

Boolean + Ranking: Querying a Database by K-Constrained Optimization*

Zhen Zhang¹, Seung-won Hwang², Kevin Chen-Chuan Chang¹
Min Wang³, Christian A. Lang³, Yuan-chi Chang³

¹University of Illinois at Urbana-Champaign

²Pohang University of Science and Technology

³IBM T.J. Watson Research Center

ABSTRACT

The wide spread of databases for managing structured data, compounded with the expanded reach of the Internet, has brought forward interesting *data retrieval* and *analysis* scenarios to RDBMS. In such settings, queries often take the form of *k-constrained optimization*, with a Boolean constraint and a numeric optimization expression as the goal function, retrieving only the top-*k* tuples. This paper proposes the concept of supporting such queries, as their nature implies, by a functional optimization machinery over the search space of multiple indices. To realize this concept, we combine the dual perspectives of discrete state search (from the view of indices) and continuous function optimization (from the view of goal functions). We present, as the marriage of the two perspectives, the OPT* framework, which encodes *k*-constrained optimization as an A* search over the composite space of multiple indices, driven by functional optimization for providing tight heuristics. By processing queries as optimization, OPT* significantly outperforms baseline approaches, with up to 3 orders of magnitude margins.

1. INTRODUCTION

The wide spread of databases for managing structured data, compounded with the expanded reach of the Internet, has brought forward interesting *data retrieval* and *data analysis* scenarios. While databases have been applied predominately in business settings with well-defined query logic, they are now frequently used in retrieving or analyzing data: In these scenarios, the target answers are described with some *qualifying* constraint \mathcal{B} , which specifies what tuples should be considered valid, and a *quantifying* function \mathcal{O} , which measures their degree of matching, and the query returns only some *k* top-matched answers—thus overall with a query form $Q = (\mathcal{B}, \mathcal{O}, k)$. As a simple example, a query $(\mathcal{B}: dept = CS \wedge (year = 2 \vee year = 3), \mathcal{O}: gpa, k: 5)$ will return the top-5 2nd or 3rd-year students in CS with highest *gpa*.

We refer to such a query as a *k-constrained optimization* query,

*This material is based upon work partially supported by NSF Grants IIS-0133199, IIS-0313260, and the 2004 and 2005 IBM Faculty Awards. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...\$5.00.

which effectively specifies a *goal function* \mathcal{G} and retrieval size *k*. The goal function \mathcal{G} consists of both Boolean *constraint* expression \mathcal{B} and a numeric *optimization* expression \mathcal{O} , i.e., $\mathcal{G} = \mathcal{B} \cdot \mathcal{O}$ (by treating \mathcal{B} as a function of $\{0,1\}$ values). As its semantics, intuitively, such a *k*-constrained optimization query, over a database \mathcal{D} , is to optimize the goal function \mathcal{G} over the domain defined by \mathcal{D} , i.e., find the *k* top tuples $t \in \mathcal{D}$, such that $\mathcal{G}(t)$ is maximized.

To begin with, such *k*-constrained optimization queries, by flexibly specifying the *retrieval* criteria over relational structured *data*, are well suited for *data retrieval* (by which we intend to parallel *information retrieval* over unstructured text). In such scenarios, user requests often involve some “hard” constraints \mathcal{B} and “soft” criteria \mathcal{O} , resulting in the overall goal \mathcal{G} . Meanwhile, as a retrieval task over large data, users are often interested in a relatively small number *k* of best matches.

Example 1 (Data Retrieval): To search for houses with a reasonable tradeoff of *size* and *price*, from a House relation *h*, we may formulate query $Q = (\mathcal{B}: h.price \leq 200k \vee h.price \geq 400k, \mathcal{O}: \frac{h.size}{|h.price - 300k|}, k: 1)$, or in the SQL form as below. The query will select the top-1 house from *h*, by qualifying only those with *price* in the given range, and quantify their scores with some ratio of *size* over *price* (i.e., some form of per-dollar size).

```
select h.address from House h
where h.price ≤ 200k ∨ h.price ≥ 400k
order by  $\frac{h.size}{|h.price - 300k|}$  limit 1
```

Similarly, such queries can also retrieve from multiple relations. To illustrate, we may now decide to also consider the *safety* of the district, and thus join another relation CrimeRate (with alias *c*). The new join query, say for top-10, is thus, in our simplified form: $Q_c = (\mathcal{B}: (h.price \leq 200k \vee h.price \geq 400k) \wedge h.zipcode = c.zipcode, \mathcal{O}: \frac{h.size}{|h.price - 300k|} \times c.crimerate^{-1}, k: 10)$ ■

Further, such queries, with \mathcal{B} as a range specifier for *categorizing* objects and \mathcal{O} for *aggregating* them, are also useful for *data analysis*. By joining multiple relations (say R_1 and R_2), the \mathcal{B} expression will select qualified tuples (say $r_1 \in R_1$ and $r_2 \in R_2$), and the \mathcal{O} expression will evaluate their aggregate scores (i.e., $\mathcal{O}(r_1, r_2)$). Such scenarios often arise in decision support tasks:

Example 2 (Data Analysis): Consider a data warehouse scenario for a retailer store, with a table Sale(*itemid*, *year*, *sale*) for collecting the sales history data over the past 10 years, with one entry per item per year, e.g., (a012, 1998, 600k). A data analyst may execute the following query, looking for the top-10 items that have the largest increase of sales in any consecutive years. The query joins the same table Sale to itself, which we refer to as aliases s_1 and s_2 , resulting in $Q_d = (\mathcal{B}: s_1.itemid = s_2.itemid \wedge s_2.year - s_1.year = 1, \mathcal{O}: s_2.sale - s_1.sale, k: 10)$ ■

However, while useful, such queries are not *fundamentally* supported in RDBMS, and thus their evaluation is rather naive and inefficient. We note that k -constrained optimizations are *already* syntactically expressible in most SQL variants (e.g., **where \mathcal{B} order by \mathcal{O} limit k** of Example 1 follows the PostgreSQL syntax). While expressible, however, these optimization queries are evaluated in naive ways; the limitations come in two ways: First, *condition separation*: As current systems are not aware of optimization across both \mathcal{B} and \mathcal{O} , they often process the two expressions step-by-step, lacking an overall integrated search in the \mathcal{G} -function space. Second, *function restriction*: While some existing algorithms can be adapted to deal with “optimization”– or finding top- k – they essentially rely on a rigid assumption, that \mathcal{G} functions are *monotonic*. The monotonicity requires \mathcal{G} to be non-decreasing if all its parameters are non-decreasing. With the general combined Boolean and ranking conditions of \mathcal{B} : \mathcal{O} , \mathcal{G} is rarely monotonic– In fact, *none* of the above examples is monotonic.

As the main thesis of this paper, we propose to process such a query *by* what the query actually means: That is, we believe such queries, as their nature implies, should be evaluated simply *by* k -constrained optimization of \mathcal{G} over the database \mathcal{D} as the domain. Much like *function optimization* in numeric analysis, we are to maximize a function \mathcal{G} . Conceptually, with the focused search that optimization schemes (e.g., hill climbing) typically achieve, this concept of “processing queries by optimization” is appealing. However, unlike functional optimization, we shall search in a database, instead of a continuous numeric domain, for the maximizing tuples.

Toward realizing this concept, we take dual perspectives: Essentially, for efficient evaluation, our objective is to optimize \mathcal{G} , with the help of indices as access methods, over tuples in \mathcal{D} . First, from the view of using *indices*, we are to search the maximizing tuples on the index nodes as “discrete states”– and thus the perspective of *discrete state search*. Second, from the view of maximizing *goal functions*, we are to optimize \mathcal{G} – and thus the perspective of *continuous function optimization*. We stress the two complementary perspectives– While function optimization helps us to focus on the goal, state search helps to navigate the index. A satisfactory solution, hence, hinges on the “marriage” of the two.

To realize k -constrained optimization over databases, this paper develops the OPT* framework, which integrates the two concepts. The gist of OPT* lies in correctly transforming the optimization problem into search on the indices, thus achieving the marriage. On the one hand, OPT* builds upon the state search perspective: To enable such search, it constructs a state space, or a “map,” of the index nodes and their interlinks, upon which the optimization problem (of maximizing the score of tuples) is equated to an A^* search (of minimizing the path to reach the tuples). On the other hand, OPT* leverages the function optimization perspective: To ensure the correctness and optimality of the embedded A^* mechanism, OPT* resorts to functional optimization to measure the “landscape,” so as to configure the state space with a right heuristics function and sound initial states. Together, with A^* search driven by functional optimization, OPT* completes the encoding of the k -constrained optimization problem, and thus the search algorithm naturally follows.

In summary, OPT* framework achieves the challenge of optimizing a goal function over index structures that access a database– for any goal functions (not necessarily monotonic), any access *paths* (not necessary hierarchical parent-child links), and over a compound space of *indices*. While we develop OPT* for the new problem of evaluating general k -constrained optimization, we stress that, in this general form, it also conceptually unifies several previously proposed frameworks: e.g., *KNN* and spatial joins in spatial

queries and TA in monotonic top- k queries, which Section 7 will discuss in details.

We have implemented the OPT* framework, and evaluated it over both real datasets with benchmark queries, as well as over synthetic datasets with controlled queries. We compared OPT* with several baseline approaches, albeit with their limitations, that can evaluate some forms of k -constrained optimization. The performance margin with such optimization-driven search is often significant – in the range of up to 3 orders of magnitude. In summary, the contributions of this paper are:

- We propose to evaluate queries by the *concept* of k -constrained optimization over databases.
- We realize the concept with the *framework* of OPT*, which builds functional optimization upon discrete state search.
- We extensively *evaluate* OPT* for its performance.

We formalize and motivate the problem in Section 2. To develop the OPT* encoding, Section 3 begins with the state search perspective, and Section 4 completes with the functional optimization perspective. We then discuss the OPT* framework in Section 5, report experiment in 6, and summarize the related work in 7.

