Processing and Optimizing Main Memory Spatial-Keyword Queries

Taesung Lee[1] Jin-woo Park[2] Sanghoon Lee[2]
Seung-won Hwang[1] Sameh Elnikety[3] Yuxiong He[3]
[1]Yonsei University [2]POSTECH [3]Microsoft Research

ABSTRACT

Important cloud services rely on spatial-keyword queries, containing a spatial predicate and arbitrary boolean keyword queries. In particular, we study the processing of such queries in main memory to support short response times. In contrast,current state-of-the-art spatial-keyword indexes and relational engines are designed for different assumptions. Rather than building a new spatial-keyword index, we employ a cost-based optimizer to process these queries using a spatial index and a keyword index. We address several technical challenges to achieve this goal. We introduce three operators as the building blocks to construct plans for main memory query processing. We then develop a cost model for the operators and query plans. We introduce five optimization techniques that efficiently reduce the search space and produce a query plan with low cost. The optimization techniques are computationally efficient, and they identify a query plan with a formal approximation guarantee under the common independence assumption. Furthermore, we extend the framework to exploit interesting orders. We implement the query optimizer to empirically validate our proposed approach using real-life datasets. The evaluation shows that the optimizations provide significant reduction in the average and tail latency of query processing: 7- to 11-fold reduction over using a single index in terms of 99th percentile response time. In addition, this approach outperforms existing spatial-keyword indexes, and DBMS query optimizers for both average and high-percentile response times.

1. INTRODUCTION

Several important applications generate queries that contain a spatial predicate and arbitrary boolean keyword predicates. For example in display advertising [36] (e.g., Microsoft Ads platform), a large set of geo-tagged advertisements is queried. A query contains a spatial predicate reflecting user current location, and complex keyword predicates consisting of disjunction and conjunction of keywords from the user profile. Similarly, this query type is used to customize web pages dynamically: Major web portals (e.g., MSN.com) exploit the user location and keyword tags from the user profile to select the set of components forming the requested web page.

Processing the generated spatial-keyword queries is part of a complex distributed system with layered components. Each component is designed to provide consistently low response times [17, 24], imposing latency constraints of typically a few milliseconds for the average latency and 10s of milliseconds for the tail latency, which is the high percentile response time (e.g., $99^{th}\%$). Consequently, the spatial-keyword query response time must be short and predictable: A query that takes too long results in lost revenue or customer dissatisfaction [31]. To meet these requirements, data is typically managed in main memory to provide fast access to data. Processing these queries quickly is, however, much harder.

Several spatial-keyword indexes have been proposed, and a recent paper [12] categorizes them into spatial-first, keyword-first or tight integration of spatial and keyword indexes. These indexes are mainly disk-based, and are therefore effective in reducing IO; they are, however, not designed for reducing the tail latency in main memory. We implement these state-of-the-art indexes, and measure their response time while used in main memory. Table 1 shows that they do not meet the latency requirements.

Table 1: Comparison to prior indexes (details in Section 8.6).

	Spatial-keyword indexes				This work
	SFC-QUAD	*SKIF*	*S2I*	*IR-tree*	$\mathcal{I} + \mathcal{P}$ *Opt*
Average (ms)	537.4	625.8	91.8	54.3	1.8
$99^{th}\%$ (ms)	9691	5892	39.5	156.5	24.1
Size (GB)	9.8	8.0	19.8	22	9.5

Rather than building a new main memory spatial-keyword index, we take a different approach using two base indexes, a spatial index and a keyword index as depicted in Figure 1. This approach achieves substantially lower latencies; the average is 1.8 ms compared to more than 50 ms from prior indexes. Furthermore, we demonstrate that this approach offers significant advantages by leveraging the underlying index capabilities or physical properties, such as "interesting orders" [32] for spatial proximity or popularity, or prefix search for keyword predicates. There are, however, some challenging problems to realize this approach.

Both spatial and keyword predicates must be treated as first-class predicates that can be reordered, and processed using efficient main-memory operators. A relational engine with spatial and keyword indexing can process such queries, but they lack the techniques we propose. For example, our approach obtains substantially lower average response times (more than 10x reduction) compared to PostgreSQL (disk-based engine) and MonetDB (main-memory engine), both equipped spatial and keyword indexing extensions as shown in Table 2. In addition, Table 2 shows that rewrit-

Proceedings of the VLDB Endowment, Vol. 9, No. 3

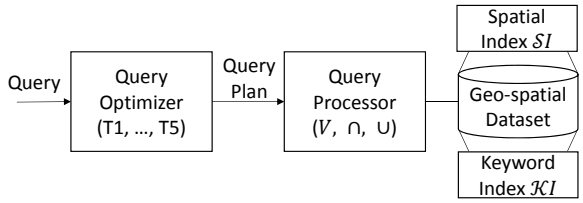


Figure 1: Framework for query processing and optimization.

ing the queries using some of the techniques from our proposal reduces average latency by over 60% in for both MonetDB and PostgreSQL.

Table 2: Comparison to DBMS (more details in Section 8.7).

	Existing DBMS		Modified Query on DBMS		This work
	MonetDB	PostgreSQL	MonetDB +T1-5	PostgreSQL +T1&5	$\mathcal{I}+\mathcal{P}$ *Opt*
Average (ms)	1405	21.3	529	8.38	1.8
$99^{th}\%$ (ms)	3462	255.3	2372	94.8	24.1
Size (GB)	10.5	7.0	10.5	7.0	9.5

In particular, a spatial index often retrieves the superset of objects satisfying the spatial predicate, allowing room for optimization that has been overlooked by prior systems. A predicate is classified as sargable if it can be processed solely with an index, and as non-sargable otherwise (e.g., processed on each object) [32, 8]. Traditional approaches are limited to heuristic rule-based techniques such as pushing down sargable predicates, and popping up non-sargables. In contrast, we provide a cost-based optimization handling sargable and non-sargable predicates, resulting in rewriting the spatial and keyword predicates.

We develop the solution in three steps. We first introduce main-memory efficient operators and develop their cost model. Prior work does not provide operators and cost models for this class of queries with both spatial and complex keyword predicates. Second, we introduce five transformation rules to optimize queries, including novel transformation rules as well as rules inspired by traditional query optimization techniques [5, 11, 26]. We provide formal analysis of the complexity and optimality of the transformations which has not been explored in existing boolean [29] and relational [32] query optimization. These rules significantly reduce the latencies. While the search space is exponential in the number of operators, the proposed transformation rules are computationally efficient with near linear time cost in the number of operators. Moreover, they generate a plan with an approximation bound compared to the optimal in case of independent predicates, which has not been established before. Third, we extend the framework to leverage interesting orders.

To assess the benefits of this approach, we implement the proposed techniques as well as several base indexes and other spatial-keyword indexes, and evaluate them experimentally using several real-life datasets (Flickr, Wikipedia, and Twitter) with diverse properties using a workload derived from a Bing Mobile log. The empirical results show that the proposed approach provides low latencies over a wide range of scenarios. In particular, using a pyramid index and an inverted index, we observe significant latency reduction: The $99^{th}\%$ response time shows 7- to 11-fold reduction over using a single index. Furthermore, we evaluate our approach with a different pair of base indexes, namely *R-tree* and *Trie* to show the applicability of this approach, enabling keyword prefix search. We also report how our approach compares to two relational engines, MonetDB [7] and PostgreSQL [2, 1].

We structure the paper around our contributions as follows: (1) We define three main memory operators as building blocks to represent query plans and develop a cost model (Section 3 and 4). (2) We introduce the optimization techniques to efficiently reduce the search space, with formal approximation bounds (Section 5). (3) We show how to leverage index features such as prefix search and interesting orders (Section 6 and 7). (4) We evaluate our approach experimentally with real-world datasets and compare it to prior work (Section 8).

2. DATA AND QUERY MODELS

2.1 Data Model

We assume a spatial-keyword dataset $\mathcal{D}$, and its size $D = |\mathcal{D}|$. Each object $o \in \mathcal{D}$ is represented as $(\text{ID}, location, keywords)$, where ID is a system-generated primary key, $location$ =[latitude, longitude] is a point location, and $keywords$ is a set of keywords.

2.2 Query Model

A spatial-keyword query specifies spatial and keyword predicates. We focus first on processing queries with boolean predicates, and discuss queries that exploit the available interesting orders such as nearest neighbour queries in Section 7. Formally, a BASE spatial-keyword query is a pair $Query = (S, T)$ where S is a spatial predicate, and T is a keyword predicate. The spatial predicate S consists of a point location p =[latitude, longitude] and a radius r so that only the objects whose distance from p is within r are selected. The keyword predicate T is a combination of operators on keywords, represented by *AND* and *OR*. The keyword predicate follows this grammar:

$$T \quad \rightarrow \quad (T \text{ AND } T) \mid (T \text{ OR } T) \mid keyword$$

AND indicates objects that satisfy both operand predicates, OR indicates objects that satisfy at least one of the two operand predicates, and $keyword$ is a keyword such as 'travel'. This grammar allows rich combinations of keyword predicates with arbitrary conjunctions and disjunctions.

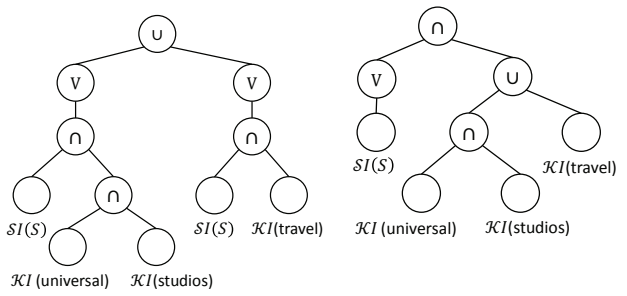
Running Example. We use a query as a running example throughout the paper. A user travels in Los Angeles and requests a web page on Universal Studios from a hotel. The ads system generates the following (simplified query) to display three ads with the requested web page: $Q_1 = (S_1, T_1)$ where spatial predicate S_1 = (location=[34.053490,-118.245323], radius=1.6 mile), and keyword predicate T_1 ="((universal AND studios) OR travel)". Notice that we want to process all matching ads that satisfy the query (or, perfect recall) rather than only a subset as in typical search scenarios, such that results are further processed to optimize for other business objectives: increasing the diversity with respect to ad providers, maximizing revenues, and budgeting.

