaf values in f64 rather than f32. Against an exactly rounded f64 reference (the correctly rounded sum of GTIL's per-tree outputs, with the same base score), TreeWalker's raw margins are closer than native XGBoost's on all 117,096 audited rows of the 292 XGBoost models (one cell each), with no ties, on both machines; TreeWalker differs from the reference on 7 rows, by at most 8.9×10^{-16}. Native XGBoost differs on every row, by at most 3.95×10^{-5}, and equals the reference rounded to f32 on 7.8% of them. The improvement is a roundoff property of f64 versus f32 accumulation, not a model-quality change.

A.4 Blocked Paired Measurement Protocol

All methods run inside a single Rust binary via C FFI, pinned to core 0 (`taskset -c 0`), timed with the same tick counter (`rdtsc` at 2.3 GHz on Intel, `cntvct_el0` at 1 GHz on Arm), with no Python in the measurement path. Each cell runs in a fresh process, so no method inherits a heap that earlier models churned. Per cell:

Validation. Before timing, every method's outputs on the timed groups are checked against TreeWalker's production predict, within the model's rounding bound (at least 10^{-13}, or 10^{-5} for a method whose output passes through f32); TreeWalker also matches its own full walk bit for bit and agrees within 10^{-12} (relative) with a correctly rounded per-row reference (the stage oracle) on a subset of each cell's groups. tl2cgen's known defect is checked apart (Appendix A.5). A method that fails is not timed.

Interface probe. Each baseline with two interfaces is timed through the faster one (Table 6), chosen by an untimed run on the warm-up batch. One-row groups use the single-row interface.

Sampling. A cell whose groups exceed 65,536 rows per round times a seeded sample of them, the same for every method, stratified by group size and giving every group the same inclusion probability, with at least 200 groups. The SUPPORT 104-row cells time 630–631 of 1,774

patients, Expedia's sessions 2,641–2,642 of 10,000, and FLCHAIN's 1,008-row cells 200 of 1,304. Medians are unweighted, and inference is for the cell's pool.

Rounds and blocks. The timed groups are split into 12 batches. A round gives each method its own phase: a warm-up on the batch it times last, then a timed pass over every batch, with the same groups in the same order for every method. Method order alternates across rounds, forward then reversed. A *block* is one batch in one round; its value is the median time per group in it.

Estimand. A method's latency is the median over its blocks, each weighted equally. A speedup is the ratio of two latencies. Its 95% interval comes from 1,000 bootstrap resamples that draw groups and rounds as crossed clusters, pair the two methods, and recompute the medians. p5–p95 over blocks describes dispersion and is not a confidence interval.

Stopping and re-timing. A cell times at least four rounds, two forward-and-reversed cycles, and stops when every method's per-round mean time per row has a relative standard error below 1%, or at 10 rounds or its time budget. A cell and mode whose speedup or ablation-effect interval is wider than $\pm 2\%$ is timed once more with twice the rounds, on the same VM; the longer attempt is reported, and both stay in the data.

XGBoost processes. On Intel, XGBoost's batch walk runs at one of two speeds from process to process, so each XGBoost cell also times XGBoost alone in six extra one-round processes. The reported number is the first process, timed and paired like every method; the faster mode across processes is a sensitivity check, against which TreeWalker is 5.3–9.0× faster at the reference (row-weighted), and the extra processes validate their first 32 groups.

Sentinel. A fixed cell (SUPPORT, LightGBM, T=500, L=4, G=16) runs at the start of each run and after every 25 cells, and a method whose latency drifts beyond 2% from the first measurement is flagged.

Cross-architecture reproducibility. Models are trained once on the Intel machine (seed 42) and the full artifact set (native models, treelite exports, test data, group offsets) is shared with the Arm machine, which checks every file's hash. Compiled baselines and `sweep_bench` are built per architecture from the shared model files. Work counters are identical on both machines.

A.5 Coverage

All 712 cells completed on both architectures, every required method validated, and no cell failed. tl2cgen reads each feature through a union of an integer and a double and tests the integer for its missing-value marker, so in an f64 model a finite value whose low 32 bits are all ones is read as missing. Such values occur in 68 of the 356 LightGBM cells; their 2,673 affected rows are checked apart, 2,579 of them differ, by at most 0.88, and tl2cgen stays timed as shipped. QuickScorer has no categorical splits or missing-value routing and its implementation limits trees to 128 leaves, so it is excluded up front from every LightGBM model with a categorical split, every cell with missing values a model may route right, and every model with a larger tree. It validates on the five original-Expedia cells of LightGBM's T=50, L=2 model and on 11 of the 16 cells of the encoded Expedia variant; the other 5 fail validation and are not timed (Appendix D).