2. MOTIVATION

2.1 Query Model

As Section 1 discussed, querying a database can be successfully modeled as a k -constrained optimization problem, with the dual goals of optimizing both constraint expression \mathcal{B} and optimization expression \mathcal{O} to retrieve the top results of size k .

Toward seamless optimization of both \mathcal{B} and \mathcal{O} , we view that the two expressions together form a unified goal function \mathcal{G} : That is, the score of \mathcal{G} for a tuple t that satisfies the constraint expression \mathcal{B} is simply its score of optimization expression $\mathcal{O}(t)$. In contrast, a tuple that fails to satisfy \mathcal{B} is assigned with a low score such that it can never make to the top- k results.

We develop this intuition into a formal definition of k -constrained optimization query. Let $A_i, i = 1, \dots, m$ denote m query attributes (either all from a single relation, e.g., as in \mathcal{Q} , or from multiple joined relations, e.g., as in \mathcal{Q}_c), \mathcal{D} a database instance, and $dom(A_i)$ the domain values of attribute A_i . A k -constrained optimization query can be formally defined as follows:

Definition 1 (Query model): Let a k -constrained optimization query \mathcal{Q} be defined over query attributes $A_i, i = 1, \dots, m$ and join n relations D_1, \dots, D_n . Let $rel(A_i)$ denote the relation D_j that A_i belongs to, $dom(A_i)$ the domain values of A_i and $\mathcal{D} = D_1 \times \dots \times D_n$. A k -constrained optimization query \mathcal{Q} is a two tuple $\langle \mathcal{G}, k \rangle$, where:

- *Goal function \mathcal{G}* : $dom(A_1) \times \dots \times dom(A_m) \rightarrow \mathcal{R}^+$ maps a tuple t of m attribute values to a positive numeric score. \mathcal{G} is composed from *optimization expression \mathcal{O}* : $dom(A_1) \times \dots \times dom(A_m) \rightarrow \mathcal{R}^+$ and *constraint expression \mathcal{B}* : $dom(A_1) \times \dots \times dom(A_m) \rightarrow \{0, 1\}$ as follows:

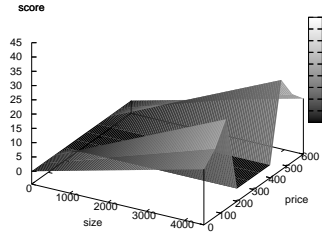
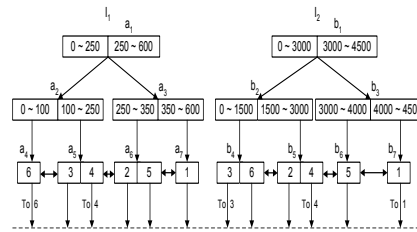
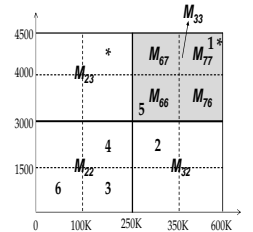
$$\mathcal{G}(t) = \mathcal{O}(t) \cdot \mathcal{B}(t)^1 \quad (1)$$

- *result size $k \in \mathcal{N}$* : specifies the number of tuples in result.

The result of a k -constrained optimization query \mathcal{Q} is thus a sorted list of k tuples in the database \mathcal{D} that maximizes \mathcal{G} . ■

¹This definition assumes \mathcal{O} maps to a positive real number in \mathcal{R}^+ and \mathcal{B} maps to a binary value of 0 or 1. More rigorously, we can let \mathcal{O} map to any real number and set $\mathcal{G}(t)$ as $-\infty$ when $\mathcal{B}(t) = 0$.

	A1:Price	A2:Size	Score
1.	600K	4500	15
2.	350K	2000	0
3.	150K	1000	6.67
4.	250K	2000	0
5.	300K	3500	0
6.	80K	500	2.27

(a) Database \mathcal{D} with relation D_1 .(b) Landscape \mathcal{G} .(c) Indices \mathcal{I} .

(d) Value space.

Figure 1: k -constrained optimization.

To illustrate our query model, consider query \mathcal{Q} in Example 1 with two query attributes *price* and *size*. Figure 1(a) describes an example database instance \mathcal{D} , while Figure 1(b) plots the landscape of \mathcal{G} over the domains of *price* and *size*. $\mathcal{Q} = \langle \mathcal{G}, 1 \rangle$ retrieves the top tuple in \mathcal{D} maximizing \mathcal{G} , i.e., (600k, 4500), which corresponds to the high-scoring point in the landscape. With this abstraction, Section 2.2 develops a query mechanism to efficiently search \mathcal{D} for top- k answers.

2.2 Query Mechanism

As Section 2.1 has defined a semantic model of querying databases, we now develop how to answer such queries with the optimal cost. More specifically, our goal is to *search* for database tuples in \mathcal{D} that maximize the goal function \mathcal{G} with the *minimal* cost. Toward the goal, the first requirement of searching over database tuples in \mathcal{D} clearly suggests the use of access methods, e.g., table scan or index scan, over \mathcal{D} . In particular, as table scan always requires an exhaustive search and thus not optimizable, we rather use indices that enable a “focused” search by organizing data tuples into discrete states preserving “attribute value locality” (as we will discuss later). The second requirement of minimizing the cost suggests to effectively guide the search toward maximizing \mathcal{G} .

To satisfy these dual requirements, we view the problem from the following two perspectives:

- **Discrete state search perspective:** From the view of using indices, our problem is essentially to search over a discrete set of index nodes to find the satisfying data tuples.
- **Continuous function optimization perspective:** From the view of optimizing \mathcal{G} , our problem is essentially to optimize the goal function \mathcal{G} over the domain of a database.

First, from *discrete state search*, our goal is to search over indices for top- k data tuples. An index is essentially a set of nodes (p, ptr) with pointer ptr to reach data tuples (either directly or through multiple “reachable” index nodes) preserving the locality to satisfy predicate p , e.g., $100K < price < 200K$. Such structures conceptually discretize domains into nodes preserving locality of values. Each of such nodes clusters data tuples with close values, with efficient traversals provided among them by node pointers. In particular, we focus on B+-trees, as commonly available in DBMS. Such an index essentially presents a graph with internal, leaf, and tuple nodes, and realizes locality in the following two types of internode linkages: First, *hierarchical traversals* among internal nodes realize locality of *containment*, ensuring tuples within the child node also fall within the parent node. That is, following such pointers can be conceptually viewed as “zooming into” a subrange. Second, *interleaf traversals* among leaf nodes realize the locality of *contiguity*, ensuring two sibling nodes refer to contiguous ranges. Ultimately, leaf nodes point to data tuples that satisfy the given locality condition.

With this abstraction, query answering is essentially performing a search on indices to reach data tuples of top- k results with mini-

mal use of indices. In a Boolean query like $\mathcal{B} = price > 100K$, such a search is straightforward as the constraint expressions \mathcal{B} explicitly suggests how to carry out a focused search, e.g., visiting only the nodes with locality potentially satisfying \mathcal{B} . In contrast, for a general k -constrained optimization query potentially involving arbitrary ranking combined with Boolean conditions and joining multiple relations, e.g., \mathcal{Q} maximizing $\frac{size}{price}$ ratio, it is no longer clear how to focus the search.

Second, from *continuous function optimization*, our goal is to optimize \mathcal{G} over the domain of the database. To perform a focused search toward optimizing \mathcal{G} , we may consider using existing function optimization schemes, e.g., hill climbing or genetic algorithm [16]. However, such schemes identify the values optimizing the given function over *continuous value space*, defined by domains of the query attributes $dom(A_1) \times \dots \times dom(A_m)$. In contrast, a k -constrained optimization query optimizes over database \mathcal{D} with arbitrary “membership” restriction. Meanwhile, existing function optimization schemes optimize over either continuous space (e.g., reals) or discrete space (e.g., integers) with regular structures, and thus cannot support arbitrary database membership.

Putting together, neither discrete state search nor continuous function optimization itself can stand as a solution to answer k -constrained optimization queries. Our challenge is thus to develop a seamless integration of the two— We can view such integration as an “informed” discrete search, guided by function optimization on \mathcal{G} , to minimize the overall cost. We state the goal of such an integrated framework below:

Definition 2 (Query evaluation): Given a database \mathcal{D} and indices $\mathcal{I} = \langle I_1, \dots, I_m \rangle$ on query attributes $A_i, i = 1, \dots, m$, the goal of answering a given query $\mathcal{Q} = \langle \mathcal{G}, k \rangle$ is to find top- k results t_1, \dots, t_k in \mathcal{D} , such that $\mathcal{G}(t_i)$ is maximum, over a state space constructed by \mathcal{I} (as we will discuss in Section 3) with a minimal access cost, which we formulate as

$$cost = w_l * N_l + w_t * N_t + w_i * N_i \quad (2)$$

where w_l, w_t and w_i are the costs of visiting a leaf, tuple and internal state respectively, and N_l, N_t and N_i are the numbers of leaves, tuples and internal states visited during the search. ■

To illustrate our evaluation goal, consider our running example query \mathcal{Q} . Figure 1(c) describes two indices on our query attributes *price* and *size*, which essentially partition the domains of *price* and *size* into a discrete set of regions, or states, preserving the value locality, as Figure 1(d) demonstrates. Our goal is to use hierarchical and interleaf traversals provided by the two indices effectively to get to the top- k results with the minimal access cost.

2.3 Challenges

The essential challenge in realizing the marriage of function optimization with discrete state space search is to encode k -constrained optimization query as an appropriate search problem, which looks for solutions that optimize the goal function. To illustrate the challenge, recall from Section 2.2 that indices essentially lay out a

“map” of a discrete set of regions and provide effective traversals among them (Figure 1(d)). However, this map of regions is flat with no distinction among regions, while each region differs in \mathcal{G} scores as Figure 1(b) illustrates. To enable an efficient search, we thus let regions to reflect the landscape of \mathcal{G} (as in Figure 1(b)), to pursue an informed search guided by such a landscape. In particular, such informed search schemes should be guided properly with some heuristics. Among the informed discrete space search schemes, A^* [18], which we will describe algorithmically in Section 5, is a well-known search algorithm that finds the shortest path, given an initial and a designated goal state (or alternatively, a “well-specified” goal test condition). A^* has been proven to be *complete* and *optimal* with a proper heuristics, that is, under certain restrictions (which we will discuss in Section 4), A^* is guaranteed to find the correct answer (completeness) by visiting the least number of states (optimality).

