Answering Imprecise Structured Search Queries

by

Arnab Nandi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:
Professor Hosagrahar V. Jagadish, Chair
Professor Dragomir R. Radev
Assistant Professor Eytan Adar
Assistant Professor Michael J. Cafarella
Assistant Professor Kristen R. LeFevre

Acknowledgments

I would like to thank my advisor H. V. Jagadish for guiding me through this journey. His infinite energy towards doing great research, and his passion towards shaping the minds of his students will always be an inspiration. It has been a privilege to be his student.

I am fortunate to have had many mentors on the path towards my Ph.D. I would like to thank S. Sudarshan and Soumen Chakrabarti at the Indian Institute of Technology Bombay for introducing me to database research. I am indebted to Phil Bernstein at Microsoft Research for his guidance and support both during and after my internship. I would like to express my utmost gratitude to my friend, colleague and mentor Cong Yu. His insights have immensely impacted my research, and his mentorship has made me a better researcher.

It has been an honor to be part of the University of Michigan Database Group. I would like to thank former and current colleagues Aaron Elkiss, Adriane Chapman, Allie Mazzia, Anna Shaverdian, Bin Liu, Cong Yu, Dan Fabbri, Fernando Farfan, Glenn Tarcea, Jing Zhang, Lujun Fang, Magesh Jayapandian, Manish Singh, Michael Morse, Nate Derbinsky, Neamat El Tazi, Nuwee Wiwatwattana, Qian Li, Rajesh Bejugam, Sandeep Tata, Stelios Paparizos, Willis Lang, You Jung Kim, Yuanyuan Tian, Yun Chen, Yunyao Li and Yuqing Wu for all the support over the years.

I am lucky to have a wonderful set of friends and housemates. Thank you Akash Bhattacharya, Alex Gray, Amy Kao Wester, Benji Wester, Bindi Dharia, Carol Girata, Dan Peek, Darshan Karwat, Katherine Goodwin, Kaushik Veeraraghavan, Meg Allison, Neha Kaul, Nick Gorski, Rachna Lal, Tim Whittemore and Urmila Kamat for making Ann Arbor a fun and memorable experience.

I would also like to thank my committee members Dragomir Radev, Eytan Adar, Kristen Lefevre and Michael Cafarella for their reviews and helpful comments towards my proposal and dissertation.

And above all, I would like to thank my parents for their love, support and blessings.

Table of Contents

List of Tables

Table

List of Figures

Figure

Chapter 1
Introduction

Humans are increasingly becoming the primary consumer of structured data. As the volume and heterogeneity of data produced in the world increases, the existing paradigm of using an application layer to query and search for information in data is becoming infeasible. The human end-user is overwhelmed with a barrage of diverse query and data models. Due to the lack of familiarity with the data sources, search queries issued by the user are typically found to be imprecise.

1.1 Motivation

1.1.1 Structured Search on Unfamiliar Data

This dissertation addresses the often-encountered problem of querying a database without complete information about it. The average computer user encounters many such databases in her daily activities, whether it is her email, her RSS feed, her social networking application or even the structured parts of the internet, such as IMDb or Wikipedia. Each of these databases is often searched upon, but their schema or the data contained in them is often beyond the user's knowledge or control.

In this scenario, the user faces a multitude of challenges. First, the user needs to know about the *query language*. Since keyword search is the simplest form of search, the user tends to type in a few words. Any richer interaction requires informing the user about the query language. Second, the user is typically *unaware of the schema* of the database.

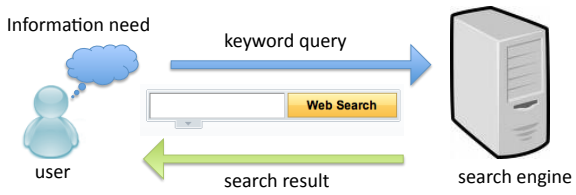


Figure 1.1: Conventional Search interfaces discourage expressibility.

