List Intersection for Web Search: Algorithms, Cost Models, and Optimizations

Sunghwan Kim
POSTECH
sunghwan08@gmail.com

Taesung Lee
IBM Research AI
Taesung.Lee@ibm.com

Seung-won Hwang*
Yonsei University
seungwonh@yonsei.ac.kr

Sameh Elnikety
Microsoft Research
samehe@microsoft.com

ABSTRACT

This paper studies the optimization of list intersection, especially in the context of the matching phase of search engines. Given a user query, we intersect the postings lists corresponding to the query keywords to generate the list of documents matching all keywords. Since the speed of list intersection depends the algorithm, hardware, and list lengths and their correlations, none the existing intersection algorithms outperforms the others in every scenario. Therefore, we develop a cost-based approach in which we identify a search space, spanning existing algorithms and their combinations. We propose a cost model to estimate the cost of the algorithms with their combinations, and use the cost model to search for the lowest-cost algorithm. The resulting plan is usually a combination of 2-way algorithms, outperforming conventional 2-way and k-way algorithms. The proposed approach is more general than designing a specific algorithm, as the cost models can be adapted to different hardware. We validate the cost model experimentally on two different CPUs, and show that the cost model closely estimates the actual cost. Using both real and synthetic datasets, we show that the proposed cost-based optimizer outperforms the state-of-the-art alternatives.

PVLDB Reference Format:
Sunghwan Kim, Taesung Lee, Seung-won Hwang, Sameh Elnikety. List Intersection for Web Search: Algorithms, Cost Models, and Optimizations. *PVLDB*, 12(1): 1-13, 2018.
DOI: https://doi.org/10.14778/3275536.3275537

Keywords

list intersection, cost-based query optimization, statistical analysis, top-down analysis

*corresponding author

Proceedings of the VLDB Endowment, Vol. 12, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275536.3275537

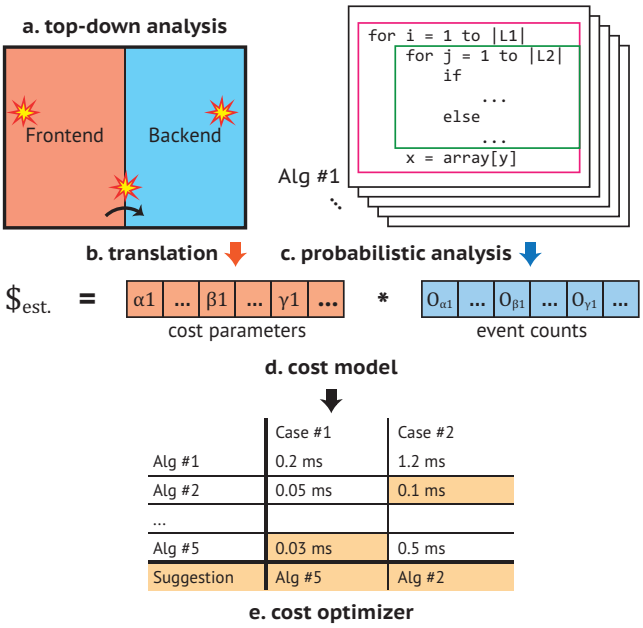


Figure 1: Overview of cost-based optimizer. We build a cost model with following two steps: (1) translate the cost factors of the top-down cycle accounting method into cost parameters, and (2) translate an intersection process into event counts through probabilistic analysis. The cost-based optimizer searches for the optimal plan.

1. INTRODUCTION

List intersection is a fundamental operation that is widely used in data processing and serving platforms, such as web search engines and databases. In web search engines, a multi-word query goes through a "matching phase", which is essentially an intersection of sorted lists corresponding to the query keywords. Each list (also called inverted or postings list) contains the IDs of web documents matching the keyword [11]. For example, a query with (*e.g.*, "VLDB 2019 Research Papers") is dispatched to the servers managing the web corpus inverted index, and each server intersects the corresponding postings lists and returns its intersection outcome as a result. Due to this significance, many algorithms have been proposed to execute this operation, but no list

intersection algorithm is cost-optimal in all scenarios. For example, merge-based algorithms are effective when the lists are of similar lengths and are accessed sequentially. Meanwhile, search-based algorithms perform better when one list is significantly shorter than others, as items in the shortest list work as pivots to skip accessing on non-matching items in long lists.

A cost-based approach aims at having an accurate cost model, so to generate the optimal query plan for the given scenario, as commonly used in the database systems [9, 20]. However, the existing list intersection cost models estimate the cost using a number of comparisons [2, 4, 5], which does not reflect the execution time on modern processors that support out-of-order execution, predictive speculation, and hardware prefetching [29]. For this reason, we develop a cost model that uses several hardware parameters rather than inventing new specialized intersection algorithms.

Considering the dependency on hardware parameters, a desirable cost model should estimate the cost of a given algorithm with a set of raw parameters, rather than with a single scalar parameter, such as the number of comparisons. We use profiling to measure the hardware parameters and develop cost models for several intersection algorithms. We build on related efforts that use hardware parameters, such as cycle accounting (CA) [29] which targets analyzing the bottleneck for given CPU architecture and algorithm. We use the hardware parameters to estimate the cost of executing the intersection algorithm rather than identifying bottleneck. We compute the total execution cost using three sets of hardware parameters reflecting (1) instruction execution, (2) branch misprediction, and (3) memory references.

We address several challenges to develop a principled cost-based approach for list intersection. The first challenge is **hardware cost parameterization**, by overcoming taxonomy mismatch. The state-of-the-art top-down method (Figure 1a) for CA divides architecture into two major parts, backend and frontend, and analyzes the cost with respect to three types of overhead—backend, frontend, and between. This does not map well with the causes we want to identify, for which we propose to translate the cost factors of top-down method into the unit cost parameters (Figure 1b).

The second is **probabilistic counting**. How many elements in a list are accessed (or skipped) is specific to the intersection algorithm and the data distributions. We propose a probabilistic model (Figure 1c), translating cost factors into the an expected total cost (Figure 1d).

Last challenge is **search space reduction**, to reduce search overhead without compromising the optimality of the algorithm found. In addition, we show that considering 2-way algorithms is often sufficient, which reduces the search overhead of the proposed approach. This finding promotes nonblocking pipelined implementation scenarios when some of the required information on input lists are not available at query time.

We demonstrate that the proposed model performs well in two CPUs on both synthetic and real-world datasets, and show that our cost-based optimizer (Figure 1e) outperforms the existing list intersection approaches.

Our main contributions are the following:

- We parameterize intersection scenarios to reflect the architecture and input characteristics, and build cost models for major intersection algorithms.
- We build a cost-based optimizer that suggests the optimal intersection plan for intersecting two or more lists.
- We analytically show that the proposed cost models are precise in a wide range of scenarios.
- We empirically demonstrate the proposed cost models and the cost-based optimizer perform well in a wide range of scenarios.

Algorithm 1 2-GALLOP

Input: L_1, L_2: Two lists to intersect
Output: $R = L_1 \cap L_2$: intersection of two lists
1: $R = \phi$
2: $c_1 = i = 0, c_2 = j = 0$
3: **for** $i = 0$ **to** $|L_1| - 1$ **do**
4: $\quad \delta = \text{GALLOPSEARCH}(L_2, j, L_1[i])$
5: $\quad j = j + \delta$
6: $\quad$ **if** $L_1[i] = L_2[j]$ **then**
7: $\quad\quad R = R \cup \{L_1[i]\}, L_1[i] = \phi$
8: **return** R

Algorithm 2 2-MERGE

Input: L_1, L_2: Two lists to intersect
Output: $R = L_1 \cap L_2$: intersection of two lists
1: $c_1 = i = 0, c_2 = j = 0$
2: $u = L_1[i], v = L_2[j]$
3: **while** $i < |L_1|$ **and** $j < |L_2|$ **do**
4: $\quad$ **if** $u < v$ **then** // $<$
5: $\quad\quad i++, u = L_1[i]$ // $\text{SB}_<$
6: $\quad$ **else if** $u > v$ **then** // $>$
7: $\quad\quad j++, v = L_2[j]$ // $\text{SB}_>$
8: $\quad$ **else** // $=$
9: $\quad\quad$ // $\text{SB}_=$
10: $\quad\quad R = R \cup \{u\}$
11: $\quad\quad i++, j++$
12: $\quad\quad u = L_1[i], v = L_2[j]$
13: **return** R

Algorithm 3 2-SIMD

Input: L_1, L_2: Two lists to intersect
Output: $R = L_1 \cap L_2$: intersection of two lists
1: $i = 0 (= c_1), j = 0 (= c_2)$
2: $u = L_1[i], v = L_2[j]$
3: **while** $i < |L_1|$ **and** $j < |L_2|$ **do**
4: $\quad d_1 = (L_1[i], L_1[i+1], \cdots, L_1[i+\Delta-1])$
5: $\quad d_2 = (L_2[j], L_2[j+1], \cdots, L_2[j+\Delta-1])$
6: $\quad R = R \cup (d_1 \cap d_2)$
7: $\quad i = i + \Delta \cdot (L_1[i+\Delta-1] \leq L_2[j+\Delta-1])$
8: $\quad j = j + \Delta \cdot (L_1[i+\Delta-1] \geq L_2[j+\Delta-1])$
9: **return** R

2. LIST INTERSECTION ALGORITHMS

This section describes existing representative 2-way and k-way intersection algorithms, as preliminaries to model their behaviors in our cost models.

2.1 2-way algorithms

2-way algorithms accept two lists L_1 and L_2 as inputs, and return its intersection $L_1 \cap L_2$.

2.1.1 2-Gallop

In our application scenario of search, input lists are ordered, which enables a search algorithm, skipping some elements, instead of incrementing the cursor one step at a time. 2-GALLOP (Algorithm 1) is a representative 2-way search-based algorithm that uses each element of the shorter list L_1 as a pivot and searches for the match in another list L_2.