3. QUERY PLAN

This section is the first step in our solution where we introduce three algebraic operators and use them to construct the query plan which is a tree. Then, we show how to generate a plan for a query. For example, Figure 2 shows two plans for Q_1.

3.1 Leaves

A leaf in the plan tree represents a set of objects from a base index. $\mathcal{KI}(t)$ is the set of objects from the keyword index $\mathcal{KI}$ with keyword t. $\mathcal{SI}(S)$ is the set of objects from a spatial index $\mathcal{SI}$ in



(a) Plan A. (b) Plan B.

Figure 2: Two example plans for Q_1.

region S. Since spatial indexes approximate the spatial predicate using cells or minimum bounding boxes, some objects in $\mathcal{SI}(S)$ may not satisfy S, requiring further verification.

3.2 Operators

We introduce three algebraic operators: (1) **Verify:** $V(Plan, Pred)$ returns a set of objects in $Plan$ satisfying the boolean predicate $Pred$ on the object attributes. (2) **Union:** $Plan_1 \cup Plan_2$ returns a set of objects in any of $Plan_1$ or $Plan_2$. (3) **Intersect:** $Plan_1 \cap Plan_2$ returns a set of objects in all of $Plan_1$ and $Plan_2$. We denote consecutive intersections $Plan_1 \cap (Plan_2 \cap (\ldots \cap Plan_M))$ as $\bigcap_{1 \le i \le M}^{W} Plan_i$, and we use a similar notation for consecutive unions.

3.3 Query Plan

We define a query plan recursively using the three operators:

$$\begin{aligned} PLAN &\rightarrow (PLAN \cup PLAN) \mid (PLAN \cap PLAN) \\ PLAN &\rightarrow V(PLAN, Pred) \mid \mathcal{KI}(t) \mid \mathcal{SI}(S) \end{aligned}$$

3.4 Generating a Query Plan

We map a query to a plan using the following mapping:

$$\begin{aligned} &BasePlan(Query(S,T)) = BasePlan(S) \cap BasePlan(T) \\ &BasePlan(T_1 \text{ OR } T_2) = BasePlan(T_1) \cup BasePlan(T_2) \\ &BasePlan(T_1 \text{ AND } T_2) = BasePlan(T_1) \cap BasePlan(T_2) \\ &BasePlan(keyword) = \mathcal{KI}(keyword) \\ &BasePlan(S) = V(\mathcal{SI}(S), S) \end{aligned}$$

For example, $BasePlan(Q_1)$ is depicted as Plan B in Figure 2. The query optimizer uses the resulting plan as input.

4. COST MODEL

The optimizer compares query plans using their costs. In this section, we develop a cost model for executing a plan in main memory assuming *independence* among spatial and keyword predicates. We can use several indexes for $\mathcal{KI}$ and $\mathcal{SI}$ as the leaves of a plan as discussed in Section 6 provided that they return sorted ID lists.

Section 4.1 discusses how each operator is implemented, and defines its cost as the number of comparisons and memory accesses. Section 4.2 completes the cost model as an aggregation of operator costs, weighted by unit costs.

4.1 Operator Implementation and Cost

Verify: The verify operator accesses each ID in the input list, uses the ID to access the object attributes in main memory to evaluate a boolean predicate. The output is a list of IDs for the objects satisfying the predicate. The cost of verification operation is dominated by the main memory access to the object attributes. Thus, we define its cost as the number of main memory accesses.

$$C_V(L) = L \tag{1}$$

Intersect: Intersection of two sorted lists outputs all IDs appearing in both lists. Unlike the merge algorithm requiring to scan all items from both lists, we implement gallop search [6]: The key idea is to choose the shorter list and locate its head ID, and then search for this ID in the longer list. Given two lists with lengths L_1 and L_2 (and $L_1 \le L_2$), the cost is a function of the lengths:

$$f_\cap(L_1, L_2) = \sum_{i=1}^{L_1} (\log_2 2d_i + \log_2 d_i) \tag{2}$$

where d_i is the distance between the $(i-1)$-th target ID and i-th target ID in the longer list.

To reduce tail latency, we set the cost of intersection as the upper bound using Lagrange multipliers as below:

$$C_\cap(L_1, L_2) = L_1 \cdot (2 \log_2 (L_2/L_1) + 1) \tag{3}$$

Union: The inputs are two sorted lists with lengths L_1 and L_2. The output is a sorted list of IDs that are in any of the two lists. The implementation scans both lists, making the cost a function of their lengths due to data access locality.

$$C_\cup(L_1, L_2) = L_1 + L_2 \tag{4}$$

Length of operator output: Computing the costs requires the lengths $L(\cdot)$ of the operands, and they are estimated recursively as follows, where D is the total number of objects in the dataset $\mathcal{D}$:

$$\begin{aligned} L(l_1 \cup l_2) &= D \cdot (1 - (1 - L(l_1)/D)(1 - L(l_2)/D)) \\ L(l_1 \cap l_2) &= L(l_1)L(l_2)/D \end{aligned} \tag{5}$$

For the base cases, $L(\mathcal{SI}(S))$ and $L(\mathcal{KI}(t))$ are directly obtained from indexes. We omit the length of the verification operator output, as it is not needed for the optimizations.

4.2 Cost Model

We define the cost of a plan by combining both comparison cost and memory access cost using two parameters: α is the unit cost for ID access in the CPU cache, and β is the cost of object access in main memory. α is the unit cost of intersection and union operation due to their locality friendly access, and β is the unit cost of verification since verification requires main memory access to exploit detailed attributes of objects. We determine α and β empirically. For a plan P, we define its cost $C(P)$ as follows:

$$\begin{aligned} C(P_1 \cap P_2) &= \alpha C_\cap(L(P_1), L(P_2)) + C(P_1) + C(P_2) \\ C(P_1 \cup P_2) &= \alpha C_\cup(L(P_1), L(P_2)) + C(P_1) + C(P_2) \\ C(V(P, Pred)) &= \beta C_V(L(P)) + C(P) \\ C(\mathcal{SI}(S)) &= 0 \\ C(\mathcal{KI}(t)) &= 0 \end{aligned} \tag{6}$$

Note that the index lookup operations $\mathcal{SI}(S)$ and $\mathcal{KI}(t)$ only retrieve a pointer to the prematerialized list of the indexes. Therefore, their costs are marginal and hence we set $C(\mathcal{SI}(S)) = 0$ and $C(\mathcal{KI}(t)) = 0$.

5. QUERY OPTIMIZATION

This section proposes our cost-based optimizer, aiming to find the plan with the least estimated cost (among all possible plans

represented as $\mathcal{G}$) assuming independence among spatial and keyword predicates. $|\mathcal{G}|$ is, however, exponential with respect to the number of operators making exhaustive enumeration too expensive. We introduce five transformations which reduce the complexity of the search algorithm from exponential to near linear time while offering approximation guarantees on the cost of the optimized plan compared to the optimal.

5.1 Overview

Table 3 summarizes the five transformations T1-T5. Each transformation is applied in sequence, gradually reducing the search space and enabling the next transformation. A desirable transformation has three requirements: (a) low transformation complexity, (b) effective reduction of search space, (c) good approximation guarantee on the optimality. As shown in Table 3, the transformations satisfy all these requirements.

Among the five transformations, we propose novel rules (T1 and T5) which significantly reduce the the cost of verification operation together with intersection, which has not been explored in existing boolean query optimization [29]. T2, T3 and T4 follow the intuition used in keyword or database query optimization [5, 11, 26]: we prove their complexity and optimality that the prior work does not offer. In addition, we integrate these five transformations into our optimizer and formally analyze their effectiveness to identify a solution that is provably close to theoretic optimal requiring exponential times. We overview these techniques:

T1. *Single verification pop up:* Delaying verification to the last does not compromise optimality, while reducing search space by considering only the plans in the form of $V(Plan_{Core})$ where $Plan_{Core}$ is a plan without any verify operator (Section 5.2).

T2. *Intersection push down:* We observe that, for a plan containing unions and intersections, pushing intersections down (*i.e.*, performing intersections before unions) often reduces the cost. This step generates a query plan that has unions near the top and intersections at the lower levels, motivating the next two transformations that reorder intersections and unions (Section 5.3).

T3. *Most selective intersection first:* To intersect k lists, we show that an optimal solution is to intersect them in an increasing order of their lengths (Section 5.4).

T4. *Enhanced Huffman union plan:* To union k lists, we show that an optimal solution is to build a (modified) Huffman tree (Section 5.5).

T5. *Verification selection:* We can skip intersections at the expense of verifying more objects. We propose two algorithms: (1) *ExamAll* makes an optimal decision, and (2) *ExamBest* is a linear time algorithm with theoretical guarantee (approximately $2\times$ the optimal in the worst case) and effective in practice (Section 5.6).

Note that each of these five transformations produces a plan with a specific form that the next transformation takes as input. Therefore, they must be applied in the sequence from T1 to T5. For example, the two plans shown in Figure 2 are not in the form that we can apply T2 directly; T1 must be applied first. We describe the algebraic form of the plan before and after each transformation when elaborating these techniques.