A.6 Measurement Variability

Of the (cell, mode) pairs, 97% on Intel and 96% on Arm stopped at the 1% precision target and the rest at the round cap or, for two Arm pairs, the time budget. The 95% interval's half-width is 0.87% of the speedup at the median (cell, method) pair on Intel (0.98% on Arm) and about 2% at the 95th percentile. The re-timing rule re-timed 155 (cell, mode) pairs on Intel and 164 on Arm, and every attempt ran on its parent's VM. In the grid 94 (Intel) and 71 (Arm) pairs stay wider than $\pm 2\%$ after their re-time; in all but 2 (6) of them the widest interval is a baseline's speedup, most often tl2cgen's, which reaches $\pm 38\%$ on FLCHAIN's 1,008-row panels on Intel ($\pm 22\%$ at most on Arm). No method's sentinel latency drifted by more than 0.6%. The two ablation runs agree within 0.9 percentage points on Intel and 2.5 on Arm. As a sensitivity check, the row-weighted speedup (total time over total time) is within 0.5% of the latency ratio at the median panel or cohort cell, and 4–6% lower on Expedia sessions. Figure 3 shows p5–p95 over blocks as error bars.

B Implementation Details

B.1 Data Layout

The algorithm's constant factors depend on the node representation and memory layout. TreeWalker uses a compact 16-byte node (Table 7). The `value` field stores the split threshold, an inline categorical bitset or an offset into the bitset pool, or the leaf output, depending on node type. The varying-feature type (constant, monotonic, or non-monotonic) is embedded in the `flags` byte at parse time, avoiding per-node lookups during the hot loop. A row mask is one `u16`, `u32` or `u64` word for groups of up to 64 rows and an array of 64-bit words above that.

Table 7: Node layout (16 bytes).

Field	Type	Description
value	f64	Threshold, inline category bitset or bitset-pool offset, or leaf output
skip	i16	Tree-local index of the light child (-1 = leaf)
varying_pred_id	u16	Index into varying-predicate array
feature	u16	Feature index
flags	u8	Bitfield: default_left, categorical, inline_cat, heavy_is_left, walkable, vary_type (2 bits: constant, increasing, decreasing, non-monotonic)
cat_n_words	u8	Bitset word count (categorical splits)

All trees share a single contiguous node array, reordered at parse time via heavy-path DFS: the child with more training weight (LightGBM's sample count, XGBoost's hessian sum; the left child on a tie) occupies position `idx + 1` in memory, making the constant-walk fall-through path a sequential memory scan.

B.2 Engineered Variant of Algorithm 1

The conceptual algorithm in §3.1 treats both children of every varying split as recursive calls. The implementation augments this with four optimizations (named in §3.2); the engineered variant is shown in Algorithm 2.

Unsplit shortcut. When a varying-split partition leaves one side empty, the shortcut loops on the non-empty side without making a recursive call. This is the mechanism behind Lemma 3: TreeWalker makes exactly $B(\tau, X)$ non-tail recursive calls under the unsplit-enabled evaluator, eliminating the $U = V - B$ would-be empty descents.

Heavy-path DFS structural recursion. At parse time each split's structurally heavy child (the one with more training weight) is placed at `idx + 1` in the contiguous node array; the light child is reached via the `skip` pointer. The recursion descends into the light child and tail-loops on the heavy child, so the tail-loop side stays sequential in cache. The corresponding stack-depth bound is $O(L)$ (Lemma 4); we make no $O(\log_2 G)$ claim.

Monotonic early-termination. When a varying split uses a feature monotone along the row sequence ($r = 0, \ldots, G-1$), the partition stops comparing thresholds at the cutoff, since the rows before and after it go opposite ways; missing values are still checked and follow the default direction. It applies to the trace evaluator only, since the precompute evaluates every predicate the same way. It is a constant-factor reduction in Q_{trace} whose wall-clock effect changes sign across the ablation anchors, -3% to $+7\%$ on Intel (-4% to $+7\%$ on Arm; §5.7).

Predicate precomputation. At each varying split TreeWalker must evaluate the split predicate for every active row. Without precomputation this costs $O(P \cdot G)$ per group, where P is the total varying-predicate count across all trees. We reduce this to the cost in Eq. (3), $O(F_v \cdot G \log G + |\Pi^{\text{num}}| + G \cdot |\Pi^{\text{cat}}|)$, by precomputing every varying predicate before traversal: at parse time, extract the unique predicates per varying feature; at predict time, sort the G row values once per feature and sweep a two-pointer merge against the sorted thresholds, emitting one bitmask per predicate in amortized $O(1)$. Tree traversal then looks up precomputed masks by predicate index. This strategy is the Q_{pre} instantiation analyzed in Corollary 2.

Algorithm 2 Engineered variant of Algorithm 1: adds the unsplit shortcut, heavy-path DFS recursion, and tail-call elimination. Constant walk on lines 9–12; leaf accumulation on lines 3–7; varying partition on lines 15–22.

```
1:  procedure PARTIALEVALIMPL(tree, node, mask)
2:      loop                                              ▷ constant walk + tail-loop site
3:          if node is leaf then
4:              for all bit r set in mask do
5:                  results[r] ← results[r] + node.value
6:              end for
7:              return
8:          end if
9:          if node.feature is constant then
10:             node ← child matching constant value
11:             continue
12:         end if
13:         break                                         ▷ hit a varying split
14:     end loop
15:     (m_L, m_R) ← PARTITION(node, mask)                ▷ may use precomputed mask
16:     if m_L = 0 then node ← right child; mask ← m_R; loop   ▷ unsplit shortcut
17:     else if m_R = 0 then node ← left child; mask ← m_L; loop
18:     else
19:         (light, heavy) ← structurally light / heavy children   ▷ heavy-path DFS
20:         PARTIALEVALIMPL(tree, light.node, light.mask)
21:         node ← heavy.node; mask ← heavy.mask; loop    ▷ tail-call
22:     end if
23: end procedure
```