We therefore ask two important questions: Why and how do we encode our problem into an A^* search problem?

First, we ask *why*. Why do we need to encode as a general search problem, while existing algorithms do not? As we will discuss further in Section 7, existing algorithms build upon their problem-specific assumptions on the goal functions or index traversal. To illustrate, a representative top- k algorithm TA [6] assumes the monotonicity of \mathcal{G} and the use of sorted accesses, or interleaf navigation, based on which the search is implicitly “hard-wired”. In contrast, by encoding into a generic search with no problem-specific assumptions on \mathcal{G} or how we traverse on index, we generalize our search to support (1) arbitrary \mathcal{G} , (2) over potentially multiple indices, and (3) a combination of both hierarchical and interleaf traversals, in order to enable a general support for k -constrained optimization queries.

Second, with the need of encoding identified, we now move on to ask *how*. For this purpose, we connect back to our two perspectives: From the *discrete state search perspective*, we need to define our “map” for A^* search, by mapping index nodes into states and interconnecting them in a correct way to ensure the correctness of the problem. As one of the challenges, A^* search requires either a designated goal state or a well-specified goal test condition testing each state independently. However, in our context of k -constrained optimization, it is challenging to identify such an independent test condition, as our objective of identifying top- k results is essentially “context-dependent”—it depends on the rest tuples to score lower to decide the top ones. Another challenge is to transform our problem into a shortest path problem, which A^* aims at. Such transformation is non-trivial, as the purpose of the search in k -constrained optimization is to *reach* the goal state maximizing \mathcal{G} , while the purpose of search in A^* is to find out *how to reach* the goal state with the shortest distance. Therefore, we need to encode the search space in a way to distinguish the quality of states, rather than distinguishing the quality of paths as in a typical shortest path problem. From the *continuous function optimization perspective*, we develop the “landscape” to determine where to start search and how to proceed. As overviewed above, with a proper heuristics that guides the search correctly and efficiently as well as initial states that can reach the goal, A^* search satisfies the completeness and optimality, as desired by query answering. To claim the completeness and optimality, we need to develop a proper heuristics and appropriate initial states using A^* .

Putting together, to enable this encoding, we will address the challenges identified above in the following sections. First, Section 3 will discuss the challenges from the discrete state search perspective, including constructing states from individual indices into a map and encoding our problem as a shortest path problem

on the map. Second, Section 4 will discuss the challenge from the continuous function optimization perspective, by introducing the landscape of \mathcal{G} to the search space, which involves defining search heuristics to quantify the “promise” of states and identify valid initial states.

3. INDEX-INDUCED STATE SPACE: A^* -DRIVEN CONSTRUCTION

As motivated in Section 2, from the view of indices, k -constrained optimization is essentially to find tuples satisfying the query by traversing indices. Such indices thus induce a discrete state space that lays out the “map” for traversing and eventually reaching tuples in the database. This index view motivates answering k -constrained optimization queries from the perspective of discrete state search.

To enable such discrete state search, we first need to construct the state space induced by indices. As an analogy, this is to lay out a “static” map that reflects the available index structures in the database. The map gives “locations” (as states) and “routes” (as transitions), and will be further “dynamically” configured with goal function induced landscape to complete an efficient exploration of the map to find query answers.

While an index defines an individual search map over a particular attribute, a k -constrained optimization query usually involves multiple related attributes, which together optimize the goal function. Therefore we need to construct a joint space over multiple indices, which Section 3.1 discusses. Further, after constructing states and transitions, we need to set up a well-defined destination on the map which the search heads to, and capture the distance of route. Identifying such a destination and capturing their distance effectively transform our k -constrained optimization problem to finding a shortest path to reach the destination, which Section 3.2 discusses. As the main product of state space construction, Figure 3 formally defines such a joint space over a database using indices, which we will refer to along our discussion.

As a joint state space is composed from individual search graphs induced from indices, we thus first present an index in terms of search graph. An index I_i over relation D_i defines a search graph $I_i = (V, E)$, where $V = R \cup T$ is the union of the set of index nodes R and database tuples T . We use $dom(n_i), n_i \in R$ to denote the range of values defined by index node n_i . For instance, in Figure 1(c), node a_2 has $dom(a_2) = [0, 250k]$. The edges E in index graph contain a set of parent-child links between index nodes, a set of sibling links between leaf nodes, and a set of TID links from leaf nodes to the containing tuples. Figure 1(c) shows the nodes and edges of index graph for index I_1 and I_2 . As we can see, given a node n , the reachable nodes from n depend on the type of n . For an *internal* node $n \in I.V$, the reachable nodes from n are the children of n , denoted as $Child(n)$. For a *leaf* node, the reachable nodes consist of sibling nodes $Sibling(n)$ reached through the bidirectional interleaf pointers and tuples $Tuple(n)$ reached through TID pointers stored in n . In the following discussion, we will use object-oriented notation to refer to a component of an index graph. For instance, the nodes of index graph are referred to as $I.V$ and edges as $I.E$.

3.1 Mapping the Space: State and Transition

Given individual search graphs from multiple indices, we first need to compose a joint graph, as a map of the space which will be searched for query answers. In principle, such a composite graph, as a cartesian product of those individual index graphs, describes all paths to reach tuples in \mathcal{D} . Specifically, to construct the state space from a set of index graph $\mathcal{I} = \langle I_1 \dots I_m \rangle$, we need to define a composite graph $I = (V, E)$ over \mathcal{I} , where V is the set of

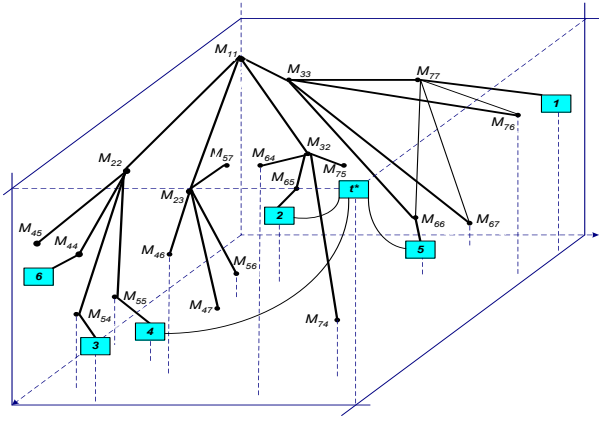


Figure 2: Illustration of state space: States and Transitions.

states and E is the set of transitions between states.

3.1.1 States

States in a search graph represent “localities” of values at different granularity— from coarse to fine, and eventually reach tuples in the database. Therefore, a state effectively summarizes a set of tuples within this locality. In parallel to the two types of vertices— index nodes and tuple nodes in an index graph, there are two types of vertices in the composite state graph. R nodes represent “regions” of values defined by a set of index nodes and T nodes represent tuples in database \mathcal{D} .

Region State: While an individual index node defines a *range* of values along an attribute, a composition of multiple index nodes thus defines a *region*. For instance, a pair of index nodes $[a_3, b_3]$ from index I_1 and I_2 in Figure 1(c) defines a region $M_{33} = [a_3, b_3]$ in Figure 1(d). In general, any combination of index nodes $\langle n_1, \dots, n_m \rangle$ from $\bowtie_i I_i$. R is a valid state representing a region defined by $\text{dom}(\langle n_1, \dots, n_m \rangle) = \bowtie_i \text{dom}(n_i)$. We call such a state *region state*. A region state r represents a set of tuples with attribute values falling into $\text{dom}(r)$. As Figure 3 formally defines, given a set of index graphs, the set of region states is the cartesian product of index nodes from each index. (As convention, we will use region M_{ij} and state $[a_i, b_j]$ interchangeably to refer to a state or region defined by index node a_i and b_j .)

Similar to nodes in an index, region states in the composite graph can be categorized into *leaf state* and *internal state*. A state $r = \langle n_1, \dots, n_m \rangle$ is a leaf state if all nodes $n_i, i = 1, \dots, m$ are leaf nodes in corresponding indices, otherwise, r is an internal state. A leaf state directly reaches all tuples in region r . For instance, state M_{66} is a leaf state defined by a_6 and b_6 , and it reaches tuple 5 that falls in region M_{66} , as shown in Figure 1(d).

Tuple State: In parallel to tuple component T in an index graph, each tuple $t \in \mathcal{D}$ also defines a state, which we call *tuple state*. Tuple states correspond to potential query answers. The goal of search is to find the tuple states that maximize the goal function \mathcal{G} .

3.1.2 Transitions

While states of space give “locations” in the map, transitions further capture possible *paths* followed to reach our destination of query answers. With the states of composite graph being defined upon the nodes of index graphs, the transitions between states are further defined upon the edges in the index graphs. Different types of transitions lead to different behaviors of traversing the space, *e.g.*, hierarchical and interleaf traversals as motivated in Section 2.

Essentially, to construct transitions between states, we need to define a *Next* function, which returns the possible states directly reachable from a given state. That is, for two states u and v , there is a transition (u, v) if $v \in \text{Next}(u)$. Similar to individual index

graphs, in a composite graph, the reachable states from a state r depend on the type of the state.

Internal state–Branch in: For internal state r , the reachable states are generated by following the parent-child links in the index nodes of r . Therefore, such a transition effectively branches from a parent state to subsuming children states. Such branch-in transitions enable a *top-down* search approach, which starts from root region and gradually zooms into query answers.