5.2 T1 - Single Verification Pop Up

Although a verification operation can be performed on any stage of a plan tree, it only needs to be performed once as the very last operation to maintain the optimality. In other words, we do not need to consider any plan containing a verification operation as an intermediate operation, which greatly reduces the search space. Theorem 1 formally analyzes the principle behind the transformation.

THEOREM 1. *Any plan can be transformed to a plan with a single verification operation as the last operation of the plan without increasing its cost.*

PROOF. We prove the claim in three cases.

Case 1: The optimal plan has a verify operation before intersection. For some $Plan_1$, $Plan_2$, and $Pred_2$, we can assume the optimal plan is $Plan_1 \cap V(Plan_2, Pred_2)$ without loss of generality. An alternative plan $V(Plan_2, Pred_1 \text{ AND } Pred_2)$ is computed to the same result where we use $Pred_1$ to denote the predicate that $Plan_1$ is verified for, *i.e.*, verifying each object from the entire date set $\mathcal{D}$ using $V(\mathcal{D}, Pred_1)$ returns the same set of objects as $Plan_1$. Thus, we have

$$\begin{aligned} &C(Plan_1 \cap V(Plan_2, Pred_2)) \\ &\quad = \alpha C_{\cap}(L(Plan_1), L(V(Plan_2, Pred_2))) \\ &\quad + \beta L(Plan_2) + C(Plan_1) + C(Plan_2) \\ &\quad \geq \beta L(Plan_2) + C(Plan_2) \\ &\quad = C(V(Plan_2, Pred_1 \text{ AND } Pred_2)) \end{aligned} \tag{7}$$

Therefore, we can rewrite $Plan_1 \cap V(Plan_2, Pred_2)$ as $V(Plan_2, Pred_1 \text{ AND } Pred_2)$ without increasing cost.

Case 2: The optimal plan has a verify operation before union. We can assume the optimal plan is $V(Plan_1, Pred_1) \cup V(Plan_2, Pred_2)$ without loss of generality. We have an alternative plan $V(Plan_1 \cup Plan_2, Pred_1 \text{ AND } Pred_2)$ for the same query. Then, we have

$$\begin{aligned} &C(V(Plan_1, Pred_1) \cup V(Plan_2, Pred_2)) \\ &\quad = \beta L(V(Plan_1, Pred_1)) + \beta L(V(Plan_2, Pred_2)) \\ &\quad + C(Plan_1) + C(Plan_2) + \alpha C_{\cup}(L(Plan_1), L(Plan_2)) \\ &\quad \geq \beta L(Plan_1 \cup Plan_2) \\ &\quad + C(Plan_1) + C(Plan_2) + \alpha C_{\cup}(L(Plan_1), L(Plan_2)) \\ &\quad = C(V(Plan_1 \cup Plan_2, Pred_1 \text{ AND } Pred_2)) \end{aligned} \tag{8}$$

Therefore, we can always rewrite $V(Plan_1, Pred_1) \cup V(Plan_2, Pred_2)$ as $V(Plan_1 \cup Plan_2, Pred_1 \text{ AND } Pred_2)$ without increasing cost.

Case 3: There are two consecutive verification operations. We can always merge them into one to reduce the redundant memory accesses:

$$V(V(Plan, Pred_1), Pred_2) = V(Plan, Pred_2 \text{ AND } Pred_1) \tag{9}$$

Thus, by repeatedly applying the above three cases, we can transform any optimal plan to an equal-cost plan with a single verification as the last operation. □

Theorem 1 supports that considering only the plans in the form of $V(Plan_{Core})$ is sufficient for ensuring optimality, where $Plan_{Core}$ is a plan without any verification operator. This reduces search space to $Plan_{Core}$, as Table 3 shows such a plan for Q_1. This reduction is significant, as $\mathcal{G}$ includes all plans where verification operators are independently placed at any node (2^K possible ways for plans with K leaves). By only optimizing $Plan_{Core}$,

Table 3: The five transformation overview. Underline and boldface indicate the transformation in the example plan. Overall space reduction: $2^{K+F+\sum_{i=1}^{N}(M_i-1)} \cdot C_N \cdot \prod_{i=1}^{N} M_i! C_{M_i}$, and overall approximation bound: $(2+\frac{\alpha}{\beta}\lceil \log_2 N \rceil)(\frac{5}{3})^F$ or $(\frac{5}{3})^F$.

T_i	Transformation Complexity	Space Reduction	Approximation Bound	Example plan
Base	-	-	-	$V\big(\mathcal{SI}(S)\big) \cap \Big(\big(\mathcal{KI}(universal) \cap \mathcal{KI}(studios)\big) \cup \mathcal{KI}(travel)\Big)$
T1	$O(\sum_i M_i)$	2^K	1	$V\underline{\Big(}\mathcal{SI}(S) \cap \big(\mathcal{KI}(universal) \cap \mathcal{KI}(studios)\big) \cup \mathcal{KI}(travel)\underline{\Big)}$
T2	$O(F)$	2^F	$\big(\frac{5}{3}\big)^F$	$V\Big(\big(\underline{\boldsymbol{\mathcal{SI}(S)}} \cap \mathcal{KI}(universal) \cap \mathcal{KI}(studios)\big) \cup \big(\underline{\boldsymbol{\mathcal{SI}(S)}} \cap \mathcal{KI}(travel)\big)\Big)$
T3	$O(M \log_2 M)$ where $M = \max_i M_i$	$\prod_{i=1}^{N} M_i! C_{M_i}$	1	$V\Big(\underline{\big(\boldsymbol{\mathcal{KI}(studios)} \cap \boldsymbol{\mathcal{KI}(universal)}\big) \cap \boldsymbol{\mathcal{SI}(S)}}\Big) \cup \big(\mathcal{SI}(S) \cap \mathcal{KI}(travel)\big)\Big)$
T4	$O(N)$	C_N	1	$V\Big(\big(\big(\mathcal{KI}(studios) \cap \mathcal{KI}(universal)\big) \cap \mathcal{SI}(S)\big) \cup \big(\mathcal{SI}(S) \cap \mathcal{KI}(travel)\big)\Big)$
T5	$O(\sum_{i=1}^{N} M_i)$ or $O(\prod_{i=1}^{N} M_i)$	$2^{\sum_{i=1}^{N}(M_i-1)}$	$2+\frac{\alpha}{\beta}\lceil \log_2 N \rceil$ or 1	$V\Big(\big(\big(\mathcal{KI}(studios) \cap \mathcal{KI}(universal)\big) \cap \boldsymbol{\mathcal{SI}(S)}\big) \cup \big(\mathcal{SI}(S) \cap \mathcal{KI}(travel)\big)\Big)$

search space is reduced by $\Omega(2^K)$ time. T2 takes this reduced plan space in the form of $V(Plan_{Core})$ as input.

5.3 T2 - Intersection Push Down

We observe that, for a plan with a mix of unions and intersections, pushing down intersections (*i.e.*, performing intersections before unions) often reduces the cost. Intuitively, with intersection first, we are likely to reduce the lengths of lists to union. On the other hand, if we union first, we must read all objects in the lists. Specifically, out of two cases: union first ($l_0 \cap (l_1 \cup l_2)$) and intersection first ($(l_0 \cap l_1) \cup (l_0 \cap l_2)$), if l_0 is not the longest list, the intersection first costs less. Even if l_0 is longer than the union of the two lists l_1 and l_2, which is less likely, the intersection first costs at most $5/3$ times the union first.

THEOREM 2. *For three lists l_0, l_1, l_2, intersection first $(l_0 \cap l_1) \cup (l_0 \cap l_2)$ costs less than union first $l_0 \cap (l_1 \cup l_2)$ if l_0 is shorter than either of l_1 or l_2. Otherwise, the cost of the intersection first is at most $\frac{5}{3}$ times the union first.*

The proof is omitted due to the page limit.

Supported by Theorem 2, we perform intersection push down on the core plan. That is, we rewrite any intermediate node $Plan_0 \cap (Plan_1 \cup Plan_2)$ as $(Plan_0 \cap Plan_1) \cup (Plan_0 \cap Plan_2)$. We extend Theorem 2 to quantify the impact of intersection push down for complex query plans which have more than one possible push-downs (*e.g.*, $Plan_0 \cap ((Plan_1 \cap (Plan_2 \cup Plan_3)) \cup Plan_4)$ in Section 5.7.

Note that after applying T2, we refine the form of the core plan as Equation 10 that T3 takes as input:

$$Plan_{Core} = \bigcup_{1 \le i \le N}^{\breve{W}} \bigcap_{1 \le j \le M_i}^{\hat{W}_i} Plan_{i,j} \ . \tag{10}$$

where $\breve{W}$ denotes a union tree and $\hat{W}_i$ denotes an intersection tree. For instance, as shown in Table 3, we transform the core plan into T2 example form.

Using this transformation, the space reduction is 2^F where F is "the maximum number of factorizations," indicating the number of intersection first or union first decisions we can make (which is often called 'factorization' or 'distribution' in algebra). For example, the core plan after T1 shown in Table 3 has only one possible factorization (factoring out $\mathcal{SI}(S)$), and hence we have 2^1 times space reduction.

5.4 T3 - Most Selective Intersection First

To intersect K lists l_1, ..., l_K, we show that an optimal intersection plan follows the order of the increasing lengths of the lists. Intuitively, the cost of an intersection operation mainly depends on the size of the shorter list, and intersecting the shortest two lists produces an even shorter list. More precisely, we choose the two shortest lists to intersect first. Then, we intersect their results with the shortest list among the remaining. Theorem 3 states the optimality of this transformation under one assumption: The length of every list is less than 1/2 of the number D of the total objects, *i.e.*, $L(l_i) < \frac{1}{2}D$. This assumption is reasonable as we usually do not keep a keyword whose index contains more than half of the total objects in the entire data set, but rather take it as in the stop list.