In particular, for a search, [6] implements an exponential search of looking for the first exponent, j, where the 2^j-th value in L_2 is greater than the search key. This step limits the range where the match exists, from which a binary search follows. Because we may expect that the pivot element resides not far from the cursor, gallop search is more efficient than directly applying a binary search in the entire range. We denote this procedure as GallopSearch.

2.1.2 2-Merge

2-Merge (Algorithm 2) adopts a sequential scan for searching of an element. This method compares two values pointed by cursor i and j from the two lists L_1 and L_2. Once the identical pair is found, the method advances both cursors. Otherwise, the method moves the cursor with smaller value forward. The method repeats comparison until either cursor i or j reach to the end of list.

This algorithm takes a different form according to the hardware changes. For example, SIMD instructions (Algorithm 3) [17, 21] support operations on 128- or 256-bit vector registers, for which set elements are packed (Line 4, 5). For this scenario, vector intersections are replaced by arithmetic operations (Line 6). This implementation compresses multiple instructions on elements, into few SIMD instructions on larger vectors.

2.2 k-way algorithms

k-way algorithms [3] accept k lists $L_1, L_2, \ldots, L_k$ then return their intersection.

2.2.1 Skeleton of k-way algorithm

Most k-way algorithms are abstracted as three steps (Algorithm 4) including Pivot, Search and Output.

Pivot (Line 2, 10) chooses a pivot from a given list, initialized as the first element in all lists. Once the pivot is selected, in the Search step (Line 4), the pivot element is tested with elements in all other lists. Depending on base algorithms, either Gallop or Merge, this implementation can differ, which we later present as GallopSearch and MergeScan respectively. In either case, this test moves the cursors by displacement δ to reduce the search space until it reaches the last element, where displacement δ is the distance between current cursor and the index of lowest value matching or exceeding pivot.

In the Output step (Line 6-9), we add the pivot element into the result, if the pivot is found in all k list. Otherwise, move on to the Pivot step until any cursor reaches to the end of list.

2.2.2 k-Gallop

We denote by k-Gallop the k-way intersection algorithm with GallopSearch procedure. Given a list L_i, a cursor c and a pivot element p, k-Gallop performs GallopSearch (Algorithm 5) as Search for finding pivot element p from the list L_i.

2.2.3 k-Merge

k-Merge (Algorithm 6) adopts a merge-based scan for Search. Specifically, for given L_i, c, and the pivot value p, MergeScan increases δ one by one until the element under $c+\delta$ is equal or greater than p. We denote by k-Merge the k-way intersection algorithm with MergeScan procedure.

Algorithm 4 General k-way intersection

Input: L
Output: $R = \bigcap_{1 \le i \le k} L_i$
1: $R = \phi, C = \{c_i = 0 | \forall i, 1 \le i \le k)\}$
2: $pivot = L_1[c_1], j = 2$
3: **while** $c_i < |L_i| (\forall i, 1 \le i \le k)$ **do**
4: $\quad \delta_j = \text{Search}(L_j, c_j, pivot)$
5: $\quad c_j = c_j + \delta_j$
6: $\quad count = count * (pivot == L_j[c_j]) + 1$
7: $\quad$ // count is 1 if $pivot \neq L_j[c_j]$ or increase by 1 otherwise
8: $\quad$ **if** $count = k$ **then**
9: $\quad\quad R = R \cup \{pivot\}, c_j ++$
10: $\quad pivot = L_j[c_j]$
11: $\quad c_j ++, j = (j \mod k) + 1$
12: **return** R

Algorithm 5 GallopSearch procedure

Input: L_i: given list, c: index of cursor, p: pivot value
Output: δ: displacement that is the difference between c and the index of lowest value which is no lower than p.
1: $\delta = 0$
2: $offset = 1$
3: **while** $c + \delta + offset < |L_i| \wedge p > L_i[c + \delta + offset]$ **do**
4: $\quad \delta = \delta + offset, offset = offset * 2$
5: **while** $\lfloor offset/2 \rfloor > 0$ **do**
6: $\quad offset = \lfloor offset/2 \rfloor$
7: $\quad$ **if** $c + \delta + offset < |L_i| \wedge p > L_i[c + \delta + offset]$ **then**
8: $\quad\quad \delta = \delta + offset$
9: **return** δ

Algorithm 6 MergeScan procedure

Input: L_i: given list, c: index of cursor, p: pivot value
Output: δ: displacement
1: $\delta = 0$
2: **while** $c + \delta < |L_i| \wedge p > L_i[c + \delta]$ **do**
3: $\quad \delta ++$
4: **return** δ

2.3 Intersecting k lists

To perform the intersection of k lists, we may can apply either or both 2-way or/and k-way algorithms. For example, we can intersect k lists, by executing a k-way algorithm once $\bigcap_{L_i \in L} L_i$, or 2-way algorithms $k-1$ times $(((L_1 \cap L_2) \cap L_3) \cdots \cap L_k)$.

Formally, we define intersection plan tree T recursively as follows:

$$T(L) = \begin{cases} L_1 & L = \{L_1\} \\ \bigcap_{L_i \in L} L_i & k\text{-way} \\ T(L') \cap T(L - L') & L' \subset L, \text{2-way} \end{cases} \quad (1)$$

where $L = \{L_1, L_2, ..., L_k\}$ $(|L_i| \le |L_j|, \forall i \le j)$ is given set of lists to intersect. Our cost model in Section 4 estimates the cost of each plan tree T, so that cost optimizer finds an efficient plan tree T_{opt} for computing the intersection of the given lists L.

3. HARDWARE PROFILING

On modern processors, using the number of comparisons O_{inst} is not sufficient to estimate execution time of list intersection which depends on memory access, branch misprediction among other factors. Our goal is to develop new model to reflect new cost factors. For example, as illustrated with SIMD examples, underlying hardware may expedite some instructions while penalizing others, for which a new specialized 2-way algorithm (Algorithm 3) had to be custom-designed. In contrast, we aim to parametrize new factors

Table 1: Cause-based pivoting of top-down factors.

Top-down Category	Our category		
	Base latency(α)	Branch MISP(β)	Memory Stalls (γ)
Frontend bound	✓	✓	
Bad Speculation		✓	
Retiring	✓		
Backend	✓		✓

such as slow memory accesses (of additional latency β) or branch misprediction (of γ).

For the new parameterizations, we divide cost of instruction execution into those incurring additional latency β or γ. For this goal, we adopt the state-of-the-art cycle accounting analysis for a modern out-of-order architecture called the top-down method [29].

In this work, we illustrate how the cost factors from top-down profiler is mapped into our new cost factors. **Top-down model** provides four groups of cost factors:

- *Frontend bound* – overhead in frontend.
  It represents fetching instructions and issuing micro-operations to backend. This is not dominant in list intersection with few instructions.
- *Bad speculation* – overhead between frontend and backend.
  It represents wasted pipeline slots due to incorrect speculations. This is mainly caused by branch mispredictions in list intersection.
- *Retiring* – no overhead.
  It represents cycles where superscalar performing in full potential. This is mainly related to the number of instructions computed.
- *Backend bound*: overhead in backend.
  It represents wasted cycles due to lack of resources. Main causes are either by memory stalls or long latency instructions such as division.

Table 1 guides how each of the above groups affects each cost factor, especially over the following three cost factors:

- *Base latency* (α) – mainly including latency of successfully retired instructions with few other overhead such as frontend bound or backend bound.
- *Branch misprediction* (β) – overhead caused by branch misprediction.
- *Memory stalls* (γ) – overhead caused by memory reference.

The cost of a particular instruction is closely related to the context of execution, thus it is hard to predict without grouping instructions that share the same context. Based on this observation, we set a "code hierarchy" based on loop structure as the basic unit of cost analysis. In the next section, we introduce how we model the cost of each algorithm.

4. PROBABILISTIC COUNTING

In this section, we analyze and build cost models for list intersection algorithms for 2-way algorithms then generalize them for k-way. For each algorithm, we first introduce how to parametrize the cost model of the algorithm, then discuss the probabilistic model of the algorithm. Table 2 describes the symbols used in this section, and the details of the machine specific parameters will be discussed in Section 6.1.

Table 2: Symbols used in cost model.

Symbol	Description
p_i	selectivity of a list L_i to entire set of list L, $p_i = \frac{\bigcap_{j=1}^{k} L_j}{\lvert L_i \rvert}$
$p_{i,j}$	selectivity of L_i to L_j, $p_{i,j} = \frac{\lvert L_i \cap L_j \rvert}{\lvert L_j \rvert}$
$r_{i,j}$	list size ratio of L_i to L_j, $r_{i,j} = \frac{\lvert L_j \rvert}{\lvert L_i \rvert}$
δ	displacement, moving distance of cursor c_2 as a result of a 2-way loop.
δ_i	displacement, moving distance of cursor c_i as a result of a k-way loop.
$\delta_{i,j}$	subdisplacement, moving distance of cursor c_i that caused by pivot change for j-th round-robin iteration after round-robin iteration on L_i.
$\alpha_{\mathcal{A}}^{(event)}$	unit cost parameter of base latency (for the *event*) in algorithm $\mathcal{A}$
$\beta_{\mathcal{A}}^{(event)}$	unit cost parameter of branch misprediction (MISP) (for the *event*) in algorithm $\mathcal{A}$
$\gamma_{\mathcal{A}}^{(event)}$	unit cost parameter of memory reference (for the *event*) in algorithm $\mathcal{A}$
$\mathrm{O}_{\mathrm{event}}$	computational occurrence of each *event*.
$f_{\mathrm{MISP}}(p)$	function of misprediction that takes p as the probability of branch taken. $f_{\mathrm{MISP}}(p) = \frac{p(1-p)}{p^3+(1-p)^3+p(1-p)}$
Δ	number of elements that can be stored in a 128- or 256-bit vector register.
N_{ln}	number of elements in a cache line. eg. 16 for 64-byte cache line size with 4-byte elements.