Production kernel. Production predict inlines the first walk of each tree into the tree loop; with prefix grouping, on by default with depth 2, that walk starts where the tree's group of shared prefixes ends. When the constant walk reaches a leaf before any varying split, as in every tree of the measured $G{=}1$ cells, whose models split on no varying feature, the leaf's value is added to every row directly, with no recursive call. In models of $4{,}096$ nodes (64 KiB) or more, at least the L1 data cache of either machine, the loop prefetches each next tree's start node. Neither changes an output or a work counter.

C Proof of Theorem 1

We prove four supporting lemmas and then close out the bound stated in §4.1. The setup, definitions $(A(u), C, V, T_c, Q)$, cost model, and provisioning step are as in §4.1. We additionally use, only in Lemma 3, the partition counts

$$B(\tau, X) = \big|\{u : u \text{ varying}, |A_L(u)|{>}0 \wedge |A_R(u)|{>}0\}\big|, \qquad U(\tau, X) = V(\tau, X) - B(\tau, X).$$

Lemma 1 (Active-mask disjointness)**.** *Under the default unsplit-enabled evaluator, for every node $u \in \tau$ with $A(u) \neq \emptyset$, TreeWalker visits u exactly once during the partial evaluation of τ on X.*

Proof. By induction on the depth of u in τ. The root has $A(\mathrm{root}) = [G]$ and is visited once at the call entry. At a constant split u with $A(u) \neq \emptyset$, the algorithm forwards $A(u)$ unchanged to the deterministic child (the other child is excluded from T_c). At a varying split u with $A(u) \neq \emptyset$, the algorithm partitions $A(u)$ into disjoint masks $(A_L(u), A_R(u))$ with $|A_L| + |A_R| = |A(u)|$, and *continues once on each non-empty side*: one recursive call to one child plus a tail-loop to the other if both are non-empty (Lemma 3 for the unsplit-enabled call-count rule). The unsplit shortcut ensures that empty-mask sub-calls are not made: a child with empty mask is skipped, so no empty-mask visit occurs. By induction, every node reached with non-empty active mask is visited exactly once. □

Remark (ablation). The unsplit-disabled ablation (`disable_unsplit=true`) makes a recursive call regardless of whether the child mask is empty, so empty-mask entries are possible. The theorem and

lemmas in this section are stated for the default evaluator; ablation-path counters lie above these bounds, not below.

Lemma 2 (Visited-set bound)**.** *For every group X, $C(\tau, X) + V(\tau, X) \leq |T_c(\tau, X)|$.*

Proof. Every split node visited by TreeWalker (with non-empty active mask) lies in $T_c(\tau, X)$ by construction, since the constant projection retains every branch a non-empty mask can reach. By Lemma 1 each such visit is counted once, so $C + V$ is the count of internal split nodes of T_c that have non-empty masks under the partial evaluation. The internal split nodes of T_c number $|T_c| - L_c(\tau, X)$, where L_c is the count of T_c leaves; the visited-with-non-empty-mask set is a subset of those internal nodes, hence $C + V \leq |T_c| - L_c \leq |T_c|$. □

Remark (tightness). The bound is conservative. T_c retains both subtrees below every varying split, even when the partition sends all active rows to one side. The looseness can be $\Theta(|T_c|/L)$ in adversarial cases (e.g. a varying split at the root with all rows going one way, leaving a large unvisited subtree on the other side). The conservative form suffices for Theorem 1; tighter bounds follow from $C(\tau, X) + V(\tau, X)$ themselves.

Lemma 3 (Unsplit call-count)**.** *Under the default unsplit-enabled evaluator, TreeWalker's partial evaluation of τ on X makes exactly $B(\tau, X)$ non-tail recursive calls. The unsplit shortcut eliminates $U(\tau, X) = V(\tau, X) - B(\tau, X)$ continuations into empty subtrees.*

Proof. At a varying split u with $A(u) \neq \emptyset$: (i) if both children have non-empty masks (u counted in B), the algorithm recurses once on one child and tail-loops on the other, producing exactly one non-tail recursive call; (ii) if exactly one child has a non-empty mask (u counted in U), the unsplit shortcut tail-loops on the non-empty child and makes no recursive call. The total non-tail call count is therefore $B(\tau, X)$, and all U would-be empty-side descents are eliminated. Lemma 3 bounds call count, not row work; it is not used in the proof of Theorem 1 and is recorded here for completeness (the unsplit ablation adds at least U calls, §5.7). □

Lemma 4 (Stack depth)**.** *Stack depth during partial evaluation of τ is at most L (the maximum tree depth).*

Proof. Each recursive call descends to a child of the current node. The non-recursive (tail-looped) side does not increase stack depth. Since the height of τ is at most L, the depth of the recursion is at most L. □

Remark (heavy-path DFS). A tighter $O(\log_2 G)$ stack-depth bound would require the recursive call to always descend to the side with smaller active popcount. TreeWalker instead recurses on the *structurally light* child (under heavy-path DFS, fixed at parse time) so that the tail-loop side stays contiguous in cache. We make no $O(\log_2 G)$ claim.