THEOREM 3. *Let $l_1, \ldots, l_K$ represent K lists. Assume that $L(l_i) < \frac{1}{2}D$ for all i. Without loss of generality, suppose the lists are numbered in an increasing order of their lengths, i.e., $L(l_i) \le L(l_j)$ when $i < j$. To minimize the cost of intersecting the K lists when they are independent, an optimal plan is $(((l_1 \cap l_2) \cap l_3) \ldots \cap l_K)$, which intersects the lists in an increasing order of their lengths.*

The proof is omitted due to the page limit.

Therefore, whenever we have consecutive intersections in our plan, we rewrite them to intersect in the order of increasing lengths; we denote this intersection tree by $\hat{W_i}^*$ in $\bigcap_{1 \le j \le M_i}^{\hat{W_i}^*} Plan_{i,j}$ where $Plan_{i,j} \in \{\mathcal{SI}(S), \mathcal{KI}(t)\}$. For example, $\mathcal{SI}(S) \cap \mathcal{KI}(studios) \cap \mathcal{KI}(universal)$ will be processed as $(\mathcal{KI}(studios) \cap \mathcal{KI}(universal)) \cap \mathcal{SI}(S)$ as shown in Table 3.

The space reduction of this transformation is the number of possible intersection trees, which is the number of full binary trees times the number of permutations of the leaves. The number of full binary trees with K leaves is Catalan number $C_K = \prod_{k=2}^{K} \frac{K+k}{k}$ [25] and there are $K!$ ways to place the leaves, i.e., we have $K! \cdot C_K$ cases. As we have M_1, M_2, ..., M_N such intersections of length in the core plan (as shown in Equation 10), this transformation achieves up to $\prod_{i=1}^{N} M_i! \cdot C_{M_i}$ times space reduction.

5.5 T4 - Enhanced Huffman Union Plan

This section discusses the optimal order to perform union operations on K lists, l_1, ..., l_K. The cost of a union operation depends on the lengths of the two input lists. Intuitively, we choose two lists that minimize the output, which are the two shortest lists.

We first introduce our transformation with a simple case where we assume the lists are non-overlapping, *i.e.*, $L(l_i \cup l_j) = L(l_i) + L(l_j)$ for $i \neq j$. We map the problem of finding optimal union tree to the problem of minimizing the weighted path length in the Huffman code problem with lists as symbols, and their lengths as weights [21]. The obtained Huffman tree is the optimal union tree for the non-overlapping lists.

We extend the Huffman tree algorithm above to address general *independent* lists: We revise the intermediate result size with proper estimation. More precisely, we maintain a heap containing the lengths of all lists. After choosing the shortest two lists from the heap, we estimate the result size as $L(l_i \cup l_j) = L(l_i) + L(l_j) - L(l_i)L(l_j)/D$, and insert this value into the heap instead of using a simple sum of $L(l_i)$ and $L(l_j)$. We repeat this process until we have only one node in the heap. This order of choosing lists is an optimal union tree for the K lists, formally analyzed in Theorem 4.

THEOREM 4. *The enhanced Huffman union transformation produces an optimal union tree for multiple lists with the minimum cost.*

The proof is omitted due to the page limit.

Therefore, whenever we have consecutive unions, we build a union tree using the enhanced Huffman tree algorithm; we denote this tree by $\breve{W}^*$ as in $\bigcup_{1\leq i\leq N}^{\breve{W}^*} Plan_i$. The complexity of this transformation is $N \log_2(N)$. Since the number of union trees is exactly the same as the number of intersection trees, and we have unions of N lists in the core plan, this transformation reduces the search space by $N \cdot C_N$ where C_N is the N-th Catalan number.

5.6 T5 - Verification Selection

For a given plan $Plan = V(Plan_{Core})$, we can relax its core plan $Plan_{Core}$ to obtain another core plan ${Plan_{Core}}'$, where ${Plan_{Core}}'$ produces a superset of results of $Plan_{Core}$, i.e., ${Plan_{Core}}' \supset Plan_{Core}$. But $V({Plan_{Core}}')$ can still produce the same results as $V(Plan_{Core})$ by examining potentially more objects by verification. Specifically, instead of intersecting a list to reduce the number of objects to verify, we can leave them to the verification operation, saving intersection cost but potentially increasing verification cost. For example, instead of performing $V(\mathcal{KI}(universal) \cap \mathcal{KI}(studios))$, we can *skip* the intersection with $\mathcal{KI}(universal)$ and use $V(\mathcal{KI}(studios))$ to save the cost of intersecting $\mathcal{KI}(universal)$, and let $V(\cdot)$ check the existence of the keyword 'universal'. We call this technique, of finding the scope of core plan and verification, verification selection.

Exhaustive search on verification selection is too expensive with exponential cost with respect to the total number of operators in the original core plan. Suppose we have $Plan = V(Plan_{Core})$ and $Plan_{Core} = \bigcup_{1\leq i\leq N}^{\breve{W}^*} \bigcap_{1\leq j\leq M_i}^{\hat{W_i}^*} Plan_{i,j}$, where $Plan_{i,j} \in \{\mathcal{SI}(S), \mathcal{KI}(t)\}$, $\hat{W_i}^*$ and $\breve{W}^*$ are obtained in Section 5.4 and Section 5.5 respectively. A brute-force enumeration of all possible selections would have $2^{\sum_{i=1}^{N}(M_i-1)}$ cases.

For a group of consecutive intersections, we show that an optimal plan only needs to consider skipping the *least selective* list one by one in sequence. More precisely, let $\bigcap_{1\leq j\leq M}^{\hat{W}_0^*} Plan_j$ represent a core plan with one group of consecutive intersections. Without loss of generality, assume that the sub-plans are labeled according to their selectivity, *i.e.*, $L(Plan_j) \leq L(Plan_l)$ if $j < l$. Although we have the option of skipping any intersection, we find that, among the choices of skipping one intersection, skipping $Plan_M$ is always the best with both lower intersection cost and lower verification cost. We state this claim formally in Theorem 5.

If skipping $Plan_M$ is better than not skipping any intersection, we skip $Plan_M$ in the core plan and continue this process recursively. However, if skipping $Plan_M$ is worse than not skipping any intersection, we choose not to skip any intersection and the core plan remains unchanged. Therefore, when the core plan only has M consecutive intersections (without union), the computational complexity of this sequential algorithm is at most M. Theorem 5 formally states its optimality on verification selection.

THEOREM 5. *Let $Plan = V(Plan_{Core})$ denote a plan for a query where $Plan_{Core} = \bigcap_{1\leq j\leq M}^{\hat{W}^*} Plan_j$, and $Plan_j \in \{\mathcal{SI}(S), \mathcal{KI}(t)\}$. Without loss of generality, assume that sub-plans $Plan_j$ are labeled in an increasing order of their lengths, i.e., $L(Plan_i) \leq L(Plan_j)$ if $i < j$. For two plans $Plan' = V(\bigcap_{1\leq j\leq M-1}^{\hat{W}_0^*} Plan_j)$, where the intersection with $Plan_M$ is skipped, and $Plan'' = V(\bigcap_{j\in\{1,\ldots,k-1,k+1,\ldots,M\}}^{\hat{W}_1^*} Plan_j)$, where the intersection with another sub-plan $Plan_k$ is skipped, we have $C(Plan') \leq C(Plan'')$.*

PROOF. We first compute the costs of the core plans.

$$C({Plan_{Core}}') = \sum_{i=1}^{M-2} D(\prod_j^i L(l_j)/D) \left[2\log_2 \frac{L(l_{i+1})/D}{(\prod_j^i L(l_j)/D)} + 1\right] \quad (11)$$

$$C({Plan_{Core}}'') = \sum_{i=1}^{k-2} D(\prod_j^i L(l_j)/D) \left[2\log_2 \frac{L(l_{i+1})/D}{(\prod_j^i L(l_j)/D)} + 1\right] \quad (12)$$

$$+ \sum_{i=k}^{M-1} D(\prod_j^i L(l_j)/D)/(L(l_k)/D) \quad (13)$$

$$\cdot \left[2\log_2 \frac{(L(l_{i+1})/D)(L(l_k)/D)}{(\prod_j^i L(l_j)/D)} + 1\right] \quad (14)$$

$$+ D(\prod_j^{k-1} L(l_j)/D) \left[2\log_2 \frac{L(l_{k+1})/D}{(\prod_j^k L(l_j)/D)} + 1\right] \quad (15)$$

Their difference $\Delta = C({Plan_{Core}}') - C({Plan_{Core}}'')$ can be expanded as

$$\Delta/D = \sum_{i=k}^{M-1} (\prod_j^{i-1} L(l_j)/D) \left[2\log_2 \frac{L(l_i)/D}{(\prod_j^{i-1} L(l_j)/D)} + 1\right] \quad (16)$$

$$- \sum_{i=k}^{M-1} (\prod_j^i L(l_j)/D)/(L(l_k)/D) \quad (17)$$

$$\cdot \left[2\log_2 \frac{(L(l_{i+1})/D)(L(l_k)/D)}{(\prod_j^i L(l_j)/D)} + 1\right] \quad (18)$$

$$= \sum_{i=k}^{M-1} \left\{ (\prod_{j=1}^i L(l_j)/D) \left(\frac{1}{L(l_k)/D} - \frac{1}{L(l_i)/D}\right) \right. \quad (19)$$

$$+ 2(\prod_{j=1}^i L(l_j)/D) \left[\log_2 \frac{(r_{i+1}r_k)^{\frac{1}{L(l_k)/D}}}{((L(l_i)/D)(L(l_i)/D))^{\frac{1}{L(l_i)/D}}}\right] \quad (20)$$

$$\left. + \left(\tfrac{1}{L(l_k)/D} - \tfrac{1}{L(l_i)/D}\right) \log_2 \left(\prod_j^i L(l_j)/D\right) \right\} \quad (21)$$