4.1 2-way algorithms

4.1.1 2-Gallop

Cost estimation. We parametrize the cost of 2-GALLOP into six components including two types of base latency ($\alpha_{\mathrm{2gallop}}^{\mathrm{in/out}}$), two of branch mispredictions (MISPs) ($\beta_{\mathrm{2gallop}}^{\mathrm{in/out}}$), and two types of memory subsystem costs ($\gamma_{\mathrm{2gallop}}^{\mathrm{mem/miss}}$). Then, we formulate the cost model by aggregating them with their corresponding event counts ($\mathrm{O}_{\mathrm{event}}$).

$$\begin{aligned}\$_{\mathrm{2gallop}}(L_1, L_2) = {} & \alpha_{\mathrm{2gallop}}^{\mathrm{in}}\mathrm{O}_{\mathrm{in}} + \alpha_{\mathrm{2gallop}}^{\mathrm{out}}\mathrm{O}_{\mathrm{out}} \\ & + \beta_{\mathrm{2gallop}}^{\mathrm{in}}\mathrm{O}_{\mathrm{MISP_{in}}} + \beta_{\mathrm{2gallop}}^{\mathrm{out}}\mathrm{O}_{\mathrm{MISP_{out}}} \\ & + \gamma_{\mathrm{2gallop}}^{\mathrm{mem}}\mathrm{O}_{\mathrm{mem}} + \gamma_{\mathrm{2gallop}}^{\mathrm{miss}}\mathrm{O}_{\mathrm{miss}}. \end{aligned} \quad (2)$$

Models of loop iterations. 2-GALLOP is generally implemented in the structure of 2-level nested loop: one *outer loop* (Line 3-7, Algorithm 1), that selects a pivot to search, and two *inner loops* (Line 3-4 and 5-8, Algorithm 5), that computes search of the pivot.

We make the model of each loop counts: $\mathrm{O}_{\mathrm{out}}$ for the outer loop and $\mathrm{O}_{\mathrm{in}_1}/\mathrm{O}_{\mathrm{in}_2}$ for each inner loop. The outer loop count $\mathrm{O}_{\mathrm{out}}$ is equal to $|L_1|$ because the loop linearly scans the shorter list L_1 for picking each element on the list as the pivot.

$$\mathrm{O}_{\mathrm{out}} = |L_1| \quad (3)$$

The behavior of inner loops is probabilistic modeled with respect to the displacement δ of the exponential search, which is the number of skipped elements by a search. Formally, we define the displacement δ as the distance between current cursor c and the index c' of lowest value matching or exceeding the pivot, where the cursor c is the index of element that the search begins. If a displacement of a specific search is δ, both inner loops are computed $i = \lfloor \log_2(\delta + 1) \rfloor$ times. Because 1) the first inner loop scans are $(c + 2^k)$-th element

for k-th iteration, for which the loop stops at the i-th iteration, and 2) the second loop executes the binary search over the range of index $[c + 2^{i-1}, c + 2^i)$ which requires i iterations. Therefore, we model the probability distribution of the loop count $\mathrm{O_{in}}$ shared by both inner loops as follows:

$$\mathrm{O_{in}} = \mathrm{O_{in_1}} = \mathrm{O_{in_2}} = |L_1| \sum_{\delta=0}^{|L_2|} \lfloor \log_2(\delta+1) \rfloor \cdot P[D=\delta], \quad (4)$$

where D is the random variable of displacement δ.

Next, we model the distribution associated with D. Consider a generative process of drawing balls from a jar with $|L_1 - L_2|$ red, $|L_1 \cap L_2|$ orange, and $|L_2 - L_1|$ blue balls: select a ball, give it a sequence number, and put the ball in L_1 if red, L_2 if blue, and in both if orange. This process will generate two random lists.

We can consider a search as drawing balls from a jar until drawing a red or orange ball. Then, the displacement is identical to the sum of drawn orange or blue balls. We model this process with a negative hypergeometric distribution (NHG) by considering picking red/orange ball as a 'success' and blue balls as a 'failure'. Specifically, we define the random variable of D as follows:

$$D = (1-p_1) \cdot NB_{1,prob}(\delta) + p_1 \cdot NB_{1,prob}(\delta - 1), \quad (5)$$

where $prob$ is $|L_2|/|L_1 \cup L_2|$ and p_1 is $|L_1 \cap L_2|/|L_1|$. Since we have large $|L_1|$ and $|L_2|$, the NHG distribution can be approximated by the negative binomial (NB) distribution [16].

Branch mispredictions. We next model the count of MISPs, based on the model of loop iterations. In particular, we identify the following three branches and their count.

- $\mathrm{O_{MISP_{in}}}$: The count of mispredictions caused by the two inner loops in GallopSearch (Algorithm 5). The first branch (Line 2-3) is taken $\mathrm{O_{in}}$ times, and not taken $\mathrm{O_{out}}$ times out of $\mathrm{O_{in}} + \mathrm{O_{out}}$ times total. The second branch (Line 4-7) is computed $\mathrm{O_{in}}$ times, and is taken or not taken, each with half assuming the uniform distribution of elements in lists.

$$\mathrm{O_{MISP_{in}}} = f_{\mathrm{MISP}}\left(\frac{\mathrm{O_{out}}}{\mathrm{O_{out}} + \mathrm{O_{in}}}\right)(\mathrm{O_{out}} + \mathrm{O_{in}}) + \frac{\mathrm{O_{in}}}{2}. \quad (6)$$

- $\mathrm{O_{MISP_{out}}}$: The count of mispredictions caused by the *if* statement in Output (line 6-7, Algorithm 1) after GallopSearch. The branch is computed $\mathrm{O_{out}}$ times, and taken $|L_1 \cap L_2|$ times when the search is successful.

$$\mathrm{O_{MISP_{out}}} = f_{\mathrm{MISP}}\left(\frac{|L_1 \cap L_2|}{\mathrm{O_{out}}}\right)\mathrm{O_{out}}. \quad (7)$$

Memory references & cache misses. We identify memory subsystem costs including memory references, and cache misses. The count of memory references $\mathrm{O_{mem}}$ is easily modeled from the algorithm structure: both inner loops have one each, and the outer loop has two memory load instructions. Thus, we compute $\mathrm{O_{mem}}$ as $2\mathrm{O_{in}} + 2\mathrm{O_{out}}$.

We next model the cache miss overhead which is also an important memory subsystem cost. In 2-Gallop, few or no capacity misses happen due to the small working set size, however, cold/compulsory misses may occur frequently. For modeling the cold miss, the stride, the memory address distance of subsequent accesses, is an important variable; if the stride fits in the cache line size N_{ln}, the cold misses do not happen thanks to speculative caching.

As the cursor move distance δ determines the stride, the cache misses are decided by δ and N_{ln}.

- $\delta < N_{\mathrm{ln}}/2$ – no cold cache miss happens.
- $N_{\mathrm{ln}}/2 \le \delta < N_{\mathrm{ln}}$ – then only one cold cache miss occurs, because the maximum stride of galloping jump (first inner loop) is N_{ln}.
- $\delta \ge N_{\mathrm{ln}}$ – the search has two additional cache misses than the search of jump distance $\delta/2$. As δ increases twice, both inner loops have additional scan that the stride is larger than N_{ln}.

For example, if δ is N_{ln}, we have three cold cache misses, two in first inner loop (stride: N_{ln}, $2N_{\mathrm{ln}}$), and one in second inner loop (stride: N_{ln}). We formulate the number of cache misses by the following function:

$$\mathrm{O_{miss}} = |L_1| \sum_{\delta=0}^{|L_2|} \max\left(0, \left\lfloor \log_2\left(\frac{2\delta}{N_{\mathrm{ln}}}\right)\right\rfloor\right) \cdot P[D=\delta]. \quad (8)$$

4.1.2 2-Merge

We parametrize the cost of 2-Merge into base latency (α_{2merge}) and MISP (β_{2merge}), and model the following cost model with corresponding counts:

$$\$_{\mathrm{2merge}}(L_1, L_2) = \alpha_{\mathrm{2merge}}\mathrm{O_{ST}} + \beta_{\mathrm{2merge}}\mathrm{O_{MISP}}. \quad (9)$$

2-Merge is generally to be implemented in a single loop with three cases $\{<, >, =\}$, which are separated by two conditional branches. Thus, total loop count $\mathrm{O_{ST}}$ is computed as follows:

$$\mathrm{O_{ST}} = \sum_{\mathrm{BT} \in \{<,>,=\}} \mathrm{O_{BT}}. \quad (10)$$

where $\mathrm{O_{BT}}$ is computed as $\mathrm{O_{=}} = |L_1 \cap L_2|$, $\mathrm{O_{<}} = |L_1| - |L_1 \cap L_2|$, and $\mathrm{O_{>}} = |L_2| - |L_1 \cap L_2|$. As 2-Merge reads data sequentially, the memory access cost of the algorithm is marginal, for which we group this cost with the base latency.

We model the count of MISP $\mathrm{O_{MISP}}$ as follows

$$\mathrm{O_{MISP}} = f_{\mathrm{MISP}}\left(\frac{\mathrm{O_{<}}}{\mathrm{O_{ST}}}\right)\mathrm{O_{ST}} + f_{\mathrm{MISP}}\left(\frac{\mathrm{O_{>}}}{\mathrm{O_{>}} + \mathrm{O_{=}}}\right)(\mathrm{O_{>}} + \mathrm{O_{=}}), \quad (11)$$

based on our two-level nested branch implementation in 2-Merge (*if* $<$ *else (if* $>$ *else* $=$*)*).

