LARGE SCALE INFORMATION INTEGRATION ON THE WEB:
FINDING, UNDERSTANDING AND QUERYING WEB DATABASES

BY

ZHEN ZHANG
B.S., Peking University, 1998
M.S. Peking University, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois
Abstract

The Web has been rapidly “deepened” by myriad searchable databases online, where data are hidden behind query interfaces. Guarding data behind them, such query interfaces are the “entrances” or “doors” to the deep Web. To open this door to the deep Web, we have been building the *MetaQuerier* system— for both exploring (to find) and integrating (to query) databases on the Web through their query interfaces. To find Web databases, we need to provide search functionalities that dynamically discover databases relevant to user’s information needs. To query those Web databases, we need to “understand” what a query interface says— i.e., what *query capabilities* a source supports through its interface, in terms of specifiable conditions. Further, to help users query “alternative” sources, we need to mediate heterogeneous query capabilities across different sources discovered on-the-fly. Finally, to process queries submitted to a database, we need to design efficient query processing techniques. To address those challenges, this thesis presents several key components in *MetaQuerier* system: First, a *search facility* searches for useful databases by their schemas; Second, *form extractor* extracts query capabilities of databases by applying a best-effort parsing approach based on hidden syntax; Third, *form assistant* translates queries across pairs of interfaces on-the-fly by deploying a *light-weight, domain-based* translation framework. Fourth, OPT* framework processes ranked queries by a $k$ constraint optimization problem. We evaluate our techniques upon real databases on the Web. The experiment results show the promise of our system.
# Table of Contents

List of Figures ................................................. viii

List of Figures ................................................. x

Chapter 1  Introduction .......................................... 1

1.1  *MetaQuerier*: Finding and Integrating Web Databases ............... 1

1.2  Technical Challenges and Our Contributions .......................... 2

Chapter 2  Finding Web Databases: Searching by Schemas ................... 8

2.1  Problem Overview ............................................ 8

2.2  Motivation .................................................. 10

2.2.1  Observations: Page-Oriented Bookmark Search Is Not Sufficient .... 12

2.2.2  Implications: Source-Oriented Generalized Search Is Needed ........ 15

2.3  Associativity Search by Schemas ................................ 16

2.3.1  Schema-Based Source Modeling ................................ 17

2.3.2  Propagation-Based Associativity Search .......................... 19

2.4  Realization of Online Associativity Search .......................... 23

2.4.1  Basic Computation Framework .................................. 24

2.4.2  Matrix Optimization: Speed Up Convergence ...................... 25

2.4.3  Prioritize Convergence: Probabilistic Incremental Convergence .... 29

2.5  Experimental Evaluation ....................................... 34

2.5.1  Evaluation Environment ..................................... 34

2.5.2  Accuracy of Search Results ................................... 34

2.5.3  Utility of Probabilistic Incremental Convergence ................. 36

2.5.4  Speed Up of Spatial and Temporal Optimizations .................. 39
### Chapter 3: Understanding Web Databases: Extracting Query Capabilities From Query Interfaces

#### 3.1 Problem Overview

#### 3.2 Towards parsing paradigm

- **3.2.1 Observations: Concerted Structure**
- **3.2.2 Hypothesis: Hidden Syntax**
- **3.2.3 Challenges: Hypothetical Syntax**
- **3.2.4 Solutions: 2P Grammar & Best-effort Parser**

#### 3.3 2P Grammar

- **3.3.1 Productions: Capturing Patterns**
- **3.3.2 Preferences: Capturing Precedence**

#### 3.4 Best-effort parser

- **3.4.1 Overview**
- **3.4.2 Just-in-time Pruning**
- **3.4.3 Partial Tree Maximization**

#### 3.5 Experiments

- **3.5.1 Global Study**
- **3.5.2 Local Study**

#### 3.6 Related work

### Chapter 4: Querying Web Databases: Translating Queries Across Heterogenous Databases

#### 4.1 Problem Overview

#### 4.2 Formulation of Query Translation Problem

- **4.2.1 Query Model**
- **4.2.2 Semantic Closeness**

#### 4.3 System Architecture

#### 4.4 Predicate Mapping: The Motivation

- **4.4.1 A Motivating Survey**
Chapter 4  Type-based Search-driven Predicate Mapping .......................... 86
4.5 Predicate Mapping: The Solution ................................................... 87
4.5.1 Overview of Type Handler ......................................................... 88
4.5.2 Built-in Handlers ................................................................. 90
4.5.3 Domain Specific Type Handler ................................................. 93
4.6 Implementation: Form Assistant Toolkit ....................................... 94
4.7 Experiment ............................................................................... 96
4.8 Related Work ..................................................................... 100

Chapter 5  Processing Queries In Databases: Evaluating Ranked Queries By K-Constrained Optimization ......................................................... 102
5.1 Problem Overview ............................................................... 102
5.2 Motivation ........................................................................ 105
5.2.1 Query Model .............................................................. 105
5.2.2 Query Mechanism ......................................................... 107
5.2.3 Challenges .................................................................. 109
5.3 Index-induced State Space: A*-Driven Construction .................. 111
5.3.1 Mapping the Space: State and Transition ......................... 112
5.3.2 Where to Head to: Goal State ............................................ 115
5.4 Goal-induced Space: Optimization driven Configuration ........... 118
5.4.1 Measuring the Landscape: Heuristics .................................. 119
5.4.2 Where to Start: Initial States ............................................. 121
5.5 OPT* Search .................................................................... 123
5.5.1 Skeleton of OPT* ......................................................... 123
5.5.2 Optimality of OPT* ....................................................... 124
5.6 Experiments .................................................................... 127
5.6.1 Experiment Settings .......................................................... 127
5.6.2 Benchmark Queries on Real Data ...................................... 129
5.6.3 Controlled Queries on Synthetic Data ................................. 130
List of Figures

1.1 Databases on the Web. .................................................. 2
1.2 *MetaQuerier*: Finding and querying Web databases. .................. 3
1.3 Architecture of *MetaQuerier* system. .................................. 4

2.1 Queries used in Google survey. ....................................... 11
2.2 Accuracy of Google search results. ..................................... 12
2.3 Stableness of Google search results. .................................... 12
2.4 Current search engines vs. our proposal. ............................... 13
2.5 Book schemas of TPC-W and *Amazon* .................................. 16
2.6 A schema repository. .................................................... 18
2.7 Database-attribute occurrences ........................................ 19
2.8 Database-relation feature occurrences .................................. 20
2.9 Schema graph. ........................................................... 21
2.10 Spatial and temporal optimization. .................................... 25
2.11 Probabilistic incremental convergence. ................................ 30
2.12 Distribution of $D$ over sources. ..................................... 33
2.13 Top results for illustrative queries. ................................... 35
2.14 Accuracy evaluation showing high precision. .......................... 35
2.15 Overlapping analysis showing high stability. .......................... 36
2.16 Propagation vs. no-propagation ....................................... 36
2.17 Correlation of actual convergence and predicted confidence for $Q_1$ and $Q_2$. .......................... 37
2.18 Average errors and response times .................................... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.19 Distribution of errors and response times</td>
<td>38</td>
</tr>
<tr>
<td>2.20 Speed up of spatial optimization</td>
<td>40</td>
</tr>
<tr>
<td>2.21 Scalability w.r.t. dataset sizes</td>
<td>40</td>
</tr>
<tr>
<td>2.22 Speed up of temporal optimization</td>
<td>41</td>
</tr>
<tr>
<td>3.1 Query interfaces examples</td>
<td>46</td>
</tr>
<tr>
<td>3.2 Examples of condition patterns</td>
<td>47</td>
</tr>
<tr>
<td>3.3 Hidden-syntax hypothesis</td>
<td>47</td>
</tr>
<tr>
<td>3.4 Query vocabulary: condition patterns as building blocks for query interfaces</td>
<td>48</td>
</tr>
<tr>
<td>3.5 Form extractor for Web query interfaces</td>
<td>52</td>
</tr>
<tr>
<td>3.6 Tokens $T$ in fragment of interface $Q_{am}$</td>
<td>54</td>
</tr>
<tr>
<td>3.7 Productions of the 2P grammar</td>
<td>56</td>
</tr>
<tr>
<td>3.8 Two interpretations for text $s_1$</td>
<td>57</td>
</tr>
<tr>
<td>3.9 Two interpretations for radio button list</td>
<td>58</td>
</tr>
<tr>
<td>3.10 Two parse trees for query interface $Q_1$</td>
<td>59</td>
</tr>
<tr>
<td>3.11 Fix-point parsing process</td>
<td>61</td>
</tr>
<tr>
<td>3.12 The 2P schedule graph for grammar $G$</td>
<td>64</td>
</tr>
<tr>
<td>3.13 Transformation of an $r$-edge</td>
<td>65</td>
</tr>
<tr>
<td>3.14 Partial trees for interface $Q_{aa}$ fragment</td>
<td>67</td>
</tr>
<tr>
<td>3.15 Source distribution over precision and recall of 4 datasets</td>
<td>69</td>
</tr>
<tr>
<td>3.16 Precision and recall over 4 datasets</td>
<td>69</td>
</tr>
<tr>
<td>3.17 Convergence of precision and recall over sample size</td>
<td>72</td>
</tr>
<tr>
<td>3.18 Increase of grammar size over sample size</td>
<td>72</td>
</tr>
<tr>
<td>4.1 Form assistant: A translation example</td>
<td>75</td>
</tr>
<tr>
<td>4.2 Query model and example instantiations for form $T$</td>
<td>78</td>
</tr>
<tr>
<td>4.3 System architecture of the form assistant</td>
<td>82</td>
</tr>
<tr>
<td>4.4 Example mapping rules of source $S$ and target $T$</td>
<td>84</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Recently, the Web has been rapidly deepened with the prevalence of databases online. As Figure 1.1 conceptually illustrates, on this so-called “deep Web,” numerous online databases provide dynamic query-based data access through their query interfaces, instead of static URL links. A July 2000 study [?] estimated 43,000-96,000 such search sites (and 550 billion content pages) on the Web. Our recent survey [?] in April 2004 estimated 450,000 online databases. As current crawlers cannot effectively query databases, such data are invisible to search engines, and thus remain largely hidden from users.

However, while there are myriad useful databases online, users often have difficulty in first finding the right sources and then querying over them. Consider user Amy, who is moving to a new town. To start with, different queries need different sources to answer: Where can she look for real estate listings? (e.g., realtor.com.) Looking for a new car? (cars.com.) Looking for a job? (monster.com.) Further, different sources support different query capabilities. After source hunting, Amy must then learn the grueling details of querying each source.

1.1 MetaQuerier: Finding and Integrating Web Databases

To enable effective access to databases on the Web, we have been building a “metaquerying” system, the MetaQuerier (metaquerier.cs.uiuc.edu), as Figure 1.2 shows. Our goal is two fold. First, to make the deep Web systematically accessible, it will help users find online databases useful for their queries. In Amy’s case, we will help her to find sources such as realtor.com or apartment.com for house listings, and monster.com or careerbuilder.com for job search. Second, to make the deep Web uniformly usable, it will help users query
online databases. In Amy’s case, we will query realtor.com and apartment.com, and combine the results to her in a unified format.

Toward this goal, we have designed the system architecture, developed several key components, and started system integration. As the architecture of Web integration system in general and MetaQuerier in specificity, Figure 1.3 shows the basic components of the system. At the bottom is a source repository which stores deep Web sources we collected online. It has three components: a crawler to collect Web databases online, an indexer to build indices of Web databases for retrieval and a modeler to describe what a database is about and how to query it. Upon the repository are two functional components: explorer to find Web databases and integrator to query those databases through their query interfaces. In the explorer, we build a search facility that allows users to search for Web databases in the repository. The integrator has two components: a query translator to map queries across different databases, and result wrapper to extract data records from query results of individual databases and compile them together. The two function modules are interactive: a user may start with search for relevant databases, and then an integrator further queries those databases returned by the explorer.

1.2 Technical Challenges and Our Contributions

Among many challenges in developing the system, this thesis focuses on four components— a search facility to finding entrances to Web databases, a source modeler to extract query capabilities of Web databases, a query translator to translate queries across heterogeneous Web databases, and a query processor to evaluate ranked queries inside Web databases.
Finding Web databases: A search facility. With massive databases online, the very first challenge in exploring the deep Web is to help users find useful Web databases. Specifically, we need to route users to the appropriate “entrances,” i.e., interface pages of Web databases to query. To satisfy users’ information needs, there are often many alternative sources, e.g., Amazon.com or BN.com for book purchase, and AA.com or Delta.com for flight reservation. To find such databases, currently users have to refer to search engines, such as Google. However, as those search engines are designed specifically for finding pages on the surface Web, they are not suitable for finding Web databases.

The reasons for such insufficiency are two fold. First, current search engines lack of an appropriate source model for describing Web databases. They adapt page-oriented models to describe individual pages, rather than source-oriented models using intuitive features inherent to the databases. Therefore, users have to formulate queries in terms of describing a page rather than describing the type of objects they want. Second, current search engines lack of a generalized search mechanism. They perform keyword matching against pages to return pages that directly contain the query keywords, which is too restricted to capture users’ intention and to discover useful databases.

Tackling these challenges, this thesis proposes to search databases by their schemas. Specifically, we propose a source-oriented modeling to model Web databases using their schemas. As a schema captures intuitive features describing the type of objects stored in the database, a schema gives a natural and intuitive way for searching databases. Further, upon the schema model, we propose to build an associativity search...
facility, which captures the relevance of sources by how they are associated by their schemas. Aiming at providing online search, we further study optimization techniques that speed up the computation to ensure fast response time.

**Understanding Web databases: A form extractor.** Finding the entrance, i.e., query interface, to a databases is only the first step towards integration. To enable querying over those sources, we further need to “understand” what a query interface says—i.e., what query capabilities a source supports through its interface, in terms of specifiable conditions. For instance, amazon.com in Figure 1.1 supports a set of five conditions (on author, title, ... , publisher). Those query conditions establish the semantic model underlying the Web query interface.

Such extraction is challenging, since query forms are created autonomously. This task seems to be rather “heuristic” in nature: with no clear criteria but only a few fuzzy heuristics—as well as exceptions. For instance, to associate elements with corresponding labels indicating their semantic meanings, a natural heuristics is to apply pairwise association of elements with the texts closest to them. However, as Chapter 3 will illustrate, such association may be n-ary, and therefore the heuristics may fail with exceptions.

To address the problem of form understanding, our approach builds on the observation that, across
myriad sources, query forms seem to reveal some “concerted structure” by sharing a small set of condition patterns as building blocks. To capture this insight, we hypothesize the existence of a hidden syntax behind Web query interfaces, across different sources. This hypothesis effectively transforms the problem into a new paradigm: We view query interfaces as a visual language, whose composition conforms to a hidden, i.e., non-prescribed, grammar. As an inverse process, their semantic understanding is thus a parsing problem.

**Querying Web databases: A query translator:** As query capabilities extracted from different Web databases are heterogeneous, to integrate those sources, we need to mediate such heterogeneities across different databases. The core component of such mediation is a dynamic query translator, which translates users’ queries between dynamically selected query forms. In our problem setting, we focus on translating queries across two sources of the same domain, i.e., we assume the sources have the same underlying schema. However, the query capabilities, as the presentation of underlying schemas, differ across sources at essentially three levels–attribute level, predicate level and query level, as Chapter 4 will elaborate. Therefore, query translation is essentially to mediate these three levels of heterogeneities. While existing works mainly focus on isolated subproblems of translation (e.g., schema matching reconciles attribute level heterogeneity, query rewriting resolves query level heterogeneity), our work aims at building a complete dynamic query translator, which handles three levels heterogeneity all together, has thus not been extensively investigated.

Such dynamic translation is challenging: As the applications mandate, our query translator translates queries on-the-fly, i.e., dealing with new sources without pre-configured source-specific knowledge. Therefore, the dynamic translator should have two properties: First, source-generality: We require the built-in translation techniques can generally cope with new or “unseen” sources. Second, domain-portability: We require the translator can be easily customized with domain-specific knowledge and thus deployed for new domains.

As a realization of such a query translator, this thesis develops a light-weight domain-based form assistant. To realize source-generality, we propose to use data types as scopes for encoding translation knowledge. As types are shared by sources in the same domain and even across domains, our form assistant can generally handle translations without pre-configured knowledge specific to a source. Further, to realize domain-portability, we design our architecture with both built-in knowledge handling domain-independent
translation as well as domain-customizable knowledge handling domain-specific translation. Such an architecture is domain-based because it deals with alternative sources in the same domain with a manageable size of domain-specific knowledge. It is also light-weight because customizing to a new domain only needs a small amount of efforts to encode domain-specific knowledge.

**Processing queries in databases: A ranked query evaluator.** Upon acceptance of a translated query, databases need to process the query to return qualified answers. The large data volume and diversified users query needs render traditional boolean based query semantics insufficient, where a tuple in a database either satisfies the query or not. Such boolean queries either retrieve too few results or too many. Therefore, it is important to incorporate ranking semantics into queries so that the results are returned in a ranked fashion according to how well they match users preferences.

While top-k query processing techniques have been extensively studied, those techniques are lacking a global search mechanism that optimizes both boolean and ranking conditions in a unified framework. On one hand, some existing work [?, ?, ?, ?, ?, ?, ?] optimizes boolean and ranking components piece by piece—they assume that either the boolean component is processed first then the ranking, or the ranking component first then the boolean, and therefore focus only on optimizing the ranking component. On the other hand, some other work [?] treats boolean and ranking components as a global “goal” function, and tries to optimize this goal function by interleaving boolean and ranking components. However, those techniques require the goal function to be monotonic. With boolean components present in the query, this monotonicity assumption hardly holds. For instance, suppose our query looking for a house has a boolean condition $10,000 < price < 20,000$ and a ranking criteria $1/size$. A house $a$ with price $15,000$ will have a higher score than a house $b$ with price $30,000$ and a house $c$ with price $5,000$ regardless of their sizes. Therefore, the overall goal function is not monotonic due to the existence of the boolean condition, although the ranking criteria itself can be monotonic.

Aiming at supporting a global function with boolean and ranking components, in this thesis, we propose to process such a query by $k$-constrained optimization of a goal function $\mathcal{G}$ over the database $\mathcal{D}$. From the goal function $\mathcal{G}$’s perspective, query processing is function optimization problem, that is, to find data points that maximize $\mathcal{G}$. However, unlike continuous function optimization, query processing is to search
in a discrete database $\mathcal{D}$, where tuples can only be accessed through certain access methods. Specifically, to avoid a brute-force scan over an entire database, indices are widely used in a database to locate tuples. Therefore, effectively, indices provide us a discrete state search space. Combining the two perspectives, we propose our OPT* framework, that is, to search discrete state space defined by index, under the guidance of function optimization.

In summary, this thesis makes the following contributions:

• **Problem**: the thesis studies several key problems in building Web integration systems— to find, understand, query Web databases, and to process queries inside database engines.

• **System**: the thesis integrates the isolated components and develops a complete system Dewex for searching and querying Web databases.

• **Techniques**: the thesis addresses the technical challenges in building the system— to find a source, we build an associativity search facility; to understand a source, we develop a hidden-syntax based best-effort parsing framework for query capability extraction; to query a source, we develop a light-weight domain-based translation framework; and to process queries of a source, we propose an OPT* framework to evaluate ranked queries as $K$-constraint optimization problem.

• **Evaluation**: the thesis evaluates the system performance using real Web databases. The results show the effectiveness of our approaches.

The rest of the thesis is organized as follows: Chapter 2 studies the schema-based search facility for finding Web databases, Chapter 3 presents the hidden-syntax based parsing framework for understanding Web databases and Chapter 4 presents light-weight domain-based form assistant for translating queries across different databases. Chapter 5 discusses OPT* framework for processing ranked query inside database engines. Chapter 6 reviews the related work and Chapter 7 concludes the thesis.
Chapter 2

Finding Web Databases: Searching by Schemas

2.1 Problem Overview

With databases prevailing on the Web, we are facing the very first challenge in exploring the deep Web, that is, to find useful databases for users to query. Specifically, we need to route users to appropriate “entrances,” i.e., query interfaces, of the databases, through which queries can be issued.

To fulfill users information needs, there are often many alternative sources, e.g., aa.com or expediam.com for flight reservation, and amazon.com or bn.com for purchasing books. To find those databases, current users have to resort to search engines such as Google or Yahoo, which are designed specifically for finding interlinked Web pages. Are they sufficient for finding Web databases? Our survey answers the question: current search engines cannot provide an effective way for finding Web databases.

The reasons for such insufficiency are two fold. First, current search engines lack of an appropriate source model for describing Web databases. They adapt page-oriented models to describe individual pages, rather than source-oriented models using intuitive features inherent to the sources. Therefore, users have to formulate queries in terms of describing a page rather than describing the type of objects they want. Second, current search engines lack of a generalized search mechanism. They perform keyword matching against pages to return pages that directly contain the query keywords. Therefore, searching for “cheap hotels” cannot reach expedia.com although it is very much like hotels.com which does contain query terms in the interface page.

Tackling those challenges, we propose to build an online associativity search facility that searches alter-
native Web databases using their schemas to fulfill users information needs. Specifically, such a facility has the following features:

**Source modeling: schema-based.** To begin with, we adapt source-oriented modeling to model Web databases by their *schemas*. Like schemas in relational databases, Web database schemas describe the type of objects stored in the databases. Such description includes the *relation name* to describe the name of objects and *attributes* to describe properties of objects. Different from those sophisticated schema models such as relational schema or XML schema model which are often manually crafted to capture complex (nested) object structures and various semantics constraints, our Web database schemas are obtained automatically from Web pages presented to end users. Therefore, Web database schemas are much more simplified and intuitive. It is simplified in that the schema often exposes simple structural view of the objects by hiding many internal attributes and various integrity constraints. It is intuitive in that the schema usually uses intuitive names to indicate the nature of the objects. Therefore, in our work, we propose a schema model with relation name and attributes, and generalization of this model to handle more sophisticated structures is an interesting topic for future study. With schemas capturing intuitive features of Web databases, we therefore propose to search databases by schemas.

**Associativity measure: locality-based.** The schema-based source modeling naturally exhibits a co-clustering phenomenon among databases and schemas. *Similar databases tend to share common schema features, and similar schema features tend to occur in similar databases.* Therefore, databases and schemas form occurrence localities. Such locality suggests that relevance of databases can be nicely captured by how they are associated by database-schema occurrences. We therefore propose to use associativity as the relevance measurement, and apply the *mutual reinforcement principle*, which Section 2.3.2 will discuss formally, to define the associativity capturing the co-clustering phenomenon.

**Search mechanism: propagation-based.** The formulation of the associativity by mutual reinforcement of source and schema relevance enables a generalized search mechanism. The associativity is computed by iteratively propagating relevance between databases and schemas starting from an initial query. Such an iterative propagation mechanism generalizes the one-step propagation of vector space model [?] commonly
practiced in current search engines. Therefore, databases directly or indirectly matching the query will be reached through propagation.

**Online computation: optimization-enabled.** As we aim at providing an online search system, we propose optimization techniques to ensure fast response time. As our dataset naturally exhibits localities, we take advantage of the data locality to effectively bound the scope of propagation and also to accelerate the “stride” of propagation. Further, as a user’s query often falls into a locality, this leads to highly skewed relevance score and computation time distribution for different sources. We thus take advantage of such skewness to aggressively generate high ranked results, which are guaranteed to be correct, i.e., the same as the theoretical answer defined in Section 2.3.2, with certain confidence.

In the rest of chapter, we will discuss our schema-based search facility in detail. Specifically, Section 2.2 studies and analyzes the insufficiency of current search engines by a formal survey. Tackling those issues, Section 2.3 proposes our solution for searching databases by schemas. We then discuss the realization of the solution for online search in Section 2.4. We validate the effectiveness of our approach in Section 2.5. We discuss related work in Section 2.6.

### 2.2 Motivation

Current search engines view the Web as a collection of hyperlinked pages, where each page is modeled with keywords appearing in the page. The search matches query terms with pages, and only those pages that contain those query terms are possibly returned. While such techniques are effective for finding pages on the surface Web, are they also suitable for finding Web databases?

To understand the performance of general search engines for finding Web databases, we conducted a survey using a popular search engine Google. The survey uses a set of queries over 6 domains—airline, hotels, used cars, jobs, books and movies. We choose those domains, because they represent different behavior in terms of intuitively how easily users can find sources of those domains. For airlines and hotels domain, as most sources support only query-based access, it is usually easy for users to reach interface pages of sources in those domains. For books and movies domain, as many sources support primarily link based access with query based access as auxiliary mechanism, to find their query interfaces often involves
<table>
<thead>
<tr>
<th>Domain</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airfares</td>
<td>airline tickets, cheap airline tickets, discount airline tickets, Last minute cheap tickets, All airline tickets</td>
</tr>
<tr>
<td>Hotels</td>
<td>hotel search, hotel discounts, cheap hotel room, hotel websites, hotel finder</td>
</tr>
<tr>
<td>Used cars</td>
<td>used car search, used car, used car finder, used vehicle, used car sites</td>
</tr>
<tr>
<td>Jobs</td>
<td>job search, find a job, job finder, job listings, job boards</td>
</tr>
<tr>
<td>Books</td>
<td>bookstore, used books, search for books, buy books, online book purchase</td>
</tr>
<tr>
<td>Movies</td>
<td>movie times, movie tickets, movie theaters, movie soundtrack, movie listings</td>
</tr>
</tbody>
</table>

Figure 2.1: Queries used in Google survey.

navigation within the sources, which makes it hard to reach deep Web sources in those domains directly. For sources in used car and jobs domain, both link-based and query-based accesses are supported, and therefore the easiness of finding those deep Web sources lies in between. For each domain, we use five popular domain queries obtained from the MSN query log. These domain queries generally look for alternative sources of a domain (such as airlines), rather than a particular source in a user’s mind (such as aa.com). To obtain a set of domain queries, we scan the query log to rank queries in a domain based on frequency, and pick the top 5 domain queries. For instance, for book domain, we collect all queries containing keyword “book” or “books,” and find the top five domain queries are “bookstore,” “used books,” “buy books,” “online book purchase” and “search for books.” Figure 2.1 shows all queries used in the survey.

For each domain query, we manually examined the top 40 results obtained from Google. We, as human experts, classify the results into three categories: page relevant, site relevant and not relevant. A result is page relevant if the page represents a Web database. A result is site relevant if the page does not directly contain the query interface, but by navigating within the site, we can find the appropriate query interface. A result is not relevant if the site is either not relevant to users query or not a deep Web source. We use precision out of top 40 to measure the performance of the search engine.