By examining the derivatives and the boundary, we can find this function is less than zero. Therefore, $C({Plan_{Core}}') < C({Plan_{Core}}'')$. Moreover, because $L(Plan_M) \geq L(Plan_k)$ for any k, ${Plan_{Core}}'$ is shorter than ${Plan_{Core}}''$, and thus $C({Plan_{Core}}') \leq C({Plan_{Core}}'')$. □

When the core plan is a union of N groups of intersections, *i.e.*, $Plan_{Core} = \bigcup_{1\leq i\leq N}^{\breve{W}^*} Plan_i$ where $Plan_i = \bigcap_{1\leq j\leq M_i}^{\hat{W_i}^*} Plan_{i,j}$,

finding optimal verification selection is more challenging. It involves different combinations of the core plan from each intersection group. Examing all combinations leads to optimal solution but with higher computational complexity. We consider two algorithms, exploiting the tradeoff on the optimality of the algorithm and its complexity:

***ExamAll*:** For each group of intersections, $Plan_i$, we can find an optimal solution by examining up to M_i cases (Theorem 5). To union N groups of intersections, they are at most $\prod_{i=1}^{N} M_i$ number of total combinations. *ExamAll* evaluates all of them and finds the one with the lowest cost. This algorithm makes optimal verification selection with computational complexity $\prod_{i=1}^{N} M_i$.

***ExamBest*:** To further reduce the search cost, we propose to make selections independently for each group of intersections and combine the best local selections to find the solution. This approach results in complexity of $\sum_{i=1}^{N} M_i$, a major reduction from exponential- to linear-time complexity. Theorem 6 bounds the cost of the plan produced by *ExamBest* with respect to *ExamAll*, an optimal verification selection algorithm. Note that, when we optimize each group of intersections, we replace the unit cost parameter β by $\alpha\lceil\log_2 N\rceil + \beta$ to assure the approximation bound, by factoring in union cost in the separate selections. Section 8.2 empirically shows that *ExamBest*, much more computationally efficient, produces plans with total cost very close to *ExamAll* in practice.

THEOREM 6. *By performing verification selection on a query plan with* $V(\bigcup_{1\le i\le N}^{\check{W}^*} \bigcap_{1\le j\le M_i}^{\hat{W}_i{}^*} Plan_{i,j})$*, the cost of the plan produced by* ExamBest *is at most* $2 + \frac{\alpha}{\beta}\lceil\log_2 N\rceil$ *times of that produced by* ExamAll.

PROOF. Let $Plan = V(Plan_{Core})$ where $Plan_{Core} = \bigcup_{1\le i\le N}^{\check{W}^*} Plan_i$, and $Plan_i = \bigcap_{1\le j\le M_i}^{\hat{W}_i{}^*} Plan_{i,j}$. If $N = 1$, this is trivially true by Theorem 5.

Therefore, suppose $N \ge 2$. Then,

$$C(Plan) = \alpha \textstyle\sum_{i=1}^{N} C_{\cap}(Plan_i) + \alpha C_{\cup}(Plan_{Core}) \quad (22)$$
$$+ \beta C_V(Plan_{Core}) \quad (23)$$

Note that $C_{\cup}(Plan_{Core})$ denotes the sum of all union costs performed in $Plan_{Core}$.

Let us denote the cost model with the parameter β replaced by $\alpha\lceil\log_2 N\rceil + \beta$ by $C_{\alpha\lceil\log_2 N\rceil+\beta}(\cdot)$. Then, we obtain

$$C(Plan) = \alpha \textstyle\sum_{i=1}^{N} C_{\alpha\lceil\log_2 N\rceil+\beta}(Plan_i) - \Delta(Plan_{Core}) \quad (24)$$

where $\Delta(Plan_{Core}) = (\alpha\lceil\log_2 N\rceil + \beta)\sum_{i=1}^{N} L(Plan_i) - \alpha C_{\cup}(Plan_{Core}) - \beta C_V(Plan_{Core})$.

Now, suppose $V(OPT)$ where $OPT = \bigcup_{1\le i\le N}^{\check{W}^*_O} OPT_i$ is a plan with the optimal verification selection for $Plan$, and SEQ_i is a plan obtained by the sequential algorithm (Theorem 5) for $Plan_i$ with the cost model $C_{\alpha\lceil\log_2 N\rceil+\beta}(\cdot)$. Then, we have

$$C(OPT) = \alpha \textstyle\sum_{i=1}^{N} C_{\alpha\lceil\log_2 N\rceil+\beta}(OPT_i) - \Delta(OPT) \quad (25)$$
$$\ge \alpha \textstyle\sum_{i=1}^{N} C_{\alpha\lceil\log_2 N\rceil+\beta}(SEQ_i) - \Delta(OPT) \quad (26)$$

since SEQ_i is the optimal solution for $Plan_i$ with $C_{\alpha\lceil\log_2 N\rceil+\beta}(\cdot)$ by Theorem 5.

Now, we build a plan $SEQ = V(\bigcup_{1\le i\le N}^{\check{W}^*_S} SEQ_i)$, and we see that

$$C(SEQ) = \alpha \textstyle\sum_{i=1}^{N} C_{\alpha\lceil\log_2 N\rceil+\beta}(SEQ_i) - \Delta(SEQ) \quad (27)$$
$$\le C(OPT) + \Delta(OPT) - \Delta(SEQ) \quad (28)$$

by applying Inequation 26. As we have $\Delta(OPT) - \Delta(SEQ) \le (1 + \frac{\alpha}{\beta}\lceil\log_2 N\rceil)C(OPT)$, we obtain $C(SEQ) \le (2 + \frac{\alpha}{\beta}\lceil\log_2 N\rceil)C(OPT)$. □

Note that the approximation bound is close to 2 since β usually much larger than α, and N is small. In our experiments, $\frac{\alpha}{\beta} = 0.04$ and N is up to 5, which gives the approximation ratio 2.1. For a more pessimistic situation with $N = 1$ million, the approximation ratio is 2.86.

T5 reduces the search space from $2^{\sum_{i=1}^{N}(M_i-1)}$ (naive enumeration) to $\prod_{i=1}^{N} M_i$ (*ExamAll*), and further down to $\sum_{i=1}^{N} M_i$ (*ExamBest*) times, linear to the query length.

5.7 Complexity and Guarantees

The total complexity of the optimizer is $O(n \log_2 n)$: T1 and T2 are all linear time operations; T3 and T4 have cost $O(n \log_2 n)$; T5 using *ExamBest* is also a linear time algorithm. Therefore, our optimization algorithm is computationally efficient in practice.

Next we show the effectiveness of our optimizer by comparing it with a theoretical optimal optimizer requiring exponential running time. Theorem 7 provides the worst-case performance bound of our plan with respect to optimal for any queries.

THEOREM 7. *The optimized plan* $Plan$ *generated by all five transformations (using* ExamAll *verification selection algorithm of Transformation 5) satisfies*

$$C(Plan) \le \left(\frac{5}{3}\right)^F C(Plan_{OPT})$$

where $Plan_{OPT}$ *denotes an optimal plan and* F *denotes the maximum number of factorizations.*

The proof is omitted due to the page limit.

Combining Theorem 6 and 7, we get the worst-case performance bound of our optimizer (using *ExamBest* at verification selection) as follows.

COROLLARY 8. *The optimized plan* $Plan$ *generated by all five transformations (using* ExamBest *verification selection algorithm of Transformation 5) satisfies*

$$C(Plan) \le (2 + \frac{\alpha}{\beta}\lceil\log_2 N\rceil)\left(\frac{5}{3}\right)^F C(Plan_{OPT})$$

where $Plan_{OPT}$ *denotes an optimal plan and* F *denotes the maximum number of factorizations.*

In practice, the maximum number of factorizations F is often small. For example, the maximum number of factorizations for Q_1 is 1. Moreover, the verification selection transformation may skip intersections, which reduces the common factors further. For a plan with intersections only or without any common factors, we achieve optimal costs as shown in Corollary 9.

COROLLARY 9. *If a given query is not factoriazable or has no union, the optimized plan generated by all five transformations has the optimal cost using* ExamAll *verification selection algorithm, and at most* $(2 + \frac{\alpha}{\beta}\lceil\log_2 N\rceil)$ *times the optimal cost using* ExamBest *verification selection algorithm for T5.*

6. BASE INDEXES

This section introduces the indexes for processing spatial and keyword predicates. We can exploit any index that quickly returns a single list of ordered IDs for a given predicate, where ordering

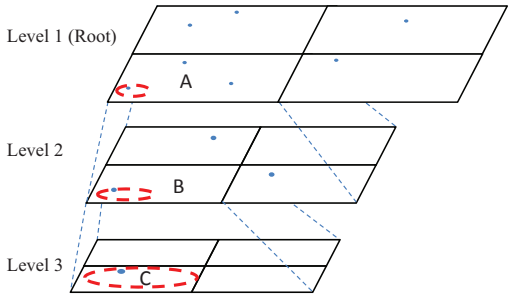


Figure 3: An illustration of a pyramid index.

is required for efficient operations. The index usually stores identifiers to objects for efficiency and for smaller memory footprint, and the object details are managed in a relation or a main memory key-value store. Among several possible indexes, we employ an inverted index [39] and a pyramid index [3] to retrieve objects satisfying keyword and spatial predicates, respectively, as they consume little memory space due to the simple structures. Moreover, both indexes can provide *interesting orders*, which we exploit in Section 7.