Proof of Theorem 1. Every visited split node contributes one unit of control work (one visit, by Lemma 1), giving $C(\tau, X) + V(\tau, X)$ in total. Every row reaches exactly one leaf per tree (the active-mask partition is exact: $\sum_{\ell \in \text{leaves}(\tau)} |A(\ell)| = G$), contributing exactly G leaf writes. Provisioning queries at varying splits aggregate to $Q(\tau, X)$. Equality at unit cost follows. The structural inequality $C + V \leq |T_c|$ is Lemma 2. □

Remark. Theorem 1 is stated for the per-tree evaluator. The default forest path includes a prefix-grouping pass that begins per-tree evaluation at a shared continuation node rather than the root. Prefix grouping only lowers the structural control term at ensemble level by sharing visits across trees; the per-tree theorem remains an upper bound on each summand of the ensemble cost.

Proof of Corollary 1. Counting provisioning queries: each row visits each varying split on its path exactly once. Summing per-row contributions gives $Q_{\text{trace}} = \sum_x d_{v,\tau}(x) = G \cdot d_v$ (an exact identity, not an upper bound). Substituting Theorem 1 with $Q_{\text{trace}} = G \cdot d_v$ and dividing by $G(d + 1)$ yields (2), since $|T_c| + G + Gd_v = |T_c| + G(d_v + 1)$. The asymptote follows from $|T_c|/(G(d+1)) \to 0$ as $G \to \infty$ under the boundedness assumption; the leaf-inclusive denominator avoids a spurious $1/d$ term. □

Proof of Corollary 2. Per varying feature: sorting the G group row values is $O(G \log G)$. Numeric predicates use a two-pointer merge in $O(|\Pi_f^{\text{num}}|)$ amortized over the sort. Categorical predicates evaluate per row in $O(G \cdot |\Pi_f^{\text{cat}}|)$. Summing over $f \in F_v$ gives Eq. (3). Each tree's per-visit provisioning access is $O(1)$ (bitmask lookup), so the per-tree control term in Theorem 1 reduces to $C(\tau_i, X) + V(\tau_i, X) + G$. Summing over trees and adding the shared provisioning term Q_{pre} yields the bound. □

Empirical validation by counter. The decomposition (1) is an identity on every group, not a predicted-vs-observed ratio. We validate componentwise against TreeWalker's emitted counters:

Term	Counter (proxy)	Source
$C(\tau, X)$	`constant_steps`	per-tree stats (exact)
$V(\tau, X)$	`varying_splits`	per-tree stats (exact)
G	group size	known (exact)
Q_{pre}	`precompute_row_evals` + `partition_row_evals`	per-group stats (proxy)
$\lvert T_c(\tau, X)\rvert$	offline replay against (τ, X)	one-time post-process

Table 8: Trace counters used to validate Theorem 1 componentwise. C, V, G are tracked exactly; Q_{pre} is a row-evaluation proxy that misses the $G \log G$ sort term and the predicate-set-size terms in Eq. (3).

D QuickScorer Head-to-Head

QuickScorer [Lucchese et al., 2015] represents each tree as a bitvector of leaves and intersects per-feature interval-bitvectors across all trees to short-circuit traversal. The only available open-source implementation [Manneschi, 2025] caps trees at $\leq$ 128 leaves, an implementation choice rather than a limit of the QuickScorer family. It also reads no categorical splits, sends every missing value left, and compares thresholds in f32. On our datasets that is a different function: the survival models split on categorical features, and Expedia has missing values. The exception is Expedia's LightGBM model at T=50, L=2, which has no categorical split and sends no missing value right: on its five cells (sessions and cohorts of 4–32 rows) QuickScorer validates and is faster than TreeWalker on both machines, by 1.3–2.5× on Intel and 1.1–2.1× on Arm.

We therefore compare on an encoded Expedia variant, with every missing value replaced before training by a constant below its column's training minimum, the same in train and test, so that QuickScorer's left-going default never applies. The encoding does not make QuickScorer compute LightGBM's function, because it still compares thresholds in f32: 5 of the 16 cells ($T \in \{50, 500, 1000, 2000\}$, $L \in \{2, 4\}$, both frameworks) fail validation and are not timed. Table 9 reports the other 11. QuickScorer is faster than TreeWalker in all 11 on Intel, by up to 1.75× on the smallest models, and in 5 on Arm, where TreeWalker is faster in the L=4 cells from T=500 on and at T=2000, L=2 (1.03–1.72×). QuickScorer's f64-to-f32 input conversion is inside its timed call, as every method's conversion is.

The two methods optimize complementary axes (per-walk vs. number-of-walks) and could in principle compose: a partial-evaluation outer loop with a QuickScorer-style inner kernel for the varying subtrees. We have not implemented this composition.