Further, we also study how stable those search results are. Although users may not use exactly the same query terms, they often have the same intention. For instance, for finding sources about used cars search, users may say “used cars” or “used car listings.” We want to study how well search engine techniques capture
such intentions. To answer this question, we did overlap analysis over top 40 and top 100 query results for each query. In each domain, for each pair of queries, we analyze how much the results overlap. We use percentage of overlaps as the stableness measurement.

2.2.1 Observations: Page-Oriented Bookmark Search Is Not Sufficient

In this section, we will report and analyze the survey results to make some important observations that motivate the need for a new search engine for Web databases. Figure 2.2 and 2.3 show the results of performance study and stableness study respectively. Specifically, from the survey results, we can observe:

Observation 1: Page-level accuracy may be poor, but site-level accuracy is much improved. As Figure 2.2(a) shows, the precision of top 40 results on page relevance for many domains are not satisfactory. For books and movies domain, the precision is generally less than 50%. For airlines and hotels domain, the precision is generally good, around 70%, but still have exceptions as low as 50%. For cars and jobs domain,
the precision varies a lot, from as low as 20% to as high as 90%. However, when combining site relevance, as Figure 2.2(b) shows, the performance is improved. The performance on site relevance generally reaches 50% for all queries. For domains like airline and hotels, the precision generally achieves 80 – 90%.

**Observation 2: Precision varies among queries.** As Figure 2.2 shows, for most domains, the precision varies greatly according to queries. For instance, in searching for jobs, the precision can be as high as 94% when using query “job search,” while as low as 20% when using “job boards” or “job listings.”

**Observation 3: Results are not stable.** As Figure 2.3 shows, the overlapping between the search results are very small, all far below 50%. Although those queries are domain queries, intending for finding Web databases in the domain, the results returned by the search engine do not converge. For instance, the five queries used in hotels domain are “hotel search,” “cheap hotel rooms,” “hotel websites,” “hotel discounts” and “hotel finder.” While those queries are similar in nature, the overlapping results from those queries are less than 10%, which is very low.

The survey results suggest that the current search engines are not particularly suitable for finding sources on the deep Web. What are the reasons for the above observations?

**Reason 1: Search engines adapt page-oriented modeling with keywords.** Search engines use pages as their units of modeling and search, and therefore finding a database amounts to reaching its interface page. Figure 2.4(a) illustrates the search paradigm of current search engines. As indicted by the solid path,
starting from a user to a database, the user need to specify the keywords that appear in the interface page of the database. However, many queries reach pages surrounding the databases, as indicated by the dot lines between pages and databases, rather than interfaces themselves. Therefore, modeling sources only by its interface page, we can only obtain poor precision, as indicated by page-level accuracy in the survey. While taking into account other pages related to the source, we obtain much improved performance, as indicated by site-level accuracy.

Further, the high variance of query performance indicates that users had difficulties to design queries suitable for search engines— they need to think in terms of finding pages instead of finding sources. For instance, in order for finding job sources, users have to use queries such as “job search” that are likely to appear in the job search page, instead of just “jobs” or “job listings” which directly describe the types of objects that users are interested in. Therefore, the performance of search engines largely relies on the ability of users to formulate “good” queries.

**Reason 2: Search engines only perform simplified “bookmark” search.** The current search engines only search for pages that are “bookmarked” by query terms. Therefore, users need to issue “bookmark” queries using terms appearing in potential pages rather than describing the type of objects itself. When using different query terms even for the same intention, users reach different pages. As illustrated in Figure 2.4(a), while all sources are relevant to all users, not being able to make connections between sources, current search engines can only return those bookmarked pages. Therefore, user $U_1$ reaches nothing, $U_2, U_3$ reaches only $S_2$ and $S_3$ respectively. The results of different queries cover different subsets of relevant sources, and thus become very unstable.

Putting together our reasoning, current search engines cannot make robust connections between users and relevant sources, as Figure 2.4(a) depicts. First, the connection between users and queries is not intuitive. Users have to use keywords that likely appear in the interface page rather than just keywords describing the type of objects they are looking for. Second, the connection between source models and sources is restricted. Sources are only modeled by interface pages, not other related pages. Third, the connection among sources is absent. Not being able to connect databases, users can only reach a subset of relevant sources that containing the query terms. To get better coverage, users need to revise and manipulate queries.
with different keywords.

### 2.2.2 Implications: Source-Oriented Generalized Search Is Needed

The survey results from Google suggest that current search engines are not sufficient for finding Web databases due to their inability to connect users with sources in a robust way. Therefore, to address those problems, an effective search engine needs to satisfy the following requirements.

**Implication 1:** **We need a source model that captures intuitive features of objects in databases.**

Because the features are directly about objects themselves rather than arbitrary keywords, they are intuitive, and thus strengthen the connection between users and queries. Further, because the features reflect the commonality of sources providing the same type of objects, they are shared by sources, and thus make explicit the connection between sources. Finally, as such intuitive features may appear in other pages besides interfaces, this model enhances the connection between models and sources.

**Implication 2:** **We need a search mechanism that generalizes beyond “bookmark” queries.** To realize the connection between sources, we need a generalized search mechanism that searches beyond the bookmark query semantics. Instead of simply matching the query terms with source models, the search should be able to expand from those directly matched sources to other related ones as well. By making such connections, we give users more flexibility in formulating their queries, and we take the responsibilities to generalize from their queries. Further, such a connection between sources also provides a new type of search function, that is, to find similar sources e.g., expedia.com or united.com, to a given one e.g., aa.com. Such a search functionality would be very useful in many applications, such as comparison shopping for users to get best deals. While Google also provides “similar page” links in their search results, it is not clear how such similarity is captured, and it does not seem to capture the intention of finding similar sources.

In a nutshell, this paradigm depicted in Figure 2.4(b) makes the connection between users and sources more robust. Users search through intuitive features, which strengthen the connection between users and queries. Sources are modeled with features from multiple related pages, which improves the connection between sources and models. Queries are answered by a generalized search mechanism, which realizes the connection among sources.
2.3 Associativity Search by Schemas

Motivated by the implications discussed above, we now develop our framework for searching Web databases by their schemas. To realize this search, we need a proper mechanism for source modeling and a principled search mechanism.

For source modeling, we propose to use schemas to capture the intuitive features of the databases, which Section 2.3.1 will discuss. Unlike text databases or Web pages which are very diverse and hard to categorize into subjects or topics, databases are intended for storing objects, and thus can often be characterized by the type of objects served in the database. Further, because of the unstructured nature of general documents and pages, there are few other ways to describe them except their contents. However, databases store structured objects, and thus can be concisely described by the schema of the objects. Therefore, schemas are an effective way for searching Web databases.

Because of such schema-based modeling, sources and schemas naturally have a co-clustering property: sources that are relevant tend to share common schema features, and vice versa, similar schema features tend to occur in relevant sources. Such a co-clustering characteristic motivates an associativity search mechanism that captures the relevance of sources by how they are associated through their schemas, and vice versa the relevance of schemas by how they are associated through their sources. This mutual reinforcement of associativity inspires a principled mechanism to realize the generalized search by iterative relevance propagation, which Section 2.3.2 will discuss in details.
2.3.1 Schema-Based Source Modeling

As motivated in Section 2.2, our source models should capture features that describe the type of objects in the databases. *Schemas* in databases are such a mechanism. A conceptual schema of a relation describes the *name* and *attributes* of the relation. When presented to end users, such conceptual schemas are often carefully designed to be intuitive for users to understand. For instance, a TPC-W benchmark schema for books shown in Figure 2.5(a) may be complex and abstract in order to respect various normalization requirements. However, when presented online as in *amazon.com* shown in Figure 2.5(b), they are much simplified and intuitive. They use standard object names such as *book* and intuitive attributes such as *author* rather than *A_LNAME, A_FNAME* as in TPC-W schema.

Besides its intuitiveness, such schema features are easily obtainable from surface pages without probing into the content of the databases, which would require querying a source. As finding sources is only a preamble to querying or integrating the sources, source modeling should not rely on sophisticated techniques that would require deep understanding of a source.

With its intuitiveness and availability, we therefore propose to use conceptual schema as our source model. A schema captures two types of features: *relation feature* to describe the name of relation and *attribute feature* to describe the attributes of the relation. Just like in databases, an attribute captures both the name and type of the attribute. More formally, we define our source model as follows:

**Definition 1 (Schema):** The schema of a source $s$ is a two tuple $<N, A>$ where *relation features* $N = \{k_1, \ldots, k_m\}$ is a set of keywords describing the name of relation, and *attribute features* $A = \{a_1, \ldots, a_l\}$ is a set of *attributes* describing the attributes of relation. Each attribute $a_i$ is a two tuple $a_i = <n_i, t_i>$, where $n_i$ is the name of the attribute and $t_i$ is the type of the attribute.

**Schema Extraction:** To construct schema models, we need to extract relation and attribute features from Web pages.

*Attribute feature extraction:* Attribute extraction is built upon a *form extractor*, which Chapter 3 will develop in details. Form extractor extracts query conditions from interfaces. Upon those query conditions, we apply stemming, stop word removal, and alphabetic re-ordering on their attribute names to obtain labels for normalized representation. To recognize the type of attributes, we build a type recognizer. In particular,
<table>
<thead>
<tr>
<th>SID</th>
<th>Relation feature</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>book</td>
<td>author[String], title[String], isbn[String], keyword[String]</td>
</tr>
<tr>
<td>$s_2$</td>
<td>textbook</td>
<td>author’s name[String], book title[String], isbn[String], keywords[String], price[Float]</td>
</tr>
<tr>
<td>$s_3$</td>
<td>book</td>
<td>isbn[String], price range[Float]</td>
</tr>
<tr>
<td>$s_4$</td>
<td>movie, dvd</td>
<td>title[String], keywords[String], director: str, actor[String]</td>
</tr>
<tr>
<td>$s_5$</td>
<td>movie, top seller</td>
<td>director[String], actor[String], title[String]</td>
</tr>
<tr>
<td>$s_6$</td>
<td>movie</td>
<td>director[String], actor[String], release[String]</td>
</tr>
<tr>
<td>$s_7$</td>
<td>flight, airline</td>
<td>from[String], to[String], leave on[Date], return on[Date], number of passengers[Int]</td>
</tr>
<tr>
<td>$s_8$</td>
<td>ticket, reservation</td>
<td>from[String], to[String], departure date[Date], return date[Date], price[Float]</td>
</tr>
<tr>
<td>$s_9$</td>
<td>flight, reservation</td>
<td>from (airport code)[String], to (airport code)[String], departure date[Date], return date[Date], passengers[Int]</td>
</tr>
<tr>
<td>$s_{10}$</td>
<td>airline, vacation</td>
<td>departure city[String], arrival city[String], depart on[Date], return on[Date], passenger[Int]</td>
</tr>
</tbody>
</table>

Figure 2.6: A schema repository.

since the data type of an attribute can often be hinted by its syntactic features, we explore those syntactic clues to implement the type recognizer. For instance, we can exploit distinctive operators (e.g., all, any for text type), or the value instances associated with the attribute name.

Relation feature extraction: In principle, relation features may appear in many different places, in addition to the interface page, including pages along the path from a root page to the interface page, anchor texts to the interface page, or even queries that lead users to this interface page (which can be obtained from query logs of search engines). In those pages, keywords with high frequency, keywords that are highlighted, or keywords in the title of pages are all possible candidates. In our implementation, we apply a simplified scheme, which uses keywords with high frequency except stop words as relation features. For every source, we choose the 10 most frequent keywords with a restriction that they occur in at least 0.3% of sources.

For ease of understanding of our framework, we now introduce a running example.

Example 1: Consider that we have a schema repository $S$ with 10 sources $S = \{s_1, \ldots, s_{10}\}$ from 3 domains Books, Movies and Airfares. Figure 2.6 shows the schema for each source. For instance, source $s_1$ is modeled with relation feature “book” and attribute features author[String], title[String], isbn[String] and keyword[String].
2.3.2 Propagation-Based Associativity Search

The schema-based source models naturally exhibit a co-clustering behavior among databases and schemas. That is, relevant sources share similar schemas and relevant schemas appear in similar sources. To understand and validate this behavior, we surveyed attribute and relation feature occurrences for 500 sources in 8 domains from the UIUC Web Integration Repository. Figure 2.7 and 2.8 show the survey result, where a dot \((x, y)\) means attribute (or relation features) \(y\) occurs in source \(x\). We can observe the densely populated triangles along the diagonal line. Those triangles indicate such a co-clustering phenomenon, which we refer to as locality. Each locality suggests that a small set of attributes (or sources) often cooccur with a set of “similar” sources (or attributes). Further, the degree or the density of locality varies between attributes and relation features. The locality of attributes in a domain can be conceptually defined as the ratio of the attribute occurrences (i.e., dots) in a domain vs. the total occurrences. The figure indicates that attribute features exhibit stronger localities, while relation features weaker. This indicates that relevant databases are better associated by their attributes than by relation features. The reason is mainly because the mechanism for attribute extraction is more effective than relation feature extraction.

The co-clustering phenomenon of sources and schemas suggests that relevance can be nicely captured by
how things are associated in the locality. Specifically, we can capture the relevance of sources by how they are associated through schemas, and the relevance of schemas by how they are associated through sources. We therefore propose associativity as a mechanism to capture relevance.

The realization of associativity essentially hinges on a mutual reinforcement principle, as implied by the co-clustering phenomenon. The principle, originated from HITS algorithm [7], defines the importance of interlinked documents recursively based on a link structure. That is, a document is important if it is pointed to by other important documents. This principle has been exploited in many problems to discover desired semantics by analyzing the link structure among objects. For instance, PageRank [7] uses hyperlinks between pages to find important pages, and citation analysis uses reference links between papers to find related papers. In addition to such explicit links, many other works explore implicit links. For instance, a co-clustering algorithm [?] exploits the term-document occurrence links to cluster relevant documents and terms. SimRank [?] measures the similarity between objects by their interlinked contextual structure. LSI [?] essentially also utilizes the term-document occurrence links to find the salient concepts from the documents.

Inspired by the above work, to the best of our knowledge, we are the first to novelty apply the mutual
reinforcement principle– to be formally defined soon in the section– to capture the co-clustering phenomenon between databases and their schemas. Utilizing this principle, we define our associativity over a bipartite graph between databases and schemas. To realize the associativity, we first establish the bipartite graph to capture mutual reinforcement links between databases and schemas, and then we develop a link analysis mechanism over the bipartite graph to numerically compute the associativity.

Establishing Link Structures: Schema Graph

To capture the mutual reinforcement between databases and schemas, we propose to use a bipartite graph, which we refer to as schema graph, to represent the schema repository. Figure 2.9 illustrates such a schema graph of the source repository of Figure 2.6. The graph has two types of nodes– source nodes \( S \) on one side of the graph and schema feature nodes \( F \) (including both attribute and relation features) on the other side. If a schema feature occurs in a source, we add an edge between the corresponding nodes. Such a schema graph captures connections at different orders: the first order connection corresponds to direct edges between nodes, and higher-order connection corresponds to paths with multiple hops.

To construct schema graph, we merge the same schema features from different sources based on textual similarity and type similarity for attribute matching (e.g., “depart on[date]” to “departure date[date]”) and textual similarity for relation feature matching (e.g., “books” and “book”). More sophisticated matching mechanism utilizing semantics information such as values of attributes is under further investigation.

More formally, we define schema graph as follows:

**Definition 2 (Schema Graph):** Given a source repository \( S \), with each source \( s_i \) modeled with relation features \( s_i.N \) and attribute features \( s_i.A \), a schema graph of the repository is a bipartite graph \( G = ([S, F], E) \) where \( F = \cup_i(s_i.N \cup s_i.A) \) and \( E \subseteq S \times F \). There exists an edge \((s_i, f) \in E\) if and only if \( f \in s_i.N \cup s_i.A\).
Analyzing Link Structure: Associativity Propagation

The essence of mutual reinforce principle is the recursive definition of associativity enabled by the links in the bipartite graph. Numerically, if we associate with \( i \)-th source an initial associativity value obtained from a query as \( s^0_i \) and an associativity value obtained from the link structure as \( s_i \), and similarly for \( j \)-th feature with \( f^0_j \) and \( f_j \), the mutual reinforcement principle is expressed as:

\[
\begin{align*}
s_i &= \lambda \sum_{j: j \sim i} f_j + (1 - \lambda)s^0_i \\
f_j &= \lambda \sum_{i: i \sim j} s_i + (1 - \lambda)f^0_j
\end{align*}
\]

where \( j \sim i \) means that the \( j \)-th feature co-occurs with the \( i \)-th source, and a control parameter \( \lambda \) controls the relative contribution of associativity from an initial query and from the link structure. If we denote the initial associativity of all sources and features by vectors \( S^0 \) and \( F^0 \) respectively, and the final associativity as \( S \) and \( F \), we can express the mutual reinforcement principle as:

\[
\begin{align*}
S &= \lambda W_{SF}F + (1 - \lambda)S^0 \\
F &= \lambda W_{FS}S + (1 - \lambda)F^0
\end{align*}
\]  

(2.1)  
(2.2)

where \( W_{SF} \) and \( W_{FS} \) are normalized source-feature occurrence matrix, that is, there is a non-zero entry \( w_{ij} \) (or \( w_{ji} \)) in \( W_{SF} \) (or \( W_{FS} \)) if source \( s_i \) contains feature \( f_j \), and each column of the matrix sums up to 1. A \( j \)-th column in the matrix essentially specifies the distribution of associativity from node \( j \) to all its neighbors. With the recursive definition of associativity in Equation 2.1 and 2.2, the solution to the equations is thus the associativity vector \( S \) and \( F \).

Mathematically, such an abstraction follows the personalized page rank formulation [?], which computes the PageRank of pages with respect to a set of initial bookmarked pages. This set of bookmarked pages is equivalent to initial queries in our settings. So the importance of a page, i.e., Personalized PageRank, is determined by both the bookmark pages and the link structures.

Notice that our graph has two types of links between sources and features: source to relation feature and source to attribute features. Therefore, to differentiate the two types of links, we introduce parameters \( \alpha \).
and $\alpha_k$, both in the range of $[0, 1]$ with $\alpha_a + \alpha_n = 1$, to control the fraction of associativity that a source node propagates to attributes and relation features, respectively. Among the neighbors of the same type, the associativity is divided uniformly. Therefore, for the normalized matrix $W_{FS}$, an entry $w_{ji}$ is defined as:

$$w_{ji} = \begin{cases} 
\frac{\alpha_n}{|s_i.N|} & : \text{if } f_j \in s_i.N \\
\frac{\alpha_a}{|s_i.A|} & : \text{if } f_j \in s_i.A
\end{cases}$$

Abstracting the associativity using the mutual reinforcement principle enables a generalized search mechanism by propagation. As Section 2.4 will discuss, the solution to the equation is reached by iteratively propagating associativity from the initial query nodes to their neighbors until converged. In fact, the vector space model deployed by current search engines is essentially a one-step process of relevance propagation from keywords to their containing documents. As generalization, our propagation captures not only the “first-order” relevance by one-step propagation, but also “high-order” relevance by iterative propagation.

### 2.4 Realization of Online Associativity Search

In this section, we further develop our computation framework to enable online search. To analyze links in a graph structure, various techniques are proposed. However, so far they are all for offline computation, and not suitable for online search. The reasons are two fold. First, there is no need for online computation due to the lack of query locality. While our problem focuses on finding top scored nodes in the graph w.r.t. query nodes, which we refer to as query locality, their problems aim at finding a global solution to every node in the entire graph. For instance, PageRank computes pagerank for every page, LSI finds salient concepts for all documents and co-clustering forms clusters for all terms and documents. Since there is no notion of query, online computation is not an issue in those problems.

Second, it is not feasible to apply their techniques online due to the lack of data locality. The lack of data locality implies that the solution can only be found by global analysis over entire graph, which would require huge matrix manipulation, and thus is not practical for online computation. However, with the naturally exhibited data locality in our problem, we can localize the propagation into near neighbors of the query node. Further, the data locality leads to highly skewed distribution of scores and computation times for different nodes in the graph. Taking advantage of such data locality, Section 2.4.2 proposes
matrix optimization techniques to bound the scope of link analysis, and Section 2.4.3 proposes probabilistic incremental convergence technique to eagerly generate top results whenever they are ready.

### 2.4.1 Basic Computation Framework

For ease of discussion, we rewrite the two equations in Equation 2.1 and 2.2 into the following equation, by combining the source vector $S$ and feature vector $F$ into an associativity vector $A$, that is,

$$A = \lambda W A + (1 - \lambda) A_0$$  \hspace{1cm} (2.3)

where $A = (S, F)$, $A_0 = (S_0, F_0)$ and

$$W = \begin{pmatrix} 0 & W_{SF} \\ W_{FS} & 0 \end{pmatrix}$$

Given Equation 2.3, the problem of computing $A$ can be casted into computation of the eigen vector of the matrix $W$, which can be iteratively computed using the Power method, or its improvements - Jacobi method and Gauss-Seidel method [?]. Because of the way we defined $w_{ij}$, the matrix $W$ is stochastic and as a result, these iterative methods are guaranteed to converge. (To guarantee convergence, $W$ is required to be irreducible as well. In practice, we expect $W$ to be irreducible due to the fuzzy boundary between localities observed in Figure 2.7 and 2.8. If it is not, we can always make the matrix fully connected with a very small weight for new edges.) Briefly speaking, the Power method iteratively computes the associativity vector in $k$-th iteration denoted as $A^{(k)}$ through its $(k - 1)$-th iteration till convergence, i.e.,

$$A^{(k)} = \lambda \times W \times A^{(k - 1)} + A_0$$  \hspace{1cm} (2.4)

Further, the Power method can be reformulated in terms of the change in the associativity vector [?]. Let us define $\Delta A^k = A^k - A^{(k - 1)}$. Then $\Delta A^k$ can be recursively formulated as: $\Delta A^{(k)} = \lambda \times W \times \Delta A^{(k - 1)}$. We can thus compute $A$ by aggregating the $\Delta$ values over successive iterations. This reformulation enables a large degree of flexibility in prioritizing updates. In particular, instead of updating the whole $A$ at each iteration, we prioritize the updates of nodes based on their $\Delta$. Updating nodes with larger $\Delta$ triggers more
changes, which results in a faster progress towards convergence.

Algorithm 2.4.1 outline our incremental propagation algorithm for computing associativity. When ignoring the bold faced lines (line 4, 5 and 11 about the incremental convergence mechanism to be discussed soon), the Figure gives us the basic computation algorithm. The baseline algorithm takes as input, a propagation matrix $W$ prepared offline, a query $Q$ given by a user and a pre-configured decay factor $\lambda$. It outputs the associativity vector $A$. For each source, the algorithm keeps track of its current associativity value $A$ and the $\Delta A$ yet to be propagated. At each iteration, it picks the node with the largest pending $\Delta$ to process (line 4). The processing includes: propagating $\Delta$ to its neighbors (lines 5-7), updating its associativity value (line 8), and resetting its pending $\Delta$ to zero (line 9).

However, as discussed earlier, this basic algorithm cannot satisfy the online computation requirement as it requires repeated disk IOs. In the following sections, catering to the data locality characteristics of our problem, we develop an incremental propagation technique by matrix optimization to reduce the scope of propagation and probabilistic incremental convergence to generate the top results whenever they stabilize.

### 2.4.2 Matrix Optimization: Speed Up Convergence

As Figure 2.10 shows, the definition of associativity vector in Equation 2.4 indicates that the iterative computation of $A$ essentially involves multiplying the propagation matrix repeatedly to the initial vector. Therefore, to optimize, there are two dimensions, as Figure 2.10 intuitively shows: First, for “individual” $W$, the spatial dimension explores the spatial localities of $W$ to decompose the large matrix into several smaller ones. Second, across “multiple” $W$, the temporal dimension explores the temporal repetition of matrix production to pre-compute the aggregated matrix.
**Spatial Optimization: Matrix Locality**

To optimize along the spatial dimension, we want to “localize” the accesses to matrix $W$ to a small region, which can largely fit into memory without being repeatedly retrieved from disks. With 30,000 sources used in our experiments in Section 2.5, we extract totally 4198 relation features and 19,687 attribute features. As each entry stores the weight of a feature as a double, the size of the source-attribute matrix is approximately $4.7G$ KB and the size of the source-relation feature matrix is approximately $1GKB$. When the number of sources increases, the matrix will grow proportionally. Matrices of such big sizes are usually disk-resident, which would render repeated disk accesses to retrieve the matrix.

However, the existence of data locality in our problem indicates that relevance propagation will be largely bounded within the locality instead of spanning across the entire graph. Therefore, we could exploit these localities by decomposing the propagation matrix into smaller blocks. By doing so, we will be able to cache these blocks in memory, and thereby reduce the number of disk IOs.

An ideal decomposition is to arrange all non-zero entries into separated submatrices along the diagonal line of the original matrix $W$ ($W_{SF}$ or $W_{FS}$). That is,

$$W = \begin{pmatrix} W_1 & 0 & \cdots & 0 \\ 0 & W_2 & \cdots & 0 \\ 0 & 0 & \cdots & W_l \end{pmatrix}$$

If such an decomposition exists, Equation 2.3 can be decomposed into:

$$A[1] = \lambda W_1 + (1 - \lambda)A_0[1] \quad (2.5)$$

$$\cdots$$

$$A[l] = \lambda W_l + (1 - \lambda)A_0[l] \quad (2.7)$$

The equation indicates, given a query node at block $i$, the propagation can be localized within submatrix $W_i$. Suppose the size of the submatrix $W_i$ is $\frac{W}{m}$. The computation complexity will be reduced by a factor of $m^2$ at each iteration.
Therefore, we are looking for a decomposition that results in blocks as dense as possible, and as separate as possible. Denseness means each submatrix is filled with many non-zero entries, and separateness means different submatrices have little overlap. In addition, we require that this decomposition should not result in any loss of information, that is, every entry in the original matrix should belong to some submatrix. While a perfectly non-overlapping decomposition may not exist due to the fuzzy boundaries between localities, an overlapped decomposition may still be “good-enough” so as to provide significant speed-up.

Such a decomposition can be obtained by various existing techniques developed for different purposes, such as, reordering a matrix for fast graph partitioning \([?]\), matrix compression \([?]\), or partitioning the sources using term-document occurrence matrix \([?]\). In our implementation, we develop a hyperclique \([?]\) based blocking scheme which partitions the sources into groups that share a common vocabulary. The partitioning of sources thus obtained is then used for decomposition of the propagation matrix.

**Temporal Optimization: Matrix Pre-aggregation**

To optimize along the temporal dimension, we now want to reduce the number of online matrix multiplications. The computation approaches discussed so far basically follow “one-hop at a time” propagation paradigm. As Equation 2.1 and 2.2 indicate, at each iteration, the basic algorithm propagates associativity from a source one-hop to its neighbor features, which in the next iteration propagates one-hop back to the neighbor sources. We thus ask: whether we can expand “multiple-hops” at each iteration, so that we can reach the convergence with much fewer number of matrix multiplications.