4.1.3 2-SIMD

We model the following cost of 2-SIMD with only one parameter – base latency (α_{2simd}). In 2-SIMD, both the cost of MISPs and memory loads are marginal, because the algorithm does not require branching and has sequential memory access pattern.

$$\$_{\mathrm{2simd}}(L_1, L_2) = \alpha_{\mathrm{2simd}}\mathrm{O_{ST}}. \quad (12)$$

For each iteration of 2-SIMD, the process selects a 128- or 256-bit vector of elements under the cursors, then finds intersections between the vectors with SIMD computation. If two lists are identical, both cursors of two lists are advanced in every iteration, otherwise, both are advanced together only if the last element of two selected vectors are equal to each other. Thus, we formulate $\mathrm{O_{ST}}$ as follows:

$$\mathrm{O_{ST}} = \begin{cases} \frac{|L_1|}{\Delta} & L_1 = L_2 \\ \frac{|L_1|+|L_2|}{\Delta} - \frac{|L_1 \cap L_2|}{\Delta^2} & L_1 \neq L_2, \end{cases} \quad (13)$$

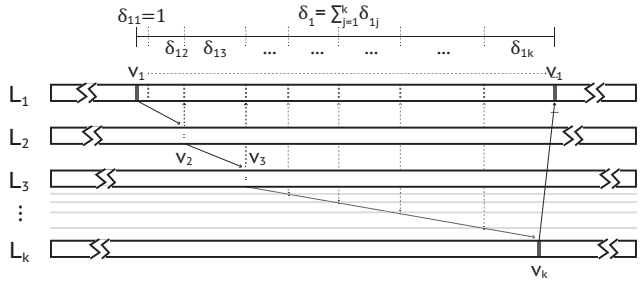


Figure 2: Illustration of displacement δ_1.

where Δ is the number of elements fit in a vector register. For example, if we intersect lists of 32-bit elements through a SIMD algorithm with 128-bit vectors, then one vector can handle four elements ($\Delta = 4$).

4.2 k-way algorithms

The cost model of k-way algorithms requires understanding the cursor movements after searches, or conversely, the number of SEARCH operations in the algorithm skeleton. Thus, we first discuss the cursor movements (Section 4.2.1), for a round robin strategy, commonly used in k-way algorithms. Then, we discuss the cost models of k-way algorithms with each procedure: GALLOPSEARCH or MERGESCAN (Section 4.2.2, 4.2.3).

4.2.1 Common structure analysis

The k-way algorithms (Algorithm 4) share the same outer loop structure, and thus share the outer loop count. The outer loop count is modeled by cursor displacements and their subdisplacements based on a negative binomial distribution and Markov chain. Since each outer loop iteration moves only a cursor, the outer loop count is equal to the total number of cursor moves, which is determined by displacements (*e.g.*, long displacements reduce the counts).

Without loss of generality, the general k-way algorithm searches for the pivot v_1 in L_2 (Figure 2). The displacement for this search is δ_2 such that the value v_2 under the index $c_2 + \delta_2$ is the smallest value which is equal to or greater than v_1. The search process continues and returns to L_1 to find a sequence of *displacements* $\delta_2, \ldots, \delta_k, \delta_1$ that updates cursors $\{c_2, ..., c_{k-1}, c_k, c_1\}$ to new values $\{c_2 + \delta_2, \ldots, c_k + \delta_k, c_1 + \delta_1\}$. We aim to model the distribution of each δ_i as the random variable D_i, then estimate the outer loop count $\mathrm{O}_{\mathrm{out}}$ with the variable D_i as follows:

$$\mathrm{O}_{\mathrm{out}} = \sum_{i=1}^{k} \frac{|L_i|}{E[D_i]}. \tag{14}$$

As depicted in Figure 2, each δ_i is influenced by each search trial of round-robin computation. Thus, we model how each search affects δ_i. Formally, we can divide δ_i into k **subdisplacements** $\{\delta_{i,1}, \delta_{i,2}, \ldots, \delta_{i,k}\}$ such that the value under $c_i + \sum_{j=1}^{k} \delta_{i,j}$ is the smallest value but no smaller than v_{i+k-1}. More formally, we formulate D_i using random variable $D_{i,j}$ for $\delta_{i,j}$:

$$D_i = \sum_{j=1}^{k} D_{i,j}. \tag{15}$$

Subdisplacement $\delta_{i,j}$ can be interpreted as a contribution of search on L_j for the move of c_i, and search of v_j in L_i (Figure 2). Similar with two list case (Section 4.1.1), the subdisplacement can be modeled by a negative binomial distribution. We model $D_{i,j}$ as follows:

$$P[D_{i,j} = \delta_{i,j}] = \begin{cases} 1 - \pi_{0,j} - \pi_{(k-1),j} + \\ (\pi_{0,j} + \pi_{(k-1),j}) NB_{1,prob_{i,j}}(0) & \delta_{i,j} = 0 \\ (\pi_{0,j} + \pi_{(k-1),j}) NB_{1,prob_{i,j}}(\delta_{i,j}) & \delta_{i,j} > 0, \end{cases} \tag{16}$$

where $prob_{i,j}$ is $\frac{|L_i - L_j|}{|L_i \cup L_j|}$ and $\pi_{l,i}$ is the probability of l number of preceding pivots are equal to the pivot v_i. For example, $\pi_{0,j}$ is the probability that $v_{j-1} \notin L_j$ and $\pi_{2,j}$ is the probability that $v_{j-1} \in L_j$ and $v_{j-1} = v_{j-2}$ but $v_{j-1} \neq v_{j-3}$. The subdisplacement $D_{i,j}$ follows NHG distribution, only when the pivot value is changed after searching the list L_j, formally $v_{j-1} \neq v_j$. The probability of this case is the sum of $\pi_{0,j}$ by definition, and $\pi_{(k-1),j}$ as well, because when we find $k-1$ constructive elements equal to v_j, then we will advance L_j to get a new pivot v_j, which is no more equal to v_{j-1}. Otherwise, v_{j-1} is equal to v_j, then the subdisplacement always zero.

We model $\pi_{l,i}$ as the stationary probability of markov chain π (Figure 3) that computes how many consecutive pivots are equal to each other during intersection procedure. Formally, we define $\pi_{l,i}$ as follows:

$$\pi_{l,i} = P[\pi^{(\infty)} = s_{l,i}], \tag{17}$$

where $s_{l,i}$ is the markov state (Figure 3c) – v_i, pivot of i, is found in l consecutive proceeding searches.

Each markov state $s_{l,i}$ has two types of state transition, (1) successful search, $v_i = v_{i+1}$ (Figure 3a) and (2) unsuccessful search, $v_i \neq v_{i+1}$ (Figure 3b). For each successful search, we set the next state as $s_{l+1,i+1}$, when l is less than $k-1$, because we have $l+1$ constructive pivots after search on L_{i+1}. Otherwise, when $l = k-1$ (Figure 3e), then as we start with new pivot v_i, we set the next state as $s_{1,i+1}$. For each unsuccessful search, we set the next state as $s_{0,i+1}$ (Figure 3d). We formally define transition probability function of π as follows:

$$\begin{aligned} P[\pi^{(t+1)} = s_{l,i}] &= \Gamma \cdot P[\pi^{(t)} = s_{l,i}] \\ &+ (1-\Gamma) \cdot \begin{cases} \sum_{l'=0}^{k-2} \left(1 - \frac{|\cap_{j=0}^{l'} L_{i-j}|}{|\cap_{j=1}^{l'} L_{i-j}|}\right) P[\pi^{(t)} = s_{l',i-1}] + \\ \frac{|L_{i-1} - L_i|}{|L_{i-1}|} P[\pi^{(t)} = s_{k-1,i-1}] & l = 0 \\ \frac{|L_{i-1} \cap L_j|}{|L_{i-1}|} \left(P[\pi^{(t)} \in \{s_{0,i-1}, s_{k-1,i-1}\}]\right) & l = 1 \\ \frac{|\cap_{j=0}^{l} L_{i-j}|}{|\cap_{j=1}^{l} L_{i-j}|} P[\pi^{(t)} = s_{l-1,i-1}] & 1 < l < k. \end{cases} \end{aligned} \tag{18}$$

In the model, we adopt a damping factor Γ on the equation to prevent the equation from diverging or not converging.

4.2.2 k-Gallop

Now we build the following cost model of k-GALLOP with six parameters – two types of base latency ($\alpha^{\mathrm{in}}_{\mathrm{kgallop}}$, $\alpha^{\mathrm{out}}_{\mathrm{kgallop}}$), two types of MISP ($\beta^{\mathrm{in}}_{\mathrm{kgallop}}$, $\beta^{\mathrm{out}}_{\mathrm{kgallop}}$) and two types of memory system cost ($\gamma^{\mathrm{mem}}_{\mathrm{kgallop}}$, $\gamma^{\mathrm{miss}}_{\mathrm{kgallop}}$):

$$\begin{aligned} \$_{\mathrm{kgallop}} = {} & \alpha^{\mathrm{in}}_{\mathrm{kgallop}} \cdot \mathrm{O}_{\mathrm{in}} + \alpha^{\mathrm{out}}_{\mathrm{kgallop}} \cdot \mathrm{O}_{\mathrm{out}} \\ & + \beta^{\mathrm{in}}_{\mathrm{kgallop}} \cdot \mathrm{O}_{\mathrm{MISP_{in}}} + \beta^{\mathrm{out}}_{\mathrm{kgallop}} \cdot \mathrm{O}_{\mathrm{MISP_{out}}} \\ & + \gamma^{\mathrm{mem}}_{\mathrm{kgallop}} \cdot \mathrm{O}_{\mathrm{in}} + \gamma^{\mathrm{miss}}_{\mathrm{kgallop}} \cdot \mathrm{O}_{\mathrm{miss}}. \end{aligned} \tag{19}$$

We model the counts for inner loop computations (O_{in}) and cold cache misses ($\mathrm{O}_{\mathrm{miss}}$) with the distribution of displacement. When the displacement of a search is δ_i, then

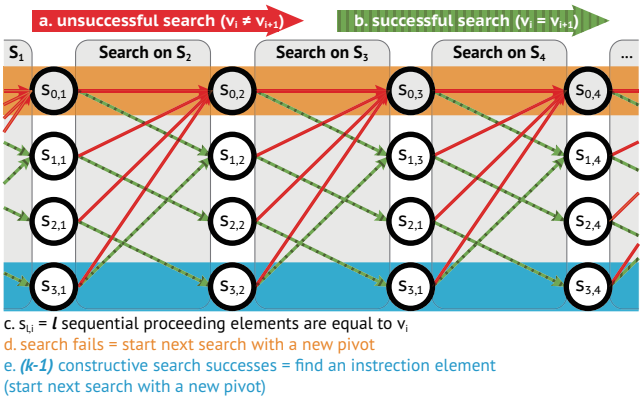


Figure 3: Illustration of markov chain π.