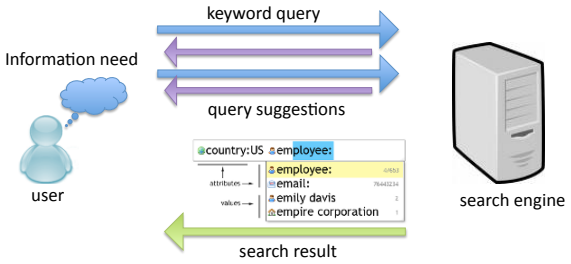


Figure 1.2: Autocompletion-based Search encourages rich queries through rapid interaction.

Without this information, it becomes hard for the user to express queries that return the right kinds of information. Third, the user is *unaware of the actual values* in database, and hence spends time issuing queries with spelling errors, queries that lead to no results, or queries that are not as expressive as they could be if the user had an idea of what is contained in the database. The instant response interface attempts to solve these problems by providing insights into the database during the query construction process.

1.1.2 A Fatter Pipe for Query Specification

Search has conventionally been a strictly stateless interaction. Given a query q, a search engine is expected to return a set of results $\mathbf{R}$. The query q is typically a simple set of words typed into a text input box. Hence, it appears that no matter how complex our information needs are, or how complex the search engine is, the convention of a single search box has restricted us to an extremely hard problem of communicating back and forth rich information through a *thin pipe* – the query input box. The thinness of this pipe is motivated by the simplicity of the interface. A single input box is less daunting than a complicated query

language / UI workbench and puts less of a cognitive load on the user.

While being a user-friendly approach, the thinness of this interface has undesirable implications for both the user and the search engine. The user now has to translate all information needs into a simple string of words, no matter how complex the need is. The search engine has to take this simplified encoding, translate it back into the correct information need, and then evaluate the correct query. Due to the loss of information on both sides, a lot of effort has to be put into techniques such as query disambiguation, amongst others.

Search engines have evolved over time to support more expressive queries. Ideas such as *key:value* operators, faceted search and clustered search are commonly implemented and let the user express her queries better. However, since each new feature adds further complexity (and hence cognitive load) to the search interface and needs to be taught or explained to the user, it diminishes the ease of immediate use.

In the *Qunits* approach, we limit the complexity of our query language and interface. Instead, we follow the idea that *many iterations over a thin pipe can aggregate to a thick pipe*. By overloading each interaction of the user to refine information need, we can derive more information than just a string of words. This is done by extending the query suggestion mechanism that is used to autocomplete search terms, and using it to guide the user to the desired piece of information.

1.1.3 Query Construction vs Query Execution

As mentioned in the previous section, search has conventionally been a " formulate query $\rightarrow$ execute query $\rightarrow$ show result" paradigm.

The qunits system intends to deviate from this approach and merge the query construction and execution stages into a iterative but seamless *qunit guidance* step. This involves execution of partial queries in near-instant ($\leq$ 200ms) time and returning qunit suggestions to the user, guiding her to the most appropriate qunit that answers her information need. Since this is done while the user is typing, the user is left with the exact answer to her

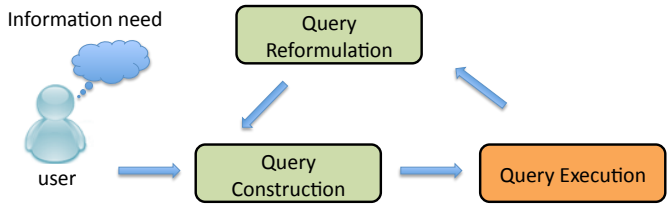


Figure 1.3: Conventional Search Process.

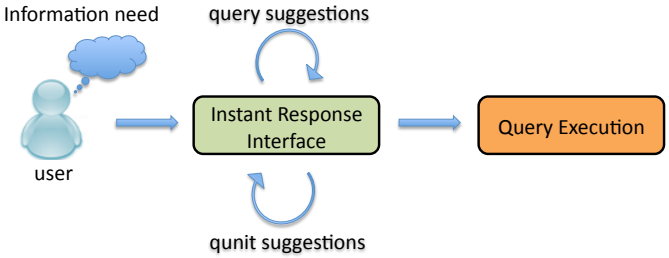


Figure 1.4: Qunits-based search process.

information need by the end of typing out the full query.