**Two-hops Aggregation:** The first optimization is to aggregate the two-step propagation \(source \rightarrow feature \rightarrow source\) into a one-step propagation of \(source \rightarrow source\). Let us consider the definition of vector \(S\) and \(F\) in Equation 2.1 and 2.2. As these definitions are mutually recursive, we can rewrite \(S\) by substituting \(F\) into:

\[
S = (\lambda^2 W_{SF} W_{FS}) S \\
+ (1 - \lambda)(\lambda W_{SF} F^0 + S^0)
\]
Let $M = \lambda^2 W_{SF} W_{FS}$, and $Q_0 = (1 - \lambda)(\lambda W_{SF} F^0 + S^0)$. We have:

$$S = MS + Q_0$$  \hspace{1cm} (2.8)

Applying the same algorithm in Algorithm 2.4.1 with propagation matrix $M$ (instead of $W$) and initial vector $Q_0$, we proceed two steps at a time, propagating associativity from source to source directly. This transforms the repeated accesses to the two matrices $W_{SF}$ and $W_{FS}$ to accessing $M$ only.

**Multiple-hops aggregation:** Applying the same idea, we can perform multiple-hops propagation to expand from a source to its length-$l$ neighbors. As Equation 2.8 suggests, the associativity vector $S^{2l}$ at $2l$-th iteration can be defined upon that at $l$-th iteration as follows:

$$S^{(2l)} = M^l \times S^{(l)} + \left( \sum_{i=0, \ldots, l-1} M^i \right) \times Q_0$$  \hspace{1cm} (2.9)

Let $X = M^l$ and $L_0 = (\sum_{i=0, \ldots, l-1} M^i)$. More generally, we can verify that the following equation holds:

$$S^{(kl)} = X \times S^{(k-1)l} + L_0 \times Q_0$$  \hspace{1cm} (2.10)

The equation states that the associativity vector at step $k \times l$ can be obtained through the vector at step $(k - 1) \times l$ using $X$ and $L_0 \times Q_0$. Therefore, we effectively proceed $l$ steps at a time towards computing $S$.

The feasibility of such pre-computation again lies in the locality characteristics of our data. The pre-computation is not feasible for personalized pagerank because the size of matrix is at the order of billion, and computing $M^l$ is impractical. However, in our problem, the big matrix multiplication can be decomposed into independent submatrix multiplication, as the following equation indicates. So if the size of a submatrix is $1/m$ of original matrix $M$, the multiplication complexity will be reduced by a factor of $1/m^3$, which is significant. In our experiment, the average size of source-attribute blocks is $400 \times 240$, and thus the size of the original source-attribute matrix is reduced by a factor of 6250, which is very significant. The average size of the source-relation feature blocks is $400 \times 160$, and thus the original matrix is reduced by a factor of 1875.
2.4.3 Prioritize Convergence: Probabilistic Incremental Convergence

While the matrix optimization speeds up the overall convergence of all nodes in the graph, the probabilistic incremental convergence discussed in this section, tries to output the top-K answers as soon as they become stabilized in a probabilistic sense. As an online search facility, the results are usually returned to users in an incremental way, so the response time for top results, mostly top 10, is critical. Therefore, instead of waiting for every node to converge, we can output the top-10 as soon as they are ready.

Intuitively, results with high scores should converge earlier than results with low scores. The reason is that the propagation of associativity has a decayed factor $\lambda$, and therefore the nodes close to the query on one hand get more associativity propagated, and on the other hand are reached promptly. Our empirical study also verifies the intuition. Figure 2.11(a) shows the error-in-ranking (or errors for short) over time for various rank positions. Specifically, we adapt the Kendalls-Tao distance measure to define error-in-ranking of top-$k$ result as the average error-in-ranking over the $k$ results, when compared to ranking at $t = \infty$. As it is impractical to run the queries for $t = \infty$, we thus simulate $t = \infty$ by running our propagation for 1min, which usually is sufficiently long for the top results to converge empirically. As we can see, the errors decrease with times, which means result ranking indeed incrementally converges. Further, comparing different curves, we see the time of top $K$ convergence decreases with $K$, and thus top 10 results are converged much earlier than top 50.

Motivated by this observation, we propose an incremental convergence mechanism for early generation of top-K results. The key challenge in the early convergence is to decide when the top-K has stabilized, i.e., the set of $K$ results does not change (but the exact orders may change) over time.

Let us analyze theoretically on the stop condition. For easy explanation, we assume $k = 10$. Let $\delta$ denote the current score gap between the 10th and 11th results, and $x_i$ denote the remaining delta of source $s_i$ till
convergence, i.e., the difference between the converged score and its current score. Although knowing the individual \( x_i \) before computation ends is impossible, the total amount of \( x_i \) that remains to be added to the entire system can be computed theoretically. Recall the basic power method for propagation. At each iteration of computing \( A^k \) from \( A^{k-1} \), the amount of new associativity added to the system is decayed by a factor of \( k \). Therefore, starting with an initial associativity of \( 1 - \lambda \), the total amount of associativity at convergence is \( R = (1 - \lambda)(1 + \lambda + \cdots + \lambda^\infty) = 1 \). With simple manipulation, we can know that the total amount of associativity for all sources is \( \tilde{S} = \frac{1}{1+\lambda} \). During computation, we know exactly how much associativity \( \tilde{S}' \) currently accumulated in the system. Thus the total remaining delta \( D \) is computed as \( D = \tilde{S} - \tilde{S}' \). Ideally, if the score gap \( \delta \) is higher than \( D \), we know no other result will jump to top-10, and thus the top-10 results stabilized. However, this is a very strict requirement, as we assume \( D \) will all be poured to a single node, which is very unlikely. Actually, \( D \) is distributed among sources. Therefore if we know the distribution of \( D \) over sources, we can guarantee the top-10 early convergence with certain confidence. This is the essential idea of our probabilistic incremental convergence.

**Incremental convergence with respect to a probabilistic model**

Given a probabilistic distribution model of the remaining delta \( D \) over sources, we can infer the probability of top-K being converged. Suppose such a distribution is given by a probability density function PDF \( P(x) \), as Figure 2.12(a) illustrates. The cumulative probability that a source gets delta higher than \( x \) amounts to
the shadow area in the distribution curve. This area is computed by:

\[ P(X > x) = \int_x^\infty P(x) \]  

(2.11)

This equation implies two things: First, it tells us the probability of current top-10 being converged. We know that the top-10 converges, if all other sources have remaining delta \( X < \delta \). Therefore the probability of top-10 being converged can be estimated by:

\[ P(X < \delta) = 1 - P(X > \delta) = 1 - \int_\delta^\infty P(x) \]  

(2.12)

Second, the distribution also tells us how to find a cutoff delta \( x_c \) so that the probability of \( X \geq x_c \) is less than a certain threshold \( p_c \). This can be computed by solving the equation:

\[ P(X > x_c) = \int_{x_c}^\infty P(x) < p_c \]  

(2.13)

Given the cutoff delta \( x_c \), if we stop at \( \delta = x_c \), the probability of a source jumping to top-10 is only \( p_c \). In other words, this cutoff delta guarantees the top-10 being stabilized with confidence \( 1 - p_c \).

Therefore, such modeling gives us two modes of probabilistic incremental convergence—response-time guaranteed and response-accuracy guaranteed. First, if users prefer over responsiveness, we can fix the computation time, and output the current top-10 answers augmented with the confidence of how accurate they are. Or if the typical elapse times for top-10 convergence are very consistent over different queries, we can use this empirical time threshold as stopping condition to output top results augmented with confidence. Second, if users prefer over accuracy, we can fix the confidence threshold \( p_c \), and output the top-10 results when we are \( 1 - p_c \) confident about the convergence. In fact, in our empirical study, we do find that the convergence behaviors vary greatly over queries, as Figure 2.11(b) shows for two benchmark queries \( Q_1 \) and \( Q_3 \) which we use in the experiment of Section 5.6. As we can see, \( Q_1 \) takes 1.5 sec to converge at top 50 while \( Q_2 \) 2.8 sec. Therefore, the accuracy-guaranteed scheme is indeed needed to adapt the response time with certain confidence guarantee rather than using a fixed time threshold. A hybrid method can output the results when either the time runs out or we meet the guaranteed accuracy.
Establishment of probabilistic model: To use such a probabilistic incremental mechanism, we need to establish the PDF function of delta distribution. However, different queries will have different PDF. How can we find a right PDF for a query? We find that although the parameters of a PDF function may vary over queries, the basic shape of the distribution is usually consistent among queries over our dataset. Such a shape usually depends on the topology of the schema graph of the dataset. Because of the schema-based source modeling, the topology of the graph is rather homogeneous. First, because of the locality, the graph can be partitioned into subgraphs, and consequently the overall complex distribution is decomposed into simpler distribution within localities. Second, within a single locality, sources are more homogeneous—different source nodes have relative homogeneous connectivity to the rest of the graph, unlike in the Web graph, where incoming and outgoing links of a page vary significantly. Therefore, the type of distribution tends to be consistent among queries.

We thus propose a two-phase process for establishing distribution function— an offline function template estimation phase and an online template parameter estimation phase. The offline phase estimates the function type by typical workloads. In this phase, some representative queries will be tested over the dataset to gather some sample distributions. Then human experts will need to find out the best function type that fits those statistics. Common function types might be exponential, zipf, gaussian, or uniform. Curve fitting tools such as Matlab may help with finding the best distribution function for sample queries. Given a function template, we can derive corresponding formula to compute the parameters, and thus know the type of statistics we need gather online to support the parameter estimation. In online phase, we gather necessary statistics to estimate the parameters of the function template. Once those parameters are computed, they will be used to derive the top-k convergence probability using Equation 2.12 or the cutoff delta using Equation 2.13.

In our specific setting, we find that exponential distribution fits our data very nicely. Figure 2.12(b) shows the distribution of $D$ over sources at different time steps for a specific query. As we can see, the distribution $D$ consistently follows the exponential curve fitting at different times. We verify such distribution holds for all 230 queries we sampled. Therefore, we use exponential function as our PDF function template in our empirical study.

Given the function template, estimating function parameter is rather cost effective for exponential PDF.
Let $P(x) = \beta e^{-\beta x}$ be our probability function template and $N$ be the number of sources. Given $D$ distributed over $N$ sources, the mean of $x$ is thus $D/N$. According to the property of the exponential distribution, the mean of $P(x)$ is given by $1/\beta$. By solving the equation $\frac{1}{\beta} = \frac{D}{N}$, we get the formula for estimating parameter $\beta = \frac{N}{D}$. Consequently, using Equation 2.12, we know that when stop at this iteration, the probability that current top-10 is stabilized is:

$$P(X < \delta) = 1 - \int_{\delta}^{\infty} P(x) = 1 - e^{-\beta \delta} = 1 - e^{-N\delta/D}$$

Further, if we want the probability of current top-10 being converged to be $1 - p_c$, then the cutoff delta is computed by:

$$P(X > x_c) = \int_{x_c}^{\infty} P(x) = e^{-\beta x_c} \leq p_c \Rightarrow x_c \geq \frac{-D \ln p_c}{N}$$

**Putting together: Generating Top-K Results Incrementally**

We describe the overall incremental propagation algorithm in Algorithm 2.4.1. Upon the basic computation discussed in Section 2.4.1, the three lines in procedure `IncrProp` further realize the probabilistic incremental convergence mechanism. Line 3 initializes the remaining $D$ as the total relevance score at convergence, and line 5 initializes a priority queue of size $k + 1$ to maintain the top-$(k + 1)$ results. Procedure `stopped` realizes the hybrid approach for determining the stop condition. At each iteration, after a source
node $s_j$ is updated, procedure $\text{updateQ}$ updates $topQ$ if $s_j$ passes the minimal score of $topQ$ denoted as $\tau$. In addition, related variables, including current $\delta$, $\tau$ and remaining delta $D$ are all updated accordingly.

2.5 Experimental Evaluation

This section evaluates the effectiveness and the practicality of proposed solution as an online search mechanism. Specifically, we study: first the accuracy of search results, second the correctness of probabilistic incremental convergence, and finally, the speed up obtained by spatial and temporal matrix optimization strategies.

2.5.1 Evaluation Environment

Dataset and Queries: We perform all our experiments on a collection of 30000 deep web sources obtained by our own crawler. We do an extensive evaluation in 6 particular domains - {Jobs, Used Cars, Airline, Hotels, Book, and Movie}, which we used in our survey of current search engines. We use Source Query, a set of 180 queries (30 queries per domain) containing example sources as input, and similarly, Keyword Query a set of 180 queries (again 30 queries per domain) containing keywords as input. In all, we evaluate our system’s performance on about 360 example queries.

Implementation Details: We perform all our experiments on 2.60 GHz hyper-threaded P4 machine with 1 GB memory. The metadata extraction for 30K sources takes about 2-3 days. Preparing the propagation matrices from extracted metadata takes less than an hour. For fast access to propagation matrices, we use BTrees in python bsddb module.

2.5.2 Accuracy of Search Results

In this section, we show the accuracy of our approach. We refer to this evaluation as our baseline performance in contrast to the spatial or temporal optimization evaluation later in Section 2.5.4.

Illustration: We first illustrate the effectiveness of the system by 4 benchmark queries. First, consider users who want to find airline sources. They issue query $Q_1 = \{aa.com\}$ or $Q_2 = \{fares for flight\}$. The top 10 results for the two queries are illustrated in Figure 2.13 (a) and (b), respectively. Not only both the results are all airline sources, we also find that 25 out of the top 30 sources obtained for the two queries are same. As another scenario, users who are looking for job sources on the web, issue queries $Q_3 = \{hotjobs.com\}$ or
\textit{Q4} = \{\textit{jobs}\}. As illustrated in Figure 2.13 (c) and (d), top results for both the queries are all job domain sources. Further, we find that top 27 of their top 30 results are same. These illustrative queries indicate the accuracy as well as the stableness of results.

Next, we further extensively study our system using a more comprehensive evaluation.

\textbf{Large scale evaluation}: For each of the 360 queries, we manually inspect the top 50 results. We find that the average precision at top 30 results (Pr@Top30) is above 80\%, for most cases, as shown in Figure 2.14 (a). Pr@Top50 generally achieves above 50\% and even 80\% for many cases, as Figure 2.14(b) shows. These results also demonstrate the effectiveness of our \textit{similar source} search functionality, which is absent at current search engines. The accuracy for \textit{Source Query} is of the same order as \textit{Keyword Query} for all domains.

Unlike the present search engines’ high sensitivity to variations in queries, our search mechanism is very robust. Thus, we alleviate the users from the requirement of carefully picking the “right” keywords for their queries. Comparing with less than 10\% overlap in early survey of Google, the overlap of Top 50 is more
than 50% in all cases in our system, as reported by Figure 2.15.

While these results already indicate the utility of propagation based associativity, to better understand its benefit over the vector space model based similarity measure, we compare their Pr@Top50 in Figure 2.16. As we can see, propagation based associativity measure, referred to as with-prop, consistently outperforms vector space based similarity measure, referred to as no-prop, by an order of 10%.

2.5.3 Utility of Probabilistic Incremental Convergence

From the analysis of execution traces of large number of queries, we observe that the remaining delta $D$ follows an exponential distribution. Therefore, using formula derived in Section 2.4.3 we can online estimate the function parameter $\beta$ using $D$, and use the estimated distribution function to decide the stop condition for top-$k$ results. In this section, we evaluate the correctness of this incremental convergence method using so estimated distribution function.
Illustrating correctness of prediction: We show the correctness of the predicted convergence with our two benchmark queries $Q_1$ and $Q_2$. These queries have very different response time behaviors, as was shown earlier in Figure 2.11. For each query, now we observe how the predicted correctness confidence correlates with actual error in ranking. For $Q_1$, as shown in Figure 2.17 (a) the predicted confidence is initially very small, at time 0.5s it jumps to about 40%, and at about 1.5s it is over 80%. Clearly, this correlates very strongly with the actual convergence behavior of $Q_1$. On the other hand, for $Q_3$, as shown in Figure 2.17 (b), upto 2.5s the predicted confidence stays below 20%, and at 2.8s it is 80% confident that $Q_2$ has converged.

Large scale correctness evaluation: While we see that predicted confidence is correlated with actual accuracy for the illustrative queries, we further evaluate this effect at large scale. We vary the confidence thresholds, and for each query in Keyword Query, we note down the actual errors of top 50 results when stopping at this threshold, and then we average the errors among queries. Figure 2.18 (a) shows this average errors with respect to confidence. As we can clearly see, our incremental convergence prediction is quite accurate. When confidence is 90%, the top-50 results indeed only have 1.5 errors in average. Further, we observe a strong correlation between predicted confidence and the actual convergence— As our predication confidence grows, the error decreases accordingly.

While the average errors show the accuracy of prediction over all queries, we want to further understand how those errors distribute across individual queries. Specifically, we study, for different confidence thresholds, the fraction of queries having a certain maximal error. Figure 2.19(a) shows for different maximal
errors $e$, the percentage of queries that make less errors than $e$. For instance, the curve for $\text{conf} = 90\%$ shows that 80% of queries have actual errors smaller 1.5- the average errors for all queries, and 90% of queries have actual errors less than 3. Similarity, for $\text{conf} = 30\%$, about 70% queries have less errors than the average value. This illustrates that, not only the average error is correlated with confidence threshold, but for a significantly large fraction of queries, the actual errors are indeed smaller than the average error.

**Tradeoff between confidence and response time:** Figure 2.18(b) shows the average convergence time for top 50 results with respect to confidence values. As we expect, the higher confidence we want to achieve, the longer time we will spend. As Figure 2.18(a) and (b) together suggest, the choice of confidence threshold
can be determined based on desired balance of accuracy and response time. For instance, if we want an average error less than 2, we find the corresponding confidence is 30% in Figure 2.18(a), which corresponds to an average response time of around 0.5 in Figure 2.18(b). We therefore may choose time threshold 0.5s as the stopping condition with tolerance for less than two errors.

Further, instead of just looking at the average case, more fine-grained tuning can be made by looking at distribution of both accuracy and response times over sources. Figure 2.19 (a), as we discussed earlier, shows the percentage of queries with certain maximal error. Figure 2.19(b) similarly shows the percentage of queries with certain maximal response time. From these figures, we can answer: what should be the confidence threshold such that 90% queries have less than one error (answerable from Figure 2.19(a))? For this threshold, what fraction of queries have response time less than 1.5s (answerable from Figure 2.19(b)). Thus, the confidence threshold as a parameter of our system provides not only a control on accuracy of queries, but can more generally be used to find the balance between accuracy and response time from the workload.

**Absolute response time:** The experiment results show that with 80% confidence, an average response time are around 0.1s for top-10 results and about 1.5s for top-50 results. Even users do not find even one of the top-10 results as relevant, the amount of time spent just in deciding whether the results are relevant or not is more than sufficient for the next batch of results to be ready. Thus, by combining incremental convergence into our search system, we can not only assure high quality of results, but at the same time, also assure that the effective response time is as small as finding the top-10 results.

### 2.5.4 Speed Up of Spatial and Temporal Optimizations

This section evaluates the effectiveness of spatial and temporal optimization techniques. Specifically, we study: (a) How much speed up they can provide, and (b) Do they degrade the accuracy of search results?

To compare the accuracy after applying the optimization techniques to the accuracy without the optimization techniques, we define a new metric: *accuracy-convergence*, which is the ratio of observed Mean Average Precision (MAP) \(^1\) to the MAP obtained by *baseline* algorithm upon convergence.

---

\(^1\)MAP is a standard metrics used for evaluating performance of IR systems. For each recall value, we take the average precision of different queries at this recall. Among the average precisions with various recall, we take the mean as MAP.
### Evaluation of Spatial Optimization Technique

We compare the blocking strategy - *hc-block* against the baseline method using the *Keyword Query*. Figure 2.20 shows the accuracy-convergence and the time for convergence. We find that the *hc-block* method is able to provide significant speed up and still achieves the accuracy-convergence of nearly 100%.

To further study the scalability of *hc-block*, we compare its performance against baseline approach with variation in dataset sizes by incrementally adding sources from our collection. Using the benchmark queries, we compare the accuracy-convergence and time for convergence. As shown in Figure 2.21, for all sizes of datasets *hc-block* is able to achieve the accuracy-convergence of nearly 100%. More interestingly, the time for convergence for *hc-block* is less than half of the baseline, showing a speed up by at least a factor of 2. Further, the increase of convergence time with increase of dataset size is slower for *hc-block* as compared to the baseline. This suggests that not only *hc-block* significantly speeds up the computation, but it is also more scalable than the baseline algorithm.

### Evaluation of Temporal Optimization Technique

We now study if the temporal optimization can
Figure 2.22: Speed up of temporal optimization

<table>
<thead>
<tr>
<th>n</th>
<th>(Th_L) 0.001</th>
<th>(Th_L) 0.0001</th>
<th>(Th_L) 0.00001</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.22</td>
<td>0.51</td>
<td>4.30</td>
</tr>
<tr>
<td>5</td>
<td>0.20</td>
<td>3.21</td>
<td>3.05</td>
</tr>
<tr>
<td>8</td>
<td>0.19</td>
<td>2.87</td>
<td>2.78</td>
</tr>
</tbody>
</table>

Further improve the performance. There are two possible implementations of temporal optimization. First, we compute \(L_0 \times Q_0\), and use it directly as the final result. Note that although applying further propagation using the precomputed matrix \(X\) is possible, we believe that if \(n\), which controls the degree of propagation, is large enough, further propagation would be unnecessary. In implementation, we find that the matrix \(L_0\) is very dense but with many close-to-zero entries. To speed up, we prune the entries of \(L_0\) that are below a certain threshold \(Th_L\). Larger the value of \(Th_L\), more aggressive the pruning is. However, we observe that as long as \(Th_L\) is small enough, its choice does not affect the accuracy-convergence much.

In Figure 2.22, for different values of \(n\) and \(Th_L\), we show the ratio of response time of hc-block with over without temporal optimization. For all cases in Figure 2.22, the accuracy-convergence for hc-block with temporal optimization is about 83%, which suggests that the value of \(n\) does not affect the quality of results. The reason for such a counter-intuitive observation is that our Keyword Query seek sources in a very specialized domain. Even a small amount of propagation is sufficient to explore such a narrow domain. We believe that queries exploring broader domains, e.g., travel related sources, entertainment sources, would show greater dependency on the degree of propagation. For all values of \(n\), the smaller the value of \(Th_L\), the greater the computational speed up is. In fact, with \(Th_L = 0.001\) and \(n = 2\), temporal optimization reduces the response time to about 23%, a significant speed up of about 4.4 times.

Although intuitively temporal optimization should linearly speed up the computation by “striding” multiple steps at a time, in practice, we did not obtain such a speed up. The reason is two folded: First, unlike the basic progressive approach, the temporal optimization has non-negligible startup time to convert the initial associativity vector \(A_0\) to \(L_0 \times Q_0\), which involves matrix multiplications. Second, the precomputation may potentially lose the benefit of early termination of propagation. Since precomputation of \(n\)-hops essentially aggregates all paths within length \(n\), it may overdo for some sources, which can stop earlier.
Overall Speed-up: Recall that hc-block already performs more than twice faster than the baseline approach. Thus, overall, combining our spatial and temporal optimization strategies, we can achieve a speed up by a factor of about 10 times over the baseline approach.

2.6 Related Work

Databases on the deep Web have gained much attention recently. Several problems have been studied, such as, focused crawling [? ,?] which can be adopted for finding interfaces, interface understanding [?], schema matching [? ,? ,? ,? ,? ] and query translation [?]. To the best of our knowledge, ours is the first attempt towards developing an online search facility for finding Web databases by schemas.

Halevy et al[?] studied the problem of finding similar Web services based on service descriptions. We are different from their techniques in that: First, we focuses on more structured and intuitive databases schemas while they uses unstructured Web service descriptions. Second, we capture the mutual reinforcement principle by a link analysis mechanism, while they uses clustering techniques to find connections between descriptions. Finally, we realize the computation by incremental propagation, while they basically adopt the vector space model for computing similarity.

Source modeling: Exploiting a large repository of schemas with associated data and metadata has been proposed earlier [?]. However, it focuses on the application to schema matching, while we focus on using schemas for searching databases. Extracting query interfaces has been well studied in [? ,? ]. Built upon their techniques, we model sources with schemas including both relation features and attribute features. Probing-based content modeling is made possible by techniques studied in [? ,? ]. However, such modeling is rather expensive and specific queries need to be designed for selected sources, which is not suitable to our scenarios with massive sources dynamically crawled online.

Locality based semantic discovery: The locality phenomenon has been explored in early works [? ,?] for interface matching and integration. Inspired by the work, we propose to explore the associativity of such metadata to find deep Web databases. While the work explores the locality by a clustering or statistic analysis approach, we realize the locality in a link analysis framework.

Link structure based semantic discovery: Many works explore the mutual reinforcement principle
on link structures to discover semantics, as Section 2.3 discussed. This includes PageRank and citation analysis for explicit links, and co-clustering and LSI for implicit document-term links. Different from those techniques designed for offline computation, our link analysis is optimized for online search.

**Propagation based link analysis:** As mentioned early, propagation based link analysis is used for computing PageRank or Personalized PageRank. Different from their offline techniques, we are among the first to adapt propagation based techniques for online search. Towards this goal, we develop our incremental propagation technique to leverage the query and data locality for fast generation of query results. ObjectRank has also modeled the associations into a linkage graph, and applied PageRank like computation. Their focus is on identifying the weights of different types of edges, while we find that to be less of our concern, and scaling up the computation to support online search is much more critical in our scenario.

Query expansion is also a technique for realizing the “high-order” relevance by expanding both queries and consequently matched documents iteratively. However, due to the lack of data locality, their realization is rather ad-hoc and turns out not effective. First, they need to decide how to pick the terms for expansion, and how to fix the number of iterations, which are all hard decisions made in an ad-hoc way. Second, the co-clustering phenomenon between documents and terms are not strong. Therefore on one hand, propagation on the entire document-term graph is not very meaningful to capture the relevance; on the other hand, it is also not feasible because of the size of the graph. In contrast, the co-clustering phenomenon between databases and schemas makes our propagation not only meaningful but also practical.

Spreading Activation or SA techniques use sophisticated algorithms for relevance propagation. They focus on how to use the constraints based on the semantic information to build links of inference or neural network. In contrast, we derive links purely using “syntactic” occurrence analysis, and mathematically model them with Markov chain model.