the search requires exact $2\lfloor \log_2(\delta_i - 1) \rfloor$ memory references. Thus, we model O_{in} as follows:

$$\mathrm{O}_{\mathrm{in}} = \sum_{L_i \in S} \sum_{\delta_i=2}^{|L_i|} 2P[D_i = \delta_i] \cdot \lfloor \log_2(\delta_i - 1) \rfloor. \tag{20}$$

The cold cache miss occurs when the stride of memory access is longer than the number of elements in a cache line N_{ln}. When δ_i is longer than $N_{\mathrm{ln}} + 1$, all the memory references have stride longer than N_{ln}. Thus, we mathematically formulate following $\mathrm{O}_{\mathrm{miss}}$:

$$\mathrm{O}_{\mathrm{miss}} = \sum_{L_i \in S} \sum_{\delta_i=N_{\mathrm{ln}}+1}^{|L_i|} P[D_i = \delta_i] \cdot \left(2 \left\lfloor \log_2 \left(\frac{\delta_i - 1}{N_{\mathrm{ln}}} \right) \right\rfloor + 1 \right). \tag{21}$$

We model the count of MISPs for branches in two inner loops ($\mathrm{O}_{\mathrm{MISP_{in}}}$) in GallopSearch as

$$\mathrm{O}_{\mathrm{MISP_{in}}} = \mathrm{O}_{\mathrm{in}} \cdot \left(f_{\mathrm{MISP}} \left(\frac{\mathrm{O}_{\mathrm{out}}}{\mathrm{O}_{\mathrm{in}}} \right) + 0.5 \right), \tag{22}$$

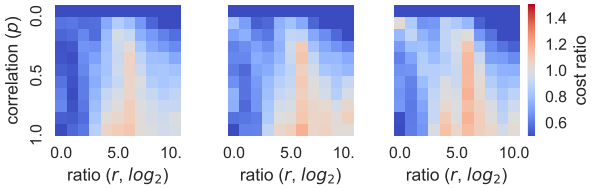
because the first loop condition is not taken $\mathrm{O}_{\mathrm{out}}$ times out of total O_{in} executions, and the second loop condition is taken or not taken with a probability of 50%. We also model the following MISP count for the branch in the outer loop ($\mathrm{O}_{\mathrm{MISP_{out}}}$) as $\frac{|\cap_{L_i \in L} L_i|}{\mathrm{O}_{\mathrm{out}}}$.

4.2.3 *k-Merge*

We parameterize the cost of k-Merge into four components – two types of base latency ($\alpha^{\mathrm{in}}_{\mathrm{kmerge}}$, $\alpha^{\mathrm{out}}_{\mathrm{kmerge}}$), two types of MISP ($\beta^{\mathrm{in}}_{\mathrm{kmerge}}$, $\beta^{\mathrm{out}}_{\mathrm{kmerge}}$). Then, we model the cost of k-Merge as follows:

$$\begin{aligned} \$_{\mathrm{kmerge}} = {} & \alpha^{\mathrm{in}}_{\mathrm{kmerge}} \cdot \mathrm{O}_{\mathrm{in}} + \alpha^{\mathrm{out}}_{\mathrm{kmerge}} \cdot \mathrm{O}_{\mathrm{out}} \\ & + \beta^{\mathrm{in}}_{\mathrm{kmerge}} \cdot \mathrm{O}_{\mathrm{MISP_{in}}} + \beta^{\mathrm{out}}_{\mathrm{kmerge}} \cdot \mathrm{O}_{\mathrm{MISP_{out}}}. \end{aligned} \tag{23}$$

We model the count of MISP for the inner loop condition ($\mathrm{O}_{\mathrm{MISP_{in}}}$) as $\mathrm{O}_{\mathrm{in}} \cdot f_{\mathrm{MISP}} \left(\frac{\mathrm{O}_{\mathrm{out}}}{\mathrm{O}_{\mathrm{in}}} \right)$, because the branch is not taken $\mathrm{O}_{\mathrm{out}}$ times out of total O_{in} executions. O_{in} is equal to $\sum_{L_i \in L} |L_i|$ because k-Merge scans all elements in linear, and the count of MISP for the branch in outer loop ($\mathrm{O}_{\mathrm{MISP_{out}}}$) is equal to the $\mathrm{O}_{\mathrm{MISP_{out}}}$ of k-Gallop.



(a) 2-Gallop **(b)** 2-Merge **(c)** 2-SIMD

Figure 4: 2-way vs. k-way in various setting. ($|L_1| = 2^{12}$, blue: 2-Gallop faster, red: k-Gallop faster.)

5. OPTIMIZATION SPACE REDUCTION

Though cost model itself suggests an optimization scheme of enumerating all possible algorithms and their costs, then execute the minimal cost one, it is too time consuming. Thus in this section, we argue two ideas for reducing the optimization space. First, in Section 5.1, we discuss that reducing the space to include only the 2-way algorithms does not compromise optimality much. Second, in Section 5.2, we discuss another rule for intersecting multiple posting lists.

5.1 2-way vs. k-way

We claim that, to intersect k-list, applying 2-way intersection $k-1$ times is faster than a single k-way intersection in most scenarios. To show this, we first identify representative input constraints and analytically show inputs satisfying the constraints meet our hypothesis on a common system (*e.g.*, IvyBridge) which fall into one of the four cases (Case 1 to 4). Then, we empirically show the claim indeed holds for a wide range parameter combinations. In the analysis, we consider that L_1 is the shortest list among the input.

Case 1 *All k inputs have equal lengths and no element is in more than one list* → 2-SIMD *outperforms both k-*Gallop *and k-*Merge*:*

In modern architecture, the cost of 2-SIMD ($\frac{\alpha_{\mathrm{2simd}}}{2}|L_1| = 3.12|L_1|$ in our IvyBridge) is usually smaller than the **lower bounds** of base latencies in outer loop and inner loop cost for both k-Gallop ($(\alpha^{\mathrm{out}}_{\mathrm{kgallop}} + \alpha^{\mathrm{in}}_{\mathrm{kgallop}})|L_1| = 3.32|L_1|$) and k-Merge ($(\alpha^{\mathrm{out}}_{\mathrm{kmerge}} + \alpha^{\mathrm{in}}_{\mathrm{kmerge}})|L_1| = 3.45|L_1|$). That is, although we have the similar numbers of micro-operations for 2-SIMD, and the outer loop of k-Gallop or k-Merge both k-way algorithms have additional inner loop costs thus they are slower than 2-SIMD. Considering the other costs such as branch misprediction or memory miss, this difference is even higher.

Case 2 *All k inputs are identical to each other* → 2-Merge *outperforms both k-*Merge *and k-*Gallop*:*

In this case, the outer loop cost of k-Gallop ($2.97k|L_1|$) and k-Merge ($2.91k|L_1|$) is higher than the cost of 2-Merge ($1.16(k-1)|L_1|$), because 2-Merge tends to have fewer micro operations. $\$_{\mathrm{2merge}}$ is equal to $(k-1)\alpha^{\mathrm{out}}_{\mathrm{2merge}}|L_1|$, $\$_{\mathrm{kgallop}}$ is greater than $k(\alpha^{\mathrm{in}}_{\mathrm{kgallop}} + \alpha^{\mathrm{out}}_{\mathrm{kgallop}} + \beta^{\mathrm{mem}}_{\mathrm{kgallop}})|L_1|$, and $\$_{\mathrm{kmerge}}$ is equal to $k(\alpha^{\mathrm{in}}_{\mathrm{kmerge}} + \alpha^{\mathrm{out}}_{\mathrm{kmerge}})|L_1|$.

Case 3 *The length ratio r is high* → *both search-based algorithms* 2-Gallop *and k-*Gallop *dominate all scan-based algorithms, and* 2-Gallop *is faster than k-*Gallop *in most of the scenarios:*

The number of searches in 2-Gallop is at most that in k-Gallop. The number of searches in k-Gallop is approx-

imately $k|L_1|$, because k-GALLOP selects most of elements in the L_1. However, the number of searches in 2-GALLOP is **upper-bounded** to $(k-1)|L_1|$, and still has much chance to have a lower value, because each time of 2-GALLOP uses intermediate results as the pivot instead of picking all elements in L_1.

For example, consider a scenario with a short list L_1 and 1024 times longer $k-1$ lists. If there is no element in more than one list, then 2-GALLOP execute the searches $|L_1|$ times, while k-GALLOP execute the searches $k|L_1|$ times, because the first intermediate result of 2-GALLOP is an empty list, while k-GALLOP picks $\frac{|L_1|}{1+((k-1)/1024)} \approx |L_1|$ elements from each list, total $k|L_1|$. If L_1 is subset of all input lists, 2-GALLOP execute searches $(k-1)|L_1|$ times, while k-GALLOP do $k|L_1|$ times.

Case 4 *The number of input lists k is large → the costs of each k-way algorithm are higher than the corresponding 2-way algorithms:*

In this case, the characteristics of the k-way cost models are different from the other scenarios. When we execute a k-way algorithm with large k, we suffer a capacity cache miss problem, because all k-way algorithms require to store lines/pages including each k cursor elements to the cache/TLB entry.

We also empirically compare the costs of using consecutive 2-way intersections, and a k-way intersection in Figure 4 and in more detail in Section 6.3. While there are some corner cases that a k-way intersection performs better, both methods have a negligible absolute difference. Thus, a wise selection of consecutive 2-way algorithms outperforms a single k-way algorithm in most scenarios.

We can generalize this to a general k-way intersection tree. In the above situation, we can replace any intersection tree T with non 2-way intersection operations into 2-way intersection tree T_2 whose cost is equal or less than T. For any non 2-way operation T' in T, we can replace it into T_2' with equal or lower cost following the above observations. By repeatedly applying this replacement, we can transform T into a faster 2-way intersection tree in most scenarios.

5.2 Case study: web search

Even after reducing the space to 2-way algorithms, in web search scenarios with multiple keywords, the size of search space may expand exponentially, because the number of possible 2-way binary tree plans is exponential. Existing work for pruning exponential search space, such as "shortest vs shortest (SvS)" [5] can be used for 2-way binary tree selection, but our problem still requires to decide which algorithm to use for each intersection.