Table 9: QuickScorer vs. TreeWalker on the encoded Expedia variant, in the 11 cells whose QuickScorer outputs validate. Latencies are Intel's, in μs per session. QS/TW is QuickScorer's latency divided by TreeWalker's; values below 1 mean QuickScorer is faster. The two 0.98 ratios on Intel are within $\pm 2\%$.

Framework	T	L	TreeWalker (μs)	QuickScorer (μs)	QS/TW (Intel)	QS/TW (Arm)
LightGBM	50	2	5.38	3.44	0.64	0.75
LightGBM	500	4	67.13	61.22	0.91	1.43
LightGBM	1000	4	107.35	105.35	0.98	1.61
XGBoost	50	2	6.85	3.90	0.57	0.69
XGBoost	50	4	16.95	10.35	0.61	0.88
XGBoost	500	2	34.85	24.09	0.69	0.77
XGBoost	500	4	72.67	65.01	0.89	1.45
XGBoost	1000	2	55.77	45.29	0.81	0.89
XGBoost	1000	4	116.57	111.23	0.95	1.60
XGBoost	2000	2	91.41	82.67	0.90	1.03
XGBoost	2000	4	201.93	198.71	0.98	1.72

E Supplementary Figures

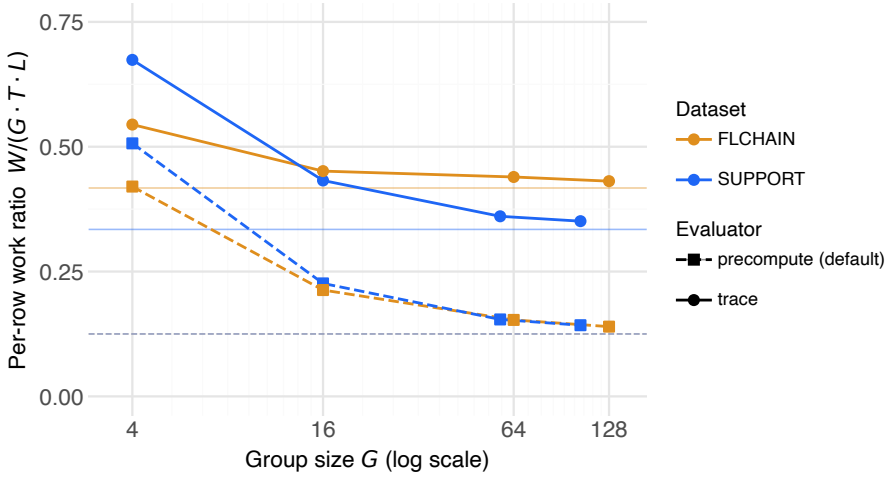

Figure 5: Empirical convergence of the per-row work ratio $W(\mathcal{F}, X)/(G \cdot T \cdot L)$ vs. group size G (log scale) at the reference configuration ($T = 500$, $L = 8$, LightGBM) on the fixed-G datasets; the counts are the same on both machines. Solid lines: trace provisioning (no precompute). Dashed lines: precompute provisioning (default). Horizontal reference lines mark the limits under this baseline: the Corollary 1 asymptote $(d_v + 1)/L$ for trace (per-dataset, in matching color) and the structural leaf-write floor $1/L$ for precompute (dataset-independent, the lowest line at 0.125). Trace approaches its limit from above as row-dependent predicate work scales with G; precompute drops faster because its per-group provisioning cost is spread across more rows. Visualizes the data in Table 1 (§4.4).

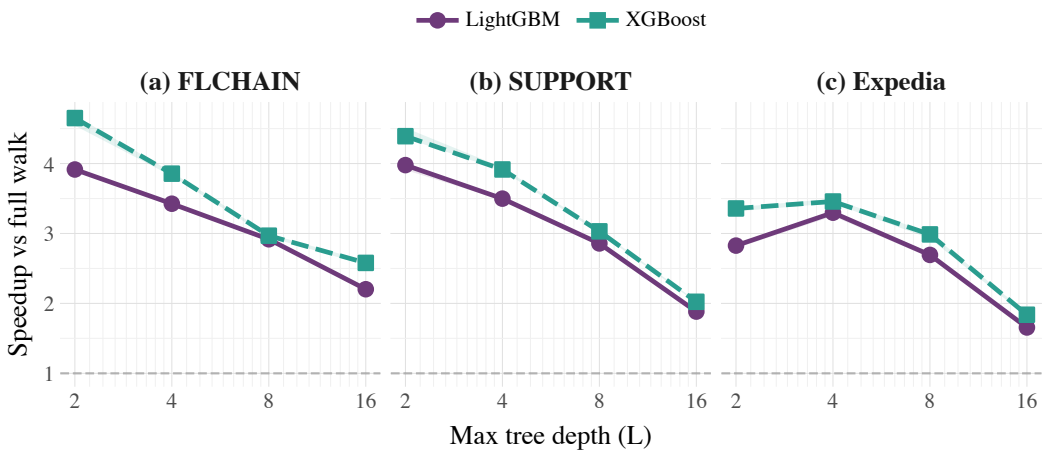

Figure 6: Algorithmic speedup vs. tree depth L (T=500, G=16, Intel). On the survival datasets shallower trees yield higher speedup. Ribbons: 95% intervals.