43
Algorithm 2.4.1 Incremental propagation algorithm.

Procedure IncrProp($W, Q, \lambda, k, p_c, t_c$)

Input: Propagation matrix $W$, query $Q$, decay factor $\lambda$, number of results $k$, accuracy threshold $p_c$, time threshold $t_c$.

Output: Associativity vector $A$.

1. initialize $A_0$ with $Q$
2. $\Delta A = A_0$
3. $D = 1/(1 + \lambda)$
4. initialize priority queue $topQ$ as empty
5. while not stopped do
6. $j = \arg\max_{j} \Delta A[j]$
7. $W_j = j$th column of $W$
8. for each $W_{i,j}$ in $W_j$ do
9. $\Delta A[i] + = \lambda W_{i,j} \times \Delta A[j]$
11. updateQ($j, \Delta$)
12. $\Delta A[j] = 0$

Procedure updateQ($j, \Delta$)

Input: source $s_j$, its delta $\Delta$.

Output: updated $topQ$, $\tau$, $\delta$ and $D$.

1. if $A[j] > \tau$ then
2. insert source $s_j$ into $topQ$
3. $\tau = topQ[k+1]$
4. $\delta = topQ[k+1] - topQ[k]$
5. $D - = \Delta$

Procedure Stopped($D, \lambda, P, p_c, t_c$)

Input: Remaining delta $D$, decay factor $\lambda$, PDF template $P$, accuracy threshold $p_c$, time threshold $t_c$.

Output: Confidence of current top-k results $\theta$.

1. estimate the parameter of $P$ with $D, \lambda$
2. if computation time $> t_c$ then
3. $\theta = 1 - \int_{\delta}^{\infty} P(x)$
4. return stopped
5. else
6. $x_c = \arg\left(\int_{x_c}^{\infty} P(x) < p_c\right)$
7. if $\delta \geq x_c$ then
8. return stopped
9. else
10. return continued
Chapter 3

Understanding Web Databases: Extracting Query Capabilities From Query Interfaces

3.1 Problem Overview

As “entrance” to Web databases, query interfaces guard the data behind them from direct access. To query Web databases, we thus need to “understand” what an interface says—i.e., what query capabilities a database supports through its interface, in terms of specifiable conditions. For instance, amazon.com (Figure 3.1(a)) supports a set of five conditions (on author, title, . . ., publisher).

Specifically, these interfaces, or HTML query forms, express a set of query conditions for accessing data from databases behind. Each condition, in general, specifies an attribute, one or more supported operators (or modifiers), and a domain of allowed values. A condition is thus a three-tuple [attribute; operators; domain], e.g., \( c_{au} = \) [author; "{first name...", "start...", "exact name"}; text] in interface Qam (Figure 3.1(a)). Users can then use the condition to formulate a specific constraint (e.g., [author = "tom clancy"]) by selecting an operator (e.g., "exact name") and filling in a value (e.g., "tom clancy"). Such query conditions establish the semantic model underlying the Web query interface. This chapter studies this “form understanding” problem: to extract such form semantics.

Automatic capability extraction is critical for large-scale integration. Any mediation task, including MetaQuerier system, generally relies on such source descriptions that characterize sources. Such descriptions, largely constructed by hands today, have been identified as a major obstacle to scale up integration scenarios [?]. For the massive and ever-changing sources on the Web, automatic capability extraction is essential for
Figure 3.1: Query interfaces examples.

many tasks: e.g., to model Web databases by their interfaces, to classify or cluster query interfaces [?], to match query interfaces [?] or to build unified query interfaces.

Such form understanding essentially requires both grouping elements hierarchically and tagging their semantic roles: First, grouping associates semantically related HTML elements into one construct. For instance, \( c_{au} \) in \( Q_{am} \) is a group of 8 elements: a text "author", a textbox, three radio buttons and their associated text's. Such grouping is hierarchical with nested subgroups (e.g., each radio button is first associated with the text to its right, before further grouping). Second, tagging assigns the semantic roles to each element (e.g., in \( c_{au} \), "author" has the role of an attribute, and the textbox an input domain.)

Such extraction is challenging, since query forms are created autonomously. This task seems to be rather “heuristic” in nature: with no clear criteria but only a few fuzzy heuristics – as well as exceptions. First, grouping is hard, because a condition is generally n-ary, with various numbers of elements nested in different ways. ([heuristics]: Pair closest elements by spatial proximity. [exception]: Grouping is often not pairwise.) Second, tagging is also hard – There is no semantic labelling in HTML forms. ([heuristics]: A text element closest to a textbox field is its attribute. [exception]: Such an element can instead be an operator of this or next field.) Finally, with various form designs, their extraction can be inherently confusing – The infamous Florida “butterfly” ballots in US Election 2000 indicates that ill-designed “forms” can be difficult, even for human voters, to simply associate candidates with their punch holes. This incident in fact generated discussions\(^1\) on Web-form designs.

To build a principled solution to this problem that appeared to be heuristic at the first glance, this thesis

\(^1\)e.g., www.larrysworld.com/articles/ups_ballot.htm.
proposes a hidden-syntax based parsing framework. The framework is enabled by our key observation that across myriad sources, query forms seem to reveal some “concerted structure.” They appear to be “modularly” constructed upon a small set of building blocks. Those condition patterns present query conditions in certain visual arrangement—Figure 3.2 shows several examples. For instance, pattern 1 represents a common format for conditions of the form [attribute; {contains}; text], by arranging attribute to the left of a textbox. Such conditions represent keyword search (by an implicit contains operator) on a textual attribute (e.g., author).

To rationalize the phenomenon of concerted-structure, as Figure 3.3 illustrates, we hypothesize that query-form creation is guided by a hypothetical hidden-syntax, which connects semantics to presentations. This hypothesis effectively transforms the problem into a new paradigm: We view query interfaces as a visual language, whose composition conforms to a hidden, i.e., non-prescribed, grammar. Their semantic understanding, as the inverse, is thus a parsing problem.

The following sections will motivate and develop this parsing framework in details. Specifically, we motivates the hidden-syntax based parsing paradigm for form understanding in Section 3.2. We then develop
a grammar mechanism for capturing the hidden syntax in Section 3.3. Coping with the derived grammar that is inherently ambiguous and incomplete, we develop a best-effort parsing algorithm in Section 3.4. We extensively evaluate the effectiveness of the framework in Section 3.5.

3.2 Towards parsing paradigm

This section develops our key insight of the hidden-syntax hypothesis and the approach it enables. We first report our motivating observations (Section 3.2.1), which lead to the key hypothesis (Section 3.2.2). As the hypothesis brings forward, we must also address the challenges entailed by a hypothetical syntax (Section 3.2.3). Overall, our solutions develop a language-parsing framework for building an automatic form extractor, which Section 3.2.4 gives an overview.

3.2.1 Observations: Concerted Structure

As query interfaces are created autonomously, automatic extraction of form semantics is clearly challenging. Is such “form understanding” even possible? As Section 3.1 hinted, there seems to be some common “patterns” emerging from heterogeneous query forms. This impression suggests that Web forms are not entirely chaotic (which, if so, would render automatic extraction unlikely). Consider these patterns as the building blocks, or vocabulary, for constructing query forms. We ask: What is this vocabulary? How large is it?

To answer these puzzles, we performed an informal survey: Using search engines (e.g., google.com) and Web directories (e.g., invisibleweb.com), we collected a total of 150 sources, which we call the Basic
dataset, with 50 in each of Books, Automobiles, and Airfares domains. (Many sources are familiar ones, e.g., amazon.com and aa.com in Figure 3.1.) We chose these domains as they are schematically dissimilar and semantically unrelated– and thus constitute a diverse “sample” of Web sources.

Our survey found that the query interfaces reveal some concerted structure: there are only 25 condition patterns overall– which is surprisingly small as a vocabulary for online queries. Figure 3.4(a) summarizes the occurrences of 21 “more-than-once” patterns. The figure marks \((x, y)\) with a “+” if pattern \(y\) occurs in source \(x\). As more sources are seen (along the \(x\)-axis), the growth (along \(y\)) of the vocabulary slows down and thus the curve flattens rapidly. Further, we observe that the convergence generally spans across different domains (e.g., Automobiles and Airfares are mostly reusing the patterns from Books), which indicates that most condition patterns are quite generic and not domain specific.

Second, we observe that the distribution is extremely non-uniform: Figure 3.4(b) ranks these 21 patterns according to their frequencies, for each domain and overall. We observe a characteristic Zipf-distribution, which means that a small set of top-ranked patterns is very frequently used.

As implications, first, the small and converging vocabulary, which occurs across autonomous sources and even across diverse domains, indicates that there are conventions (or “design patterns”) emerging among Web query forms. While each form is different, together they seem to share a relatively small set of vocabulary. Second, the non-uniform distribution of patterns suggests that, to leverage such conventions, even if we can not exhaustively cover all patterns, a few frequent ones will likely pay off significantly.

3.2.2 Hypothesis: Hidden Syntax

The concerted-structure hints that form understanding can be promising, by leveraging presentation conventions. Intuitively, given a query form, we may build our understanding of it by decomposing into some known patterns, each of which we have seen before– Thus, we assemble an interpretation of an interface unseen before, by the patterns we know of. This “divide-and-conquer” approach seems promising, since we have observed a small vocabulary of such patterns shared across diverse query forms.

While this approach is intuitively appealing, to realize it, what would be a principled computation paradigm? The task seems to be heuristic in nature– To use these layout patterns (as previous works also explored; Section 3.6), it is tempting to “simply” code up each pattern as a rule-of-thumb, e.g., the
pairwise-proximity grouping heuristic (Section 3.1). To specify these patterns, such procedural description will involve convoluted code, lacking both generality and extensibility. Further, to recognize these patterns, it is far from clear, beyond individual heuristics, how they together form a coherent interpretation of the query form.

Toward our insight, we hypothesize the existence of a hidden syntax behind query interfaces, across different sources. This hypothesis rationalizes the observed concerted structure. As Figure 3.3 illustrates, we view query-form creation as guided by such a hypothetical syntax, which connects semantics (i.e., query conditions) to presentations (i.e., query forms). Such a hidden syntax represents the presentation conventions across Web forms. Unlike traditional string languages (e.g., programming languages), this syntax uses visual effects to express the embedded semantics (e.g., pattern 1 in Figure 3.2 arranges the attribute to be left-adjacent and bottom-aligned to the input field).

This hypothesis brings forward a new paradigm: We now view query interfaces as a formal language, and in particular, a visual language, whose composition conforms to a hidden, i.e., non-prescribed, grammar. Their semantic understanding, as the inverse, is thus a parsing problem.

This “language” paradigm further enables a principled algorithmic framework for form understanding—a task that appears inherently heuristic in the beginning. By the hidden-syntax hypothesis, we can now resort to a formal framework for languages: The dual notions of a grammar and a parser together provide a systematic framework for both specifying and recognizing common patterns.

For pattern specification, the grammar provides a declarative mechanism. Such patterns (e.g., Figure 3.2) are simply declared by productions (i.e., grammar rules) that encode their visual characteristics. The specification of patterns is thus declarative, fully separated from and independent of how they are recognized individually and assembled globally—by the parser. It is general: By incorporating arbitrary spatial relations (instead of, say, only proximity), we can describe complex visual patterns. By building productions upon productions, we can describe patterns of different “orders.” It is also extensible: We simply augment the grammar to add new patterns, leaving parsing untouched.

For pattern recognition, the parser provides a global mechanism for systematically constructing a parse tree as a coherent interpretation of the entire query interface. Such a parse naturally structures elements in
nested subtrees, and thus satisfies the grouping requirement (Section 3.1). Further, it assigns grammatical alphabet symbols (terminals and non-terminals) to each construct, and thus satisfies the tagging requirement. Finally, we stress that, such parsing leverages not only individual patterns but also their coherent assembly into an entire query form, and thus resolves local conflicts by a global context. Parsing thus systematically realizes the intuitive “divide-and-conquer” approach.

3.2.3 Challenges: Hypothetical Syntax

As the hidden syntax is hypothetical, while it enables a new paradigm, we must address the challenges it entails: As this hypothetical nature implies, our grammar is non-prescribed: Instead of being prescribed before query forms are created, it is derived from whatever conventions naturally emerge. Further, our grammar is thus secondary to any language instance: Instead of dictating form creation, it relies on the language’s natural convergence to derive any convention. Our challenges are thus two-folded:

First, for capturing this hypothetical syntax, what should such a derived grammar encode? Such a syntax represents “conventions” used for Web form presentation. What types of conventions are necessary to enable parsing? While ideally we want to capture all patterns across many forms, unlike in a carefully-orchestrated grammar, these patterns may not be mutually “compatible.” We must thus rethink the right mechanism for such a derived grammar, to capture necessary conventions for enabling parsing.

Second, for working with a hypothetical syntax, what should be the semantics and machinery for a soft parser? A derived grammar will be inherently incomplete (with uncaptured patterns) and ambiguous (with conflicting patterns). Thus, such a grammar is only secondary to any input: Unlike traditional parsing, our parser cannot reject any input query form, even if not fully parsed, as “illegal.” That is, our parser is no longer a language “police” for checking and enforcing grammar rules. It must now be a “soft” parser that accepts any input. We must thus rethink the right semantics for such a soft parser, and further, its realization.

3.2.4 Solutions: 2P Grammar & Best-effort Parser

Our solutions build upon the traditional language framework, dealing with the specific challenges outlined above. First, as a derived grammar for capturing the hypothetical syntax, the 2P grammar encodes not only “patterns” but also their “precedences.” Second, as a soft-parser directed by the hypothetical syntax,
when a single perfect parse does not exist, the best-effort parser resolves ambiguities as much as possible and constructs parse trees as large as possible.

**2P Grammar:** To capture the hidden syntax, we develop our grammar to encode two complementary types of presentation conventions. On one hand, we must ideally capture all conventional *patterns*. On the other hand, however, by capturing many patterns, some will conflict, and we must also capture their conventional *precedence* (or “priorities”).

Our 2P grammar mechanism, as Definition 3 specifies, encodes both conventions by *productions* and *preferences* respectively (and thus the name). That is, it captures knowledge for both pattern construction (by productions) and ambiguity resolution (by preferences), as Section 3.3 will discuss.

**Definition 3 (2P Grammar):** A 2P grammar is a 5-tuple \( \langle \Sigma, N, s, P_d, P_f \rangle \): \( \Sigma \) is a set of *terminal* symbols. \( N \) is a set of *nonterminal* symbols. \( s \in N \) is a *start* symbol. \( P_d \) is a set of *production* rules. \( P_f \) is a set of *preference* rules.

In our framework, we use this 2P grammar mechanism to express the hypothetical syntax. Such a grammar is to be derived, from analyzing and abstracting common patterns. In our implementation, we use a “global” grammar derived from query interfaces in the Basic dataset– which thus essentially captures the observations that Section 3.2.1 reported. We discuss this grammar and the related issues in Section 3.5.

**Best-effort Parser:** To work with a hypothetical syntax, we develop our parser to perform “best-effort.” As explained earlier, a derived grammar will be inherently ambiguous and incomplete. We thus need a “soft parsing” semantic—The parser will assemble parse trees that may be *multiple* (because of ambiguities)
and partial (because of incompleteness), instead of insisting on a single perfect parse. First, it will prune ambiguities, as much (and as early) as possible, by employing preferences (as in the 2P grammar). Second, it will recognize the structure (by applying productions) of the input form, as much as possible, by maximizing partial results. Our parser pursues this new “philosophy” of best-effort parsing, which Section 3.4 will present.

**Overall Framework– The Form Extractor:** Upon the core components of the 2P grammar and the best-effort parser, we build our form extractor in a language-parsing framework, as Figure 3.5 shows. Given an input HTML query form, the form extractor outputs its semantic model (or the query capabilities) of the form. At the heart, the best-effort parser works with a derived 2P-grammar to construct multiple and partial parse trees. As preprocessing, the tokenizer prepares the input to the core parser, by converting the input HTML form into a set of basic tokens, which are the atomic units in the visual grammatical composition. As post-processing, the merger integrates the output of the parser to generate the final semantic model.

At the front-end, the tokenizer converts an HTML query form (in a Web page) into a set of tokens, each representing an atomic visual element on the form. Note that these tokens are instances of the terminals $\Sigma$ as the 2P grammar defines (Definition 3). Each token thus has a terminal type and some attributes recording properties necessary for parsing. For instance, given the HTML fragment (as part of interface $Q_{am}$), as Figure 3.6 shows, the tokenizer extracts a set $T$ of 16 tokens. In particular, token $s_0$ is a text terminal, with attributes $sval = "$\text{Author}$"$ (its string value) and $pos = (10, 40, 10, 20)$ (its bounding-box coordinates). Although different terminals have different attributes, this $pos$ attribute is universal, as our grammar captures two dimensional layout. Such a tokenizer builds on a layout engine for rendering HTML into its visual presentation. In particular, our tokenizer uses the HTML DOM API (available in browsers, e.g., Internet Explorer), which provides access to HTML elements and their positions.

At the back-end, the merger combines the multiple partial parse trees that the parser outputs, to compile the semantic model and report potential errors (if any). Since our parser is rather generic, this step applies application (i.e., query form) specific processing. First, since our goal is to identify all the query conditions, the merger combines multiple parse trees by taking a union of their extracted conditions. As each parse covers different parts of the form, this union enhances the coverage of the final model constructed. For
example, given a fragment of interface $Q_{aa}$, as Figure 3.14 shows, the parser will generate three partial parses (trees 2, 3, 4 in the figure) (Section 3.4.3). Their union covers the entire interface and generates all the conditions.

The merger also reports errors, which will be useful for further error handling by the “client” of the form extractor. It reports two types of errors: First, a conflict occurs if the same token is used by different conditions. In Figure 3.14, tree 2 associates the number selection list with number of passengers, while tree 3 with adults and thus they conflict by competing for the number selection. (In this case, tree 3 is the correct association.) Second, a missing element is a token not covered by any parse tree. The merger reports both types of errors for further client-side handling.

In summary, for automatic form understanding, we make the key hypothesis of hidden syntax and thus pursue a parsing framework for the form extractor. As the focus of this chapter, we will now concentrate on the dual cores of this framework– the 2P grammar and the best-effort parser.

### 3.3 2P Grammar

As the key component in the parsing framework, 2P grammar captures presentation conventions of Web interfaces. Specifically, the 2P grammar declaratively and comprehensively specifies both condition patterns and their precedences, as a principled way to express a derived syntax and to resolve potential ambiguities. In particular, productions formally specify common condition patterns and preferences their relative precedence,
as Section 3.3.1 and Section 3.3.2 will present respectively.

### 3.3.1 Productions: Capturing Patterns

Since the condition patterns establish a small set of building blocks for query forms, we need to explore appropriate presentational characteristics to capture those condition patterns as productions. In particular, in query interfaces, visual effects such as *topology* (e.g., alignment, adjacency) and *proximity* (e.g., closeness) are frequently used for expressing semantically related components and thus are the candidates to be captured by productions. We find that some features such as proximity work well for simple interfaces; however, it is hard to extend to complex interfaces and can often result in incorrect interpretations. On the other hand, we observe that the topology features such as alignment and adjacency (e.g., *left*, *above*) accurately indicate the semantic relationships among the components in query interfaces. Therefore, in the 2P grammar, we extensively explore such topological information, in the productions, to capture condition patterns.

Many two-dimensional grammars have been proposed in visual languages to realize such specifications of visual patterns, e.g., relational grammar [?], constraint multiset grammar [?], positional grammar [?]. Our 2P grammar (without considering the preferences), is a special instance of attributed multiset grammar [?], where a set of spatial relations capturing topological information (e.g., *left*, *right*) is used in productions.

The main extension of two dimensional grammars from string grammars (e.g., for programming languages) is to support general constraints. In two dimensional grammars, productions need to capture spatial relations, which essentially are constraints to be verified on the constructs. For example, consider production $P_5$ in Figure 3.7. To capture the pattern $\text{TextOp}$ (used by $\text{author}$ in interface $Q_{am}$), we specify that $\text{Attr}$ is *left* to $\text{Val}$ and $\text{Op}$ *below* to $\text{Val}$. (Note that, in the 2P Grammar, *adjacency* is implied in all spatial relations and thus omitted in the constraint names). In contrast, productions in string grammars only use one constraint, the *sequentiality*, among components.

As a consequence, such extension leads to adaptations in other aspects of the productions. Specifically, to support the general constraints, each symbol has a set of attributes (e.g., $\text{pos}$ of $\text{Attr}$, $\text{Op}$ and $\text{Val}$), which store the information used in constraints evaluation (e.g., *left*, *below*). Further, each production has a *constructor*, which defines how to instantiate an *instance* of the head symbol from the components. For example, after applying the production $P_5$ to generate a new $\text{TextOp}$ instance $i$, the constructor computes
Figure 3.7: Productions of the 2P grammar.

\textit{i}’s position from its components. Formally, we define the production as:

**Definition 4 (Production):** A production $P$ in a 2P grammar $G = \langle \Sigma, N, s, P_d, P_f \rangle$ is a four-tuple $\langle H, M, C, F \rangle$: Head $H \in N$ is a nonterminal symbol. Components $M \subseteq \Sigma \cup N$ is a multiset of symbols. Constraint $C$ is a boolean expression defined on $M$. Constructor $F$ is a function defined on $M$, returning an instance of $H$.

**Example 2 (Grammar $G$):** We show a simple grammar $G$ (without the component $P_r$) that we will use in later explanation. As Figure 3.7 shows, grammar $G$ specifies 11 productions labeled from $P_1$ to $P_{11}$. Each production defines a nonterminal (e.g., TextOp and EnumRB) as its head. The start symbol is QI and the terminal symbols are text, textbox and radiobutton. Note that, to simplify the illustration, we omit the production constructors in Figure 3.7.

Productions $P_3$ to $P_{11}$ capture three patterns (patterns 1 and 2 in Figure 3.1(c) in addition to TextOp introduced above). Productions $P_1$ and $P_2$ capture the form pattern by which condition patterns are arranged into query interfaces. In particular, we consider a query interface QI as composing of vertically aligned “rows” HQI, where each HQI further composes of horizontally aligned condition patterns CP.

Our productions provide a quite general and extensible mechanism for describing patterns. First, it can express patterns of different “orders”: Complex patterns are built upon simpler ones. For example, pattern TextOp is constructed from simpler patterns Attr, Op and Val, and in turn serves as the basis to higher order patterns such as QI. Second, it is very extensible to incorporate new patterns and new constraints, leaving
the parsing algorithm untouched.

### 3.3.2 Preferences: Capturing Precedence

For derived grammars, precedences are essential for resolving the conflicts among patterns and thus an integral component of the 2P grammar. While our grammar intends to capture as many common (but non-prescribed) patterns as possible, those patterns may not be “compatible,” which result in significant ambiguities (Section 3.3.2). To resolve those ambiguities, we explore a preference framework, which captures the conventional precedences among condition patterns (Section 3.3.2).

**Inherent ambiguities**

Ambiguity happens when there exist multiple interpretations for the same token, and therefore these interpretations conflict on such a token. Example 3 shows an example of a conflicting situation.

**Example 3 (Ambiguity):** To capture the condition pattern TextVal used by from condition in $Q_{aa}$ and pattern RBU used in $Q_{am}$, we define productions $P_4$ and $P_9$ respectively. However, such generality brings ambiguities, allowing a token to be interpreted differently by different patterns. Consider the text token $s_1$ (i.e., “first name/initial and last name”) in Figure 3.6. Patterns TextVal ($P_4$) and RBU ($P_9$) have different interpretations on $s_1$, as Figure 3.8 shows. In particular, TextVal interprets it as an Attr instance A1 in a TextVal instance I1 (Figure 3.8(a)). In contrast, RBU interprets it as the text of an RBU instance I2 (Figure 3.8(b)). Since conflicting on $s_1$, I1 and I2 cannot appear in the same parse tree.

The existence of ambiguities may cause parsing inefficient and inaccurate. It is inefficient because of local ambiguities: The parser may generate “temporary instances” that will not appear in any complete parse tree. An ambiguity between two instances is local if at least one of them is a temporary instance. In
Example 3, \(I_1\) is a temporary instance, since we cannot further derive a complete parse tree from \(I_1\). In contrast, we can derive complete parse trees from \(I_2\) (as Figure 3.10 shows two). Hence, such an ambiguity is *local* because it will eventually be resolved at the end of parsing. As we will show in Section 3.4, the parsers used in visual language generally follow a bottom-up exhaustive approach which explores all possible interpretations. Therefore, the existence of local ambiguities makes parsing very inefficient due to the generation of many “temporary instances.”

In contrast, *global* ambiguities make parsing results inaccurate: The parser may generate more parse trees than the semantically correct one. An ambiguity between two instances is *global* if they lead into different parse trees, and thus cannot be resolved even at the end of parsing.

**Example 4 (Global ambiguity):** To capture radio button lists of arbitrary length, production \(P_{8}\) is defined in a recursive way. As a result, a radio button list of length three can have four interpretations, depending on how elements are grouped. Figure 3.9 shows such two—(a) as a single list or (b) as three individual lists with each of length one. The ambiguity between these two interpretations is global, because they eventually lead to two different parse trees, as Figure 3.10 shows. The first one takes the entire list as an operator of \texttt{author}, while the second takes each list (of length one) as a condition of pattern \texttt{EnumRB}.

The effect of the inherent ambiguities is significant. For instance, the simple query interface in Figure 3.6 has one correct parse tree containing 42 instances (26 non-terminals and 16 terminals). However, applying the basic parsing approach (to be discussed in Section 3.4) that exhausts all possible interpretations by “brute-force,” we get 25 parse trees and totally 773 instances (645 temporary instances and 128 non temporary ones). The reason to have such a significant amount of ambiguities is that conflicting instances may further
participate in generating other instances, which in turn conflict. Such exponential aggregation of conflicts makes ambiguity a significant problem in parsing.

Preference

To resolve the ambiguities among condition patterns, it is essential for a derived grammar to prioritize these patterns. The derived nature of our hidden syntax implies that such precedence comes from “hidden priority conventions” across patterns. In predefined grammars, the creation of a grammar is prior to that of the corresponding language, and therefore how to resolve ambiguities is determined apriori. However, in derived grammars, the precedence itself is a part of conventions to be derived from the language, and thus cannot be designed arbitrarily. In this thesis, we propose to explore preference to encode such conventional precedence across patterns.

Example 5 (Preference): Consider the two conflicting instances, A1 and I2, in Example 3. Is there any convention that indicates which one is better? We observe that text and its preceding radio button are usually tightly bounded together. Therefore when conflicting, I2 is more likely to have a higher priority than A1. Such a convention of the precedence between the two patterns establishes our knowledge to resolve the ambiguity. In particular, we encode such a precedence convention as a “preference” $R_1$: When an RBU instance and an Attr instance conflict on a text token, we arbitrate unconditionally the former as the winner.