In case of search engines, where word occurrences are known to typically follow zipfian distribution, we find 2-GALLOP is a good choice when length ratio is higher than some threshold, which holds mostly true for the second and later intersections with zipfian distribution. For example, an intermediate result after 1^{st} intersection, input lists for 2^{nd} intersection, is short, because it covers posting lists of two rare words. It is even more unlikely that two rare words to occur in the same document, such that length ratio rarely exceeds the threshold found. We can thus order input lists in the ascending order of length, then decide an algorithm for the first intersection, and apply 2-GALLOP for the rest. We empirically validate that this simple rule significantly reduces the space without compromising optimality much in Section 6.3. As an added bonus, this rule relieves the cost model from estimating the output list size of preceding intersections, such that supporting nonblocking execution as in search engines is also straightforward.

Table 3: Synthetic dataset parameters.

argument	symbol	values
# lists	k	**2,3,4 ... 15,16**
scale	$\|L_1\|$	$2^{10}, 2^{10.1}, 2^{10.2}$ ... $\mathbf{2^{12}}$ ... 2^{20}
length ratio	r	$\mathbf{2^0}, 2^{0.1}, 2^{0.2}$ ... $\mathbf{2^1}$ ... $\mathbf{2^2}$... $\mathbf{2^9}$ ... $\mathbf{2^{10}}$
correlation	p	**0**, 0.01, 0.02 ... **0.1** ... **0.2** ... **0.9** ... **1**

6. EXPERIMENTS

In this section, we empirically show that (1) our cost models are accurate, and (2) our cost-based 2-way optimizer outperforms any existing intersection algorithm, including k-way algorithms and the state-of-the-art methods in various scenarios.

In particular, we consider both (1) different hardware architectures, and (2) different input characteristics. We evaluate our cost models on two machines with different processors: Intel IVYBRIDGE equipped with four processors of 2.93 Ghz clock and 32GB DDR3 RAM, and SANDYBRIDGE equipped with four processors of 3.60 Ghz clock and 64GB DDR3 RAM. All experiments are implementations in C++.

6.1 Experimental setup

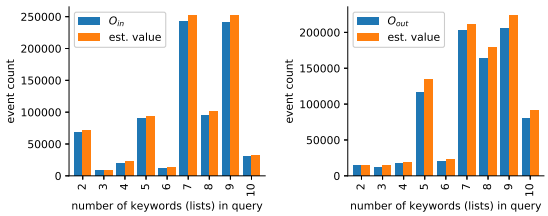
We evaluate our method on a synthetic dataset and two real-world corpora: Wikipedia corpus[1] and CommonCrawl web corpus[2]. We generate synthetic data with four parameters: the number of lists n, the length of the smallest list $|L_1|$, the correlation between all lists p which is the length ratio between the shortest list and the intersection of all lists, and the length ratio between the shortest list and the other lists r. We first generate $p|L_1|$ elements randomly that are common in all lists, then, fill the rest of each list with distinct elements, so that the intersection of lists is set as we intended. We consider diverse combinations of these parameters to see the robustness of the models and the algorithms for various input characteristics as shown in Table 3. We set the default values shown in bold in Table 3, unless otherwise noted.

We use two corpora to consider a web search scenario querying keywords over a corpus, where posting lists for the keywords are intersected. Wikipedia corpus has 9.0M documents and 412M unique keywords, and CommonCrawl web corpus has 2.9B documents. We extract the entire Wikipedia corpus and a sampled set of 3.3M documents from CommonCrawl web corpus to build an inverted index with words as the key over the documents. This index takes a word and returns a list of documents containing the keyword.

On each corpora, we compute the queries extracted from the real-life search engine log on Wikipedia, used in [28]. We extract 450 queries from this query log that contains at least two keywords, and execute the intersection of the lists from the inverted index for the keywords. The number of keywords in queries ranges from 2 to 10 for each 50 queries.

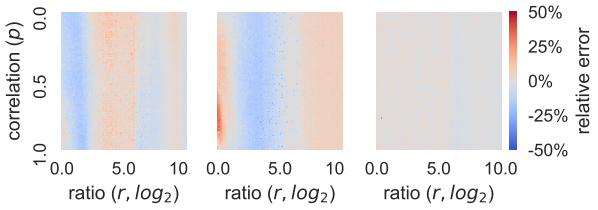
[1]https://dumps.wikimedia.org/enwiki/20160701

[2]https://commoncrawl.org/2017/12/december-2017-crawl-archive-now-available/



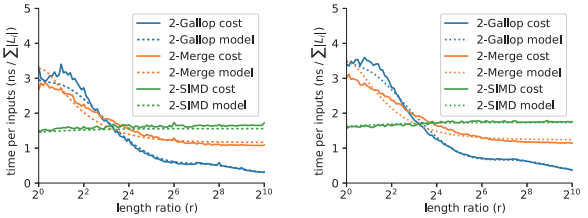
(a) O_{in} of 2-GALLOP **(b)** O_{out} of k-way

Figure 5: Accuracy of selected event counts in randomly selected real queries.



(a) 2-GALLOP **(b)** 2-MERGE **(c)** 2-SIMD

Figure 6: Heatmap of relative error for 2-way algorithms in IVYBRIDGE ($|L_1| = 2^{12}$).



(a) on IVYBRIDGE system **(b)** on SANDYBRIDGE system

Figure 7: Cost model of 2-way algorithms on different length ratio r ($|L_1| = 2^{12}, p = 0.5$).

As shown in Table 4, we see most of queries include a small shortest list, and have large ratios.

For each hardware system, we learn the cost model parameters by running intersection algorithms over randomly generated lists, and measuring its running time and top-down hardware counters using Intel VTune Amplifier. Then, we apply a gradient descent solver to obtain the unit cost parameters. In computation of the stationary probability of Markov chain $P[\pi^t = s_{i,j}]$ (Eq 18), we propagate probabilities sufficiently large t times (300 in our experiments). Table 5 describes the obtained unit cost parameters from our machines.

In the evaluation, we represent response time per total input size as the cost metric to compare average time to process an element in various scenarios. For measuring accuracy of methods, we consider both additive and relative errors. The *relative error* shows the overall accuracy of the cost model over various test input sizes. The *additive error* complements the relative error, for very short queries where relative error deviates significantly for a minute difference. Formally, we define the errors as follows:

$$\text{time per input} = \frac{\overline{\$_{\mathcal{A}}}}{\sum_{L_i \in S} |L_i|} \tag{24}$$

$$\text{additive error} = \frac{\$_{\mathcal{A}} - \overline{\$_{\mathcal{A}}}}{\sum_{L_i \in S} |L_i|} \tag{25}$$

$$\text{relative error} = \left(\frac{\$_{\mathcal{A}}}{\overline{\$_{\mathcal{A}}}} - 1 \right) \cdot 100(\%). \tag{26}$$

These measures are positive if a cost model overestimates costs, and negative if a cost model underestimates costs.

6.2 Correctness of cost models

We show our cost models for algorithm selection are accurate in two granularities: event count estimation, and cost estimation.

Event count accuracy. The cost model consists of estimations of many event counts such as O_{in}, O_{out} and O_{MISP}. Many event counts can be obtained trivially through other counts or list properties. For example, O_{out} of 2-MERGE generally equals to the size of input lists. However, some event counts such as O_{in} or O_{MISP} in the gallop algorithms (2-GALLOP, k-GALLOP), and O_{out} in k-way algorithms are estimated using a negative binomial distribution or Markov chain.

We thus show that the correctness of estimation for selected complex event counts: O_{in} of 2-GALLOP, and O_{out} of both k-way algorithms. In 450 Wikipedia queries, our model provides accurate estimation for each count. In detail, the 5th and 95th percentile errors for O_{in} of 2-GALLOP are -1.6% (under-est.) and 24.1% (over-est.), and same percentile errors for O_{out} of k-way algorithms are 0.0% and 23.1% as the actual computation can early terminate. Figure 5 represents both measured and estimated count values of nine randomly selected queries in Wikipedia corpus, identified by number of keywords, which shows each event count has marginal errors. In this experiment, we collect actual counts by using the program counters, and compare each value with the corresponding value in our model.

Cost estimation accuracy. We next discuss the accuracy of cost model. All 2-way models show reasonable accuracy in most of the synthetic pairwise queries (Figure 6). For the 2-MERGE model, each $5^{th}/95^{th}$ percentile of the error per input is -0.324/0.401 (ns) on the IVYBRIDGE machine, and -0.327/0.445 (ns) on the SANDYBRIDGE machine. For the 2-GALLOP model, each $5^{th}/95^{th}$ percentile of the same value in is -0.353/0.128 (ns) on the IVYBRIDGE, and -0.388/0.151 (ns) on the SANDYBRIDGE. The 2-SIMD model has 0.1 ns of maximum error per input which is most accurate among the tested models, because of the simplicity of the algorithm and the cost model. Figure 7 shows our cost model on different length ratio. In this experiments, we build two-list synthetic test cases with all possible combinations of r and p in Table 3.

All the cost models of each k-way algorithms also have reasonable accuracy on the synthetic dataset. Both k-MERGE and k-GALLOP cost models have low relative error in more than half of test cases (Table 6), which is less than $\pm 10\%$. Figure 8 shows detailed result of relative errors in various cases. For both models, although the relative error is high for some queries, the absolute error in this case is marginal.

Table 4: Statistics of real-data queries.