Inverted index: An inverted index ($\mathcal{I}$) is an efficient data structure to find objects with a specific keyword. It is a dictionary with a keyword as a key, and a set of objects with the keyword as its value. The set of objects is implemented as an ordered list of object IDs.

Pyramid index: A pyramid index ($\mathcal{P}$) with height H is a multi-resolution spatial index which consists of cells dividing the 2D space, similar to collection of grids with different resolutions (Figure 3). Each cell partitions its region into four smaller regions *if* the number of objects is no less than user-specified parameter $T_{\mathcal{P}}$ and the depth is less than H. Each cell stores the IDs of all objects within its range.

Other possible indexes: The proposed apporach is general and we can use other indexes. For example, we later discuss an alternative implementation using an *R-tree* as the spatial index and a *Trie* as a keyword index to support prefix keyword queries.

7. LEVERAGING INTERESTING ORDERS

This section extends our framework to exploit interesting orders provided by the base indexes. We have focused so far queries with boolean spatial predicate and aribtrary keyword predicates, which we call BASE queries. We illustrate the use of interesting order with the following two example query types. (1) TOP-k: returns the top k objects (among BASE results) based on a numerical value (*e.g.*, popularity) of objects. (2) kNN: returns the k nearest objects (among BASE results) from the query point in the spatial predicate. Both queries can be abstracted as adding $OrderBy(f)$ and $Limit(k)$ operations to BASE, with ranking f defined as distance or popularity respectively. Discussion of how to exploit alternative or multiple interesting orders can be found later in the section. A basic way to support these queries is through post processing: We use the optimal plan for BASE, then add $OrderBy(f)$ and $Limit(k)$, which is supported by our framework. More advanced methods are possible if we integrate the interesting order [32] into our cost model to have query optimizer produce better plans. For example, an inverted index where object ID is aligned with popularity will access objects in the order of popularity, or executing $OrderBy(f)$ early on. This enables "early termination" of selectively considering when only the top ranked matches are needed. In comparison, the post processing approach always requires examining all matching objects, potentially incurring higher cost. Next we discuss the capability on the indexes for efficiently supporting the advanced interesting order optimization and its impact to the cost model and query plan.

Index capability. Our framework can be generalized to query types of any order f, as long as they are supported by indexes with the *global monotonicity*. A list of objects $L = \{l_1, ..., l_p, ..., l_K\}$ is *globally monotonic* if for each p there exists $q(p)$ such that $f(l_j) \geq f(l_p)$ if $j > q(p)$. For example, with a special case where the list L follows a total order, *i.e.*, $l_1 \leq ... \leq l_p \leq ... \leq l_K$, the list L satisfies global monotonicity with $q(p) = p$. More generally, we can view such a list as a concatenation of blocks (of arbitrary number/length). Inside each block, the elements may not be in the order of $f(\cdot)$, and instead they are in the order of IDs. However, any element in an earlier block is larger than any elements in a later block. So at a more global view, they are ordered. When an index returns a globally monotonic list using $f(\cdot)$, and we have already obtained k objects after processing p-th object, we only need to examine up to $q(p)$-th object to assure that the k objects are indeed the top-k results we want (early terminating at $q(p)$-th object).

The gap from $q(p)$ to the length K of the list is positively correlated with the effectiveness of early termination. For indexes producing lists with large $q(p)$ values (close to K), the savings from early termination are limited. In an extreme case where $q(p) = K$, all matched objects must be examined. On the other hand, when an index provides lists with total order, *i.e.*, $q(p) = p$, the savings are the highest. Thus, the gap from $q(p)$ to K reflects how effective the early termination of an index is: the larger, the better.

How to process and when to terminate. We stream the globally monotonic list from the index, and process each block, which is ordered by IDs so that we can apply all techniques we derived. Given predicates $Pred$, the probability $P(Pred)$ of an object to satisfy $Pred$ can be computed easily assuming that the satisfied objects are uniformly distributed among all the objects returned by the index. Then, to obtain top k objects satisfying the predicates, ordered by $f(\cdot)$, we are expected to consider $q(\frac{k}{P(Pred)})$ objects. If we do not obtain the top k objects, we process more blocks.

8. EXPERIMENTAL EVALUATION

We evaluate the following claim empirically using a variety of datasets: *The optimization techniques are fast and effective because they reduce query response times under diverse circumstances.*

8.1 Experimental Setup

Datasets. We use three geo-tagged object datasets as shown in Table 4. (1) **Flickr**: All photos geo-tagged in the U.S. from Flickr taken in 2012. (2) **Twitter**: All tweets containing hashtags and geo-tagged in the U.S. for 7 weeks in 2010. (3) **Wikipedia**: All geo-tagged Wikipedia entries in English collected in 2013.

Each object in the datasets has a location and a set of keywords. For keywords, we use the user provided photo tags in Flickr, hashtags in Twitter, and anchor text in Wikipedia.

Table 4: Datasets.

Dataset	Flickr	Twitter	Wikipedia
Object	11,021,551	15,964,134	31,610
Unique keyword	1,540,069	2,490,402	109,288
Keyword per object	6.256	1.383	42.118
Start date	2012/01/01	2011/01/15	2013/10/01
End date	2012/12/31	2011/01/27	

Query workload. We generate a query workload with 10,000 queries by creating the spatial and the keyword predicates as follows. For the spatial predicate, we determine the query point using a distribution of 1,688,717 user locations from Bing mobile query log reflecting actual user locations. The radius is chosen randomly from 0.1 to 25.6 miles as listed in Table 5 unless specified otherwise. We execute each query 10 times to report its measurements.

We construct the keyword predicate such that the query result is not empty. As we support complex keyword predicates, we do this in two steps. First, the keyword predicate has one or more sets that are ORed, and each set has one or more keywords that are ANDed. Parameter *NumSet* determines the number of sets, and *SetSize* determines the number of keywords inside each set. Second, we find the nearest *NumSet* objects to the query point, and we randomly select *SetSize* keywords from each object. For example, setting (*NumSet*=3) and (*SetSize*=2), we generate the keyword predicate "('Beach' AND 'Park') OR ('Sunrise' AND 'Cafe') OR ('Sushi' AND 'Restaurant')" by finding the three nearest objects to the query point, and obtaining keyword sets from objects. We further simplify the keyword predicate by extracting common keywords, for example we use "'State' AND ('Beach' OR 'Park')" instead of "('State' AND 'Beach') OR ('State' AND 'Park')". We vary the parameters as listed in Table 5.

Table 5: Query parameters.

Parameter	Values
Radius (mile)	0.2, 0.4, 0.8, 1.6, 3.2
NumSet	1, 2, 3, 4, 5
SetSize	1, 2, 3, 4, 5
k	1, 2, 4, 8, 16, 32, 64, 128, 256

Systems. We implement the base spatial index $\mathcal{SI}$ by the pyramid index $\mathcal{P}$, and the base keyword index $\mathcal{KI}$ by the inverted index $\mathcal{I}$. We compare the following systems which exploit both $\mathcal{I}$ and $\mathcal{P}$ indexes: (1) $\mathcal{I}+\mathcal{P}$ uses the base plan without optimizations. (2) $\mathcal{I}+\mathcal{P}$ *Opt* is our approach applying the optimization techniques with *ExamBest* for T5. (3) $\mathcal{I}+\mathcal{P}$ *Opt+All* is our approach with *ExamAll* for T5. We also use the two base indexes without our optimization techniques for reference: (1) *Baseline-*$\mathcal{I}$ is the inverted index. (2) *Baseline-*$\mathcal{P}$ is the pyramid index. Moreover, we compare to other indexes and relational engines in sections 8.6 and 8.7.

Implementation Software and Hardware. We implement all indexes and optimization techniques in C# (25,000 lines of code). We determine the cell capacity of the pyramid index empirically: cell capacity $T_{\mathcal{P}} = 128$, and height $H = 20$. We run all experiments on a server with an Intel Core i7-4820K processor with 32 GB memory running Microsoft Windows. The indexes and working set fit in main memory. The measured β is 23.2 times of α.

Performance Metric. We use the query response time as the main metric, and it includes applying optimization techniques and executing the plan. We report the average response time and the 99^{th}-percentile (denoted by $99^{th}\%$) response time which is often used in server provisioning to control the tail latency.

8.2 Impact of Five Optimization Techniques

We show the effect of successively applying the proposed five optimization techniques in Figure 4 on Flickr dataset. We apply the optimization steps cumulatively as they cannot be applied individually. Each step is annotated with its relative response time reduction (*e.g.*, adding T5 reduces the response time of using T1-4 by 7.49% in $99^{th}\%$). The results show that both average and $99^{th}\%$ response times decrease monotonically as more techniques are applied. Each of the earlier transformations from T1 to T3 significantly reduces

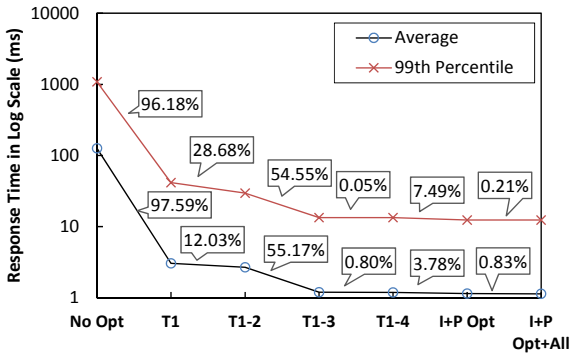


Figure 4: Impact of the optimization techniques.

the response time. Taming the response time even further becomes more challenging for the later transformations. There is small response time reduction by adding T4 as the queries use few union operators. The transformation T5, however, is rather effective, reducing about 7.5% response time further on the $99^{th}\%$ response time. We can also see the effect of two different techniques for T5: *ExamAll* and *ExamBest*, and $\mathcal{I}+\mathcal{P}$ *Opt* using *ExamBest* performs closely to $\mathcal{I}+\mathcal{P}$ *Opt+All* using *ExamAll* (less than 1% performance difference). Furthermore, these optimizations have low overhead, the optimization time is 0.036 ms on average and 0.124 ms at the $99^{th}\%$ when applying all techniques. Comparing **no optimization** to *ExamBest*, we conclude that the optimizations effectively reduce the average response time by more than 74 times (from 134.7 ms to 1.8 ms), and reduce the tail response time by 46 times (from 1081 ms to 23.7 ms).