In general, a convention may also carry a criterion for picking the winner: For example, for the ambiguity in Example 4, we observe that a row of radio buttons is usually used as a single longer list rather than separate shorter ones. Therefore, we define a preference $R_2$: When two RBList instances conflict, and if one subsumes the other, we pick the longer one as the winner.
Specifically, each preference resolves a particular ambiguity between two types of conflicting instances by giving priority to one over the other. As Example 5 motivates, such a preference needs to specify the situation and the resolution. The situation indicates the types of conflicting instances (e.g., \textit{RBlist} in preference \textit{R\textsubscript{2}}) and the conflicting condition (e.g., \textit{subsume}). The resolution describes the criteria that the winner instance should satisfy (e.g., \textit{longer}). Formally, we define the preference as:

**Definition 5 (Preference):** A Preference \( R \) in a 2P grammar \( G = \langle \Sigma, N, s, P_d, P_f \rangle \) is a three-tuple \( \langle I, U, W \rangle \):

- **Conflicting instances** \( I = \langle v_1 : A, v_2 : B \rangle \), where \( A, B \in T \cup \Sigma \), identify the types of instances \( v_1 \) and \( v_2 \) respectively.
- **Conflicting condition** \( U \): a boolean expression on \( v_1, v_2 \) that specifies a conflicting situation to be handled.
- **Winning criteria** \( W \): a boolean expression on \( v_1, v_2 \) that specifies the criteria to pick \( v_1 \) as winner.

There are several advantages of using preferences to resolve ambiguities. First, as the specification of precedences, preferences are simple and effective mechanism to encode the precedence conventions deterministically. Such simplicity helps the parser efficiently resolve ambiguities, compared with other approaches, e.g., probabilistic model for enabling precedence. Although our preferences only encode a “flat set” of precedences and thus may be nondeterministic when preferences themselves conflict, in practice we never have such a situation, because semantically meaningful preferences are consistent.

Second, as a mechanism for ambiguity resolution, preferences are particularly suitable to derived grammars. Traditional techniques, such as \textit{lookahead prediction} \[?] and \textit{grammar transformation} \[?], impose significant restrictions on the productions, which are difficult to meet for a derived grammar. In contrast, preferences can be specified independently from the productions without any specific constraint on the grammar. Further, preferences uniformly deal with both local and global ambiguities by favoring promising interpretations. Such uniform treatment is especially desirable for a derived grammar because its potential incompleteness blurs the distinction between local and global ambiguities.

### 3.4 Best-effort parser

With 2P grammar capturing the conventions of condition patterns and their precedences, this section presents a best-effort parsing algorithm that on one hand makes use of preferences to prune the wrong interpretations...
in a timely fashion, and on the other hand handles partial results to achieve maximum interpretations for the input. We start with outlining the best-effort parsing algorithm 2PParser (Section 3.4.1), and then zoom into two essential components: just-in-time pruning (Section 3.4.2) and partial tree maximization (Section 3.4.3).

### 3.4.1 Overview

With potential ambiguities and incompleteness, our best effort parser operates on a basic framework, the fix-point evaluation [?, ?, ?], that progressively and concurrently develops multiple parse trees. The essential idea is to continuously generate new instances by applying productions until reaching a fix-point when no new instance can be generated. For example, as Figure 3.11 conceptually shows, the parser starts from a set of tokens $T$ (Figure 3.6), iteratively constructs new instances and finally outputs parse trees. In particular, by applying the production $P9$, we can generate an RBU instance from the text token $s_1$ and radiobutton $r_1$. Further, with the production $P8$, the RBUs in a row together generate an RBList instance. Continuing this process, we eventually reach the fix-point. A complete parse tree corresponds to an instance of the start symbol QI that covers all tokens, as Figure 3.11 conceptually shows one. However, due to the potential ambiguities and incompleteness, the parser may not derive any complete parse tree and only end up with multiple partial parse trees.

Upon this framework, we realize the “best-effort” philosophy by essentially: First, just-in-time pruning to prune the parse trees with wrong interpretations as much and as early as possible; Second, partial tree maximization to favor the parse trees that interpret the input as much as possible.

Algorithm 3.4.1 shows the best-effort parsing algorithm 2PParser. Corresponding to the above two
components, the algorithm has two phases: parse construction with just-in-time pruning and partial tree maximization at the end of parsing. To achieve just-in-time pruning, we schedule the symbols (by procedure BldSchdGraph) in a proper order so that false instances are pruned timely before further causing more ambiguities. According to the scheduled order, we instantiate the symbols one by one with a fix-point process (by instantiate). Preferences are enforced at the end of each iteration (by enforce) to detect and remove the false instances in this round. When an instance is invalidated, we need to erase its negative effect: false instances may participate in further instantiations and in turn generate more false parent instances. Procedure rollback is used to remove all those false ancestors to avoid further ambiguity aggregation. Finally, after parse construction phase, PRHandler chooses the maximum parse trees generated in the parse construction phase and outputs them.

Inherently, visual language parsing is a “hard” problem. The complexity of the membership problem (i.e., given grammar $G$, a sentence $S$, to determine whether $S \in L(G)$) for visual languages is NP-complete \cite{?}. Therefore, the algorithm runs in exponential time with respect to the number of tokens. However, in practice, the use of preferences gives reasonably good performance. Our implementation\footnote{The experiment is conducted with a Pentium IV 1.8GHz PC with 512MB RAM.} shows that, given a query interface of size about 25 (number of tokens), parsing takes about 1 second. Parsing 120 query interfaces with average size 22 takes less than 100 seconds. (The time measured here only includes the parsing time without tokenization and merger.)

In the next two sections, we zoom into the just-in-time pruning technique for ambiguity resolution and the partial tree maximization technique for partial results handling in more details.

### 3.4.2 Just-in-time Pruning

To prune false instances as much and as early as possible, we need to find a good timing for enforcing the preferences. Such timing ideally would guarantee that any false instance is removed before participating in further instantiations, and therefore no rollback is necessary. However, applying preferences whenever a new instance is generated in the basic fix-point algorithm cannot achieve so.

**Example 6:** With preference $R_1$ (defined in Example 5) which resolves the local ambiguity in Example 3, the Attr instance $A_1$ should be removed by the RBU instance $I_2$. However, what if $A_1$ is generated at the
Algorithm 3.4.1 Best-effort parser for 2P grammar.

Proc 2PParser(TS, G):
Input: Token set TS, grammar G
Output: Maximum partial trees res
1: \[ Y = \text{BldSclldGraph}(G) \]
2: find a topological order of symbols in \( Y \)
3: for each symbol \( A \) in order: do
4: \( I \) += instantiate\((A)\)
5: for each preference \( R \) involving \( A \): do
6: \( F = \text{enforce}(R) \)
7: for each invalidated instance \( i \in F \) do
8: Rollback\((i)\)
9: \( res = \text{PRHandler}() \)

Proc instantiate\((A)\):
Input: Symbol \( A \)
Output: Instances of \( A \) inst
\[ \text{inst} = \emptyset \]
repeat
    for each production \( p \) with head being \( A \): do
        \( \text{inst} += \text{apply}(p) \)
until there is new instance added into \( \text{inst} \)

very beginning of parsing, while \( I_2 \) at the end? \( A_1 \) will still instantiate instance \( I_1 \) (and possibly others), and only be removed at the end of parsing (when \( I_2 \) is generated). This “late pruning” makes preference \( R_1 \) ineffective in preventing ambiguity aggregation.

To address this problem, we want to generate the winner instance (e.g., \( I_2 \)) before the loser (e.g., \( A_1 \)) so that the loser can be detected and pruned whenever it is generated. Essentially, we want to schedule the instance generation in some desired order consistent with preferences. As preferences are defined on symbols, to guarantee the order on particular instances, we enforce such an order on symbols so that the winner symbol produces all its instances before the loser does. Therefore, such symbol-by-symbol instantiation and winner-then-loser order can guarantee that instances are produced in a desired order to ensure just-in-time pruning.

To realize the symbol-by-symbol instantiation, symbols have to be processed in a “children-parent” direction defined by productions. For example, consider symbol TextOp, as production \( P_5 \) defines. The symbols that contribute to the instantiation of TextOp are Attr, Op and Val. Therefore, before we can process TextOp, those children symbols must be processed first. Further, to realize the winner-then-loser order, the winner symbol (e.g., RBU in Example 6) must be scheduled before the loser (e.g., Attr).
To schedule the symbols by the above two ordering criteria, we build a 2P schedule graph. The graph consists of the symbols as nodes and two types of edges—d-edges to capture the “children-parent” order defined by productions and r-edges to capture the winner-then-loser order defined by preferences.

**Example 7 (2P schedule graph $Y$):** Figure 3.12(c) shows the 2P schedule graph $Y$ for the Grammar $G$ (defined in Example 2), by merging d-edges (Figure 3.12(a)) and r-edges (Figure 3.12(b)). $Y$ has a d-edge $A \rightarrow B$ if the grammar has a production with head symbol $A$ and a component symbol $B$ (i.e., $A$ is a parent of $B$). $Y$ has an r-edge $C \rightarrow D$ if the grammar has a preference $D$ over $C$ (i.e., $D$ is the winner and $C$ is the loser). We omit the self-cycles because they do not affect the scheduling. (More precisely, we also omit the terminals, as they do not affect the schedulability in this example.) By merging these two types of edges, we get the 2P schedule graph $Y$, with solid edges denoting d-edges and dashed r-edges.

By enforcing a topological order on symbol instantiations, this 2P schedule graph captures the two requirements needed for just-in-time pruning. If the graph is acyclic, any topological order achieves such a goal. For example, as our schedule graph $Y$ (Example 7) is acyclic, we schedule RBU before Attr. Thus, instance $I_2$ is generated before $A_1$, which then is pruned promptly. More precisely, as preferences are enforced at the end of each symbol instantiation to avoid repeated calls for every instance, ambiguities may aggregate during the instantiation of a particular symbol. Such aggregation is localized to a single symbol and thus is rather insignificant.

However, a 2P schedule graph may be cyclic. For example, suppose we have two symbols $B$ and $C$, which share a construct $A$, as Figure 3.13 illustrates. Let $a$, $b$ and $c$ denote instances of $A$, $B$ and $C$ respectively.
Instances $b$ and $c$ may potentially conflict on $a$. To resolve the ambiguity, suppose we define two preferences $R_{C \rightarrow B}$ and $R_{B \rightarrow C}$. The former specifies that we prefer $b$ to $c$ if, say, the inter-component distance of $b$ is smaller than that of $c$. Similarly, the later states that we prefer $c$ to $b$ if such a distance of $c$ is smaller than that of $b$. In other words, the two preferences define that the winner instance, which can be either $b$ or $c$, is the one with smaller inter-component distance. The two $r$-edges thus form a cycle $C \rightarrow B \rightarrow C$.

When the graph is cyclic, there is no such an order that satisfies both scheduling requirements (i.e., the symbol-by-symbol and winner-then-loser orders). Therefore, some edges have to be “relaxed.” A $d$-edge must be enforced because symbol-by-symbol instantiation is the prerequisite of winner-then-loser order. (This implies that we require the $d$-edges form an acyclic graph.) In contrast, since an $r$-edge is to enhance efficiency by removing false instances early, it is mainly an “optimization.” If we relax the ordering by removing an $r$-edge, the negative effect is simply to introduce more false instances. Such negative effect can be erased by the rollback step, although it incurs certain overhead. In fact, removal of $r$-edges is not the only way of relaxation. Transformation, as we will discuss next, may relax an $r$-edge without causing the negative effect of ambiguity aggregation.

Is it possible to relax an $r$-edge while still achieving just-in-time pruning? The answer is yes, because the requirement imposed by an $r$-edge is sufficient but not necessary to achieve our goal. Consider the $r$-edge $C \rightarrow B$, assuming we remove this $r$-edge to break the cycle. We notice that if we can schedule $B$ before $C$’s parent symbol $D$, the winner $B$ can still prevent the loser $C$ from further generating false instances of $D$. 

Figure 3.13: Transformation of an $r$-edge.
Therefore, as Figure 3.13 illustrates, in case of cycle, we transform an original \( r \)-edge \( C \rightarrow B \) to a set of indirect \( r \)-edges by, for each \( C \)'s parent \( D \) (defined by a \( d \)-edge), adding an indirect \( r \)-edge \( D \rightarrow B \). By doing so, the cycle in Figure 3.13(a) is broken. We thus have a scheduling, e.g., \( E, F, A, C, B, D \), that achieves just-in-time pruning. In this schedule, since \( C \) is scheduled before \( B \), any false instance of \( B \) is pruned promptly at generation. On the other hand, although \( C \) is scheduled before \( B \), the false instances of \( C \) are removed when the winner instances of \( B \) are generated, which is still before any instance of \( D \) is generated. Therefore, by transformation, we achieve just-in-time pruning. Although such transformation cannot guarantee to break all cycles in a 2P schedule graph, in practice, it is very effective to handle all the cycles in our 2P grammar.

If transformations still cannot break cycles, we have to remove \( r \)-edges and use rollback to offset the negative effects caused by late pruning. As cycles are rare situations, we employ a greedy algorithm in building 2P schedule graph to avoid cycles at first place: we add \( r \)-edges one by one, and if an \( r \)-edge causes cycle even after transformation, we simply remove it. When an \( r \)-edge is removed, a false instance may generate its parents. We then call rollback to erase those false ancestors caused by late pruning.

### 3.4.3 Partial Tree Maximization

Last section discussed how just-in-time pruning addresses inherent ambiguities of a grammar, in this section we briefly discuss how to handle partial parse trees. The parsing algorithm generates partial parse trees when the grammar is incomplete to interpret an entire query interface.

Specifically, partial parse trees are the derivation trees that cover a subset of tokens and cannot be expanded further. When a query interface contains new condition patterns not captured by the 2P grammar, the parse construction will stop at those partial trees, not being able to further assemble more tokens. For example, consider the query interface in Figure 3.14, which is a variation from the interface \( Q_{aa} \). Grammar \( G \) does not completely capture the form patterns of that interface: The circled part is arranged “column by column” instead of “row by row.” Therefore, the parse construction generates only partial parses, as Figure 3.14 shows four of them.

To maximize the understanding of query interfaces, our parser favors the maximal partial trees that interpret as many tokens as possible. In particular, we use maximum subsumption to choose parse trees that
assemble a maximal set of tokens not subsumed by any other parse. For example, Tree 1 in Figure 3.14 is not maximal because the tokens covered by Tree 1 is subsumed by those of Tree 2. The other three, although overlapping, do not subsume each other. (It is straightforward to see that a complete parse tree is a special case of maximal partial trees.) In addition to maximizing the interpretations, such maximal parse trees also potentially achieve better interpretations, since they are looking at larger context compared with those non-maximal ones.

### 3.5 Experiments

As we expect the form extractor to be used in two settings: global setting and local setting, we design two sets of experiments - global study and local study to evaluate the issues in different settings.

First, in the global setting, the form extractor integrates with a general-purpose “crawler” to extract query interfaces collected from arbitrary domains (e.g., to build a directory of Web databases.) Therefore, the goal of global study is to evaluate whether a “globally” fixed grammar can achieve good performance for interfaces from arbitrary domains (Section 3.5.1).
Second, in the local setting, the form extractor is tailored to understand query interfaces in specific
domains (e.g., to integrate all query interfaces from Airfares.) In particular, domain experts may design
grammars specifically for such a domain. Therefore, the goal of local study is to understand the design issues
(e.g., the requirement of sample sources to observe) for grammar derivation (Section 3.5.2).

Taking a different view, the two settings represent two dual angles for evaluating the design of a grammar
and its performance. On one hand, the global setting studies how good a well-crafted grammar can perform;
On the other hand, the local setting suggests the requirement of acquiring a grammar to achieve good
performance.

For the experiments, we use 4 datasets, collected using Web directories (e.g., invisibleweb.com, bright-
planet.com) and search engines (e.g., google.com).

- **Basic** dataset, which we used in our motivating survey (as Section 3.2 reported), contains 150 sources in
three domains: Airfares, Automobiles, and Books.

- **NewSource** dataset contains 10 extra query interfaces from each of the above domains, with a total of 30
sources.

- **NewDomain** dataset contains query interfaces from 6 different domains other than the above three. Each
domain has 7 sources with a total of 42 sources.

- **Random** dataset contains 30 query interfaces that we randomly sampled from the Web. In particular, we
choose invisible-web.net as our pool of sampling, which contains about 1,000 manually complied deep Web
sources. As all the sources are sequentially numbered in the directory, we thus can easily draw random
samples using the source ID. The sampled sources cover 16 out of 18 top level domains in the directory.

**Metrics:** Since the goal of our form extractor is to extract the query capabilities for each source, as a *set*
of supported query conditions, we adopt *precision* and *recall* as our measurement of performance. For each
query interface, we manually extract the set of conditions in its semantic model and compare with the ones
extracted by the form extractor. We measure the results in two ways - per-source metric and overall metric.

- Per-source metric measures the result for each source. Given interface $q$, let $C_s(q)$ denote the set of
conditions in its semantic model, $E_s(q)$ extracted conditions by the form extractor, then formula $P_s(q)$
and $R_s(q)$ calculate the precision and recall for the interface $q$. 

$$P_s(q) = \frac{C_s(q) \cap E_s(q)}{E_s(q)}, \quad R_s(q) = \frac{C_s(q) \cap E_s(q)}{C_s(q)}$$

- Overall metric measures, given a set of query interfaces $w$, the precision and recall over all conditions aggregated in those sources. Let $C_a(w)$ denote all conditions in $w$, $E_a(w)$ extracted conditions, then the overall precision and recall over $w$ are defined as:

$$P_a(w) = \frac{C_a(w) \cap E_a(w)}{E_a(w)}, \quad R_a(w) = \frac{C_a(w) \cap E_a(w)}{C_a(w)}$$

This overall metric essentially normalizes the precision and recall according to the size of query interfaces.
3.5.1 Global Study

The global study intends to evaluate whether a single global grammar can be used for arbitrary domains. The grammar used in the experiment is derived from the Basic dataset. We manually observe the 150 query interfaces in the dataset, and summarize 21 most commonly used patterns. The derived grammar has 82 productions with 39 nonterminals and 16 terminals.

We test the derived grammar against 4 datasets. The first one, as a baseline comparison, evaluates the performance over sources from which the grammar is derived. The other three test how well such a single grammar can perform over new sources (NewSource dataset), new domains (NewDomain dataset) and even random sources (Random dataset) from highly heterogeneous domains.

Figure 3.15(a) and (b) show the distribution of precision and recall over sources for 4 datasets. For instance, in the Basic dataset, 69% sources have precision 1.0, and 72% sources have 1.0 recall. Figure 3.16(a) shows average precision and recall of sources in 4 datasets, and (b) the overall precision and recall for 4 datasets. As we can see, for the Random dataset, the overall precision $P_a$(Random) achieves 0.8 and recall $R_a$(Random) 0.89.

An interesting thing we observed is that the result from the NewSource dataset has best performance, which is even better than that of the Basic dataset. The reason may be that we may have some bias during the collection of data in the survey: We tend to collect complex forms with many conditions, since we want to know how complex query interfaces can be. However during the collection of the new sources, we are more random. Therefore those query interfaces turn out to be simpler and show better accuracy.

The results from the set of experiments show that a single grammar can achieve reasonable good performance across heterogeneous domains. The performance over 4 datasets are rather even, and we do not observe significant performance drop when extending to more heterogenous sources - about 0.85 of overall precision and recall for first three datasets, and over 0.80 for random sampled sources.

3.5.2 Local Study

The local study intends to address the design issues in customizing the form extractor to specific domains. As we stressed, the declarative nature of the grammar mechanism makes such customization easy: We only need
to design a grammar for a specific domain leaving the parsing framework unchanged. An important issue in the customization is how many samples we should draw to derive a grammar that achieves good performance. Therefore, the experiments in the local study studies the impact of sample size to the performance, and the trend of grammar growth over the sample size.

We conduct the performance study with the Basic dataset as it is domain specific and contains more interfaces than others. For each domain in the dataset, we randomly selected a set of 40 sources, for which we measure the overall performance using different grammars. The grammars are designed from samples of different sizes: We draw 4 samples from the interface collection - first 10 sources, 20 source, 30 sources and 40 sources respectively. We design each grammar specifically to only capture the condition patterns appeared in the sample. The experiment shows how the performance changes with respect to the grammars derived from the different samples.

Further, we examined how significant a grammar may change when new sources are incrementally observed in the samples. In particular, we examined the number of productions and flattened productions. The former approximates the number of basic constructs defined by a grammar factoring out the internal layout. For example, in condition pattern TextVal, we use only one production to capture 4 different internal layouts of Attribute and Val, thus count only once. Number of flattened productions then tries to take internal layout into consideration so that each one only captures a construct with specific components in a particular layout. Therefore, 4 flattened productions are counted for the above example. The increase of productions often indicates an emergence of totally new patterns, while the increase of flattened productions also includes appearance of a new layout with the same components.

Figure 3.17(a) and (b) show the convergence of overall precision $P_a(w)$ and recall $R_a(w)$ over the sample size, and Figure 3.18 (a) and (b) the increase of grammar size in terms of productions and flattened productions respectively.

The results of the experiments show that the system can achieve good performance even with small samples of query interfaces. Even only with 10 samples, all three domains reach above 0.85 overall precision and recall over 40 sources. Further, the performance stabilizes quickly as more samples are drawn. For example, in Automobiles, the performance gain from sample size 10 to 20 is about 6%, from 20 to 30 is
2.6%, and from 30 to 40 is only 1.2%.

The results on the size of grammars show that there are only very small growth in both productions and flattened productions as more query interfaces are observed, and such growth is converged. For example, in Books domain, 5 new productions are introduced for grammar 2 (with 20 sources) from grammar 1 (with 10 sources), and only 1 is added incrementally for grammar 3 and 4. Therefore, the grammar stabilizes very fast.

3.6 Related work

While understanding query interfaces automatically is an important problem for many applications (e.g., integration of Web databases), the efforts investigated are very limited. In this section, we relate our work to others mainly from two aspects: works addressing the same problem of interface understanding and works
with similar insight of exploiting structure regularity.

First, the problem of understanding query interfaces has been studied quite extensively. But to the best of our knowledge, we are the first to study this problem formally and specifically. The problem is mentioned, but not as a focus, in several early works [?, ?, ?]. Some works that rely on automatic form filling [?, ?, ?] either only deal with simple keyword query forms or make use of only selection lists for easy interaction. In particular, references [?] and [?] propose to use simple heuristics such as proximity and alignment to associate pairwise elements and texts in a form. Further, later work [?] proposes more sophisticated interface model and uses many heuristic rules to realize the extraction. In contrast, we explore a parsing framework with a hidden syntax to derive a global interpretation for the input, which can generally capture not only complex compositions but also sophisticated features other than proximity or alignment.

Second, there are many other works that also exploit the structural regularities for semantic understanding. In particular, wrapper induction [?, ?, ?], which addresses extracting the underlying structures of query result pages, relies on the regularities of such structures. Although we focus on a different type of Web artifacts, i.e., query interfaces, our work does share similar insight with those works—That is, there exists structure regularity underlying Web artifacts. In particular, the works done by Crescenzi [?] and Arasu [?] are closest to ours in that we both view Web pages as generated from a grammar. However, in their settings, the existence of a grammar is not an assumption but reality, because the collection of Web pages studied is indeed homogeneously generated from the same underlying template. In our problem, the syntax is only hypothetical based on the observations of heterogeneous query interfaces. Consequently, their works focus on the derivation of an actual grammar, while ours on the development of a parsing framework with hypothetical syntax.

Finally, by abstracting query interfaces as a visual language generated from a hidden syntax, we are able to leverage the formalisms [?, ?, ?] and techniques [?, ?, ?] developed in visual languages, while still develop our own strategy to address specific challenges of hypothetical grammars, as we have discussed in this chapter.
Chapter 4

Querying Web Databases: Translating Queries Across Heterogenous Databases

4.1 Problem Overview

The previous chapter studied form extractor for automatic understanding of query forms, this chapter further presents our form assistant, which helps querying Web databases with heterogeneous query capabilities.

With the proliferation of sources in various domains, we often need to query “alternative” sources in the same domain (e.g., Books, Airfares). We observe that such a domain-based integration scenario is useful in broad applications. For instance, we may build our MetaQuerier to integrate dynamically selected sources relevant to user’s queries, where on-the-fly translation of user’s queries to these sources is necessary; we may build a domain portal (e.g., pricegrabber.com) to provide a unified access to dynamic online sources in the same domain with general translation techniques; or we may build a query assistant toolkit to suggest users with potential queries they are likely to issue in query forms, e.g., if a user fills the query form in amazon.com, the toolkit can suggest potential queries in bn.com. (Section 4.6 will discuss the implementation of such a query assistant toolkit.)

A core component of these applications is a dynamic query translator, which translates user’s queries between dynamically selected query forms in the same domain. In particular, we define the query translation problem as translating a user’s query from a source query form to a target one, which we believe is a foundation to many translation tasks in various applications. (Section 4.2 will present our formal definition of the query translation problem.) Existing works mainly focus on isolated subproblems of translation (e.g.,
As the applications mandate, such a query translator should have two properties: First, source-generality: We require the built-in translation techniques can generally cope with new or “unseen” sources. Second, domain-portability: We require the translator can be easily customized with domain-specific knowledge and thus deployed for new domains. As a realization of such a query translator, we develop a light-weight domain-based form assistant. It is domain-based because it can generally handle alternative sources in the same domain with a manageable size of domain-specific knowledge. It is light-weight because customizing to a new domain only needs a small amount of efforts to encode domain-specific knowledge. To better understand new challenges we are facing in building this form assistant, let us look at a translation example.

**Example 8:** Assume we want to build a form assistant for Books domain, which can translate queries between any two book sources. In particular, consider a translation from a source query $Q_s$ (issued on a source form $S$) to a target form $T$ in Figure 4.1. $Q_s$ is a conjunctive query over four predicates $s_1 : [\text{author contain Tom Clancy}]$, $s_2 : [\text{title contain red storm}]$, $s_3 : [\text{age} > 12]$, and $s_4 : [\text{price} \leq 35]$, i.e., $Q_s = s_1 \land s_2 \land s_3 \land s_4$. The target form $T$ supports predicate templates on author, title, subject, ISBN one at a time with an optional predicate template on price. Figure 4.1 shows one of the possible translations.
Note that, our discussion in this chapter needs to differentiate condition templates from the concrete instantiations of such templates. Specifically, we use term **predicate template** to refer to a template of conditions and **predicate** to refer to its concrete instantiation, as Section 4.2 will formally define.