1.2 Key Contributions & Dissertation Outline

This dissertation introduces the notion of a "queried unit", or *qunit*, which is the semantic unit of information returned in response to a user's search query. In a qunits-based system, the user comes in with an information need, and is guided to the qunit that is an appropriate response for that need.

The qunits-based paradigm aids the user by systematically shrinking both the query and result spaces. On one end, the query space is reduced by enriching the user's imprecise information need. This is done by extracting information from the user during query input by providing schema and data suggestions. On the other end, the result space is reduced by modeling the structured data into a collection of qunits. This is done using qunit derivation methods that use various sources of information such as query logs. We describe the Qunits concept and paradigm in Chapter 3.

This dissertation describes the design of an autocompletion-style system called an ***instant-response interface***. This interface performs both query and result space reduction by interacting with the user in real time, providing suggestions and pruning candidate qunit results. We developed a prototype for this system and demonstrated the efficacy of this querying paradigm.

In order to provide relevant suggestions to the user, a set of challenges have to be addressed. First, the suggestions have to be relevant to the query at hand, and if accepted, should lead to a valid query. We solve this problem by generating ***schema-based suggestions***, which bias suggestions based on attribute proximity in the schema graph. The second challenge is to be able to deliver these suggestions to the user in real-time. Since querying the database for suggestions is not fast enough, we devise a suggestion architecture which stores multi-attribute schema-based suggestions in a combination of tries and inverted indices in order to instant-response suggestions. The process of query space reduction using the autocompletion-based interface is detailed in Chapter 4.

The third challenge arises when dealing with text- heavy databases. Column values may contain full sentences or even paragraphs of text. Clearly, suggestions with the entire texts are too large, while suggestions on individual words are too small. Thus, one needs to perform suggestions at the phrase level. For this, we describe a trie data structure called a ***FussyTree***, which stores only popular and significant phrases in the database. The FussyTree is then used to generate phrase based suggestions, discussed in Chapter 5.

In addition to providing the users with possible query suggestions, we also suggest possible results, i.e. relevant qunits that are results for the currently suggested queries. As the query progresses, the number of suggested qunits decreases until the user is left with the desired information. Since the universe of possible qunits is infinite, we have to first derive relevant (i.e. high-ranking, based on a quality metric) qunits, i.e. qunits that are likely answers to queries, in a preprocessing step called ***result space reduction***, where the database is transformed into a small collection of qunit definitions. While qunit definitions

can be generated using just the schema and data of the database, this often does not suffice; as the popular human information needs may not be reflected by the system-friendly schema of the database. To capture information need, we propose the use of existing unstructured keyword search query logs, by assembling qunits out of annotated keyword queries. We propose an unsupervised algorithm called ***Query Rollup*** that produces qunits that satisfy typical information needs across a wide range of databases. Result space reduction is further discussed in detail in Chapter 6. We further dive into the process of deriving qunits from keyword search query logs in Chapter 7.

We conclude with the insights gained and future work in Chapter 8. We begin by discussing the related work in the following chapter.

Chapter 2
Related Work

2.1 Database Usability

The broader concept of making databases more accessible using a presentation data model has been discussed by Jagadish et al. [68]. Work on exposing the database query interface using automatically generated forms has been discussed in [73, 74].

Managing complex databases has also been studied in the context of schema summarization [156], using characteristics of the schema. The schema clusters can then be queried upon [157] with low overhead. However, summarizing a schema does not take into account the nature of the query workload or the multiple ways to organize data. A related body of research in view selection [57, 111] attempts to select a set of views in the database that minimizes query execution time of a workload given resource costs, but does not consider the *utility* of the information provided by each view.