Specifically, to generate reachable states for an internal state r , we expand all the internal index nodes of r to their children nodes and generate children states. For instance, from an internal state M_{33} , by expanding both a_3 and b_3 , we reach four states, M_{66}, M_{67}, M_{76} and M_{77} . We choose to expand simultaneously all internal nodes because such expansion gives the shortest way to reach a leaf state. Alternatively, we may expand a subset of internal nodes in r , but such selective expansion does not improve search performance— As data tuples can be only reached at the leaf nodes, it is always better off to expand all internal nodes to find the most efficient path early on, rather than expand one index at a time.

Leaf state–Branch out and materialize: As in a leaf node of an index, which reaches both sibling nodes and tuples, the reachable states for a leaf state r also consist of two parts— neighbors of r and tuples contained in r . Expansion to the neighbor states effectively branches out from a leaf region to its surrounding leaf regions, and expansion to the tuple states materializes tuples from TID. Such branch-out transitions enable *bottom-up* search, which starts from specific leaf states and gradually spreads out to reach answers.

To generate neighbor states, the expansion follows the sibling pointers in the leaf nodes to new leaf nodes. The neighboring leaf states are generated by combining each new leaf node with leaf nodes from other indices. To illustrate, consider a leaf state $[a_6, b_6]$. By following sibling links in two indices, we can reach a_7 and b_7 respectively. Taking a cartesian product of them, we reach the neighbors M_{67}, M_{76} and M_{77} .

In addition to generating neighboring regions, at a leaf state, expansion also reaches out to tuple states by following the TIDs stored in the leaf nodes of indices. For instance, from state M_{66} , we can reach tuple 5 by following TID pointer in $a_6 \cap b_6$. Although from the two leaf nodes, we actually have the opportunity to reach all tuples covered in $a_6 \cup b_6$, the region M_{66} only defines tuples in $a_6 \cap b_6$. As we will see later, the estimated score bound of heuristics function only bounds the states in $a_6 \cap b_6$, not the others, *e.g.*, tuple 2. Therefore, allowing the transitions from a leaf state r to those tuples other than $a_6 \cap b_6$, we will violate the properties required by the heuristics function to guarantee the correctness of search. For the same reason, in expansion of the internal states, we do not follow the TID links to reach tuple states even if there exists a leaf node in this state. Based on the consideration, for a set of leaf nodes from the same relation, we allow the transitions only to TIDs defined in the intersection of those leaves. With leaf nodes from multiple relations in a state, the reachable tuple states are the cartesian product of such intersected TIDs from each relation.

By defining the *Next* function, as formally described in Figure 3, we construct transitions between states. Following different types of transitions among states, search navigates the space in different ways, as the following example illustrates. Note that although the *Next* function is defined for every state, paths to the children states are only followed and thus materialized when the search selects to follow that particular state, as Section 5 will discuss. Therefore, the graph is effectively constructed and selectively materialized “on-the-fly” rather than statically pre-computed.

Example 3: Continue Example 1. Given a set of states constructed from the set of index graph \mathcal{I} , Figure 2 further illustrates (part of)

the transitions among the states. The search, in principle, should follow those transitions to look for the tuple states maximizing the goal function. For instance, suppose we decide to start from the root of the graph M_{11} . The search may follow the path $M_{11} \rightarrow M_{33} \rightarrow M_{77} \rightarrow 1$ to reach the target tuple state. This essentially follows a top-down search strategy. Alternatively, as a bottom-up search, suppose we start from M_{67} , the search may follow an alternative route $M_{67} \rightarrow M_{77} \rightarrow 1$. ■

<p>OPT* ENCODING: k-CONSTRAINED OPTIMIZATION OF \mathcal{G} OVER \mathcal{D} USING \mathcal{I} Input: Indices $\mathcal{I} = (I_1, \dots, I_m)$, Goal function \mathcal{G}, Databases $\mathcal{D} = \bigotimes_{i=1, \dots, n} D_i$ Output: State space $I = (V, E)$, BlackLink set \bar{E}, Initial states \mathcal{S}</p> <hr/> <p>INDEX-INDUCED SPACE CONSTRUCTION</p> <p>Input: Indices $\mathcal{I} = (I_1, \dots, I_m)$, Goal function \mathcal{G}, Database $\mathcal{D} = \bigotimes_{i=1, \dots, n} D_i$ Output: State space $I = (V, E)$ /*construct states of I^*/ $I.R = \bigotimes_{i=1, \dots, m} I_i.R$ $I.T = \mathcal{D}$ $I.V = I.R \cup I.T \cup \{t^*\}$ /*construct transitions of I^*/ $\forall r = \langle n_1, \dots, n_m \rangle \in I.V$: /*construct reachable states for r^*/ $Next(r) = \{t^*\}$ if r is a tuple state $Next(r) = \bigotimes_{i=1, \dots, m} Child(n_i)$ if r is an internal state $Next(r) = Neighbor(r) \cup Tuple(r)$ if r is a leaf state, where $Neighbor(r) = \bigotimes_{i=1, \dots, m} Sibling(n_i) \cup \{n_i\} \setminus r$ /*join TIDs from multiple relations D_1, \dots, D_n^*/ $Tuple(r) = \bigotimes_{i=1, \dots, n} T_i$, where $T_i = \bigcap_{\{j rel(A_j)=D_i\}} Tuple(n_j)$ $E = \{(r, v) v \in Next(r)\}$ $\forall (u, v) \in E$: $d(u, v) = -\mathcal{G}(u)$ if $v = t^*$ $d(u, v) = 0$ otherwise</p> <hr/> <p>GOAL-INDUCED SPACE CONFIGURATION(I, \mathcal{G})</p> <p>Input: State space $I = (V, E)$, Goal function \mathcal{G} Output: Black set \bar{E}, Initial states \mathcal{S} $h_{\mathcal{G}}(r) = \text{OPTMAX}(\mathcal{G}, r)$ $\bar{E} = \{(u, v) (u, v) \in I.E \wedge h(u) < h(v)\}$ $O = \text{OPTPOINT}(\mathcal{G}, \text{dom}(I.root))$ $\mathcal{S} = \{r \in I.R \forall p \in O, \exists r \in \mathcal{S} \wedge p \in \text{dom}(r)\}$</p>
--

Figure 3: State space: OPT* encoding.

3.2 Where to Head to: Goal State

While states and transitions compose the map for search, we need further identify our destination, *i.e.*, the goal state, to head to. Among the states in the space, the actual goal states are the ones that correspond to the tuples maximizing the goal function \mathcal{G} . Therefore, our problem is to find out such “optimal tuple states” with maximal \mathcal{G} -score. This is different from the traditional shortest path problem addressed by A^* , where the search looks for an “optimal path” to reach a testable goal state. To apply A^* for k -constrained optimization we thus need to transform our problem of finding optimal tuple states to finding the optimal path to reach a goal state. The key to the transformation is: First, to encode a tuple state with a path passing the state towards a testable goal; Second, to encode the quality of those tuple states with quality of those paths so that the optimal state corresponds to the shortest path.

To begin with, to transform a tuple state to a path which passes the state towards a testable goal, we need to add a pseudo goal t^* as a goal state and connect each tuple to this pseudo goal t^* . Therefore, in the state space, the reachable states for a tuple state t is the pseudo goal t^* , *i.e.*, $Next(t) = \{t^*\}$. With this pseudo goal t^* , each path reaching t^* corresponds to a unique tuple state (since there is no edge between tuple states), and therefore finding a path corresponds to finding a tuple state. For instance, a path $P_1 = (M_{11}, M_{33}, M_{77}, 1, t^*)$ corresponds to tuple state 1.

Further, to transform the optimal tuple state into the shortest path passing this state, we need to assign proper distances to edges between the states. As Figure 2 illustrates, the key observation enabling the transformation is that the actual goal state 1 maximizes the goal function, and thus such a tuple (with a maximal score) must by definition have the shortest path to t^* . To reverse distance minimization to score maximization, we thus define the distance from a tuple state t to t^* as the inverse \mathcal{G} -score of the tuple state, *i.e.*, $d(t, t^*) = -\mathcal{G}(t)$.

Meanwhile, while the above distance assignment ensures the optimal tuple state o has the shortest distance to the goal, we need further ensure that a path passing through o has the shortest overall distance. For instance, among the two paths $P_1 = (M_{11}, M_{33}, M_{77}, 1, t^*)$ and $P_2 = (M_{11}, M_{32}, M_{65}, 2, t^*)$, it should be $d(P_1) < d(P_2)$ because tuple 1 is the top answer. Note that the distance of a path $P = (v_1, \dots, v_n, v_{n+1})$ is the overall distance of P , *i.e.*, $d(P) = \sum_{i=1, \dots, n} d(v_i, v_{i+1})$. Specifically, for any two paths $P_1 = (v_1, \dots, v_n, t^*)$ and $P_2 = (v'_1, \dots, v'_n, t^*)$, we should have $d(P_1) < d(P_2)$ if $d(v_n, t^*) < d(v'_n, t^*)$. To ensure this inequality, we therefore assign, for all internal edges, $i = 1, \dots, n - 1$, $d(v_i, v_{i+1}) = 0$, which yields $d(P_1) = d(v_n) < d(P_2) = d(v'_n)$. Therefore a tuple state with the shortest distance to t^* , or equivalently maximal score, corresponds to the shortest path to t^* .

As Figure 3 depicts, by assigning all edges between “physical” states with distance 0 and edges from tuple states to the pseudo goal with distance as inverse \mathcal{G} -score, we transform finding the tuple state with maximal score to finding the shortest path to the pseudo goal. As an extension, to find top- k results in k -constrained optimization is to find k shortest paths to the pseudo goal.

4. GOAL-INDUCED SPACE: OPTIMIZATION DRIVEN CONFIGURATION

The previous section discussed answering k -constrained optimization queries from discrete state search perspective, specifically, how to encode a static state space, reflecting the index structures available for search. In this section, we turn to the function optimization perspective. As discussed in Section 2, answering k -constrained optimization query is essentially to search in discrete state space driven by function optimization. Therefore, to complete the picture of search, we study how function optimization contributes to the evaluation of k -constrained optimization queries.