8.3 Leveraging Interesting Orders

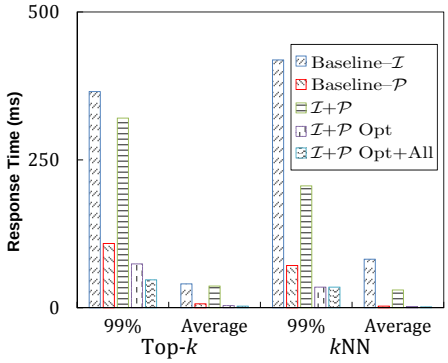


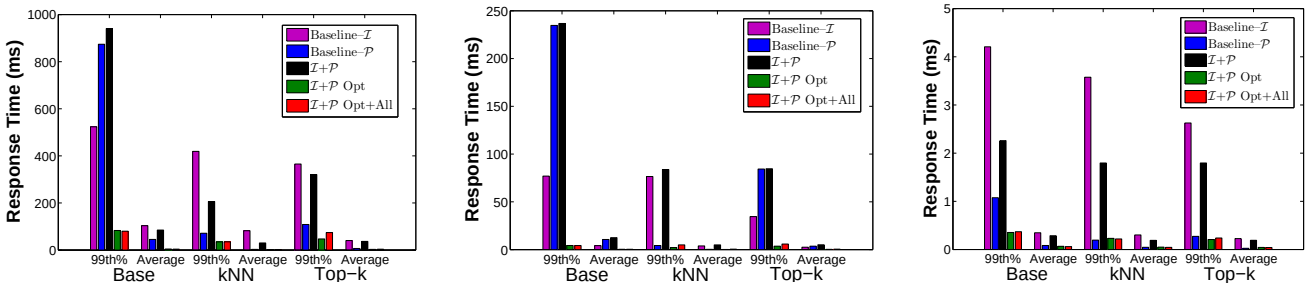
Figure 5: Response time of kNN and TOP-k queries (Flickr dataset).

This experiment shows that we can efficiently process kNN and TOP-k queries using interesting order. Figure 5 depicts the average and $99^{th}\%$ response time for Flickr dataset: We achieve the lowest response times for both kNN, and TOP-k queries.

8.4 Sensitivity Study and Discussion

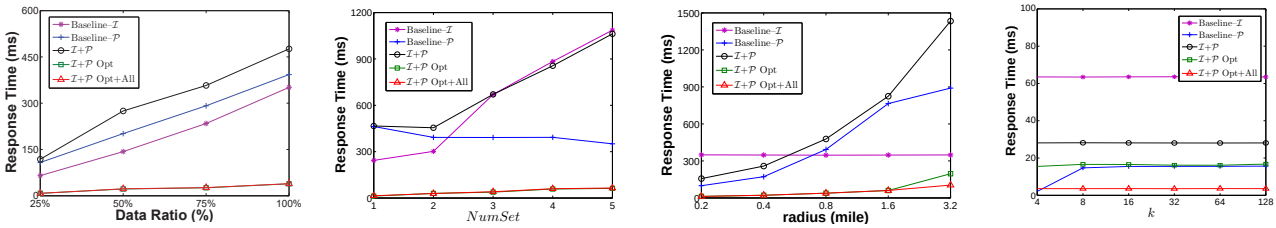
In this section, we vary each parameter to study its impact while fixing the remaining parameters. We set k to 16, radius to 0.8, *NumSet* to 3, and *SetSize* to 3 unless they are varied.

Datasets. We evaluate three query types (BASE, kNN, TOP-k) using three different real-world datasets. Figure 6 summarizes the result: The optimized plans show consistently lower response time.



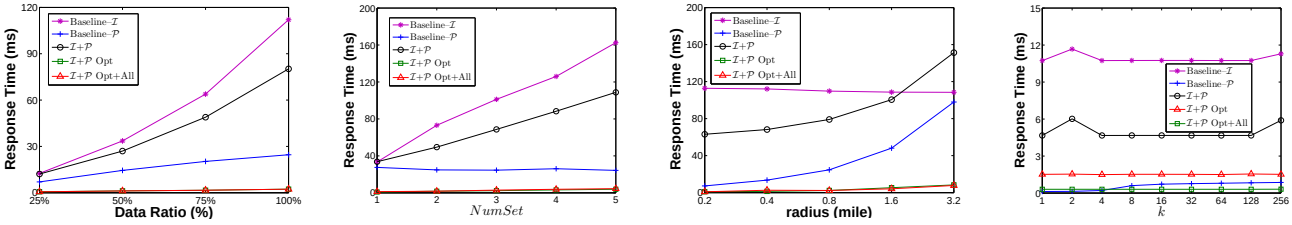
(a) Flickr (b) Twitter (c) Wikipedia

Figure 6: Response times for the three query types across all datasets.



(a) Data size. (b) *NumSet*. (c) Radius. (d) k (kNN).

Figure 7: The 99^{th} percentile of response time with varying parameters for BASE queries.



(a) Data size. (b) *NumSet*. (c) Radius. (d) k (kNN).

Figure 8: The average response time with varying parameters for BASE queries.

Scalability with dataset size. Figure 7(a) and Figure 8(a) shows the effect of changing dataset size. We randomly select 25%, 50%, and 75% of data points. We see that our optimizations are effective in reducing the response time across the different sizes.

Complex keyword predicate. Figure 7(b) and Figure 8(b) shows the effect of *NumSet*. We observe that, as *NumSet* increases, the selectivity of keyword predicate decreases and thus degrades the performance of *Baseline-*$\mathcal{I}$ requiring more intersections and more verifications. In contrast, our approach maintains low latency by keeping only useful intersections, and leveraging $\mathcal{SI}$ ($\mathcal{P}$) as needed.

Radius. Figure 7(c) and Figure 8(c) shows the effect of changing the radius r, which reflecting the selectivity the spatial predicate. A larger radius increases the cost of *Baseline-*$\mathcal{P}$. However, our approach maintains low latency, by leveraging both $\mathcal{KI}$ and $\mathcal{SI}$ ($\mathcal{I}$ and $\mathcal{P}$, respectively) to benefit from the selectivity of the keyword predicates. As a result, our work $\mathcal{I}+\mathcal{P}$ *Opt+All* outperforms *Baseline-*$\mathcal{P}$, *Baseline-*$\mathcal{I}$ and $\mathcal{I}+\mathcal{P}$ across all ranges of r.

Retrieval size k. Figure 7(d) and Figure 8(d) shows the effect of the retrieval size k for a kNN query. For kNN queries, a larger k makes the spatial predicate less selective, which degrades the performance of *Baseline-*$\mathcal{P}$ using early termination, when $k \geq 8$ compared to $k = 4$. The response time of *Baseline-*$\mathcal{P}$ soon stops increasing because enough number of objects can be retrieved from the similar number of cells. Our approach, in contrast, is less sensitive to k since we leverage selective keyword predicate. Moreover, as we discard less selective keyword predicate (*e.g.*, 'America') and leverage the more selective spatial predicate (*e.g.*, scarcely populated region), the optimized approach performs better than *Baseline-*$\mathcal{I}$.

Cost model validation. We validate the cost model by comparing the cost model estimates and the actual execution time measured for 10,000 queries. The Pearson correlation coefficient between the estimates and actual execution times is 0.8355, which shows a strong correlation.

8.5 Alternative Base Indexes

We show the effect of using indexes other than $\mathcal{P}$ and $\mathcal{I}$. As an example, we use *Trie* and *R-tree* to replace the inverted index and the pyramid index. Table 6 shows the performance of using individual indexes (*Trie* and *R-tree*), their base plan mapping (*R+T*), and their optimized plan (*R+T Opt*). We observe that the optimized plan *R+T Opt* with *R-tree* and *Trie* works better than individual indexes and their base plan *R+T*. While the same result can be observed with $\mathcal{I}$ and $\mathcal{P}$, the performance of $\mathcal{I}+\mathcal{P}$ *Opt* is slightly better than *R+T Opt*.

We may employ an index supporting a prefix search, like *Trie*. For example using the Flickr dataset, leveraging *R+T Opt* for prefix search takes 5.87 ms in average and 67 ms in $99^{th}\%$ time, while using *Trie* alone takes much longer $99^{th}\%$ time of 1,092 ms, and *R+T* without optimization takes even longer $99^{th}\%$ time of 3,563 ms. We conclude that the optimizations are effective here.

Table 6: Response time (ms) with different base indexes.

	Trie	*R-tree*	*R+T*	*R+T Opt*	$\mathcal{I}$	$\mathcal{P}$	$\mathcal{I}+\mathcal{P}$	$\mathcal{I}+\mathcal{P}$ *Opt*
Average	3.2	2408	2431	2.0	4.6	122.4	134.7	1.8
$99^{th}\%$	40.0	3531	3563	25.3	39.2	1054	1081	23.7

8.6 Comparison to State-of-the-Art Indexes

We compare the proposed approach to the state-of-the-art spatial-keyword indexes. We use the best performing methods for BASE queries as reported in a recent paper [12]: *SFC-QUAD* [14], *SKIF* [23], *IR-tree* [15] and *S2I* [30]. *SFC-QUAD*, *SKIF* and *IR-tree* are tightly integrated spatial-keyword indexes. *S2I* is a loosely integrated keyword-first index that uses an inverted index as the main structure, and each posting list can be augmented by an *R-tree*. *SKIF* is an inverted index whose key is either a keyword or a grid cell. *IR-tree* uses *R-tree* as the main structure whose nodes are augmented by a set of contained keywords, and we use this set to prune nodes that do not satisfy the keyword predicate. Variants of the *IR-tree*, including *DIR-tree*, *CIR-tree*, and *CDIR-tree*, are designed to improve *IR-tree* for relevance ranking of keywords. In our query model without such ranking, *IR-tree* represents the collective behavior of these indexes. We implement these methods and extend them to process queries with both AND and OR. For *SFC-QUAD*, we use a recent implementation [12] and port it to C#.