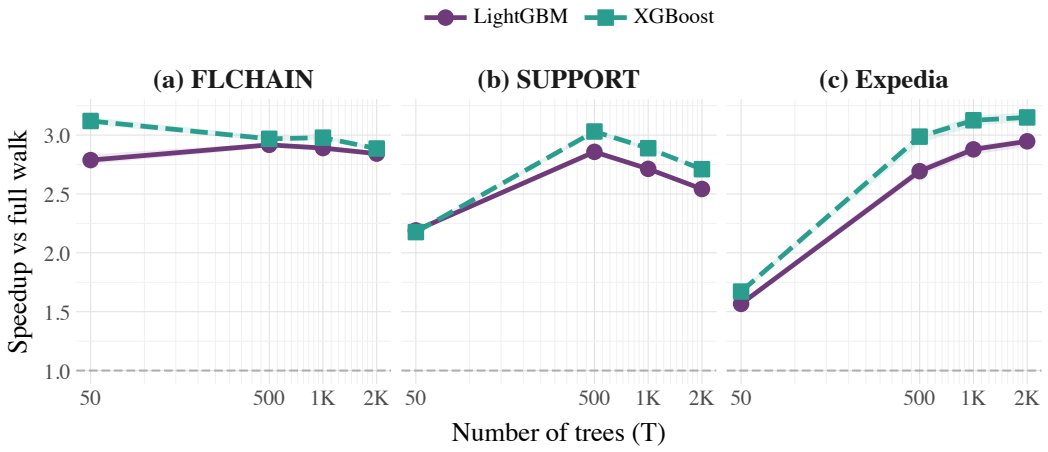

Figure 7: Algorithmic speedup vs. ensemble size T (L=8, G=16, Intel). Ribbons: 95% intervals.

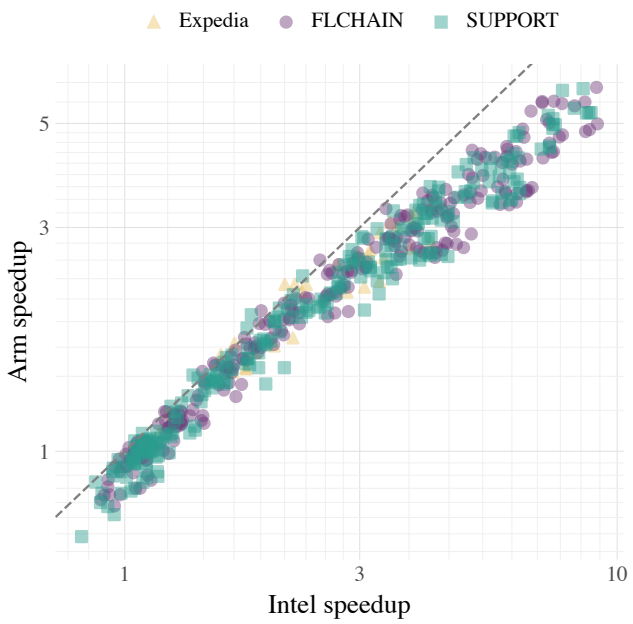

Figure 8: Intel vs. Arm algorithmic speedup for the 544 cells of the factorial grid (log-log). Points above the diagonal indicate higher Arm speedup; dashed line = parity.

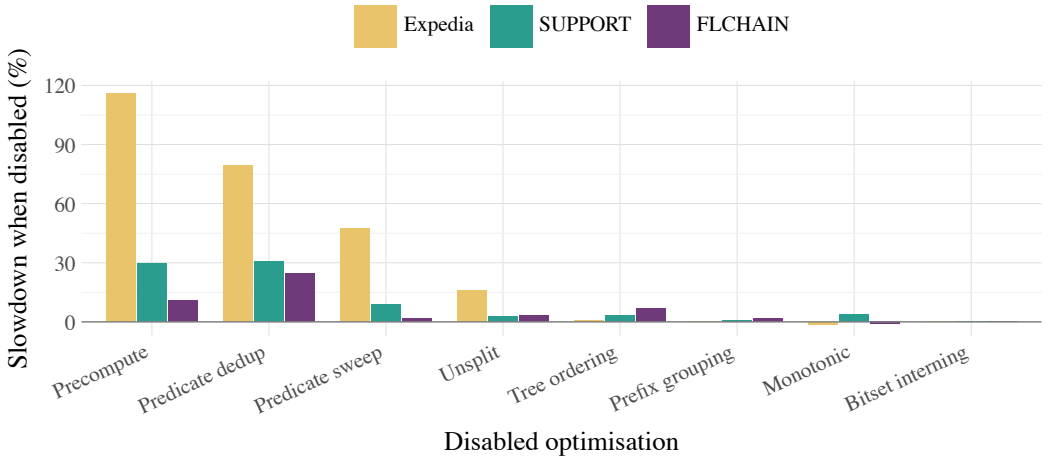

Figure 9: Slowdown when each flag is disabled in isolation, in percent: the disabled latency over a research build of the kernel with every flag on, minus one (monotonic early-termination: against the build with precomputation off), at T=500, L=8 on Intel. The research build is within -4% to $+6\%$ of production at the anchors. G is the reference G=16 for SUPPORT, empirical sessions for Expedia, and G=4 for FLCHAIN. Flags are sorted by the geometric mean slowdown over the three anchors. Bars: mean over the frameworks and both ablation runs, which agree within 0.9 percentage points.