Specifically, while indices induce a static map of state space, we further need a landscape over the map so that the search is guided efficiently towards the goal. Conceptually, such a landscape *measures* the relative “qualities” of different states with respect to the goal function, and thus gives dynamic “query-specific” configuration over the static space. Such configuration refines the state space to be searchable by A^* algorithm, and completes the encoding of k -constrained optimization query into A^* search problem. The configuration involves two aspects: First, it defines a proper heuristics based on the goal function to guide the search and configures the state space with respect to this heuristics (Section 4.1). Second, it identifies a set of initial states decided by the goal function to start the search properly (Section 4.2).

Our tool of such measurement is continuous function optimization, a well-studied technique. First, to define a heuristics to guide A^* search, as we will see in Section 4.1, we need function optimization techniques to estimate the upper bound score of tuple states reachable from the current state. Second, to identify a set of initial states, as we will see in Section 4.2, essentially boils down to finding local optimal points of the goal function in a value domain.

To achieve the two goals, we define a function optimization procedure OPT. The procedure takes as input a function $\mathcal{G}(x_1, \dots, x_m)$

and a domain of values $dom = [x_1^1, x_1^2] \times \dots \times [x_m^1, x_m^2]$ where $x_i \in [x_i^1, x_i^2]$. It returns a set of local optima O and an upper bound score U that can be achieved within the domain, *i.e.*,

$$\langle O, U \rangle = \text{OPT}(\mathcal{G}, dom) \quad (3)$$

where $O = \{p | p \in dom \wedge p \text{ is a local optima}\}$ and $U = \max_{p \in dom} \mathcal{G}(p)$.

Specifically, we use OPTMAX to denote the function that returns the U component of OPT, and OPTPOINT to denote function that returns O component of OPT.

As implementation to this procedure, there are three categories of techniques. First, analytical methods [16] for function optimization compute the derivatives (or gradient for multivariable functions) of the goal function, and get the extremum at the points where their derivatives are equal to zero. Second, search-based methods, *e.g.*, hill climbing or conjugate gradient method [16], find extremum by approximating them gradually and infinitely in the regular structured value space. Third, template-based approach “hard-codes” extremum, if the goal functions are parameterized with certain fixed form. In practice, due to the problem-specific assumptions, this approach is often very simple and useful. For instance, for monotonic functions, the upper bound score of a region is achieved by taking maximal value at each dimension. In a *KNN* problem with Euclidean distance as a goal function to minimize, the upper bound score is achieved either at the query point or one of the boundary planes of the region.

4.1 Measuring the Landscape: Heuristics

With state space laying out a map of search, we need to further measure the landscape indicating the “ups” and “downs” of the states with respect to the goal function. As Figure 2 illustrates, different states have different “promises,” or heights as indicated by the dotted line, to reach the goal—Some are closer to the goal while others are farther. For instance, at state M_{33} , the search faces multiple children states, *e.g.*, M_{67} and M_{77} , showing different promises. It can be verified that the maximal score of tuples in region M_{67} is 0 and that of region M_{77} is 45. Therefore, the search should favor the choice of M_{77} over M_{67} because it is more promising.

To drive efficient search, A^* algorithm needs a heuristics estimation for the cheapest path to reach the goal. To guarantee the completeness of the A^* algorithm, such a heuristic must be *admissible* and *descending*. First, *admissibility* requires the heuristic function does never overestimate the distance to the goal. Second, *descendence*² requires the heuristics estimation never decreases on any path possibly visited by A^* from the initial state to reach the goal state. In our problem, it means the estimated \mathcal{G} -score never increases, and we therefore call it as descendence property.

The admissibility requirement means that, given a state r , our heuristics can only estimate optimistically about the scores of tuples reachable from r . Therefore, the estimation should be an upper bound score of those reachable tuples. Further, to make the estimation as accurate as possible, the heuristics should give tightest upper bound for the state. Intuitively, given that the available information at state r is only the value ranges $dom(r)$ provided by index nodes, we may design our heuristics h as the tightest upper bound that goal function \mathcal{G} can achieve within the value ranges, *i.e.*,

$$h_{\mathcal{G}}(r) = \text{OPTMAX}(\mathcal{G}, dom(r))$$

However, although such an estimation is the best we can make, it does *not* satisfy the admissibility and descendence property be-

²To avoid ambiguity between the monotonicity property of functions, typically referred to by top- k queries, we use the term *descendence*.

cause of the sibling edges between leaf states. Let us use the following example to illustrate.

Example 4: Consider state M_{67} . Using the heuristics $h(M_{67})$, we get the upper bound score of M_{67} is 0, which bounds only the tuples located within region M_{67} . However, if following the link to its neighbor state M_{77} , we can actually reach tuple 1 with score 15. This means that the heuristics function does not give upper bound of all tuples reachable from M_{67} , and thus violates the admissibility property. Further, traversing from M_{67} to M_{77} , the heuristics score increases from 0 to 45, and thus violates descendence property. ■

The reason that such heuristics violates admissibility and descendence is that we introduce some “problematic” links, as we aim to support all access paths, not only the parent-child transitions but also sibling transitions available in indices. Such sibling links make leaf states fully connected in the state space. Therefore, at any region state r , we can reach every tuple state, and thus the upper bound of state r should be the upper bound of the entire database \mathcal{D} , instead of just tuples confined in region r . However, such an admissible heuristics will end up with giving the same estimation to every region state, and thus provide no guidance on how the search should proceed.

Given that the heuristics h is the best estimation we can make at a state r , to *enable* A^* search, we therefore need to configure the state space to make this heuristics admissible and descending. We observe that the admissibility and descendence are violated when the search takes “up-hill” edges between leaf states, such as M_{67} to M_{77} . Therefore, if we remove all those *blacklinks* that should not be followed (named as in “black list” which we will formally define later), starting at any state, the score only decreases. This on the one hand obviously satisfies the descendence property, on the other hand also meets the admissibility property because a state can only reach tuple states through downhill links, and thus the heuristics gives upper bound to the reachable tuple states, as the following example illustrates.

Example 5: Consider the state space (partially) shown in Figure 2. All leaf states M_{6i} , $i = 4, 5, 6, 7$ (corresponding to regions in the third column of Figure 1(d)) have heuristics score 0 because they disqualify the price range in query \mathcal{Q} . Therefore, the blacklink set \bar{E} contains, along with others, all edges originated from M_{6i} , *e.g.*, (M_{67}, M_{57}) , (M_{67}, M_{77}) . In the configured graph, these states become “sinks” with only incoming links. Therefore if we start from M_{57} , we can reach M_{67} and M_{56} through downhill links, but not any of the states M_{7i} , $i = 4, 5, 6, 7$ (corresponding to the states in the fourth column). ■

Definition 3: Given a state space $I = (V, E)$, a heuristics function h , we say an edge $(u, v) \in E$ is a *blacklink* if $h(u) < h(v)$. All the blacklinks in I compose the *blacklink set* $\bar{E} = \{e | e \text{ is a blacklink}\}$. A configured state space I_h of I is a subgraph of I with blacklinks removed, *i.e.*, $I_h = I \setminus \bar{E}$. ■

By removing the blacklinks, as Figure 3 formally describes, we configure the conceptual state space into a space searchable by A^* algorithm, which guarantees the admissibility and descendence properties. However, the implication of removing the blacklinks is that leaf states become not fully connected. Such disconnection impacts the reachability of the search, *i.e.*, some tuple state may be unreachable if we start at a wrong initial state. This is the problem to be addressed in the next section.

4.2 Where to Start: Initial States

While configuration of the state space with blacklinks guarantees the admissible and descending properties required by heuristics function, it also imposes the problem of *reachability*, *i.e.*, change

of the landscape makes two leaf states become disconnected and unreachable from the search. To illustrate, consider Example 5—suppose the search starts at M_{57} , it cannot reach M_{77} , which contains the top answer to query \mathcal{Q} . Therefore, the search fails.

To address this reachability problem, we need to pick up a set of states from which we can reach *every* tuple state in the space. Does such a set of states exist? Obviously, the root state of the state space satisfies the requirement, and thus top-down search can always find the correct answers. However, top-down may be inefficient because starting from root needs more hops to reach the goal than starting from, say, a leaf state. Therefore, we want to find out better alternatives, starting closer to the goal. Observe that the states which are possibly missed are those containing local optima of the goal function, *e.g.*, M_{77} . Other states, *e.g.*, M_{74} , M_{75} and M_{76} , can all be reached by taking “downhill” edges from their surrounding local optima states, *e.g.*, M_{77} .

To guarantee that every tuple is reachable during the search, we therefore initialize the search with a set of states that cover all the local optima points returned by $\text{OPTPOINT}(\mathcal{G}, \text{dom}(I.\text{root}))$, where $I.\text{root}$ is the root state of I , defined by roots of each index.

Example 6: As the landscape of Figure 1(b) shows, the goal function \mathcal{G} has two local optima $\langle 200k, 4500 \rangle$ and $\langle 400k, 4500 \rangle$. The two local optima are located in M_{57} and M_{77} respectively. Therefore $\mathcal{S} = \{M_{57}, M_{77}\}$ covers all local optima points of \mathcal{G} . Starting from \mathcal{S} —closest to the goal state, bottom-up approach leads to the most efficient search, as shown in Section 5. ■

Specifically, given state space $I = (V, E)$, heuristics function h , we say a set of states $\mathcal{S} \subseteq V$ is a *sound set of initial states*, if \mathcal{S} covers all local optima points, *i.e.*, $\forall p \in \text{OPTPOINT}(\mathcal{G}, \text{dom}(I.\text{root}))$, there exists a state $r \in \mathcal{S} \wedge p \in r$.

By initializing the search with this sound set, we can guarantee the correctness of A^* search framework, as the following theorem states. Due to space restriction, we omit the proof.