Table 1 compares the response times of our approach and the existing indexes, which perform well in several settings. Our approach, however, has lower response time than the state-of-the-art methods, both for the average and the $99^{th}\%$ response time. *IR-tree* and *S2I* use more memory space to boost the performance, which is a good tradeoff for disk-based systems. However, some queries under *S2I* take very long time up to 224,683 ms, and these extremes make the average higher than $99^{th}\%$ tail latency. As also noted in the *S2I* paper [30], such degraded performance can be observed when keywords are skewed: 'Manhattan' or 'NYC' occurs frequently in objects in New York. *SFC-QUAD* and *SKIF* require relatively less memory, but they are designed to reduce the I/O cost rather than main memory processing.

8.7 Comparison to Relational Engines

Limitations of relational engines. Table 2 compares the performance of our approach to two relational engines using the Flickr query workload (Section 8.1). We find that in the two systems, the query optimizer does not fully support intersection and union reordering (T2-T4) for spatial and keyword predicates. Furthermore, they do not employ the combination of T1 and T5 to rewrite the queries. In particular, MonetDB does not reorder the keyword and spatial predicates. PostgreSQL does not reorder keyword predicates because it uses a bitmap index to process all keyword predicates at once, which is efficient for disk-based access. This, however, limits the search space to three options only: 1) using the spatial predicate only, 2) *all* the keyword predicates only, or 3) using all the predicates. In contrast, our optimizations uses a richer plan space with plans that are not currently supported in two systems.

Applying our techniques. To show that our techniques can improve relational engines in processing spatial keyword queries, we rewrite the SQL queries for each engine and report the results in Table 2. Notice that we do not change these engines and they use different operators and different underlying index structures. For example, MonetDB uses merge and hash join, rather than gallop search. We also use hints [9] or optimization parameters to force the generation of the target plan in both systems. For MonetDB, we rewrite each SQL query to reflect the optimized intersection and union orders in addition to skipping unselective predicates. This results in 2.65 times reduction (1405 ms vs. 529 ms) for the average response time, and 1.46 times reduction (3462 ms vs. 2372 ms) for $99^{th}\%$ response time. PostgreSQL leverages k-way algorithms and hence does not reorder intersections and unions, and applies all keyword predicates using k-way algorithms. We modify our queries to reflect the combination of T1 and T5 to skip unselective predicates, resulting in 41.9% average, and 62.9% $99^{th}\%$ response time reduction compared the original queries PostgreSQL on the Flickr query workload (Section 8.1).

9. RELATED WORK

Processing spatial-keyword queries is an active area of research, and a diverse set of solutions building upon disk-based spatial and keyword indexes has been proposed. Spatial-first approaches build on *R-tree* [10, 16, 20, 37, 38] or grid [33] to process the spatial predicate, and then selectively access candidates satisfying the keyword predicate. Alternatively, spatial indexes can be augmented by keyword information to prune both by spatial and keyword predicates. For example, *IR-tree* [15] and KR*-tree [20] maintain keyword counts for each node in an *R-tree*. Grid cells can also embed similar information [23, 34]. Similarly, keyword-first approaches use inverted file [38] or bitmap [16, 37] first to loosely integrate with spatial predicate. A tighter integration has been proposed to augment inverted files with a space filling curve [13, 14]. We presented their performance comparison results in Table 1.

While this cost-based optimization is common in traditional database systems, our problem is unique in several aspects. First, since we exploit memory-based indexes, CPU cost (object comparison cost) is not negligible compared to memory access cost. Thus, we model both comparison cost and access cost as a part of cost model, unlike traditional database systems focusing only on the latter. Second, verify should be introduced as an operator, as a spatial index often retrieves the superset satisfying a spatial predicate. Therefore, a non-sargable predicate must be used to test objects in the set. This offers new optimization opportunities as what we proposed in techniques T1 and T5. In particular, the verify operator that is popped up by T1 may consider both sargable and non-sargable predicates. We exploit a cost-based optimization to select these predicates in T5. In addition, we optimize a complex boolean expression. MonetDB and PostgreSQL do not optimize the order of intersections and unions. Also, in contrast to a recent work [36] that does not conclude whether union or intersection should be processed first, we theoretically conclude that intersection first is close to optimal in our specific problem.

List intersection, an important building block of our optimization framework, has been actively studied for supporting Web search or relational join queries. State-of-the-arts can be categorized by base operation and arity. First, a merge-based algorithm reads input lists in sequential orders, which is effective for input lists with similar length [22]. In contrast, when one input list is significantly shorter, a search-based algorithm of pivoting an element in one list to search for a match in the remaining lists is more effective. Second, regarding arity, binary algorithms intersect two lists at a time, while k-way algorithms [19] intersect all lists at once [18, 4, 5].

For our target problem, we implemented a search-based binary intersection we found to be effective, as an intermediate result list is significantly shorter than the remaining input lists in general.

Regarding query model, unlike ours supporting full boolean retrieval model, existing work is often optimized for a specific subset (*e.g.*, conjunction-only [29]) arguing full model is too complicated to be formulated by end users. However, in online advertising, advertisers target user and web page attributes, which automatically translates to a complicated boolean expression according to [36]. However, verify operator, though [36] recognizes its importance as "residual", was not fully optimized, while our work proposes T1 and T5 transformations optimizing for verify and achieves 96% reduction in $99^{th}\%$ response time.

As for base index, we demonstrate the feasibility of replacing base index for varying requirements, *e.g.*, using *Trie* when prefix search is important. For spatial predicates, though we implemented a pyramid index and an *R-tree* with the advantage of materializing space of varying granularity, we note simpler structures, *e.g.*, grids found effective for highly dynamic problems [27, 28], can be similarly considered. In this paper, we assume one base index for a spatial and keyword predicate each, but it is straightforward to leverage a tightly integrated index as a base index for both predicates. Such index, pruning on both spatial and keyword predicates, can be considered as a join index studied in relational query context [35]. We leave a sophisticated optimization in the presence of multiple base indices for space, keyword, and its combination, as future work.

10. CONCLUSIONS

This paper develops a cost-based optimizer for processing spatial-keyword queries with arbitrary boolean predicates. We introduce three operators for main memory processing and use them to construct query plans. We develop a cost model and a sequence of efficient transformations to optimize a plan. We extend the framework further to support interesting orders. We validate this framework experimentally using three realistic datasets, showing that the optimization techniques reduce the average and tail latency significantly.

11. ACKNOWLEDGMENT

We thank the authors of [12] for providing the source code of spatial-keyword indexes. This work was supported by ICT R&D program of MSIP/IITP [B0101-15-0307, Basic Software Research in Human-level Lifelong Machine Learning (Machine Learning Center)], and Microsoft Research.

12. REFERENCES

[1] PostGIS, PostGIS Spatial and Geographic Objects for PostgreSQL. http://postgis.net/.

[2] PostgreSQL, SQL Compliant, Open Source Object-Relational Database Management System. http://www.postgresql.org/.

[3] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pages 265–272, 1990.

[4] R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*, pages 13–24, 2005.

[5] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM JEA*, 14:7:3.7–7:3.24, 2010.

[6] J. L. Bentley and A. C. chih Yao. An almost optimal algorithm for unbounded searching. *IPL*, 5(3):82–87, 1975.

[7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[8] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.

[9] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power hints for query optimization. In *ICDE*, pages 469–480, 2009.

[10] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *SSDBM*, pages 87–95, 2010.

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

[12] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *VLDB*, 6(3):217–228, 2013.

[13] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[14] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: Efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[15] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *VLDB*, 2(1):337–348, Aug. 2009.

[16] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[17] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[18] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, 2000.

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

[20] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (gir) systems. In *SSDBM*, page 16, 2007.

[21] D. Huffman. A method for the construction of minimum-redundancy codes. *IRE*, 40(9):1098–1101, 1952.

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

[23] A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA*, pages 450–466, 2010.

[24] S. Kim, Y. He, S. Hwang, S. Elnikety, and S. Choi. Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search. In *WSDM*, pages 7–16, 2015.

[25] T. Koshy. *Catalan Numbers with Applications*. Oxford University Press, 2008.

[26] R. Krauthgamer, A. Mehta, V. Raman, and A. Rudra. Greedy list intersection. In *ICDE*, 2008.

[27] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[28] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[29] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD*, pages 773–784, 2007.

[30] J. a. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[31] E. Schurman and J. Brutlag. Performance related changes and their user impact. *Velocity*, 2009.

[32] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[33] A. K. SKIDMORE. A comparison of techniques for calculating gradient and aspect from a gridded digital elevation model. *IJGIS*, 3(4):323–334, 1989.

[34] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pages 218–235, 2005.

[35] P. Valduriez. Join indices. *ACM TODS*, 12(2):218–246, 1987.

[36] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. Indexing boolean expressions. *Proc. VLDB Endow.*, 2(1):37–48, Aug. 2009.

[37] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE TKDE*, 24(10):1889–1903, 2012.

[38] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.

[39] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM CSUR*, 38(2), 2006.