To translate $Q_s$ from $S$ to $T$ in the above example, we need to reconcile three levels of *query heterogeneities*:

**Attribute level**: Two sources may not support querying the same concept or may query the same concept using different attribute names. For instance, $S$ supports querying the concept of *reader’s age*, while $T$ does not. Also, $S$ denotes book *price* using *price range*, while $T$ using *price*.

**Predicate level**: Two sources may use different predicate templates for the same concept. For instance, *price* predicate template in $T$ has a different set of value ranges from $S$. As a result, in the predicate level, we can only translate a predicate as “close” as possible. In particular, we set our closeness goal as *minimal subsumption*, i.e., to subsume the source query $Q_s$ with fewest extra answers.

**Query level**: Two sources may have different capabilities on querying valid combinations of predicate templates. In our example, form $T$ only supports queries on one of the attributes *author*, *title*, *subject* and *ISBN* at a time with an optional attribute *price*. Therefore, $T$ cannot query *author* and *title* together, while $S$ can.

To realize the source query, we need to reconcile the heterogeneities at the three levels and generate a query plan expressed upon the target form $T$. Such a plan, as Figure 4.1 shows, in general, consists of two parts: a **union query** $Q^*_t$ which is a union of queries upon the target form to retrieve relevant answers from the target database, and a **filter** $\sigma$ which is a selection to filter out false positives. (Further optimization is possible by logic transformation of the query plan, but this transformation does not change the semantics of the plan.) To minimize the cost of post processing, i.e., filtering, we want the union query $Q^*_t$ to be as “close” to the source query $Q_s$ as possible so that it retrieves fewest extra answers. $Q^*_t$ in Figure 4.1 is such a query. We will discuss its construction throughout the chapter.

Building such a form assistant brings us two new challenges: predicate mapping and system architecture design. First, we need to develop a general predicate mapping mechanism that can be easily adopted to
new sources and domains. While extensively studies have been done to the attribute level, known as schema matching, and the query level, known as capability-based query rewriting, little has been done to reconciling the heterogeneity at predicate level, which we name as predicate mapping. Existing work on predicate mapping relies on per-source based rules to encode the mapping knowledge, which cannot achieve our goal of source-generality and domain-portability. Our approach is inspired by an intriguing insight that matching predicates often form localities, which are consistent with the notion of data types. This observation enables us to encode more general mapping knowledge based on the types and thus motivates a type-based search-driven approach for mapping predicates.

Second, guided by the application requirements, we need to carefully design the system architecture of the form assistant. In particular, to realize source-generality, each component of the system should incorporate general techniques to cope with heterogeneities for new sources. To realize domain-portability, the system should require minimal amount of human involvement to customize the form assistant for new domains.

Tackling these challenges, this chapter develops a light-weight domain-based form assistant. Specifically, Section 4.2 formalizes the query translation problem. Section 4.3 presents the system architecture of the form assistant. Section 4.4 motivates the type-based search-driven predicate mapping machinery and Section 4.5 discusses concrete techniques for realizing it. Section 4.6 presents our development of other components in building the form assistant. Section 4.7 reports our experimental results.

4.2 Formulation of Query Translation Problem

In this section, we formalize the query translation problem. As Figure 4.1 illustrates, the input to the query translator is a source query $Q_s$ on a source form $S$ and a target form $T$. The output of translator is an expression with a filtering $\sigma$ applied upon a union query $Q^*_T$. The union query is built upon a set of form queries, $f_1, \ldots, f_n$, where each form query is a valid way of filling the form $T$.

Such a translation, as Example 8 shows, needs to meet two goals: First, the union query must be a valid query. That is, it must be built upon only valid form queries $f_1, \ldots, f_n$ of the target form. Second, the union query must be “close” to the source query to minimize the selection as the post processing. As we can see, these two objectives are placed on the union query without considering the actual selection $\sigma$. The reason is that we can always apply the source query $Q_s$ as the tightest filter to remove false positives. We notice
that some other works, e.g., [?], also studied the “optimality” of the filters, i.e., to choose a filter with fewest number of predicates. We do not consider this issue in our work since it is straightforward to get an optimal filter for a union query.

We thus define the query translation problem as: Given a source query $Q_s$ and a target form $T$, among all the valid union queries of $T$, we choose the one semantically closest to $Q_s$ as the best translation. Next, we need to formally define valid union query and semantic closeness.

### 4.2.1 Query Model

The query model of a source describes templates of acceptable queries. Many specification languages, e.g., [?], have been proposed for describing general data sources, often in the form of datalog or context-free grammar. However, since developed for general purposes, those languages, although sufficiently powerful, are not very intuitive for expressing query forms. Therefore, we adopt a simple query model to describe the source capability, which will be discussed below. This query model is built upon the semantic model which form extractor extracts. Further, as Section 4.6 will discuss, when necessary, this model can be transformed into other specification languages and thus we can apply existing query rewriting techniques.

The query model of a form consists of vocabulary $\Sigma$ and syntax $\mathcal{F}$. Vocabulary $\Sigma$ specifies a set of usable predicate templates (previously referred to as conditions in Chapter 3) on the query form. A predi-
icate template is a three-tuple \([\text{attribute \ operator \ value}]\), with one or more variables as “placeholder” to be instantiated by concrete values. For instance, Figure 4.2(a) shows the vocabulary of the target form \(T\) in Figure 4.1, which contains five predicate templates \(\{P_1, \ldots, P_5\}\) on attributes \(\text{author, title, subject, isbn and price}\) respectively. In particular, \(P_1 = [\text{author contain} \$au]\) queries on attribute \(\text{author}\) with a default operator \(\text{contain}\) applied to a value parameter \(\$au\). As we notice, our form extractor discussed in previous chapter essentially constructs such a vocabulary of query model for a query form.

Further, upon the vocabulary of predicate templates, the syntax \(\mathcal{F}\) specifies all the valid combinations of these templates with respect to the form. Like many other specification languages, e.g., \([?]\), our syntax focuses on conjunctive queries, because it is sufficient to capture the capabilities of most deep Web sources. We name a valid combination of predicate templates a \textit{conjunctive form}.

We observe that deep Web sources often have two types of \textit{constraints} on how predicate templates can be queried together. First, some predicate templates may only be queried “exclusively.” For instance, form \(T\) allows only an exclusive selection among attributes \(\text{author, title, subject and ISBN}\); \(P_1, P_2, P_3\) and \(P_4\) can thus only appear one at a time (with an optional \(P_5\)). Second, a form may have “binding” constraints, which require certain predicate templates be filled as mandatory. For instance, form \(T\) may require \(\text{price}\) not be queried alone and thus each form query must bind a predicate template from \(P_1, \ldots, P_4\). In our modeling, we thus define \(\mathcal{F}\) as all valid conjunctive forms that satisfy these two constraints. For instance, form \(T\) requires one template from \(P_1, \ldots, P_4\) with an optional \(P_5\), and thus its syntax contains eight conjunctive forms \(\mathcal{F} = \{F_1, \ldots, F_8\}\), as Figure 4.2(a) shows.

Based on the above modeling, we can define what a valid union query is, given the target form. To begin with, we define a \textit{predicate} \(p_i\) as an instantiation of a predicate template \(P_i\), i.e., assigning concrete values for the parameters in \(P_i\). For instance, \(p_1\) in Figure 4.2(b) is an instantiation of \(P_1\). Further, a \textit{form query} \(f_j\) is an instantiation of a conjunctive form \(F_j\), e.g., \(f_1\) in Figure 4.2(b) is an instantiation of \(F_1\). Overall, a \textit{valid union query} on the target form is thus \(f_1 \lor \ldots \lor f_n\), where \(f_i\) is a form query. In particular, Figure 4.2(b) shows a valid union query \(Q_t\) on form \(T\).

The reason we limit our focus on union without considering intersection is two-folded: First, the union operation is “non-blocking,” which allows results to be incrementally constructed. Specifically, Web query
results are often organized into multiple pages, with each page containing a set of data records. For union, we can pipeline the processing of each page without waiting for complete input from operands. However, intersection operation is “blocking.” We are not sure whether a data record should be in the intersection result or not until we retrieve all pages from both operands. Second, an intersection usually can be more efficiently realized by a selection. Both intersection (e.g., \( A \cap B \)) and selection (e.g., \( \sigma_B A \)) achieve the same goal of removing false positives, but intersection needs to retrieve more results of \( B \). Therefore, our translation considers only union, which is consistent with many other works, e.g., \([?, ?]\). For simplicity, we will use query to refer to union query without ambiguity in the following discussion.

4.2.2 Semantic Closeness

To capture the closeness between a valid query and the source query, we need to define a closeness metric. In particular, we adopt the \( \text{minimal subsuming} \) metric \( C_{\text{min}} \), i.e.,

**Definition 6:** Given a source query \( Q_s \) and a target query form \( T \), a query \( Q_t^* \) is a minimal subsuming translation \( w.r.t. T \) if:

1. \( Q_t^* \) is a valid query \( w.r.t. T \);
2. \( Q_t^* \) subsumes \( Q_s \), i.e., for any database instance \( D_i \), \( Q_s(D_i) \subseteq Q_t^*(D_i) \);
3. \( Q_t^* \) is minimal, i.e., there is no query \( Q_t \) such that \( Q_t \) satisfies (1) and (2) and \( Q_t^* \) subsumes \( Q_t \).

We choose \( C_{\text{min}} \) as many other query rewriting works \([?, ?]\) do, because it has several advantages: First, a \( C_{\text{min}} \) translation does not miss any correct answer and contains fewest incorrect answers. Consequently, the overhead of filtering out “false positives” (in the selection operation) is minimal. Second, the \( C_{\text{min}} \) metric is database content independent, as the above definition indicates. Since human users usually are not aware of the content before querying a source, such content independent translation is consistent with users’ behavior. Also, in Section 4.5, we will apply this property to generate a complete database for subsumption testing. Third, it enables the separation of predicate mapping from query rewriting, as Section 4.3 will discuss.

The following example demonstrates a translation using the \( C_{\text{min}} \) metric.

**Example 9:** Consider the source query \( Q_s \) in Example 8 and three valid queries \( Q_{t_1}, Q_{t_2} \) and \( Q_{t_3} \) as below, wherein predicate \( p_1, p_2, p_3^1 \) and \( p_3^2 \) were introduced in Figure 4.2.

80
We can see that $Q_{t_1}$ and $Q_{t_3}$ subsume $Q_s$, while $Q_{t_2}$ does not, because it misses the price range between 0 to 25 and thus cannot be the best translation. Further, between $Q_{t_1}$ and $Q_{t_3}$, we can prune $Q_{t_3}$ because it subsumes $Q_{t_1}$ and thus cannot be the $C_{min}$ translation. In fact, we can show that $Q_{t_1}$ is the $C_{min}$ translation.

### 4.3 System Architecture

To realize the query translation as defined in Section 4.2, conceptually we need to search among all valid queries for the minimal subsuming one. This is certainly inefficient. To avoid exhaustive “search,” we want to “construct” the best translation. In particular, since translation essentially is to reconcile the heterogeneities at the three levels, it is thus desirable to address them separately, and then construct the translation by putting them together. That is, we first find the matching predicates, then map each pair of matching predicates individually by $C_{min}$ metric, and finally find the $C_{min}$ rewriting of the mapped query on the target form.

Is such separation possible? While the separation of predicate mapping from attribute matching is obvious, how about the separation of query rewriting from predicate mapping? To guarantee the correctness of such separation, we require that the mapped query, generated by mapping predicate individually, must be rewritable to a minimal subsuming translation. This requirement has two implications: First, the mapped query must be rewritable to a valid translation (if a valid translation of the source query exists at all); second, the rewriting of the mapped query must be an overall minimal subsuming translation. It can be shown that under the $C_{min}$ translation metric, the mapped query meets these two requirements and thus predicate mapping and query rewriting can be separated.

The separation thus enables a modular design for the form assistant, which consists of three components: attribute matcher, predicate mapper and query rewriter, as Figure 4.3 shows. First, attribute matcher discovers semantically corresponding attributes. Then, for each pair of matching predicates, predicate mapper maps the source predicate to the target one using the $C_{min}$ metric. After matching attributes and mapping

<table>
<thead>
<tr>
<th>$Q_{t_1}$</th>
<th>$(f_1 : p_1 \land p_1^2) \lor (f_2 : p_1 \land p_2^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_{t_2}$</td>
<td>$f_2 : p_1 \land p_2^2$</td>
</tr>
<tr>
<td>$Q_{t_3}$</td>
<td>$f_3 : p_1$</td>
</tr>
</tbody>
</table>
each individual predicate, we get a mapped query. Finally, query rewritter rewrites the mapped query to a valid query in terms of the capability of the target form. Example 10 illustrates the functionality of each component with a concrete example.

**Example 10:** Consider the translation example in Example 8, where source query \( Q_s = s_1 \land s_2 \land s_3 \land s_4 \).

First, attribute matcher will find that \( s_1 \), \( s_2 \) and \( s_4 \) have matching attributes in \( T \), while \( s_3 \) does not. Then, predicate mapper will map each pair of matching predicates with the \( C_{min} \) metric. In particular, for \( s_1 \), the mapped predicate is \( p_1 = s_1 \); for \( s_2 \), the mapped predicate is \( p_2 = s_2 \); for \( s_4 \), the mapped predicate is \( p_5 = [\text{price between 0-25}] \) and \( p_6 = [\text{price between 25-45}] \). After matching and mapping, the mapped query is thus \( Q'_s = p_1 \land p_2 \land (p_5 \lor p_6) \). Finally, applying the query rewriting, we can rewrite \( Q'_s \) into \( \sigma_{p_2}(f_1 \cup f_2) \), where \( f_1 = p_1 \land p_5 \) and \( f_2 = p_1 \land p_6 \). The union query \( f_1 \cup f_2 \) is thus the best translation \( Q'_t \) for source query \( Q_s \), and the selection \( \sigma_{p_2} \) is a part of the final filter, which we omit here.

Such modularization on one hand reduces the complexity of translation—Each component only focuses on a subproblem to solve. On the other hand, it allows us to plug in, for each component, corresponding techniques suitable for specific application needs. In particular, attribute matching and query rewriting have been extensively studied. Our form assistant can thus take advantage of the existing techniques by taking
them as building blocks of the system.

To prepare the input for the form assistant, we explore the form extractor we developed in Chapter 3 to automatically extract query capabilities of sources. Next, we need to design each component of the form assistant to satisfy the application requirements, i.e., source-generality and domain-portability. The key is to identify, for each component, what knowledge can be built-in as generic techniques and what should be customized as domain-specific knowledge. Further, we want the customized knowledge to be per-domain based instead of per-source, and we want to keep such knowledge within human-manageable scale.

**Attribute Matcher**: To match attributes is essentially to identify synonym attributes. This problem, known as schema matching, has been an important problem and thus extensively studied in data integration. Depending on the application scenarios of the form assistant, we can employ either automatically discovered or manually encoded synonym thesaurus, based on which we can explore a general thesaurus-based matching approach. For instance, for the MetaQuerier system (discussed in Section 4.1), since sources are on-the-fly selected and thus form dynamic domains, the thesaurus has to be automatically discovered by exploring existing automatic matching techniques, e.g., [?, ?]. While for domain portals and query assistant toolkit, since sources are in pre-configured domains, the thesaurus can be manually customized with more reliable domain-specific knowledge. Our survey [?] on query forms shows that, for each domain, there only exists a small number of frequently used attributes, which makes the manual encoding of such a domain-specific thesaurus feasible in practice.

**Predicate Mapper**: To map predicates, we need to know, for each pair of matching predicates, how to choose the operators and fill in the values. To achieve the requirement of easy deployment for new domains, we thus ask whether there exists domain-generic mapping mechanism which addresses most if not all mappings in various domains. We observed that, as an enabling insight, predicates with the same data type often share similar mapping patterns. Motivated by this observation, we develop a *type-based search driven* mapping approach with several built-in type handlers to handle the majority cases of general mappings. For those domain-specific mappings (e.g., airport code to city name in airfares domain), we can customize the required mapping knowledge by adding new type handlers if needed.

**Query Rewriter**: To tackle the query-level heterogeneity is essentially to rewrite the query to satisfy
the syntax constraints of the target form. Such a task, studied as capability-based query rewriting [? , ? , ?], is not specific to any source or domain. Hence, as Figure 4.3 shows, the query rewriter component can be entirely built-in into the system. Section 4.6 will discuss our development of this component by exploiting existing rewriting techniques.

4.4 Predicate Mapping: The Motivation

With extensive existing study on schema matching and query rewriting, together with our modular separation of predicate mapping from query rewriting, the essential challenge of translation boils down to predicate mapping. As Section 4.3 motivated, the requirement of easy-customization calls for domain-generic mapping knowledge. Does there exist such generic mapping knowledge applicable to sources across different domains? How to encode such knowledge? Those are the critical questions we need to address in predicate mapping.

Existing solutions on predicate mapping usually assume a static small-scale setting. In such scenarios, it is common, e.g., as [?] studies, to use a source-based pairwise-rule driven mapping machinery to map predicates. Figure 4.4 gives some example rules to encode the mapping knowledge required for translation in Example 8.

However, such a mapping machinery, characterized by a “per-source” knowledge scope and “pairwise” rule encoding mechanism, is not suitable for our light-weight domain-based form assistant, since it lacks both generality and extensibility. To begin with, the source-based scope can only handle mappings between sources whose mapping knowledge has been specified. It cannot generally deal with mappings involving “unseen” sources. Also, the pairwise-rule mechanism to encode the knowledge is not extensible because writing rules for each pair of predicate templates is hard to scale when mapping is required among a large set of sources. Further, as sources are autonomously developed, their query capabilities may change, which
makes maintenance of rules labor intensive.

4.4.1 A Motivating Survey

To understand the feasibility of developing a more general and extensible mapping machinery, we thus ask: Are there generic scopes orthogonal to domains wherein the mappings happen? If such scopes exist, we thus can leverage them to encode our domain generic mapping knowledge.

To answer this question, we conduct a survey to examine the mapping correspondence between predicate templates. Such mapping correspondence hints the required translation scope: only templates with mapping correspondence need to be translated. In particular, we examined 150 query forms of three domains, which we have used in survey of Section 3.2. We totally find 37 template patterns in 150 sources. For instance, as two popular patterns, [attr default $val] uses a default operator and accepts arbitrary values from the user input (e.g., template $P_1$ in $T$ belongs to this pattern), and pattern [attr default $val$$val \in \{D\}] accepts a value from a set of given options (e.g., $P_5$ in $T$ belongs to this pattern). Note that the template patterns we examined here is different from those in Section 3.2. Here, the patterns capture operators and input formats, while the patterns in form extractor capture the visual arrangements. In the rest of the section, we will use template to generally refer to template pattern for simplicity.

We notice that two predicate templates have mapping correspondence only if there exists a concept expressed with these two templates in different sources. For example, to support querying book subject, a source may use [subject default $val] and another [category default $val$$val \in \{D\}]$. Since subject and category are
matching attributes, templates \([\text{attr default } \$\text{val}]\) and \([\text{attr default } \$\text{val} \in \{D\}]\) need to be translated and thus have mapping correspondence. We record such mapping correspondence between any two predicate templates \(P_i\) and \(P_j\) in a \emph{correspondence matrix} \(CM\). In particular, \(CM(i,j)\) denotes the number of concepts that are expressed using both templates \(P_i\) and \(P_j\). Figure 4.5 shows our survey result (i.e., the correspondence matrix), where the value of \(CM(i,j)\) is illustrated as the degree of grayness.

As Figure 4.5 indicates, mappings happen mostly only within certain clusters of templates. In the figure, we order the templates in a way to clearly observe such localities: In particular, we observe three localities of templates among which mappings happen. Such mapping localities thus formulate the scopes of mappings.

We further ask: \textit{What templates can share a locality?} Are there something common among templates in the same locality? Not surprising, we found there is indeed an underlying explanation for the phenomenon. While the predicate templates in a locality are used by various concepts in different domains, those concepts often share the same “data type.” In particular, the first locality in Figure 4.5 corresponds to templates usually used by concepts of \textit{datetime} type, the second one \textit{numeric} type and the third one \textit{text} type. The only outlier is the template \([\text{attr default } \$\text{val}]\), as shown in Figure 4.5. Concepts with different data types can all use this template to express their query predicates and this template thus has mapping correspondence to most other templates.

\subsection{Type-based Search-driven Predicate Mapping}

Our observations above clearly reveal that mappings between predicate templates are not arbitrary– Their associations not only suggest “locality,” but also are clearly “aligned” with their underlying data types. Such an observation motivates a \textit{type-based search-driven} predicate mapping machinery.

First, our observation shows the promise of pursuing a \textit{type-based} knowledge scope. Since mappings are only worthy inside localities of types, we can express the mapping knowledge in the scope of types. Such a type-based mapping scope is more general than the source-based or domain-based scope, since data types are widely reused across different sources and domains, and we are able to translate queries to unseen sources or adapt the knowledge to new domains as long as they reuse those data types.

Second, we find that a data type gives us a “platform” to compare semantics (i.e., the subsuming relationship) of different mappings and thus enables an extensible \textit{search-driven} mechanism. Instead of
writing rules to hard code the mapping for every pair of predicate templates, we can encode our mapping
knowledge as an evaluator for each template. This evaluator “materializes” the semantics of a query against a
type specific platform, as Section 4.5.1 will discuss. By doing so, semantic comparison can thus be performed
on the materialized results. For instance, to realize rule \( r_3 \) in Figure 4.4, we can project the source predicate
as well as all target predicates onto an axis of real numbers, and thus compare their semantics based on their
coverage. Finding the closest mapping thus naturally becomes a search problem - to search for the ranges
expressible in the target form that minimally cover the source. Such a search-driven approach achieves
extensibility by exploring evaluators rather than static pairwise rules. With a data type of \( n \) templates, we
only need \( n \) evaluators instead of \( n^2 \) rules; when adding a new pattern, only one evaluator needs to be added
instead of \( n \) rules.

4.5 Predicate Mapping: The Solution

In this section, we discuss our development of predicate mapper, which realizes the type-based search-driven
mapping machinery. According to our system design (Section 4.3), predicate mapper takes a source predicate
\( s \) and a matching target predicate template \( P \) as input, and outputs the closest target translation \( t^* \) for \( s \).
In particular, predicate mapper consists of two components: type recognizer and type handler, as Figure 4.6
shows. Type recognizer recognizes the data type of the predicates and then dispatches to a corresponding
type handler. A type handler maps predicates of a specific type with a search approach.

Since our focus in this section is type handler, we only briefly discuss the development of type recognizer.
In particular, since the data type of a predicate can often be hinted by its syntactic features, we explore
those syntactic clues to implement the type recognizer. For instance, we can exploit distinctive operators
(e.g., \texttt{all}, \texttt{any} for \texttt{text} type), the value filled in the source predicate (e.g., \( s_3 \) in Example 8 contains value
35) and the value domain in the target template (e.g., a selection list of \( P_5 \) in Example 8) to infer type
information. We have developed a type recognizer in our previous work for schema matching [?], wherein
the type information is used to help matching. Please refer to [?] for more details.
4.5.1 Overview of Type Handler

At the core of the predicate mapping, each type handler realizes the search-driven mechanism for its responsible type. Like any other search-based algorithm, a type handler needs to have three key components: search space, closeness estimation, and search strategy.

Example 11: Consider a predicate mapping problem. The source predicate is \( s = [\text{category} \ \text{contain} \ "\text{computer science}" \] and the target predicate template is \( P = [\text{subject} \ $op \ $val], \) where the operator $op$ is from \{ "any words", "all words" \} (simply as "any, all").

Defining Search Space

Defining a reasonable search space is essential to any search process, because it impacts the complexity of search significantly. Given a predicate template \( P \), we define instance space of \( P \), denoted by \( I(P) \), as all possible instantiations (i.e., predicates) of \( P \). Due to the predicate-level heterogeneity, we often need to map a source predicate into multiple target predicates (e.g., the price predicate in Example 8). We thus define the search space of \( P \), denoted by \( \Omega(P) \), as any disjunction of predicates in \( I(P) \). Each item in \( \Omega(P) \) thus corresponds to a possible mapping.

As \( I(P) \) often can be infinite, we take a “close-world” assumption to restrict the search space. Specifically, while operators are clearly limited by \( P \) (e.g., two operators any, all in Example 11), the values can often be
Instance space $I(P)$:

$p_1$: [subject any "computer"]
$p_2$: [subject all "computer"]
$p_3$: [subject any "science"]
$p_4$: [subject all "science"]
$p_5$: [subject any "computer science"]
$p_6$: [subject all "computer science"]

Search space $\Omega(P)$:

$t_1 = \{p_1\}$, $t_2 = \{p_2\}$, $t_3 = \{p_3\}$, $t_4 = \{p_4\}$, $t_5 = \{p_5\}$, $t_6 = \{p_6\}$
$t_7 = \{p_1, p_2\}$, $t_8 = \{p_1, p_3\}$, $t_9 = \{p_1, p_4\}$
$t_{10} = \{p_1, p_5\}$, $t_{11} = \{p_1, p_6\}$

... Figure 4.7: The instance space and search space of $P$.

Infinite— for a predicate template without pre-defined domain of values (e.g., an attribute name with a text input box), any value can be filled in and thus constitutes an infinite instance space. To define a reasonable space of $I(P)$, we make a “closed-world” assumption: We denote the values filled in the source predicate $s$ as $W_s$. For a target template without pre-defined domain of values, we assume the values of the target predicate can only choose from $W_s$. Therefore, we define the instance space $I(P)$ as all possible instantiations of $P$ using values either in a pre-defined domain if it is available or composed from $W_s$ otherwise. For instance, in Example 11, the target template $P$ is thus restricted to use only the words in the source predicate, i.e., $W_s = \{"computer", "science"\}$. Figure 4.7 shows the instance space and part of the search space of $P$. In particular, with the close-world assumption, the instance space of $P$ contains 6 possible instantiations.

Such a close-world assumption is reasonable, because without domain-specific knowledge, it is difficult to invent new values. In fact, the search space can be enriched (e.g., by expanding queries words with their synonyms) if more domain knowledge is available (e.g., by providing synonym thesaurus), as we will discuss in Section 4.5.3.

Closeness Estimation

Given the search space $\Omega(P)$ covering all possible mappings, finding a $C_{min}$ mapping boils down to inferring subsumption relationship between a mapping and the source predicate, and between two mappings. This inference is specific to data types— For some types, it is straightforward, while others, it is not. In particular, for numeric type, since a predicate with numeric type can be projected to an axis of real numbers,
evaluating subsumption relationship upon the numeric axis becomes an easy task. Datetime type can be processed in a similar way as numeric type and thus we discuss their handlers together in Section 4.5.2. However, for text type, the inference of subsumption relationship is not trivial since it essentially needs logical reasoning. To avoid convoluted logical inference, we develop an equivalent “evaluation-by-materialization” approach. Section 4.5.2 will discuss this approach in details.