Theorem 1: Given a query $\mathcal{Q} = \langle \mathcal{G}, k \rangle$, a state space I , a heuristics function $h(r) = \text{OPTMAX}(\mathcal{G}, r)$, a set of states \mathcal{S} , A^* search guarantees to correctly find answers to \mathcal{Q} if \mathcal{S} is a sound set of initial states. ■

By initializing the search with local optimal states, A^* search automatically ignores the blacklinks because the search always chooses the best state currently available for processing. For instance, consider again Example 5, starting with the best initial state M_{57} , the search expands to its neighbors, *e.g.*, M_{67} . However, since M_{77} has a higher score than M_{67} , the search now will not follow outgoing links from M_{67} anyway, but rather jump to M_{77} , which is the state currently with the best heuristics score.

While Theorem 1 states the correctness requirement for choosing initial states, it also leaves us with different options of initialization. For instance, a top-down search may choose to start at root of the graph, which trivially covers all local optimal points. Or alternatively, a bottom-up search starts with a set of local optimal leaf states, and gradually expands the search to query answers. Different initialization strategies will result in different search cost, as Section 5 will discuss.

5. OPT* SEARCH

Upon the state space correctly encoded (as shown in Figure 3), the A^* search naturally follows. First introduced by Hart et. al, A^* is a graph search algorithm that finds the shortest path from a given initial node to a given goal node (or one passing a given goal test). It employs a “heuristic estimate” that ranks each node by an estimate of the best route that goes through that node. As implementation, A^* maintains a priority queue, which stores the partial paths starting

```

Procedure OPT*( $\mathcal{I}, \mathcal{G}, k$ )
Input: Indices  $\mathcal{I} = \langle I_1, \dots, I_m \rangle$ 
      Goal function  $\mathcal{G}$ 
      Result size  $k$ 
Output: top- $k$  results TupleQ
begin
   $\tau \leftarrow -\infty$ 
  OPT_INITIALIZE(ToDoQ,  $\mathcal{I}$ )
  While not REACHGOAL():
     $r = \text{GETNEXTSTATE}()$ 
    if  $r \in \text{HaveDoneQ}$ : continue
    insert  $r$  to HaveDoneQ
    newR ← EXPAND( $r$ )
    for each state  $s \in \text{newR}$ :
      OPT_HEURISTIC( $s, \mathcal{G}$ )
      if ELIGIBLE( $s$ ):
        case  $s$  is:
          Region State: insert  $t$  into ToDoQ
          Tuple State:
            insert  $t$  into TupleQ
             $\tau = \min(\text{TupleQ})$ 
end

```

Figure 4: Query algorithm: OPT* search.

from the initial node, prioritized by the estimated minimal distance to the goal state. A^* thus visits the nodes in order of this heuristic estimate from the priority queue.

In this section, we briefly present our OPT* search algorithm as a specialization of A^* search algorithm. As mentioned in Section 3, by different realization of the initialization operation, OPT* ends up with visiting different set of states. This section therefore also discusses how such difference affects the performances.

5.1 Skeleton of OPT*

In this section, we present our OPT* algorithm. Figure 4 outlines the skeleton of the algorithm. Specifically, the algorithm keeps two priority queues - ToDoQ to keep the region states to be further explored and TupleQ to keep track of current top- k tuple states among all those visited thus far. Further, the algorithm also maintains a hash table HaveDoneQ to record all leaf states that have been visited. ToDoQ is initialized with a sound set of initial states. The algorithm continues to retrieve a state r from ToDoQ and expands it until we reach the goal states. At each iteration, given a state r passed by GETNEXTSTATE, procedure EXPAND first checks the type of state r . If r is an internal state, EXPAND generates new set of states newR using the children nodes of indices in r , as discussed in Section 3. If r is an unvisited leaf state, the algorithm adds r to HaveDoneQ, and then EXPAND takes two actions: to generate neighbor states and to reach tuple states respectively. The expansion will update ToDoQ if new region states are generated, or update TupleQ if tuple states are reached. The search terminates when the stop condition is met.

To initialize, we apply the function optimization procedure OPTPOINT to find out the set of local optima points. As we will show in the next section, such initialization gives the most efficient search. For each (optimal) point, we traverse the index to locate the leaf state containing this point. As an optimization, instead of fully materializing all local optimal leaf states beforehand in the ToDoQ, we can materialize them “on-demand.” Specifically, we keep track of those local optima points in OptimaQ prioritized by their \mathcal{G} -scores, and initialize the ToDoQ with only the global optimal leaf state. A point in OptimaQ will be materialized into a leaf state only when its \mathcal{G} -score is greater than the score of the top-scored state in ToDoQ. By doing this, we only materialize the leaf states that need to be visited during the search. Due to space limitation, we omit the proof of the correctness of this “on-demand” materialization.

The algorithm stops when ToDoQ is empty. This stop condition can further be sped up by comparing the current top- k answers with the top-scored state in the ToDoQ. In particular, we set the thresh-

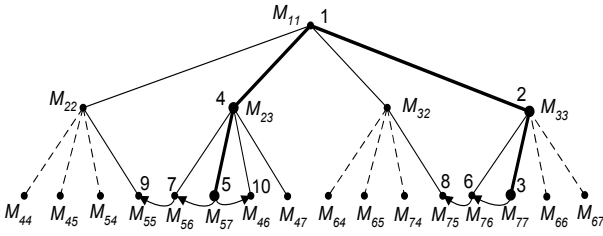


Figure 5: Search Tree.

old τ to keep track of the score of the k th-tuple in `TopleQ`. If τ is greater than the best state in `ToDoQ`, we can empty the queue. Actually, this testing can happen even earlier at the time when a candidate state is attempted to enter the queue. The eligibility test operation implemented by `ELIGIBLE` realizes this testing.

5.2 Optimality of OPT*

While A^* is an optimal algorithm in discrete state space search, such optimality does not directly address our cost function defined in Section 2. First, A^* optimizes the total number of nodes visited in the state space, *i.e.*, $N_l + N_t + N_i$, not our cost function. Second, A^* is optimal only with respect to the given initial and goal state as well as heuristics. Therefore, starting at different initial states will make a difference in A^* search cost. In this section, we discuss the optimality of OPT* in terms of our cost function and compare the cost of search with different initialization strategies.

First, we examine the optimality of OPT* search comparing with other search algorithms for a given set of initial states. It can be shown that OPT* as specialization of A^* not only optimizes the total number of states visited, but also optimizes the number of leaf states visited. That is, any leaf state visited by OPT* will be visited by other algorithms using the same heuristics function. The intuition is that each visited leaf state has a heuristics score higher than the k -th tuple in the final top- k result. Therefore, without visiting this leaf state, an algorithm cannot conclude to find the top- k answers. Visiting the least number of leaf states in turn optimizes the number of tuple states visited because the set of tuple states visited is determined by the set of leaf states. Given that the cost of internal state access is less than the cost of leaf and tuple access, we show that OPT* optimizes the cost function, as the following theorem states. The proof is straightforward, and we omit it here.

Theorem 2: Given a set of initial states and a heuristics function, OPT* search optimizes the cost function $w_l * N_l + w_t * N_t + w_i * N_i$ if $w_l > w_i$ and $w_t > w_i$. ■

While the above theorem states the optimality of OPT* search with respect to the assumption $w_l > w_i$ and $w_t > w_i$, such an assumption typically holds true in practice, as internal nodes typically reside in memory while leaf and tuple reside in disks, *i.e.*, $w_l \gg w_i$ and $w_t \gg w_i$. Observe also that, this optimality is with respect to given initial states. Therefore, OPT* framework may result in different costs if starts with different initial states. As Figure 2 shows, by starting from different initial states, the search makes different numbers of “hops” to reach the goal. For instance, taking the *top-down* approach, we start from the root state, which makes the largest number of hops to get to the the goal. Taking a *bottom-up* approach, we start from a leaf state from a sound set, *e.g.*, M_{77} , which has only one hop to reach the target tuple state.

In particular, to compare the cost of different initialization strategies, we need to examine the number of internal, leaf and tuple states visited. Let us first examine leaf and tuple state accesses using the two extreme approaches. As we observe that although top-down search does not explicitly start with the optimal leaf state, *e.g.*, M_{77} , it eventually gets there, because M_{77} maximizes goal

function \mathcal{G} and thus maximizes the bounds of all its ancestors, which in turn will have the highest heuristic score in the search. Therefore, starting with the root, top-down search essentially follows the path to reach M_{77} first. Applying the same intuition, we can prove that the second leaf state visited by top-down is also the same as bottom-up. More generally, different sound initialization strategies will visit exactly the same set of leaf states, and therefore reach the same set of tuple states. The following property formally states this.

Property 1: Given a state space and a heuristics function, OPT* always visits the same set of leaf states and thus tuple states if the search is initialized with any sound set of initial states. ■

Given that different search approaches visit the same set of leaf and tuple states, their costs thus differ only in the number of internal node accesses. As we observe, the set of internal states accessed by the top-down approach is essentially the subtree of the state space that consists of paths from root to all visited leaf states, as illustrated by the solid lines in Figure 5. Similarly, the set of internal states accessed by bottom-up consists of the paths from root to all the local optima. However, with on-demand materializing the leaf states, the internal nodes visited consist of only the paths to those visited local optima, as highlighted in Figure 5 with thick solid lines. Obviously, the leaf states visited during the search are a superset of local optima states visited, and therefore, the internal states visited by top-down are a superset of the internal states visited by bottom-up. The following property formally states this.

Property 2: Given a state space and a heuristic function, the bottom-up approach visits the least number of internal states if initialized with a sound set of initial states. ■

Combining the two properties, we know that bottom-up search always has the lowest cost than other search algorithms, given the same state space and heuristics function. Therefore, if applicable, we will prefer bottom-up search over other initialization schemes. However, bottom-up approach may not be applicable if \mathcal{G} has an infinite number of local optima. For instance, consider goal function $\mathcal{G} = \frac{1}{(x-y)^2+1}$. Any value satisfying $x = y$ maximizes the function, and therefore there is no way to initialize the search with a finite set of states. In this situation, only top-down approach is applicable. In our experiments of Section 6, we will assume the bottom-up approach. Putting together, the following example shows the search route taken by OPT*.