	Wikipedia Queries						Common Crawl Queries					
	shortest $\|L_1\|$	longest $\|L_n\|$	total $\sum\|L_i\|$	min ratio $\|L_2\|/\|L_1\|$	max ratio $\|L_n\|/\|L_1\|$	correlation $\|\cap L_i\|/\|L_1\|$	shortest $\|L_1\|$	longest $\|L_n\|$	total $\sum\|L_i\|$	min ratio $\|L_2\|/\|L_1\|$	max ratio $\|L_n\|/\|L_1\|$	correlation $\|\cap L_i\|/\|L_1\|$
mean	16.1K	1.32M	2.44	856.47	61.8K	3.9%	15.7K	435K	925K	1114.2	49.4K	9.3%
min	1	3.59M	5886	1.00	1.18	0%	1	8978	10.1K	1.00	1.05	0%
25%	108	411K	654K	2.59	109	0%	1624	152K	297K	1.68	10.45	0.13%
50%	629	819K	1.58M	9.74	1050	0.11%	6995	296K	621K	3.15	42.38	0.59%
75%	6594	1.51M	3.81M	64.14	7875	2.0%	21.8K	541K	1.43M	8.35	216	2.8%
max	559K	7.44M	13.6M	197K	4.46M	99.2%	130K	2.22M	4.97M	21.8K	831K	74.2%

Table 5: Estimated unit cost parameters.

Cost type	base latency (α)				branch misprediction (β)				memory cost (γ)			
Cost subtype (Symbol)	inner loop ($\alpha^{\text{in}}_{(y)}$)		outer loop ($\alpha^{\text{out}}_{(y)}$)		inner loop ($\beta^{\text{in}}_{(y)}$)		outer loop ($\beta^{\text{out}}_{(y)}$)		ref. cost ($\gamma^{\text{mem}}_{(y)}$)		miss. cost ($\gamma^{\text{miss}}_{(y)}$)	
System / Algorithm (y)	IVY	SANDY	IVY	SANDY	IVY	SANDY	IVY	SANDY	IVY	SANDY	IVY	SANDY
2-Gallop (2gallop)	1.36	1.31	1.36	1.64	1.29	1.36	5.47	6.64	0.55	0.65	53.6	70.8
2-Merge (2merge)	1.16	1.24	-	-	4.38	4.61	-	-	-	-	-	-
2-SIMD (2simd)	6.23	6.93	-	-	-	-	-	-	-	-	-	-
k-Gallop (kgallop)	0.35	0.36	2.97	3.04	2.91	3.03	2.53	2.01	0.74	0.80	59.4	67.4
k-Merge (kmerge)	0.53	0.66	2.91	2.69	6.35	5.91	2.49	2.12	-	-	-	-

Table 6: Correctness of the k-way intersection optimizer ($|L_1| = 2^{12}$).

Algo/SYSTEM	error type	min	25%	50%	75%	max
k-Gallop/IVY	Abs	-1.23	-0.05	-0.00	0.05	0.65
	Rel	-34.8%	-6.9%	0.4%	5.2%	37.4%
k-Gallop/SANDY	Abs	-1.10	-0.09	-0.02	0.03	0.64
	Rel	-31.7%	-11.6%	-1.4%	6.1%	23.4%
k-Merge/IVY	Abs	-1.39	-0.05	-0.00	0.02	0.87
	Rel	-25.6%	-3.4%	-0.1%	2.6%	38.5%
k-Merge/SANDY	Abs	-1.33	-0.08	-0.00	0.03	. 0.44
	Rel	-23.0%	-5.5%	-0.0%	3.4%	44.1%

Abs: ns / input size, Rel: %

Table 7: Comparison between algorithms.

Algorithm	$r_{\max}$					
	1	4	16	64	256	1024
2-GALLOP	2.51	2.36	1.19	0.63	0.43	0.21
2-MERGE	2.47	2.65	1.75	1.29	1.13	1.04
2-SIMD	2.02	2.26	2.10	1.99	1.96	1.95
2-Opt	1.54	1.73	0.96	0.46	0.23	**0.10**
SIMD V1	*1.57*	1.50	0.79	0.43	0.31	0.27
SIMD V3	2.19	**1.14**	*0.68*	*0.39*	*0.29*	0.26
SIMD Inoue	2.94	2.77	1.74	1.32	1.16	1.08
SIMD Gallop	2.30	1.70	0.87	0.53	0.40	*0.19*
STL	3.77	3.81	2.75	2.15	1.95	1.85
2-Opt All	**1.48**	**1.14**	**0.66**	**0.34**	**0.20**	**0.10**

unit: μs / input size

6.3 Comparison of Algorithms

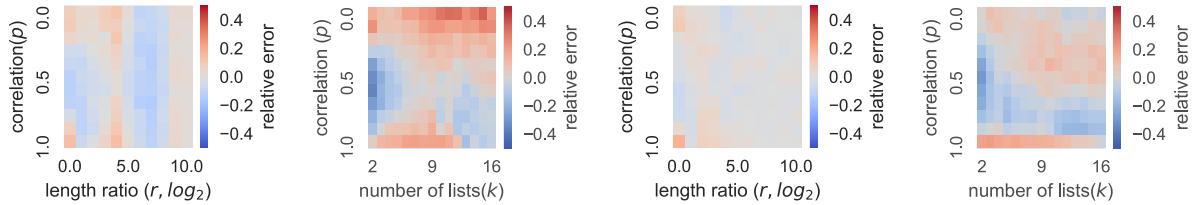
Component Algorithms. Our cost-based approach with 2-way algorithm outperforms any of the k-way algorithms in general. We compare 2-way algorithms and k-way algorithms in both synthetic dataset and real corpora. For scheduling 2-way intersection orders, we use the simple but effective state-of-the-art, namely "shortest vs shortest (SvS)" as 2-way scheduling algorithm [5], which intersects two shortest lists at a time.

2-GALLOP outperforms both of k-way algorithms in most of the synthetic data. 2-GALLOP is faster than k-GALLOP except few extreme cases such that r is close to 64 (Figure 10a-b). In this extreme case, the cost of k-GALLOP is very similar to the cost of 2-GALLOP (Figure 10c), because the cache performance of 2-GALLOP is slightly worse than k-GALLOP. That is, access pattern of 2-GALLOP causes more cache alignment problem than k-GALLOP. However, in general, 2-GALLOP has less required search trials and comparable cache efficiency compared to k-GALLOP. If all lists have similar lengths, either 2-SIMD or 2-MERGE outperforms all other algorithms in any correlation p (Figure 10d).

Also in the real-world datasets, our cost-based 2-way approach outperforms all k-way algorithms. On both Wikipedia corpus (Figure 11) and CommonCrawl web corpus (Figure 12), our approach is faster than both two k-way algorithms in **all** of the 450 queries. Moreover, the optimized approach completes 84.7% (Wiki) and 92.0% (web) of queries within 500μs on the IVYBRIDGE machine, while k-GALLOP completes only 62.2% (Wiki) and 49.6% (web) of queries within 500μs on the same IVYBRIDGE machine.

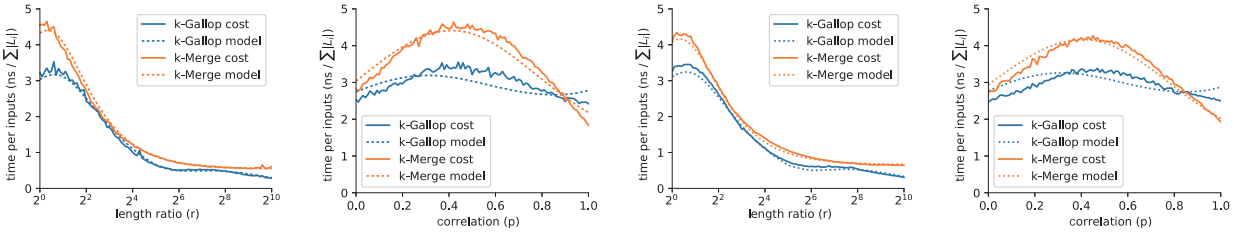
Our cost-based optimizer usually suggests 2-GALLOP extensively, and leverages 2-MERGE or 2-SIMD to process relatively rare cases of similar lengths. The optimizer applies 2-GALLOP at least 50% of the first step because the length ratio between first two shortest list is larger than 9.74 in 50% of queries. The optimizer always uses 2-GALLOP at the later steps because the minimum length ratio between the first intermediate result and the third list is at least 10.9 in 99.5% of the queries with 3+ keywords, for which 2-GALLOP is the fastest option. This supports our claims in Section 5.2 holds for web search scinario, which follows zipfian distribution.

State-of-the-art Algorithms. We compare the component algorithms and our cost-based optimizer with the state-of-the-art algorithms – SIMD Inoue, SIMD V1, SIMD V3 and SIMD Gallop. To compare algorithms, we build six scenarios with different maximum length ratio ($r_{\max}$) that is the length ratio between shortest and longest lists. Each scenario consists of 100 test cases that differ from the number



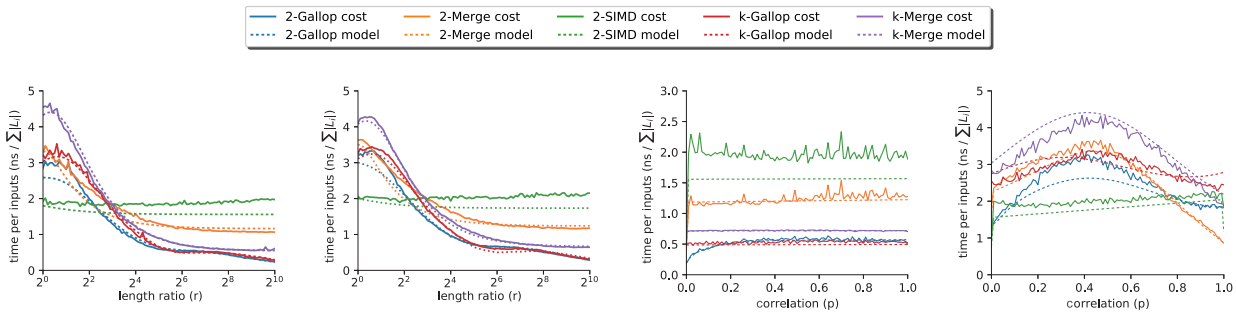
(a) k-GALLOP, $k = 4$ **(b)** k-GALLOP, $r = 1$ **(c)** k-MERGE, $k = 4$ **(d)** k-MERGE, $r = 1$

Figure 8: Heatmap of relative error for k-way algorithm in IVYBRIDGE, $|L_1| = 2^{12}$.