F Scenario-Analysis Benchmark

Workload. The dataset is UCI Default of Credit Card Clients [Yeh, 2009] (CC BY 4.0, OpenML 42477): 30,000 rows, 23 numeric features, binary target. We train one LightGBM model (T=500, L=8, `num_leaves`= 255, binary objective, seed 42) on a deterministic train split and draw 2,000

base rows from the test split. Each base row expands into G variants; variant 0 is the unperturbed row. The perturbable pool is credit limit (`x1`), bill amounts (`x12`–`x17`), and payment amounts (`x18`–`x23`); the nested k sets are ordered by descending gain importance within the pool. Multipliers are log-uniform in $[0.5, 2]$ (`np.random.default_rng(42)`). All non-selected features are bit-identical across the group.

Zero-valued features. A selected feature with base value 0 stays 0 under a multiplicative shock, which is common for the payment amounts. At k=8, 47.15% of groups contain at least one selected zero-valued feature, 15.00% of selected entries are zero, and the mean realized varying dimensionality is 6.80 of 8. We therefore measure d_v on the realized groups via the trace evaluator rather than deriving it from k.

Protocol and correctness. Timing follows the protocol of §A.4 (at least four rounds, 1% precision target), in the same campaign and on the same VMs as the main grid. TreeWalker and its full walk are validated as there, against production predict within the model's rounding bound (at least 10^{-13}) and against the per-row reference. Models are trained once on Intel and shared, so work counters are identical across architectures. Latencies are in μs per scenario set (one base row and its G variants), not per variant; Table 10 reports unit-free speedups and ratios.

Table 10: Full scenario-analysis sweep. Speedup is algorithmic speedup over the same-layout full walk. Work ratios are $W/(G \cdot T \cdot L)$ under the trace and default (precompute) evaluators; the trace prediction under this baseline is $(d_v + 1)/L$ with $L = 8$.

k	G	Speedup (Arm)	Speedup (Intel)	Trace ratio	Predicted	Precompute ratio
1	4	1.77×	1.96×	0.413	0.188	0.351
1	16	4.46×	5.16×	0.245	0.188	0.183
1	64	7.74×	11.79×	0.202	0.188	0.140
1	128	9.21×	15.15×	0.195	0.188	0.133
2	4	1.47×	1.60×	0.502	0.268	0.359
2	16	3.72×	4.16×	0.329	0.268	0.186
2	64	6.90×	10.00×	0.284	0.268	0.141
2	128	8.38×	12.82×	0.276	0.268	0.133
4	4	1.21×	1.29×	0.635	0.386	0.375
4	16	2.93×	3.21×	0.453	0.386	0.193
4	64	5.97×	8.01×	0.403	0.386	0.143
4	128	7.37×	10.32×	0.395	0.386	0.135
8	4	1.04×	1.08×	0.875	0.601	0.401
8	16	2.40×	2.57×	0.679	0.601	0.205
8	64	4.97×	6.17×	0.622	0.601	0.148
8	128	6.22×	8.12×	0.611	0.601	0.138

G Groups Wider than 128 Rows

Per-chunk decomposition. A group X of width $G > 128$ splits into $K = \lceil G/128 \rceil$ chunks $X_1, \ldots, X_K$. Applying Theorem 1 per chunk bounds per-tree work by

$$W(\tau, X) \;\le\; \sum_{j=1}^{K} |T_c(\tau, X_j)| \;+\; G \;+\; \sum_{j=1}^{K} Q(\tau, X_j).$$

The constant projection depends only on the tree and the constant feature values, which all chunks share, so $|T_c(\tau, X_j)| = |T_c(\tau, X)|$ for every j. The structural term is paid K times instead of once, and the per-row structural share plateaus at the chunk width instead of falling further in G. Leaf writes remain G in total, and Q_{trace} is additive over rows, so chunking leaves both unchanged. For Q_{pre} the sort term does not grow ($\sum_j G_j \log G_j \le G \log G$), while the predicate-sweep term is paid once per chunk. This K-fold repayment of $|T_c|$ is the analytical cost of chunking. Evaluating the group whole avoids it, at the price of masks of $\lceil G/64 \rceil$ words, whose operations cost $\Theta(G/64)$ (§4.1); the engine does so for groups of any width, in pieces of at most 1,024 rows.

Chunks against whole groups. Because the engine takes wide groups whole, the cost of chunking is measurable. On FLCHAIN's horizons of 256, 512 and 1,024 (1,008 rows) we time TreeWalker on whole groups and, in the same cells and rounds, on 128-row chunks of the same groups ($T \in \{500, 2000\}$, $L \in \{8, 16\}$, both frameworks; 200 of the 1,304 patients at 1,008 rows). Chunks are slower in every cell, by 1.07–1.38× on Intel and 1.07–1.31× on Arm (Table 11). The measurement is a net difference: chunks repay $|T_c|$ and the predicate sweep once per chunk, while whole groups pay for wider masks. It does not isolate the K-fold $|T_c|$ term.