Example 7: Consider Example 1. We initialize the search with $\mathcal{S} = \{M_{57}, M_{77}\}$. Starting from M_{77} , the search visits a sequence of leaf and internal states in the order labeled in Figure 5. The search returns the top 1 answer as tuple 1. As we can see, this bottom-up search visits a total number of 7 leaf states and 3 internal states. It can be verified that starting from root, the top-down approach will visit the same set of leaf states but 5 internal states. ■

6. EXPERIMENTS

This section reports our experiments on OPT* framework for answering k -constrained optimization queries. In particular, our goal is two-fold: First, to validate practicality in real-world scenarios, we evaluate OPT* using benchmark queries over real data. Second, to validate extensively, we then study its performance over a wider range of queries and data settings, by simulating over extensive synthetic queries and datasets. Toward the goal, we first overview our experiment settings in Section 6.1, then report our benchmark query experiments over real data in Section 6.2 and controlled query experiments over synthetic data in Section 6.3.

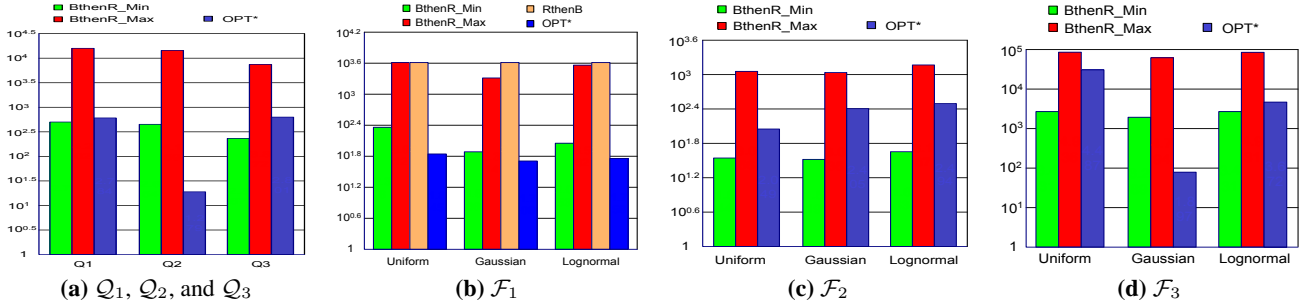


Figure 6: Average number of page accesses.

6.1 Experiment Settings

This section overviews our experiment settings, including implementation details, evaluation metrics, and baseline approaches we use for comparison.

Implementation: In our implementation, we use B+ tree as access methods, because B+ tree is the most commonly used index structure in databases. The fanout of B+ tree is set to 200, which yields a storage of around 4KB for each node. As commonly practiced in commercial DBMS, the internal nodes of B+ tree reside in memory, and leaf nodes in disks. For the implementation of our framework, we use the bottom-up approach, as it is the most efficient for \mathcal{G} with a finite number of local optima, as argued in Section 5.

Evaluation Metrics: The evaluation cost is decided by the cost sum of visiting internal, leaf, and tuple nodes, as captured in our cost model in Section 2. However, due to the fact that leaf and tuple accesses dominate the overall cost, we can adapt a simpler metric— number of leaf and tuple accesses as an approximation of the evaluation cost. That is,

$$\text{cost} = N_l + N_t \quad (4)$$

We can further infer the number of page accesses from the number of leaf and tuple accesses. In particular, assuming the number of leaves per page is 1 and the number of tuples per page is T , the number of page accesses in the worst case scenario is equivalent to $N_l + N_t$, while in the best case scenario, the number reduces to $N_l + N_t/T$.

Baseline Approaches: We introduce two baseline approaches we compare against. As we will discuss, these two approaches are representatives of the existing works, as the adaptation of the existing works for k -constrained optimization queries falls into either of these approaches. Although *RankSQL* [11] attempts a finer interleaving of the two approaches, there is no optimization to pose such interleaving yet, which thus corresponds to choosing the best of the two approaches for now.

Boolean then Rank (BthenR): This approach first evaluates the constraint expression to retrieve qualified tuples, and then applies the optimization expression to sort the results to retrieve top- k . This approach can generally process any k -constrained optimization queries, and is particularly good for queries with selective constraint expressions. We define the “boolean selectivity” of constraint expression as the ratio of tuples that satisfy the constraint expression, as typically defined for boolean queries.

To process constraint expressions we may use index scan or simply table scan with filtering. Table scan, by exhaustively accessing all N data tuples, incurs N/T page accesses. In contrast, index scan, by selectively retrieving the qualified tuples, accesses only the qualified N_t tuples— Such accesses incur N_t/T page accesses when the index is clustered (*BthenR_Min*) and up to N_t accesses otherwise (*BthenR_Max*). In DBMS, optimizer will select the desirable access method, while in our evaluation, we only report the best of the two.

To process the optimization expression, we may either rank all the qualified tuples, or alternatively apply top- k algorithms like *TA*. Recall that, top- k algorithms require sorted accesses, which can be simulated by traversing leaf nodes of index trees. However, after processing constraint expressions, the index cannot be used to provide sorted accesses on intermediate results. We thus do not consider using top- k algorithms to process optimization expression.

Rank then Boolean (RthenB): This approach first processes the optimization expression using existing top- k algorithms, specifically, *TA* algorithm, and applies the constraint expression to filter out unqualified ones. An optimization of this approach is to evaluate the constraint expression during ranking. Because *TA* performs *random access* to retrieve tuples with TID, and therefore once the tuple is retrieved, we can evaluate its \mathcal{G} -score with optimization and constraint expression together. This approach is suitable for queries with *selective* optimization constraints. We define this “ranking selectivity” of a ranking expression as the ratio of tuples accessed to generate top- k results using only ranking expression.

However, as *TA* requires monotonic ranking function, this approach only applies to those queries with monotonic ranking function. To support sorted accesses on each dimension required by *TA*, we use interleaf links in the indices to simulate. The number of pages visited includes both the leaf nodes visited by sorted access and the tuples retrieved by random access.

6.2 Benchmark Queries on Real Data

To validate the practicality of our framework over real-world data retrieval scenarios, we evaluate OPT* using benchmark queries on real data.

Datasets: Our real dataset contains 19706 houses listings crawled from *realtor.com*. The dataset has four attributes on (*price, size, bathrooms, bedrooms*).

Queries: Our benchmark queries were handcrafted simulating house search scenarios. The three queries are specified as following:

$$Q_1: \frac{\text{size} \times \text{bedrms}}{|\text{price} - 450k|} : [40k \leq \text{price} \leq 50k]$$

$$Q_2: \frac{\text{size} \times e^{\text{bedrms}}}{|\text{price} - 350k|} : [\text{price} < 400k \wedge \text{size} > 4000]$$

$$Q_3: \frac{\text{size}}{\text{price}} : [\text{bedrms} = 3 \vee \text{bedrms} = 4]$$

Results: Figure 6(a) shows the performance results for query Q_1 , Q_2 and Q_3 on real dataset. Note that, as the optimization expression is not monotonic, only *BthenR* approaches and OPT* are applicable for queries in this set. For *BthenR* approaches, we show both the best case scenario where tuples accessed are clustered (*BthenR_Min*) and worst case scenario where tuples are scattered in the disks (*BthenR_Max*). Observe that, in all queries, OPT* performs better or close to the best baseline approach. Particularly, in query Q_2 , OPT* outperforms *BthenR_Max* and *BthenR_Min* significantly by more than three and one order of magnitude respectively, as constraint expression is less selective in this query.

6.3 Controlled Queries on Synthetic Data

To evaluate our framework in a more extensive setting, we now evaluate OPT* using controlled queries on synthetic data.

Varying Data Distributions and Query Values: In this set of experiments, we evaluate over a wide range of datasets and queries, varying data distributions and query values of three representative query forms.

Datasets: Our synthetic dataset contains three randomly generated datasets from different distributions – *uniform*, *gaussian* and *log-variationalnormal*, which we believe are representative for many real world applications. Each dataset represents a relation of four attributes (x_1, x_2, x_3, x_4) , which are generated using the same distribution with pre-set parameters– In particular, we set *uniform* to generate attributes x_1, \dots, x_4 in ranges of $(0, 100)$, $(0, 1000)$, $(0, 5000)$ and $(0, 10000)$ respectively. We generate *gaussian* and *logvariationalnormal* distributions with means 100, 1000, 5000, and 10000 for x_1, x_2, x_3 , and x_4 respectively and variations proportionally to means.

Queries: We generate three sets of queries of three representative query forms. Each form can be configured with query values, which we randomly vary to generate 33 queries for each of the three datasets, to yield a total of about 300 queries by running over three synthetic datasets. For this set of parameterized queries, we use template-based approach, as discussed in Section 4, to compute the local optima for function optimization.

- \mathcal{F}_1 : The first set of queries is linear average, which are widely used in top- k algorithms. The queries are generated using:

$$\mathcal{O}(x_1, \dots, x_4) = \sum_{i=1, \dots, 4} w_i \times x_i; \mathcal{B} : x_2 \subseteq r_1 \vee x_3 \subseteq r_2$$

where parameter w_i are randomly generated from $(0,1)$, while r_1 and r_2 are randomly generated value ranges for x_1 and x_2 respectively.

- \mathcal{F}_2 : This set of queries in a form of nearest neighbor queries simulates two scenarios. First, it simulates typical house search queries where a user specifies her desired house as a query point, and wants to find “similar” houses using distance function \mathcal{O} . Second, this set of queries simulates *KNN* queries to find the closest neighbors of a query point defined by Euclidian distance function. Note that we use four individual indices on the four dimensions, unlike *KNN* which uses multidimensional spatial indices such as R-tree. The queries are generated using:

$$\mathcal{O}(x_1, x_2) = c_1 \times (x_1 - a)^n + (1 - c_1) \times (x_2 - b)^n$$

$$\mathcal{B} : x_1 \subseteq r_1 \wedge x_2 \subseteq r_2$$

where parameter c_1 is randomly generated from $(0,1)$, a from $(0,100)$, b from $(0,10000)$, and $n = 1, \dots, 10$, while r_1 and r_2 are randomly generated value ranges for x_1 and x_2 respectively.