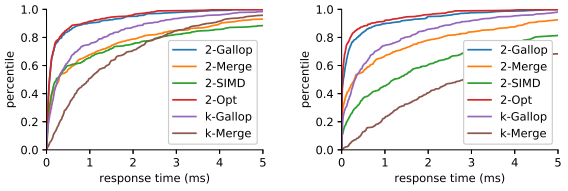
(a) IVYBRIDGE $p = 0.5$, **x-axis**: r **(b)** IVYBRIDGE $r = 1$, **x-axis**: p **(c)** SANDYBRIDGE $p = 0.5$, **x-axis**: r **(d)** SANDYBRIDGE $r = 1$, **x-axis**: p

Figure 9: Cost model of k-way algorithms in four-list scenarios, $|L_1| = 2^{12}$.



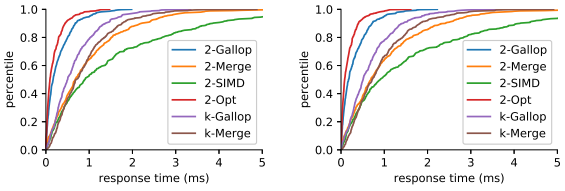
(a) IVYBRIDGE, $p = 0.5$, **x-axis** : r **(b)** SANDYBRIDGE, $p = 0.5$, **x-axis** : r **(c)** IVYBRIDGE, $r = 1$, **x-axis**: p **(d)** SANDYBRIDGE, $r = 1$, **x-axis**: p

Figure 10: Comparison of algorithms in four-list scenarios with $|L_1| = 2^{12}$.



(a) IVYBRIDGE **(b)** SANDYBRIDGE

Figure 11: CDF of response times for 450 query on Wikepedia corpus.



(a) IVYBRIDGE **(b)** SANDYBRIDGE

Figure 12: CDF of response times for 450 query on CommonCrawl web corpus.

of lists and correlation. Table 7 describes the average time for computing intersection of synthetic lists with each algorithm.

When deploying only one algorithm, using SIMD V3 has good results over a wide range of scenarios, while SIMD V1 and SIMD Gallop are optimal in some scenarios. SIMD V3 is the fastest algorithm when $4 \leq r_{\max} \leq 512$, SIMD V1 is the fastest when $r_{\max} = 1$, and SIMD Gallop is the fastest when $r_{\max} \geq 1024$. Both SIMD V1/V3 algorithms add element skipping to the merge-based algorithm, which are faster than search-based algorithms in the scenarios where the length ratio is not very large. If the length ratio is very large, then using search-based algorithm such as SIMD Gallop or Gallop is efficient. Even the SIMD Gallop is faster than 2-GALLOP in general, the difference is marginal.

Deploying the various algorithms contextually through the optimizer provides a faster option than deploying only one algorithm for diverse inputs. First, our optimizer considering only component algorithms (2-Opt) is (1) faster than all component algorithms in all scenarios, and (2) faster than SIMD V1 and V3 when $r_{\max}$ is 1 or at least 256. Second, our extended optimizer considering all component algorithms and state-of-the-art algorithms (2-Opt All) outperforms all other algorithms. Both optimizers suggest an appropriate algorithm based on the cost model of each algorithm very well, thus improve speed by recommending faster algorithms in each 2-way intersection step for diverse input scenarios.

7. RELATED WORK

List intersection operator has been widely used in many applications, for which various approaches are studied in the literature. Merge-based approaches simultaneously explore all lists, while search-based approaches [2, 5, 6, 10, 27] set one as a pivot (which may change) and use an efficient search to intersection candidates in the remaining lists. Thus, search-based algorithms is faster in case the lengths of the sets are widely distributed. On the other hand, [17, 21] claim that, with comparable length lists, a simple merge-based algorithm performs better than search-based algorithms. Hash-based approaches [11, 27] invest resources off-line to build hash structures to make intersection small, however, their total cost is much higher than merge- or search-based algorithms.

Recently, numerous works study how to speedup the list intersection algorithms by exploiting SIMD instructions. Schlegel *et al.* [26] adopt STTNI instruction (in SSE 4.2 instruction set) for intersection of 8-bit or 16-bit integer lists. However, STTNI instruction does not support data types larger than 16 bits, it which limits applicability. Lemire *et al.* [21] design two merge-based algorithms (SIMD V1 and V3), and search-based algorithm (SIMD gallop) with SIMD instructions. V1 or V3 is faster than SIMD gallop if the lengths of two lists are comparable, but SIMD gallop outperforms otherwise. Inoue *et al.* [17] improve such gain by increasing data parallelism: they use SIMD instructions to compare only a part of each item for filtering out unnecessary comparisons. SIMD instructions are also exploited to reduce branch misprediction [17, 21], an expensive overhead in some workloads, by replacing control flow into data flow called *If-Conversion* [1]. We include a representative SIMD algorithm in our optimization, and also compare our approach with these individual SIMD algorithms in Section 6.3.

In the context of web search, list intersection operator is generally executed on the regular architectures. In detail, a large search engine distributes computation by sharding the inverted index on hundreds or thousands of servers, such that each server maintains a shard corresponding to a set of web documents and their associated sorted postings lists. For a given user query, this engine processes the query in two main phases: matching and ranking. In the matching phase, each server intersects the corresponding postings lists, then results are aggregated from all servers to some servers for ranking. List intersection returns all results, rather than employing early-termination [7, 8, 12] to return a subset to reduce the ranking costs. In fact, the matching phase in a commercial search engine [15] returns a superset of the results and uses the ranking phase for further filtering. The ranking phase follows the matching phase [25], is typically performed in successive steps for pruning, and is extensively accelerated with custom hardware such as FPGAs [24] and ASICs [18] in commercial engines. Thus custom hardware reduces the ranking time, and subsequently introduces changes to the matching phase. In contrast, list intersection (or the matching phase) is still performed on the regular architectures without specialized hardware, which dominates query processes.

Shipping k-way algorithms to all servers has been favored in commercial search engine, with its pipelined implementation [23] and low memory requirement. A contradicting opinion of favoring 2-way algorithms has been considered in a different problem context where blocking, which requires storing intermediate results, is common [19]. Our work parameterizes the given hardware and generates a cost-optimal algorithm accordingly. As a result, we are applicable to systems using nonblocking pipelined algorithms, outperforms the existing state-of-the-art methods.

List compression encodings are devised for representing elements with fewer bits such as delta encoding [30, 31], and elias-fano encoding [13, 14, 22]. Such encodings are used for transferring the inverted index and its postings lists to or form storage or network devices when their bandwidth is substantially slower than main memory bandwidth. However, postings lists are managed in main memory and in an uncompressed layout for fast access hosted on server machines with large main memories, requiring a fast main memory-based intersection algorithm.

8. CONCLUSIONS

We study the problem of cost-based optimization for list intersection. We analytically and empirically show that shipping an identical k-way algorithm to all servers is inefficient. Instead, we develop a cost model adaptive to different hardware systems and inputs, and use a cost-based optimizer to generate an efficient scheduling of 2-way intersection computation using parameters reflecting dataset and modern architecture characteristics. We validate our framework experimentally and find that the cost models closely estimate the actual intersection cost, and as a result, 2-way intersection scheduling with our cost-based optimizer outperforms the state-of-the-art alternatives.

Acknowledgements

This work is supported by Microsoft Research Asia. We thank the IndexServe team in Microsoft Bing.

9. REFERENCES

[1] J. R. Allen et al. Conversion of control dependence to data dependence. *POPL*, 1983.

[2] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. *CPM*, pages 400–408, 2004.

[3] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. *SODA*, pages 390–399, 2002.

[4] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. pages 146–157. Springer, 2006.

[5] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *Journal of Experimental Algorithmics (JEA)*, 14, 2009.

[6] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inform. Proc. Lett.*, 5(SLAC-PUB-1679):82–87, 1975.

[7] A. Z. Broder et al. Efficient query evaluation using a two-level retrieval process. *CIKM*, pages 426–434, 2003.

[8] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. *ICDE*, pages 709–720, 2011.

[9] S. Chaudhuri. An overview of query optimization in relational systems. *PODS*, pages 34–43, 1998.

[10] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1), 2010.

[11] B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, Jan. 2011.

[12] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. *SIGIR*, pages 993–1002, 2011.

[13] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[14] R. M. Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[15] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He. Bitfunnel: Revisiting signatures for search. *SIGIR*, pages 605–614, 2017.

[16] D. Hu and A. Yin. Approximating the negative hypergeometric distribution. *International Journal of Wireless and Mobile Computing*, 7(6):591–598, 2014.

[17] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[18] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *ISCA*, pages 1–12, 2017.

[19] R. Krauthgamer, A. Mehta, V. Raman, and A. Rudra. Greedy list intersection. *ICDE*, pages 1033–1042, 2008.

[20] T. Lee, J. Park, S. Lee, S.-w. Hwang, S. Elnikety, and Y. He. Processing and optimizing main memory spatial-keyword queries. *PVLDB*, 9(3):132–143, 2015.

[21] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016.

[22] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. *SIGIR*, pages 273–282, 2014.

[23] E. Pitoura. *Pipelining*, page 2117. Springer US, 2009.

[24] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ISCA*, pages 13–24, 2014.

[25] K. M. Risvik et al. Maguro, a system for indexing and searching over very large text collections. *WSDM*, pages 727–736, 2013.

[26] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using SIMD instructions. *ADMS*, pages 1–8, 2011.

[27] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *PVLDB*, 2(1):838–849, 2009.

[28] M. Yang, B. Ding, S. Chaudhuri, and K. Chakrabarti. Finding patterns in a knowledge base using keywords to compose table answers. *PVLDB*, 7(14):1809–1820, 2014.

[29] A. Yasin. A top-down method for performance analysis and counters architecture. *ISPASS*, pages 35–44, 2014.

[30] J. Zhang et al. Performance of compressed inverted list caching in search engines. *WWW*, pages 387–396, 2008.

[31] M. Zukowski et al. Super-scalar ram-cpu cache compression. *ICDE*, page 59, 2006.