Table 11: 128-row chunks against whole groups on FLCHAIN: chunked latency over whole-group latency, median [min, max] over the 8 cells ($T \in \{500, 2000\}$, $L \in \{8, 16\}$, LightGBM and XGBoost) at each horizon.

Rows per group	Intel	Arm
256	1.17× [1.13, 1.21]	1.19× [1.14, 1.26]
512	1.22× [1.15, 1.28]	1.20× [1.13, 1.25]
1,008	1.19× [1.07, 1.38]	1.12× [1.07, 1.31]

NeurIPS Paper Checklist

1. **Claims**

   Answer: [Yes]

   Justification: The abstract and introduction state conditional speedup claims backed by the factorial grid evaluation and the scenario-analysis benchmark (Section 5), with limitations and boundary cases in Section 6.

2. **Limitations**

   Answer: [Yes]

   Justification: Section 6 discusses when partial evaluation does not help (low constant fraction, small groups, and CPU-only scope), the unmeasured cost of constructing groups, the model types the implementation reads, and how often a baseline is faster; Appendix D compares QuickScorer on an encoded Expedia variant. Appendix A.5 documents baseline coverage.

3. **Theory assumptions and proofs**

   Answer: [Yes]

   Justification: Theorem 1 (structural work decomposition) and Corollaries 1–2 are stated in §4.1 with the abstract unit-cost model and boundedness assumption made explicit; full proofs of Lemmas 1–4 and the theorem appear in Appendix C.

4. **Experimental result reproducibility**

   Answer: [Yes]

   Justification: Section 5.1 and Appendix A specify hardware, software versions, compiler flags, and methodology. All benchmark parameters are listed. The system is implemented in Rust with deterministic model parsing; dataset subsampling uses seed 42.

5. **Open access to data and code**

   Answer: [Yes]

   Justification: The code and benchmark harness are public at `https://github.com/Shopify/treewalker/tree/neurips2026-final`, under the MIT license. All datasets are publicly available: SUPPORT and FLCHAIN via standard R/Python packages, Expedia via Kaggle, and the credit-card dataset of the scenario analysis via the UCI repository.

6. **Experimental setting/details**

   Answer: [Yes]

   Justification: Section 5.1 summarizes the protocol and reference configuration; Appendix A gives full hardware (Table 4), software versions (Table 5), compiler flags, and method roster (Table 6). Table 2 describes the datasets.

7. **Experiment statistical significance**

   Answer: [Yes]

   Justification: Each cell uses a blocked paired measurement protocol with at least four rounds of 12 blocks, alternating method order, per-group timing, and a 1% precision target (Appendix A.4). We report median latency with p5/p95 over blocks; speedups carry paired bootstrap 95% intervals.

8. **Experiments compute resources**

   Answer: [Yes]

   Justification: Table 4 specifies GCE machine types (c4-standard-32 with 120 GB RAM, c4a-highmem-16 with 128 GB RAM); measurements pin the benchmark process to a single physical core via `taskset -c 0` with SMT disabled at the VM level (`threads_per_core=1`), both CPUs at their all-core turbo frequency, and architecture-native compilation. Models are trained once on Intel and shared with Arm, which checks their hashes. The camera-ready campaign took about 29 hours on the Intel VM and 32.5 hours on the Arm VM.

9. **Code of ethics**

Answer: [Yes]

Justification: This work optimizes inference latency for tree ensembles. It does not involve human subjects, personal data collection, or dual-use concerns.

10. **Broader impacts**

    Answer: [N/A]

    Justification: This is a systems optimization for tree ensemble inference. It reduces computation for existing models without changing their predictions. We see no direct negative societal impact.

11. **Safeguards**

    Answer: [N/A]

    Justification: The released artifact is an inference engine, not a pre-trained model or dataset with misuse risk.

12. **Licenses for existing assets**

    Answer: [Yes]

    Justification: SUPPORT and FLCHAIN are standard research datasets available under open licenses. Expedia data is from a public Kaggle competition. The UCI Default of Credit Card Clients dataset is CC BY 4.0. LightGBM (MIT), XGBoost (Apache-2.0), treelite (Apache-2.0), lleaves (MIT), tl2cgen (Apache-2.0) are cited with versions in Section 5.1.

13. **New assets**

    Answer: [Yes]

    Justification: The code and benchmark harness are released at `https://github.com/Shopify/treewalker/tree/neurips2026-final` under the MIT license, with a README that documents the engine, the experiments and how to reproduce them.

14. **Crowdsourcing and research with human subjects**

    Answer: [N/A]

    Justification: No crowdsourcing or human subjects research.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Answer: [N/A]

    Justification: No human subjects research.

16. **Declaration of LLM usage**

    Answer: [Yes]

    Justification: LLMs assisted with writing and editing the paper, implementing the Rust codebase and Python benchmark scripts, and orchestrating the benchmark sweeps on cloud VMs. LLMs were not used as a component of the core method itself. All LLM-generated code was reviewed and tested; every benchmark run validates each method's outputs before timing it (Appendix A.4).