**Search Algorithm**

With the subsuming testing, we can find the \( C_{\text{min}} \) mapping using the following simple algorithm, where \( s, P \) is the source predicate and target template, \( H \) records all subsuming mappings and \( R \) records \( C_{\text{min}} \) mappings and \( x \) is the returned mapping:

1. \( H = \emptyset, R = \emptyset \)
2. for \( \forall t \in \Omega(P) \):
3. if \( \text{subsume}(t, s) \): add \( t \) to \( H \)
4. for \( \forall t \in H \):
5. if \( \exists t' \in H \) and \( t' \neq t \) such that \( \text{subsume}(t, t') \)
6. add \( t \) to \( R \)
7. choose an \( x \in R \) with minimal number of predicates

This algorithm basically exhausts the entire search space \( \Omega(P) \) to find all \( C_{\text{min}} \) mappings, and then chooses the one with minimal number of predicates to minimize the number of form queries issued on the target form. To improve its efficiency, we may refer to an approximate algorithm. In particular, we can take a greedy approach in practice: We find the mapping iteratively and in each iteration, we look for a (not-selected) instantiation from \( I(P) \) that maximally covers the uncovered part of the source predicate \( s \) until we can entirely cover \( s \). (In case there is no subsuming mapping for \( s \), this approach will generate a maximal subsumption mapping, which is very desirable.)

**4.5.2 Built-in Handlers**

**Text Type Handler:** Text is the most commonly used type in query forms for querying string-based fields (e.g., subject in Example 11). Operators of text type predicates are typically string match operations
including all, any, exact, etc., and values are strings.

We will use the mapping task in Example 11 to illustrate how the search proceeds towards finding the best mapping of the target template \( P \). In Figure 4.7, we have illustrated the instance space and search space of \( P \). Next, we need to develop an approach to evaluate the subsumption relationship of text type predicates.

While it may be possible to logically reason the subsumption relationship, such an approach can be very convoluted. On one hand, it is usually difficult to encode the logic relationship, e.g., \( contain(v_1, v_2) = \text{any}(v_1) \cap \text{any}(v_2) \) and \( contain(v_1) = \text{any}(v_1) \). It is also non-trivial to apply those rules to reason between complex predicates. On the other hand, it is not extensible, since adding a new operator (e.g., \textit{start with}) requires encoding its relationship with every existing operator (e.g., any, all, exact).

To circumvent such reasoning, we employ an “evaluation-by-materialization” approach. The idea is that, instead of semantically reasoning the subsumption relationship between two queries \( Q_1 \) and \( Q_2 \), we can “materialize” them against a database instance \( D \) and compare the query results \( Q_1(D) \) and \( Q_2(D) \). The question is which database instance can be used to reliably test the subsumption relationship? It is obvious that we cannot arbitrarily choose a database instance \( D_i \). We observe that a “complete” database \( D \), which conceptually is the union of all possible database instances \( D_i \), satisfies our requirement.

**Property 1:** Given two queries \( Q_1 \) and \( Q_2 \), and a complete database \( D = \bigcup D_i \), \( Q_1 \) semantically subsumes \( Q_2 \) if and only if \( Q_1(D) \subseteq Q_2(D) \).

The “only if” direction obviously holds, and we only show the “if” direction informally. Let us suppose that \( Q_1(D) \subseteq Q_2(D) \) but \( Q_1 \) does not semantically subsume \( Q_2 \). Then there exists a database instance \( D_i \) such that \( Q_1(D_i) \nsubseteq Q_2(D_i) \), i.e., \( \exists x \in Q_1(D_i) \text{ and } x \notin Q_2(D_i) \). This means that \( Q_2(x) \) is false and thus for any database instance \( D_j \), \( Q_2(D_j) \) cannot contain \( x \). Therefore, \( Q_2(D) \) cannot contain \( x \), and \( Q_1(D) \nsubseteq Q_2(D) \). This is contradictory to our assumption and thus the property holds.

Next, the question becomes how to construct such a complete database? Conceptually, a complete database is a complete enumeration of all possible phrases composed from English words (assuming we only handle English). This is simply too huge to construct and we thus need to look for a practical solution.

We observe that the “completeness” of a database depends on the “logical properties” to be tested by
operators. For instance, operators contain and any concern only membership of words in a string field without caring their ordering, while operators start and exact concern both membership and sequence. In practice, we observe that membership and sequence are usually the only properties tested by the operators for text type. Therefore, we construct the database using words from $W_s$ plus some additional random words. The database is composed of all possible combinations of the words (for testing the membership) with all possible orders (for testing the sequence). Since the exact set of additional words used in the test database does not matter, we thus can use a small number of random words to generally represent all words outside $W_s$. In implementation, to avoid constructing database for every mapping, we use a set of five fixed words to construct a static database. At runtime, we dynamically designate each word to represent a particular constant in $W_s$. If the size of $W_s$ is greater than 5, we will dynamically construct a test database.

To test subsumption between a mapping and the source predicate in Example 11, we build a database $D$ with alphabet $D = \{\text{computer, science, dummy}\}$, as Figure 4.8 shows the content. For simplicity, we use only initials for corresponding words. Figure 4.8 shows, for source predicate $c$ and each mapping $t_i$ in the search space, the result set against the complete database. As we can see, all the listed mappings are subsuming mapping. Among them, $t_5$ is $C_{\text{min}}$ mapping as it subsumes no other mappings. Alternatively, the greedy approach that searches over all predicates in $I(P)$ can also find the right mapping $t_5$, but it is much more efficient.

<table>
<thead>
<tr>
<th>Source Predicate</th>
<th>Materialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c : {\text{category contain CS}}$</td>
<td>$\text{CS, SC, CSD, CDS, DSC, DCS, SCD, SDC}$</td>
</tr>
<tr>
<td>$t_1 : {\text{subject any C}}$</td>
<td>$\text{C, CS, CD, SC, DC, CSD, CDS, DSC, DCS, SCD, SDC}$</td>
</tr>
<tr>
<td>$t_2 : {\text{subject all C}}$</td>
<td>$\text{same as above}$</td>
</tr>
<tr>
<td>$t_3 : {\text{subject any S}}$</td>
<td>$\text{S, CS, SD, SC, DS, CSD, CDS, DSC, DCS, SCD, SDC}$</td>
</tr>
<tr>
<td>$t_4 : {\text{subject all S}}$</td>
<td>$\text{same as above}$</td>
</tr>
<tr>
<td>$t_5 : {\text{subject all CS}}$</td>
<td>$\text{CS, SC, CSD, CDS, DSC, DCS, SCD, SDC}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.8: Example of database and evaluation results.
**Numeric & DateTime Handler:** Numeric and datetime have very similar nature in that they all form a linear space, and have similar operators, such as before vs. ≤, during vs. between and so on. In fact, datetime is internally presented as an integer in many programming languages. Therefore, the mapping techniques for the two types are generally the same. In this section, we will thus focus on numeric type and the general discussion applies to datetime type too.

We use an example to run through the mapping process. Consider mapping between the price predicates in Example 8. Since the target predicate has a pre-defined domain of values, each representing a range, our search space is restricted to disjunctions of those ranges. To estimate the closeness, we project the source predicate \( c : [\text{price} \leq 35] \) and the target predicate, e.g., \( p_5^1 : [\text{price} \text{ between } 0,25] \) into ranges in numeric line. By projecting \( c \) and \( p_5^1 \) to the numeric line, \( p_5^1 \)'s false positive range (empty) and false negative range (25, 35) can be straightforwardly evaluated based on their coverage on the line. Using the greed search approach, we will choose the one that maximally covers the uncovered range. Therefore, we choose range (0, 25) and (25, 45) in turn, which forms our mapping \([\text{price} \text{ between } 0,25] \lor [\text{price} \text{ between } 25,45]\).

### 4.5.3 Domain Specific Type Handler

While the type handlers discussed above can handle generic types independent of domains, there are situations where domain specific knowledge is required. First, while our built-in type handler only syntactically maps values between source and target predicate values, some mapping needs domain-specific thesaurus to find out the semantic correspondence. For instance, to map from [subject equal computer science] to [subject equal computer architecture], we need a subject ontology to know “computer science” is a hyponym of “computer architecture.” Second, while the built-in type handlers only deal with 1:1 mappings, we need domain-specific procedures to realize \( m : n \) mappings, e.g., to map from lastname, firstname to author.

There are two ways to implement domain specific type handlers. The first implementation is to plug in a full-fledged type handler, which takes the matching predicates as input and outputs the mapping result. This handler is thus a black box to the system, which performs whatever computation to generate the mapping. The second implementation is to plug in a domain thesaurus, which on one hand is used to generate terms for defining the search space, e.g., to map from city name to airport code, and on the other hand is used for subsumption testing, e.g., to tell that “computer science” subsumes “computer architecture.” Therefore,
this implementation requires two interfaces from the thesaurus– interface lookup takes a value as input and outputs a set of values which are semantically related to the input; interface subsume takes two values as input and outputs whether one is a hyponym of the other. With such a thesaurus, the same search algorithm can be used for finding the best mapping.

Let us use an example to illustrate the second implementation of domain-specific type handler. Consider a mapping from [airport code equal LAX] to [city name equal $val], where a set of searchable city names is enumerated in a selection list. To realize a specific type handler for this mapping, we provide a thesaurus with an subsume interface which takes an airport code and a city name as input and outputs whether one subsumes the other. (Note that since the target template has a pre-defined domain of values, we do not need lookup interface to define the search space.) With this interface, we can perform the same search algorithm, which tries different city names one by one to find the minimal subsuming mapping.

4.6 Implementation: Form Assistant Toolkit

With form assistant as a core component to many applications, in this section, we study an example application, which we have implemented with a system demonstration [?]– building a query assistant toolkit, to concretely evaluate the effectiveness of our approach. Query assistant toolkit is a browser based toolbar, which gives users suggested translations across sources in the same domain. In particular, users can register a query of her interest, e.g., a query on amazon.com, and when she browses other sources in the same domain, e.g., barnesandnoble.com, she can ask for suggested translations. This toolbar is pre-configured with a set of supported domains, and users need to specify the domain of interaction when registering a query.

When activated, the assistant will automatically translate a source query to the target form. To prepare the input to translation, the form extractor module, built upon form understanding discussed in Chapter 3, automatically constructs the query capability of the target form. With predicate mapping addresses the predicate heterogeneity, in this section, we briefly discuss the remaining three components - form extractor, attribute matcher and query rewriter to complete the framework.

Form Extractor: Form extractor automatically constructs the query capability of a form to prepare the input for form assistant. We have studied the problem of form extraction in Chapter 3, which extracts the set
of predicate templates as vocabulary from a query form. To further construct the syntax of query capability, we identify the exclusively queried attributes and the required fields in query forms. The exclusive predicates are usually presented in a pattern, where a set of attributes are enumerated in radio-button or selection list, with a textbox accepting input. Based on this pattern, we thus refer to a set of commonly queried attributes in a domain to further identify those exclusive predicates. The set of common attributes is preconfigured as domain knowledge, which is used in attribute matching as well. Identifying required fields is a challenging task because there usually is no standard way of indicating those fields. However, most existing rewriting techniques can accommodate this problem by pushing as many predicates as possible to each form query. We will discuss this issue in query rewriter module.

**Attribute Matcher:** Attribute matcher identifies the semantical corresponding attributes. Our attribute matcher is customized with a domain thesaurus, which stores synonyms for commonly used concepts. At runtime, attribute matcher fuzzily matches the input attributes with those synonyms to check whether they express the same concept or not. In implementation, we use a hash table to map each synonym (after normalization), which we call **indexed attribute**, to the corresponding concept. In particular, the attribute matcher consists of two steps:

*Preprocess:* Given two predicates, the **preprocess** step performs some standard normalization, including stemming, normalizing irregular nouns and verbs (e.g., “children” to “child,” “colour” to “color”) and removing stop words.

*Synonym check:* After preprocessing, **synonym check** checks whether the two predicates match. We view input predicates as matched if their types match and their attributes are synonyms. The reason for checking on types is that some attributes, e.g., **departure, arrival** in airfares, must be accompanied by their types to reflect their semantics (e.g., departure city or departure date). To do so, we recognize the types of the predicates using the same type recognizer used in predicate mapping of Section 4.5. If their types match, we further check whether they are synonyms. In particular, for each input attribute, we first search for its “representative,” which is an indexed attribute with the highest similarity score above a predefined threshold. If the system successfully finds the representatives for both attributes and both representatives correspond to the same concept, we return them as matching attributes.
Query Rewriter: Query rewriter takes a mapped query and constructs a minimal subsuming query as the best translation. The core of this translation is a well-studied problem known as capability-based rewriting. Many existing solutions, e.g., [?], can apply here by transforming our query model into their specification languages. In particular, our rewriter builds upon the techniques developed in [?], which generates a minimal subsuming rewriting of a given query. This approach pushes as many predicates as possible to each form query so that it maximizes the use of source query capability.

Finally, we discuss the issue of incomplete modeling of required fields in form extraction. Considering the target query form \( T \), if we fail to identify that at least one of the four predicates on \texttt{author}, \texttt{title}, \texttt{subject}, \texttt{ISBN} is required, we may enlarge the space of valid translations by including “false” queries, e.g., \texttt{price between 0,25}. However, the query rewriter will not choose this query because it is not minimal subsuming—more predicates, e.g., \( p_1 \), can be pushed to the form query to make it more specific. Therefore, as long as the source query specifies the corresponding predicates, which can be mapped to the required fields of the target form, our translation will not abuse those “false” queries even if we do not explicitly capture those required fields. (If the source query does not specify those required fields, we cannot have any valid translation at all.)

4.7 Experiment

Experimental Data: To evaluate the complexity of deploying the form assistant and the accuracy of the translation, we collected 120 query forms from 8 domains of the TEL-8 dataset [?]. In particular, we separate the forms into two datasets - Basic dataset and New dataset. The Basic dataset contains 60 query forms with 20 from each of the three domains we surveyed (Section 4.4.1). The purpose is to evaluate the complexity of domain knowledge needed for translation and the performance over sources from which the types and their patterns are derived. The New dataset contains 60 query forms, which we sampled from 5 domains (i.e., CarRentals, Jobs, Hotels, Movies and MusicRecords) in the TEL-8 dataset. The purpose is to test how well the type handlers can apply to new domains.

Experiment Setting: The experiment evaluates the translation accuracy between two randomly picked

---

1 We observe that we can optimize the algorithm in [?] by taking advantage of the restricted format of our mapped query for more efficient rewriting. Our optimized algorithm runs in linear time.
forms in the same domain. In particular, suppose a domain has $n$ forms as $[Q_1, \ldots, Q_n]$. We random shuffle those forms to get a permutation $[Q_{i_1}, \ldots, Q_{i_n}]$, and translate queries between $(Q_1, Q_{i_1}), \ldots, (Q_n, Q_{i_n})$. With 120 forms in total, we thus evaluate 120 translations.

For each domain, we manually prepare the commonly used concepts and synonyms, as the domain-specific thesaurus. In particular, for each of the Airfares, Books and Automobiles domains, those concepts and synonyms are prepared based on 20 forms randomly sampled from the forms used in the survey of Section 4.4.1. We store only concepts appearing in more than one source. We apply those knowledge to new forms to understand the feasibility of manually preparing matching knowledge and the impact of mismatching to translation accuracy.

To avoid biased queries against different sources, we randomly generate queries. Given a source form $Q_i$, we instantiate its query templates by randomly choosing a value for each variable. In particular, if the domain of the variable is finite with a set of pre-defined options, we randomly choose one from these options. Otherwise, we randomly generate a value with the corresponding type. The query generated in the experiment is thus “maximal,” i.e., it instantiates all predicate templates in the source form. As we will see in the following discussion, as one of our performance metric measures the percentage of queries that are perfectly translated, using a maximal query thus measures the worst-case performance.

**Performance Measurement:** In the experiment, we measure two aspects of the system performance: the translation accuracy and the complexity of customization.

**Translation Accuracy:** For the translation accuracy, we measure how many mappings in the suggested translation are correct, which indicates the amount of efforts the form assistant saves for users. In particular, we adopt a *form-based* metric: For a source query $Q_s$ and a target form $T$, let $P(Q_s, T)$ denote the number of predicates in the target query, and $R(Q_s, T)$ denote the number of predicate templates correctly filled by the form assistant. The form-based metric is defined as the percentage of correctly filled predicate templates and is thus $R(Q_s, T)$ over $P(Q_s, T)$.

To validate the impact of 1:1 mapping assumption, for each set of experiments, we measure the performance with and without this assumption. Measurement with 1:1 mapping counts, among all 1-1 mappings, how many predicates are correctly filled. For complex mapping (e.g., *lastname+firstname* → *author*), as the
form assistant does not handle such mappings, those predicates - to be filled by complex mapping - are taken as a negative count in measurement with complex mapping.

To understand the impact of other components to predicate mapping, we further separately report the errors caused by form extraction and attribute matching from those caused by predicate mapping.

**Complexity of customization:** For each domain, we only customize the domain-specific thesaurus for attribute matcher. To evaluate how well the domain generic translation can achieve, we do not add any domain specific type handler. To measure the complexity of such customization, we report the number of commonly queried concepts and the corresponding synonyms for each concept.

**Experimental Result**

**Form-based Measurement:** Figure 4.9 (a) and (b) report the performance without counting the errors caused by form extraction and attribute matching. In particular, they show the frequency distribution over form-based metric for the two datasets respectively. As we can see, for 1:1 mappings, the system achieves very good results for both **Basic** and **New** datasets - 87% of the forms achieve perfect accuracy for the **Basic** dataset and 85% for **New** dataset. Further, as Figure 4.9(a) and (b) show, the system performs reasonably well even counting complex mappings - 76% of the forms achieve accuracy above 80% for the **Basic** dataset, and 70% for **New** dataset.

Figure 4.7 shows the performance impact of form extraction to the system (under 1:1 assumption). In
particular, it shows average accuracy of translation with perfect input from form extraction and without. To generate perfect input, we manually corrected the errors caused by form extraction. As we can see, in average, we achieve 90.4% for Basic dataset with raw input from form extraction, and by correcting the errors of raw input, we achieve 96.1% accuracy. For the New dataset, we achieve 81.1% with raw input and 86.7% with perfect input.

Figure 4.7 further reports, among all the mappings, the number of errors caused by each component. As we can see, the impact of attribute matching is smallest— it causes only 6 errors out of 33 in total; form extractor causes 13 errors and predicate mapping causes 14 errors. The majority errors in predicate mapping are caused by the lack of domain specific mapping knowledge. For instance, to map between subject “computer science” to subject “programming language,” we need a domain ontology; to map between zipcode to city, we need a domain dictionary. Therefore, adding domain specific type handlers will help to improve the mapping performance.

Complexity of domain knowledge: Figure 4.12 shows the number of concepts and synonyms we prepared for each domain. As it shows, the number of concepts and synonyms needed for translation is reasonably small. More importantly, such knowledge is collected on only 20 sources for each domain, and the accuracy of matching based on the prepared knowledge is very high. As Figure 4.9(d) shows, only 6 errors are caused by the mismatching of attributes, which is very acceptable.
<table>
<thead>
<tr>
<th>Domain</th>
<th>#. of Concepts</th>
<th>#. of Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airfares</td>
<td>13</td>
<td>34</td>
</tr>
<tr>
<td>Automobiles</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>Books</td>
<td>17</td>
<td>24</td>
</tr>
</tbody>
</table>

*Figure 4.12: Size of domain thesaurus.*

### 4.8 Related Work

Query translation has been actively studied in information integration systems in the literature. In particular, we observe that the existing techniques can be classified into four categories, to address data source heterogeneity at four levels. Our work, while is closely related to those works, clearly has its own focus: First, we focus on a specific subproblem of dynamic predicate mapping, which is largely unexplored in the literature. Second, we focus on the design and development of a complete query translator, which satisfies our requirements for a light-weight domain-based form assistant. In this section, we connect and compare our work with those existing solutions.

**Attribute Heterogeneity:** Schema matching [?, ?, ?, ?, ?] focuses on mediating the heterogeneity at attribute level. Those works are concrete building blocks for the form assistant. Different approaches of schema matching suit for different application settings. Some approaches, e.g., [?, ?, ?] require a collection of sources to mine the matchings holistically, which are suitable for applications such as MetaQuerier. Others, e.g., [?, ?], perform matching across pairwise sources, which are suitable for applications such as a domain portal.

**Data Heterogeneity:** Schema mapping [?, ?] focuses on addressing the heterogeneity of data format across different sources. Its main objective is to convert a set of data records from a source schema to a target one and thus it does not need to cope with constraints on predicates (i.e., available operators and values). Unlike schema mapping, which focuses on equivalence conversion, predicate mapping must deal with more complicated semantics (e.g., *any, exact* in text type and *less than* in numeric type).

**Predicate Heterogeneity:** Predicate mapping focuses on addressing the heterogeneity of the predicates across different source capabilities. Existing solutions, e.g., [?] usually assume a static system setting, where
sources are pre-configured with source-specific translation rules. In contrast, our approach dynamically maps predicates across unseen sources without prepared source knowledge. Further, the mapping in [?] depends on static rules to describe how to choose operators and fill in the values, while we propose a general search-driven approach to dynamically search for the best mapping.

**Query Heterogeneity:** Capability-based query rewriting focuses on mediating the heterogeneity of query syntax across different sources. Current query mediation works [? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? ] studied the problem of how to mediate a “global” query (from the mediator) into “local” subqueries (for individual sources) based on capabilities described in source descriptions. Such rewriting is essentially transformation of Boolean expressions upon predicates, and does not consider the heterogeneity inside predicates. Therefore, predicate mapping and query rewriting are two complementary aspects of query translation. In particular, reference [?] focuses on $C_{min}$ query rewriting at query level, and is thus a concrete building block of our form assistant to achieve the overall $C_{min}$ translation.
Chapter 5

Processing Queries In Databases: Evaluating Ranked Queries By K-Constrained Optimization

5.1 Problem Overview

So far, we have discussed various issues concerning how to generate a valid query submitted to a database. Upon acceptance of a query, database engines need to evaluate those queries to return qualified results. How to evaluate those queries efficiently? This is the problem we will address in this chapter.

While traditionally databases have been applied predominately in business settings with well-defined query logic, Web databases are now typically used in data retrieval scenarios where query semantics involves not only boolean conditions but also ranking criteria. In such scenarios, the target answers are described with some qualifying constraint $B$, which specifies what tuples should be considered valid, and a quantifying function $O$, which measures their degree of matching, and the query returns only some $k$ top-matched answers—thus overall with a query form $Q = (B, O, k)$. As a simple example, a query $(B: \text{dept} = \text{CS} \land (\text{year} = 2 \lor \text{year} = 3), O: \text{gpa}, k: 5)$ 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, which effectively specifies a goal function $G$ and retrieval size $k$. The goal function $G$ consists of both Boolean constraint expression $B$ and a numeric optimization expression $O$, i.e., $G = B \cdot O$ (by treating $B$ as a function of $\{0,1\}$ values). As its semantics, intuitively, such a $k$-constrained optimization query, over a database $D$, is to optimize the goal function $G$ over the domain defined by $D$, i.e., find the $k$ top tuples $t \in D$, such that $G(t)$ 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 $B$ and “soft” criteria $O$, resulting in the overall goal $G$. Meanwhile, as a retrieval task over large data, users are often interested in a relatively small number $k$ of best matches.

**Example 12 (Data Retrieval):** To search for houses with a reasonable tradeoff of size and price, from a House relation $h$, we may formulate query $Q = (B: h.price \leq 200k \lor h.price \geq 400k, O: \frac{h.size}{|h.price-300k|}, k: 1)$, 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 \leq 200k \lor h.price \geq 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: $Q_c = (B: (h.price \leq 200k \lor h.price \geq 400k) \land h.zipcode = c.zipcode, O: \frac{h.size}{|h.price-300k|} \times c.crimerate^{-1}, k: 10)$

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

**Example 13 (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 $Q_d = (B: s1.itemid = s2.itemid \land s2.year - s1.year = 1, O: s2.sale - s1.sale, k: 10)$
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 B order by O limit k` of Example 12 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 $B$ and $O$, they often process the two expressions step-by-step, lacking an overall integrated search in the $G$-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 $G$ functions are *monotonic*. The monotonicity requires $G$ to be non-decreasing if all its parameters are non-decreasing. With the general combined Boolean and ranking conditions of $B \cdot O$, $G$ is rarely monotonic– In fact, *none* of the above examples is monotonic.

As the main thesis of this work, 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 $G$ over the database $D$ as the domain. Much like *function optimization* in numeric analysis, we are to maximize a function $G$. 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 $G$, with the help of indices as access methods, over tuples in $D$. 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 $G$– 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 chapter 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 5.7 will discuss in details.

This chapter will develop OPT* framework. Specifically, we formalize and motivate the problem in
Section 5.2. To develop the OPT* encoding, Section 5.3 begins with the state search perspective, and
Section 5.4 completes with the functional optimization perspective. We then discuss the OPT* framework
in Section 5.5, report experiment in 5.6, and summarize the related work in 5.7.

5.2 Motivation

5.2.1 Query Model

As Section 5.1 discussed, querying a database can be successfully modeled as a $k$-constrained optimization
problem, with the dual goals of optimizing both constraint expression $B$ and optimization expression $O$ to
retrieve the top results of size $k$.

Toward seamless optimization of both $B$ and $O$, we view that the two expressions together form a unified
goal function $G$: That is, the score of $G$ for a tuple $t$ that satisfies the constraint expression $B$ is simply its
<table>
<thead>
<tr>
<th>A1:Price</th>
<th>A2:Size</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>600K</td>
<td>4500</td>
<td>15</td>
</tr>
<tr>
<td>350K</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>150K</td>
<td>1000</td>
<td>6.67</td>
</tr>
<tr>
<td>250K</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>300K</td>
<td>3500</td>
<td>0</td>
</tr>
<tr>
<td>80K</td>
<td>500</td>
<td>2.27</td>
</tr>
</tbody>
</table>

*Figure 5.1: Database $\mathcal{D}$ with relation $D_1$.*

score of optimization expression $O(t)$. In contrast, a tuple that fails to satisfy $B$ 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 $A_i, i = 1, \ldots, m$ denote $m$ query attributes (either all from a single relation, e.g., as in $Q$, or from multiple joined relations, e.g., as in $Q_c$), $\mathcal{D}$ a database instance, and $\text{dom}(A_i)$ the domain values of attribute $A_i$. A $k$-constrained optimization query can be formally defined as follows:

**Definition 7 (Query model):** Let a $k$-constrained optimization query $Q$ be defined over query attributes $A_i, i = 1, \ldots, m$ and join $n$ relations $D_1, \ldots, D_n$. Let $\text{rel}(A_i)$ denote the relation $D_j$ that $A_i$ belongs to, $\text{dom}(A_i)$ the domain values of $A_i$ and $\mathcal{D} = D_1 \times \cdots \times D_n$. A $k$-constrained optimization query $Q$ is a two tuple $< \mathcal{G}, k >$, where:

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

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

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

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

---

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

106
To illustrate our query model, consider query $Q$ in Example 12 with two query attributes price and size. Figure 5.1 describes an example database instance $D$, while Figure 5.2 plots the landscape of $G$ over the domains of price and size. $Q = < G, 1 >$ retrieves the top tuple in $D$ maximizing $G$, i.e., $(600k, 4500)$, which corresponds to the high-scoring point in the landscape. With this abstraction, Section 5.2.2 develops a query mechanism to efficiently search $D$ for top-$k$ answers.