- \mathcal{F}_3 : This set of queries in a form of join queries simulates queries involving query attributes from multiple relations in goal functions. We use two datasets as two tables. Let $S(a_1, a_2, a_3, a_4)$ and $T(b_1, b_2, b_3, b_4)$ denote the schemas of the two tables. The queries are generated using the formula

$$\mathcal{O}(s, t) = s.a_1 * t.b_1 + s.a_2 * t.b_2 : [s.a_2 = t.b_2]$$

$$\mathcal{B} : s.a_1 \subseteq r_1 \wedge t.a_1 \subseteq r_2$$

where r_1 and r_2 are randomly generated value ranges for x_1 and x_2 respectively.

Results: Figure 6(b) to (d) show the average number of page accesses of three sets of queries against three synthetic datasets. In particular, Figure 6(b) shows the average page accesses for 33 linear average queries over the three datasets. Observe that, in all queries, *RthenB* exhausts the entire database tuples. The reason is that for all three datasets, we generate the first attribute x_1 within a

small range of 0 to 100, and therefore there are only a small number of keys in the index and each key corresponds to many tuples. As a result, when we generate sorted accesses using such an index, *TA* algorithm quickly exhausts all tuples from the index. This suggests that *TA* can perform better by focusing on more effective sorted accesses than by retrieving from multiple sorted lists in parallel. Observe also that, in all queries and datasets, OPT* performs better or closely to *BthenR_Min*. In particular, in Figure 6(d) over gaussian dataset, OPT* outperforms *BthenR_Max* and *BthenR_Min* by more than three and one orders of magnitude respectively. The performance contrasts significantly in \mathcal{F}_3 , as constraint expressions generated on a_1 with the smallest range incur many duplicates and thus low boolean join selectivity.

As a remark, we discuss the issue of random vs. sequential I/O costs. Although our results measure the number of page accesses without explicitly accounting the cost of random vs. sequential I/O, with simple extrapolation, we claim that OPT* outperforms other baseline approaches in most common settings. The sequential scan can be used when 1) DBMS chooses table scan as the access method in the baseline approach (*BthenR* in particular) and 2) DBMS chooses index scan, and the index chosen is clustered.

To begin with, when table scan is used, it will sequentially access all pages, which is 4K in our experiment. When factoring in the random vs. sequential cost ratio, which is 10 to 12 in a common disk configuration (see *PCGuide.com*), OPT* will be more efficient, if the gain on the number of page accesses is more than 12. Using the results in Figure 6, OPT* accesses 13 to 80 times less pages (and 100+ times for join queries) than table scan, thus more efficient. This performance gap will enlarge when the table size scales up (or when more tables are joined), since OPT* scales sublinearly (*i.e.*, $\log n$ as in B-tree search) with the table size, while table scan only linearly.

Further, when index scan is used (which results in Figure 6), the execution cost will depend on whether the index is clustered. For an unclustered index, the most common situation, as in *BthenR_Max*, sequential scan is generally not possible, and therefore the significant performance gain shown in Figure 6 still holds. For a clustered index with sequential scan, the *BthenR* approach will outperform OPT* for some scenarios. However, we stress that our goal is to have OPT* as a viable alternative scheme, which will outperform baseline approaches in many scenarios, and thus a good choice for the query optimizer.

Varying Boolean and Ranking Selectivity Ratio: We finally also study the impact of boolean and ranking selectivity to the performance of different approaches. Due to space limitation, we omit the experimental details but only report our observations. We observe that the performance of *BthenR* increases when the constraint expression becomes increasingly selective and the performance of *RthenB* increases when the ranking expression becomes increasingly selective. Again, OPT* generally performs the best, by outperforming baseline approaches by an order of magnitude.

7. RELATED WORK

As mentioned in Section 1, many existing query processing algorithms can be considered as a special instance of OPT* framework. We examine and relate with those algorithms categorized by application scenarios.

The first category is middleware based top- k algorithms [5, 6, 7, 15, 17, 13, 3, 12, 2, 1] which act as middleware to combine the top results from the individual subsystems. The individual subsystems are responsible for evaluating partial score functions and generating the rankings along a specific dimension.

The second category is index-based spatial query algorithms (*e.g.*, [8,

category	\mathcal{G}	\mathcal{I}	traversal
MT	monotonic $\mathcal{G} = \mathcal{O}$	one idx (<i>KNN</i>)	interleaf
IS	$\mathcal{G} = \mathcal{O}$		hierarchical
DT	monotonic \mathcal{O}		hierarchical

Figure 7: Problem-specific assumptions

19]) which navigate attribute indices to process spatial queries, *e.g.*, *KNN* or spatial join queries. In particular, they navigate index structures top-down, expanding the children nodes with the minimal estimated distance to the query point (*KNN*) or the joinable pair (spatial joins) until we can safely prune out the unvisited regions.

The third category is database internal top-*k* algorithms [9, 14, 4, 11] supporting ranking inside the database system and tightly coupling with the Boolean query engine. Recent works on [9, 11] ranking-aware query optimization fall into this category. This approach augments the query optimizer to consider the ranking as an interesting order and therefore those rank-aware query plans are considered during plan enumeration. The work mainly focuses on how to incorporate ranking into Boolean query processing to generate query plans aware of the existence of ranking operator.

In contrast, our work, by encoding query answering as a generic *A** search problem, generalizes existing works summarized above to: first, support arbitrary goal function \mathcal{G} ; second, over multiple indices; and third, using both hierarchical and interleaf traversals. To begin with, *KNN* query is a special case of *k*-constrained optimization query with goal function contains only ranking expression, which is the Euclidian distance from a given query point. *KNN* algorithms follow top-down search approach to traverse indices and look for query answers. Further, existing works on spatial distance join, such as [8], can also cast as a top-down search instance of our framework, where multiple spatial indices are jointly used to support the search that aims to minimize a given distance function as the goal function \mathcal{G} . Third, while not intended, a representative top-*k* algorithm *TA* [6] also falls into our *OPT** search framework. *TA* assumes a monotonic goal function without boolean constraints, and thus starts the search from a unique local optima point over the value space. Further, as *TA* builds on sorted accesses, *i.e.*, traversing from a region to neighboring regions by value locality, which is essentially following interleaf traversals.

Figure 7 summarizes the problem-specific assumptions of the three lines of existing works discussed above, *i.e.*, middleware top-*k*, index-based spatial, and DB-internal top-*k* (which we denote MT, IS, and DT respectively), on \mathcal{G} , the indices used, and their traversals. Observe that all three lines of works build upon problem-specific assumptions on \mathcal{G} and use either hierarchical or interleaf traversal. Further, *KNN* algorithms make an additional assumption to traverse a single index. In contrast, our framework enables to support general *k*-constrained optimization queries by eliminating such problem-specific assumptions and generally supporting arbitrary \mathcal{G} , combining multiple indices, using both hierarchical and interleaf traversals.

Finally, people have studied constrained logic programming or CLP for short [10] to solve combinatorial optimization problems mostly over discrete (*e.g.*, integer) domains. Like our framework, the evaluation approaches generally fall into two paradigms— top-down and bottom-up evaluation. In this paper, we are addressing constrained optimization in a more specific setting, *i.e.*, relations in databases. Although we share the similar search paradigms, the search spaces are constructed and represented differently. The search space in our problem is constructed from indices over irregular tuple space and represents the candidate solutions to the problem. The search space in general CLP is constructed from the logic rules, and represents the transitions of logic inferences towards solutions. As the value space of CLP is regularly structured,

the exploration of this space is thus mechanic instead of assisted by index structures as in our problem.

8. CONCLUSION

This paper presents *OPT** framework for answering *k*-constrained optimization queries. The framework abstracts *k*-constrained optimization queries as a discrete space search problem over existing access methods, *i.e.*, indices. Such an abstraction imposes two perspectives for query answering— discrete space search perspective induced by index, and function optimization perspective induced by continuous function optimization. Combining both techniques, we develop our *OPT** framework by constructing a “static” state space over indices and dynamically configuring the space with function optimization. Upon this space, *OPT** employs *A** search algorithm to achieve the completeness and optimality. The experimental results show that *OPT** framework improve the performance of baseline approaches in order of magnitude.

9. REFERENCES

- [1] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.
- [2] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.
- [3] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *SIGMOD*, 1996.
- [4] P. Ciaccia and M. Patella. The M^2 -tree: Processing complex multi-feature queries with just one index. In *Proceedings of the First DELOS Network of Excellence Workshop on “Information Seeking, Searching and Querying in Digital Libraries”*, 2000.
- [5] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.
- [6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [7] U. Güntzer, W. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.
- [8] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, 1998.
- [9] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, 2004.
- [10] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [11] C. Li, K. Chang, I. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.
- [12] C. S. Li, L. D. Bergman, V. Castelli, and J. R. Smith. Spire: A progressive content-based spatial image retrieval engine. In *SIGMOD*, 2000.
- [13] A. Natsev, Y. C. Chang, J. R. Smith, C. S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.
- [14] A. Natsev, G. Y. C. Fuh, W. Chen, C.-H. Chiu, and J. S. Vitter. Aggregate predicate support in dbms. In *Proceedings of the Thirteenth Australasian Database Conference*, 2002.
- [15] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.
- [16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. CAMBRIDGE UNIVERSITY PRESS, 2nd Edition, 1992.
- [17] M. V. Ramakrishna, S. Nepal, and P. K. Srivastava. A heuristic for combining fuzzy results in multimedia databases. In *Proceedings of the Thirteenth Australasian Database Conference*, 2002.
- [18] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd Edition, 2002.
- [19] H. Shin, B. Moon, and S. Lee. Adaptive and incremental processing for distance join queries. In *SIGMOD*, 2003.