5.2.2 Query Mechanism

As Section 5.2.1 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 $D$ that maximize the goal function $G$ with the minimal cost. Toward the goal, the first requirement of searching over database tuples in $D$ clearly suggests the use of access methods, e.g., table scan or index scan, over $D$. 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 $G$.

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

Second, from continuous function optimization, our goal is to optimize $G$ over the domain of the database. To perform a focused search toward optimizing $G$, we may consider using existing function optimization schemes, e.g., hill climbing or genetic algorithm [?]. However, such schemes identify the values optimizing the given function over continuous value space, defined by domains of the query attributes $\text{dom}(A_1) \times \cdots \times \text{dom}(A_m)$. In contrast, a $k$-constrained optimization query optimizes over database $D$ 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 $G$, to minimize the overall cost. We state the goal of such an integrated framework below:

**Definition 8 (Query evaluation):** Given a database $D$ and indices $I = <I_1, \ldots, I_m>$ on query attributes $A_i, i = 1, \ldots, m$, the goal of answering a given query $Q = <G, k>$ is to find top-$k$ results $t_1, \ldots, t_k$ in $D$, such that $G(t_i)$ is maximum, over a state space constructed by $I$ (as we will discuss in Section 5.3) with a minimal access cost, which we formulate as:

$$
cost = w_l * N_l + w_t * N_t + w_i * N_i
$$

(5.2)

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 $Q$. Figure 5.3(a) 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 5.3(b) 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.

### 5.2.3 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 5.2.2 that indices essentially lay out a “map” of a discrete set of regions and provide effective traversals among them (Figure 5.3(d)). However, this map of regions is flat with no distinction among regions, while each region differs in $G$ scores as Figure 5.3(b) illustrates. To enable an efficient search, we thus let regions to reflect the landscape of $G$ (as in Figure 5.3(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, $A^*$ [?], which we will describe algorithmically in Section 5.5, 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 5.4), \( A^* \) 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 5.7, existing algorithms build upon their problem-specific assumptions on the goal functions or index traversal. To illustrate, a representative top-\( k \) algorithm \( TA \) \([?]\) assumes the monotonicity of \( G \) 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 \( G \) or how we traverse on index, we generalize our search to support (1) arbitrary \( G \), (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 \( G \), 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 5.3 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 5.4 will discuss the challenge from the continuous function optimization perspective, by introducing the landscape of G to the search space, which involves defining search heuristics to quantify the “promise” of states and identify valid initial states.

5.3 Index-induced State Space: A*-Driven Construction

As motivated in Section 5.2, 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 5.3.1 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 5.3.2 discusses. As the main product of state space construction, Algorithm 5.3.1 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 $\text{dom}(n_i), n_i \in R$ to denote the range of values defined by index node $n_i$. For instance, in Figure 5.3, node $a_2$ has $\text{dom}(a_2) = [0, 250k]$.

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 5.3 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 $\text{Child}(n)$. For a leaf node, the reachable nodes consist of sibling nodes $\text{Sibling}(n)$ reached through the bidirectional interleaf pointers and tuples $\text{Tuple}(n)$ 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 $I.E$.

5.3.1 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 $D$. Specifically, to construct the state space from a set of index graph $I = < I_1 \ldots I_m >$, we need to define a composite graph $I = (V, E)$ over $I$, where $V$ is the set of states and $E$ is the set of transitions between states.

**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 $D$.

**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 5.3(a) defines a region $M_{33} = [a_3, b_3]$ in Figure 5.3(b). In general, any combination of index nodes $<n_1, \ldots, n_m>$ from $\bowtie_i I_i.R$ is a valid state representing a region defined by $\text{dom}(<n_1, \ldots, n_m>) = \bowtie_i \text{dom}(n_i)$. 

112
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 Algorithm 5.3.1 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 \( r = <n_1, \ldots, n_m> \) is a leaf state if all nodes \( n_i, i = 1, \ldots, m \) 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 5.3(b).

**Tuple State:** In parallel to tuple component \( T \) in an index graph, each tuple \( t \in D \) 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 \( G \).

**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 5.2.

Essentially, to construct transitions between states, we need to define a \( \text{Next} \) 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 Algorithm 5.3.1, 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 5.5 will discuss. Therefore, the graph is effectively constructed and selectively materialized
“on-the-fly” rather than statically pre-computed.

Example 14: Continue Example 12. Given a set of states constructed from the set of index graph $I$, Figure 5.4 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 $M_{67} \rightarrow M_{77} \rightarrow 1$.

5.3.2 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 $G$. Therefore, our problem is to find out such “optimal tuple
states” with maximal $G$-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., $\text{Next}(t) = \{t^*\}$. 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.
Algorithm 5.3.1 State space: OPT* encoding.

OPT* Encoding: $k$-constrained optimization of $G$ over $D$ using $T$

Input: Indices $I = (I_1, \ldots, I_m)$, Goal function $G$, Databases $D = \bigcup_{i=1}^{n} D_i$

Output: State space $I = (V, E)$, BlackLink set $\bar{E}$, Initial states $S$

INDEX-INDUCED SPACE CONSTRUCTION

Input: Indices $I = (I_1, \ldots, I_m)$, Goal function $G$, Database $D = \bigcup_{i=1}^{n} D_i$

Output: State space $I = (V, E)$

/*construct states of $I*$/
$I.R = \bigcup_{i=1}^{m} I_i.R$  
$I.T = D$  
$I.V = I.R \cup I.T \cup \{t^\ast\}$

/* construct transitions of $I*$/
for $r =< n_1, \ldots, n_m > \in I.V$: do
/*construct reachable states for $r*$/
Next($r$) = $\{t^\ast\}$ if $r$ is a tuple state
Next($r$) = $\bigcup_{i=1,\ldots,m} \text{Child}(n_i)$ if $r$ is an internal state
Next($r$) = Neighbor($r$)$\cup$ Tuple($r$) if $r$ is a leaf state, where
Neighbor($r$) = $\bigcup_{i=1,\ldots,m} \text{Sibling}(n_i) \cup \{n_i\} \setminus r$
/*join TIDs from multiple relations $D_1, \ldots, D_n$*/
Tuple($r$) = $\bigcup_{i=1,\ldots,n} T_i$, where $T_i = \bigcap_{j \in \text{rel}(A_j)=D_i} \text{Tuple}(n_j)$

$E = \{(u, v) | v \in \text{Next}(r)\}$
for $(u, v) \in E$: do
$d(u, v) = -G(u) \text{ if } v = t^\ast$
$d(u, v) = 0 \text{ otherwise}$

GOAL-INDUCED SPACE CONFIGURATION($I, G$)

Input: State space $I = (V, E)$, Goal function $G$

Output: Black set $\bar{E}$, Initial states $S$

$h_G(r) = \text{OptMax}(G, r)$
$\bar{E} = \{(u, v) | (u, v) \in I.E \land h(u) < h(v)\}$
$O = \text{OptPoint}(G, \text{dom}(I.root))$
$S = \{r \in I.R | \forall p \in O, \exists r \in S \land p \in \text{dom}(r)\}$
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 5.4 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 $G$-score of the tuple state, i.e., $d(t, t^*) = -G(t)$.

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 $P = (v_1, \ldots, v_n, v_{n+1})$ is the overall distance of $P$, i.e., $d(P) = \sum_{i=1}^{n} d(v_i, v_{i+1})$. Specifically, for any two paths $P_1 = (v_1, \ldots, v_n, t^*)$ and $P_2 = (v'_1, \ldots, v'_n, t^*)$, 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, $i = 1, \ldots, n-1$, $d(v_i, v_{i+1}) = 0$, 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 Algorithm 5.3.1 depicts, by assigning all edges between “physical” states with distance 0 and edges from tuple states to the pseudo goal with distance as inverse $G$-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.

### 5.4 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 5.2, 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 5.4.1). Second, it identifies a set of initial states decided by the goal function to start the search properly (Section 5.4.2).

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 5.4.1, 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 5.4.2, 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 $\mathcal{G}(x_1, \ldots, x_m)$ and a domain of values $\text{dom} = [x^1_1, x^2_1] \times \cdots \times [x^1_m, x^2_m]$ where $x_i \in [x^1_i, x^2_i]$. It returns a set of local optima $O$ and an upper bound score $U$ that can be achieved within the domain, i.e.,

$$\langle O, U \rangle = \text{OPT}(\mathcal{G}, \text{dom}) \quad (5.3)$$
where \( O = \{ p | p \in \text{dom} \land p \text{ is a local optima} \} \) and \( U = \max_{p \in \text{dom}} G(p) \). Specifically, we use \texttt{OptMax} to denote the function that returns the \( U \) component of \( \text{OPT} \), and \texttt{OptPoint} to denote function that returns \( O \) component of \( \text{OPT} \).

As implementation to this procedure, there are three categories of techniques. First, analytical methods\footnote{To avoid ambiguity between the monotonicity property of functions, typically referred to by top-k queries, we use the term \textit{descendance}.} 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\footnote{To avoid ambiguity between the monotonicity property of functions, typically referred to by top-k queries, we use the term \textit{descendance}.}, 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 \textit{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.

### 5.4.1 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 5.4 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 \textit{admissible} and \textit{descending}. First, \textit{admissibility} requires the heuristic function does never overestimate the distance to the goal. Second, \textit{descendence} 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 $G$-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 $G$ can achieve within the value ranges, i.e.,

$$h_G(r) = \text{OptMax}(G, dom(r))$$

However, although such an estimation is the best we can make, it does *not* satisfy the admissibility and descendence property because of the sibling edges between leaf states. Let us use the following example to illustrate.

**Example 15:** 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 descendance 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 descendance 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 16:** Consider the state space (partially) shown in Figure 5.4. All leaf states \( M_{6i}, i = 4, 5, 6, 7 \) (corresponding to regions in the third column of Figure 5.3(b)) have heuristics score 0 because they disqualify the price range in query \( Q \). 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 9:** 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 Algorithm 5.3.1 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.

**5.4.2 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 16–suppose the search starts at \( M_{57} \), it cannot reach \( M_{77} \), which contains the top answer to query \( Q \). 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 \(\text{OptPoint}(G, \text{dom}(I.root))\), where \(I.root\) is the root state of \(I\), defined by roots of each index.

**Example 17:** As the landscape of Figure 5.2 shows, the goal function \(G\) has two local optima \(<200k, 4500>\) and \(<400k, 4500>\). The two local optima are located in \(M_{57}\) and \(M_{77}\) respectively. Therefore \(S = \{M_{57}, M_{77}\}\) covers all local optima points of \(G\). Starting from \(S\) - closest to the goal state, bottom-up approach leads to the most efficient search, as shown in Section 5.5.

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

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 \(Q = <G, k>\), a state space \(I\), a heuristics function \(h(r) = \text{OptMax}(G, r)\), a set of states \(S\), \(A^*\) search guarantees to correctly find answers to \(Q\) if \(S\) 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 16, 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 5.5 will discuss.

5.5 OPT* Search

Upon the state space correctly encoded (as shown in Figure 5.3.1), 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 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 5.3, 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.

5.5.1 Skeleton of OPT*

Algorithm 5.5.1 Query algorithm: OPT* search.

Procedure OPT*(I, G, k)
Input: Indices $I = \langle I_1, \ldots, I_m \rangle$, Goal function $G$, Result size $k$
Output: top-$k$ results TupleQ

$\tau \leftarrow -\infty$

OPT_INITIATE(ToDoQ, I)

while not REACHGOAL() do

$r = GETNEXTSTATE()$

if $r \in HAVEDONEQ$ then

continue

insert $r$ to HAVEDONEQ

$newR <=\ EXAND(r)$

for each state $s \in newR$ do

OPT_HEURISTIC($s, G$)

if ELIGIBLE($s$) then

if $s$ is Region State then

insert $t$ into ToDoQ

else if $s$ is Tuple State then

insert $t$ into TupleQ

$\tau = \min(TupleQ)$
In this section, we present our OPT* algorithm. Algorithm 5.5.1 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 5.3. 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 \( G \)-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 \( G \)-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.

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 threshold \( \tau \) to keep track of the score of the \( k \)th-tuple in TupleQ. If \( \tau \) 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.

5.5.2 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 5.2. First, \( A^* \) optimizes the total number of nodes visited in the state
space, i.e., $N_l + N_i + N_t$, 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

$$w_l \ast N_l + w_i \ast N_i + w_t \ast N_t \text{ if } w_l > w_i \text{ and } w_t > w_i$$

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 5.4 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

*Figure 5.5: Search Tree.*
the root state, which makes the largest number of hops to get to 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 \( G \) 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 2:** Given a state space and a heuristics function, \( \text{OPT}^* \) 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 5.5. 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 5.5 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 3:** 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 \( G \) has an infinite number of local optima. For instance, consider goal function \( G = \frac{1}{(x-y)^2+1} \). 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 5.6, we will assume the bottom-up approach. Putting together, the following example shows the search route taken by OPT*.

**Example 18:** Consider Example 12. We initialize the search with $S = \{M_{57}, M_{77}\}$. Starting from $M_{77}$, the search visits a sequence of leaf and internal states in the order labeled in Figure 5.5. 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.

5.6 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 5.6.1, then report our benchmark query experiments over real data in Section 5.6.2 and controlled query experiments over synthetic data in Section 5.6.3.

5.6.1 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 5.5.

**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 5.2. 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,

\[ cost = N_l + N_t \]  

(5.4)

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 [?] 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-	extit{k} algorithms, specifically, \textit{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 \textit{TA} performs *random access* to retrieve tuples with \textit{TID}, and therefore once the tuple is retrieved, we can evaluate its \(G\)-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-	extit{k} results using only ranking expression.

However, as \textit{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 \textit{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.

### 5.6.2 Benchmark Queries on Real Data

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

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

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

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

\[
Q_2: \frac{\text{size} \times \text{bedrooms}}{\text{price} - 350k} : [\text{price} < 400k \land \text{size} > 4000]
\]

\[
Q_3: \frac{\text{size}}{\text{price}} : \text{[bedrooms = 3 \lor bedrms = 4]}
\]

**Results**: Figure 5.6 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 \textit{BthenR} approaches and \textit{OPT*} are applicable for queries in this set. For \textit{BthenR} approaches, we show both the best case scenario where tuples accessed are clustered (\textit{BthenR\_Min}) and worst case scenario where tuples are scattered in the disks (\textit{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.

5.6.3 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 logvarianotenormal, 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 \(x_1, \ldots, x_4\) in ranges of \((0, 100)\), \((0, 1000)\), \((0, 5000)\) and \((0, 10000)\) respective. We generate gaussian and logvarianotenormal 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 5.4, to compute the local optima for function optimization.

- $F_1$: The first set of queries is linear average, which are widely used in top-k algorithms. The queries are generated using:

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

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.

- $F_2$: 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 $O$. 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:

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

$$B : x_1 \subseteq r_1 \land x_2 \subseteq r_2$$

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

- $F_3$: 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

$$O(s, t) = s.a1 \times t.b1 + s.a2 \times t.b2 : [s.a2 = t.b2]$$

$$B : s.a1 \subseteq r_1 \land t.a1 \subseteq r_2$$

where $r_1$ and $r_2$ are randomly generated value ranges for $x_1$ and $x_2$ respectively.
**Results:** Figure 5.7(a) to (c) show the average number of page accesses of three sets of queries against three synthetic datasets. In particular, Figure 5.7(b) shows the average page accesses for 33 linear average queries over the three datasets. Observe that, in all queries, \( R_{thenB} \) 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 \( B_{thenR\_Min} \). In particular, in Figure 5.7(d) over gaussian dataset, \( OPT^* \) outperforms \( B_{thenR\_Max} \) and \( B_{thenR\_Min} \) by more than three and one orders of magnitude respectively. The performance contrasts significantly in \( F_3 \), 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 (\( B_{thenR} \) 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 5.7, 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 5.7), 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 5.7 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 then focus on controlling the relative selectivity of constraint and optimization expressions. Varying such relative selectivity ratio provides interesting intuitions as such ratio determines the relative performance of baseline approaches– Intuitively, BthenR approach performs well when the boolean selectivity is high. In contrast, RthenB approach performs well when the ranking selectivity is high. We will study how our approach compares with those approaches under different settings.

Datasets: To control the relative selectivity of constraint and optimization expression, we generate a dataset with contrasting distributions over different attributes to control selectivity by how we weigh each (as we will elaborate further). In particular, the dataset contains three attributes using gaussian distribution. The first attribute follows a “low/concentrate” gaussian distribution generated using a small mean and variation. In contrast, the second attribute follows a “high/loose” gaussian distribution generated using a large mean and variation. The third attribute sets mean and variation in the middle and will be used in controlling the boolean selectivity.
Queries: Using the dataset with contrasting distributions, we can control the selectivity by varying query values of the following form: \( O : w_1 * x_1 + w_2 * x_2, B : x_2 \subseteq r_1 \land x_3 \subseteq r_2 \).

In particular, to control the selectivity of constraint expressions, we control the ranges of \( r_1 \) and \( r_2 \) of gaussian-distributed attributes \( x_2 \) and \( x_3 \), by fixing the width of ranges and varying the starting points of the range. Meanwhile, to control the selectivity of optimization expressions, we need to control how similar the scores of the tuples are. That is, if tuples have very similar scores, the algorithm needs to retrieve more tuples to “disambiguate” top-k results from the rest (i.e., low selectivity). In contrast, if the tuples significantly vary in scores, the algorithm will terminate quickly (i.e., high selectivity). We implement this idea by varying the ratio of \( w_1 \) and \( w_2 \) to control the similarity of score distributions and thus the relative selectivity of constraint and optimization expression. In particular, if \( w_1 : w_2 \) is small, \( x_2 \) with a large variation dominates the entire score distributions to vary significantly (i.e., high selectivity). In contrast, if the ratio of \( w_1 : w_2 \) is high enough, none of the attributes dominate the final ranking and tuples will have similar scores (i.e., low selectivity).

Results: Figure 5.8(a) and (b) show how the performance of different approaches changes when the selectivity of boolean and ranking changes. Figure 5.8(a) shows the number of page accesses when we vary the boolean selectivity. As we can see that, OPT* outperforms the baseline approaches by up to four orders of magnitude. In particular, note that the performance of BthenR increases when the constraint expression
becomes increasingly selective. Figure 5.8(b) similarly shows how the performance of different approaches changes when the selectivity of optimization expression changes. Again, OPT* generally performs the best among all situations, by outperforming baseline approaches by an order of magnitude. For \textit{BthenR} approach, the performance is stabilized because the number of tuples satisfying the constraint expression is fixed. Note the performance of \textit{RthenB} approach improves when optimization expressions become increasingly selective.

### 5.7 Related Work

As mentioned in Section 5.1, 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 \cite{1, 2, 3, 4, 5, 6, 7, 8, 9} 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., \cite{10, 11}) which navigate attribute indices to process spatial queries, e.g., \textit{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 (\textit{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 \cite{12, 13, 14} supporting ranking inside the database system and tightly coupling with the Boolean query engine. Recent works on \cite{15, 16} 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 $G$; 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 [?], 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 minimizes a given distance function as the goal function $G$. Third, while not intended, a representative top-$k$ algorithm $TA$ [?] 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. Finally, the recent work [?] deploys the global search mechanism over pre-computed data cubes. Those data cubes essentially materialize the state space induced by the indices, and search is thus performed in this materialized space. This work focuses on the design of data cubes to store ranking-aware information in facilitating the search.

Figure 5.9 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 $G$, the indices used, and their traversals. Observe that all three lines of works build upon problem-specific assumptions on $G$ 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 $G$, combining multiple indices, using both hierarchical and interleaf traversals.

Finally, people have studied constrained logic programming or CLP for short [?] 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.
Chapter 6

Related Work

As we have discussed works related to individual tasks, i.e., query capability extraction and query translation, in the corresponding chapters, this chapter overviews the existing integration systems related and contrast to our MetaQuerier system.

The MetaQuerier has a distinct focus as a large scale integration system— for dynamic discovery and on-the-fly mediation. On one hand, traditionally, information integration has assumed relatively small and pre-configured systems. On the other hand, the recent emergence of Web integration mostly considers only various subtasks.

Information integration has traditionally focused on relatively small-scaled pre-configured systems [?, ?] (e.g., Information Manifold [?], TSIMMIS [?], Clio [?]). In particular, relevant issues on schema matching [?, ?, ?], schema mapping [?, ?], and query mediation [?, ?, ?, ?, ?, ?, ?] have been extensively studied.

In contrast, we are facing a “dynamic” and “ad-hoc” scenario of integrating databases on the Web. Such large-scale integration imposes different requirements and thus faces new challenges: To deal with this large scale, many tasks have to be automated, unlike integration at a small scale where sources can be manually “configured.” First, for finding sources, we must dynamically select relevant sources according to user’s ad-hoc information need, but not to fix a small set of sources for integration. Second, for modeling sources, we must automatically discover their query capabilities, but not to assume pre-configured wrappers providing source descriptions. Third, for querying sources, we must “on-the-fly” translate queries for unseen sources, but not to hard-code per-source knowledge specifically for each source.

In terms of our “solutions,” we note that, as one of our unified insights, the MetaQuerier exploits hidden
“clues” revealed by holistic sources to discover underlying semantics. For instance, in form extraction, the observed condition “patterns” are such hidden clues across holistic sources. By hypothesizing a hidden-syntax to capture these hidden clues, we discover the underlying semantics, i.e., query capabilities, by a parsing based framework.

Along this line of exploiting hidden regularities, several research efforts, which have also emerged recently, share this similar holistic insight— but specifically for the schema matching task. In particular, references [?, ?] exploit clustering for holistically matching many schemas. Reference [?] proposes a “corpus-based” idea, which uses a separately-built schema corpus as a holistic “knowledge-base” for assisting matching of unseen sources.

While sharing similar holistic frameworks, in contrast to these efforts, we have developed our holistic-integration insight to generally tackle with “semantics discovery” common in many large-scale integration tasks, which generalize beyond the specific task of schema matching to interface extraction and query translation as well. Further, besides “statistical” analysis (which most other works have based upon), there are a wide range of applicable techniques (e.g., syntactical parsing [?] for interface extraction; locality-based search [?] for query translation) to generally explore holistic hidden regularity for semantics discovery.
Chapter 7

Conclusion

The Web has been dramatically deepened by the prevalence of databases online, which bring up great challenges to information access and exploration. To begin with, finding useful databases to fulfill users information needs is challenging, as relevant databases are overwhelmed by massive sources on the Web, including surface and deep Web sources. How to guide users to the right entrances to the Web databases is the first challenge. Further, after finding appropriate Web databases, to query them uniformly is a second challenge. As those alternative databases differ in details of querying, to integrate those sources hiding their discrepancy is important to alleviate users from querying details of individual sources. Finally, with queries submitted to databases, a third challenge is to process those queries efficiently.

Tackling those challenges, this thesis studies several concrete and critical issues in building large scale information integration system on the Web.

First, for finding Web databases, this thesis develops an online search facility based on schema models of Web databases. Our key insight is that: schemas of databases capture the intuitive features of the type of objects stored in the databases, and therefore they are natural and intuitive to search by. Further, inherent to the type of objects, schemas of alternative databases providing the same type of objects naturally form localities or co-clusters. Such a co-clustering phenomenon motivates an associativity based measurement for capturing semantic relevance, i.e., we can measure the relevance by how databases are associated directly or indirectly through schemas. To realize the associativity measure, we adapt the iterative computation mechanism to propagate relevance progressively. Taking advantage of the co-clustering phenomenon, we further develop online query processing techniques, which reduces the scope of computation and speeds up
the convergence of top ranked results for early output.

Second, for integrating Web databases, this thesis studies how to understand the query capability of a Web database and how to issue valid query over it. Both challenges regard to the core problem of integration, i.e., semantic discovery— to understand what a query interface says and know how to query it. Our key insight in semantic discovery is that: sources at large scale are not chaotic— they share “regularities.” Such regularities, as some type of clues, connect the underlying semantics with its presentation, as Figure 7.1 illustrates. Therefore, such regularities shed light for dynamic semantics discovery in large scale integration: By identifying the holistic regularity, our integration task, to discover the desired semantics, is thus the inverse of this semantics-to-presentations connection. That is, our integration framework can tackle large scale challenges by developing some reverse analysis, which holistically analyzes the shallow clues, as guided by the hidden regularity, to discover the desired semantics.

For understanding query interfaces, such regularities are the condition patterns as building blocks commonly used for constructing interfaces. Leveraging such regularities, we develop a parsing based framework which captures the regularities with a hypothetical grammar and understands the semantics by a best-effort parser. For querying databases, such regularities are the data types that are commonly used to present various query conditions. Conditions of the same data type share similar presentation templates and common translation knowledge. Therefore, leveraging the regularities of data types, we build a type-based translator.
which encodes the translation knowledge for a particular data type and handles translation between query conditions of this type.

Finally, for processing queries inside a database, this thesis proposes an OPT* framework to evaluate ranked queries as $k$-constrained optimization problem. 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.

In summary, this thesis studied several important problems in large-scale information integration systems, proposed novel techniques to address those problems and evaluated extensively the effectiveness of the approaches. Overall, the thesis shows that large-scale integration is not only possible but also promising under the holistic integration framework, leveraging the hidden regularities revealed among holistic sources.
Author's Biography

Zhen Zhang was born in China, on February 24, 1976. She graduated from the Peking University with a Bachelor of Science in Computer Science in 1998 and a Master of Science in Computer Science in 2001. Zhen then relocated to Champaign, Illinois, to continue the graduate study in computer science. She completed a Doctor of Philosophy in Computer Science from the University of Illinois at Urbana-Champaign in 2007. Following the completion of her Ph.D., Zhen will begin work for Cazoodle Inc. as Chief Technical Officer.