2.2 Query By Example

Query-By-Example (QBE) [161] and its successor Office-By-Example (OBE) [162] were visual query languages based on domain relational calculus [36, 84] that allowed users to query a database by creating *example tables* in the query interface. The table was then translated into a more structured language, such as SQL and then executed on the database.

While being a friendlier approach to database querying than SQL, QBE does not perform well with large schema / data scenarios. Enterprise schemas today carry hundreds of tables

and just as many columns per table. This complexity becomes an issue when representing tables as forms on a screen. Furthermore, the user is expected to be aware of the values of the database prior to the query evaluation.

2.3 Visual Query Interfaces

Visual approach to querying databases is a well studied area. Implementations have been researched to query a variety of data models. The QBD[7] system uses the ER schema of the database as a starting point, allowing the user to graphically express a query on it, with the help of a "metaschema" to aid actions such as recursive queries. Kaleidoscape [117] allows users to navigate through an Object-oriented database using a 3D virtual environment, selecting object attributes to perform a set of query operations.

In an early survey, Catarci interfaces for a database: one based on a restrictive natural language, and the other based on "direct manipulation languages", characterized by rapid visual feedback that provides evaluative information for every executed user action. It further classifies direct manipulation interfaces based on database / query representation, result representation, query intent and query formulation paradigm. The Qunits method combines the restrictive and direct manipulation language paradigms by guiding the user through a restrictive language using a highly interactive interface.

A popular paradigm of querying works by constructing mappings between actions in the user interface and an existing restricted database language. VQL [113] addresses the design issues involved with querying complex schema. It introduces graphical primitives that map to textual query notations, allowing users to specify a "query-path" across multiple tables, and also express recursive queries visually. The Polaris System [58] constructs a similar mapping between common user interface actions such as zooming and panning, and relational algebra. This allows users to visualize any standard SQL database using a point and click interface. Work by Liu and Jagadish [90] model relational database as

a spreadsheet, providing user interface methods for selection and projection over SQL databases. While interface cues in the Qunits *query space reduction* phase can be considered as schema discovery, projection and rank operations; they are executed on a normalized collection of qunits and not the actual database.

The need for usable interfaces to data extends to the world of visualization and visual interfaces on data analytics as well [47]. Interactive data mining has been approached by implementing query languages and combining them with an easier-to-use visualization layer [48]. Systems like Tableau[93] perform this by translating VizQL queries to SQL, leveraging the optimizer to remove inefficiencies. In the *Control* project [63], Hellerstein et al. propose a re-architecting the entire database system to incorporate online execution, sampling and result reordering.

2.4 Keyword Search in Databases

Previous efforts towards solving keyword search for databases have concentrated predominantly on ranking. Initial work such as BANKS [22, 77] and DBXplorer [6] use inverted indices on databases to return tuple joins in the form of spanning trees of matched query words and discuss efficient methods of deriving such trees. In the case of XML, one strategy is to identify the smallest element that contains some or all keywords [56], or to identify the smallest element that is *meaningful* [88]. Later work [14] in this area concentrates further on the ranking aspects of the problem, using a metric of authority transfer on a data graph to improve result quality. While these methods provide solutions towards the improvement of ranked results, the composition of the results themselves is not explored.

Work by Koutrika, Simitsis and Ionnadis on *Précis*[139] receives special mention in this context. Précis responses to keyword queries are natural language compositions of "information directly related to the query selections" along with "information implicitly related to them in various ways". Results are not intended to be complete, and instead con-

centrate on being “comprehensive”, akin to the idea of providing users with most probable next browse-points in a browsable database. Their contributions include a rich data model to express facts for the sake of representation, and semi-automated methods to infer these structures by way of annotating a schema graph with weights, providing detailed analysis on execution time and satisfaction of the user.

A major challenge for Précis, and other keyword query systems for databases, is that new database-specific ranking algorithms have to be invented. These are often very complex, and yet tend not to have the quality of ranking algorithms in IR, where they have been studied much more intensively and for much longer. With the qunits proposal, our primary goal is to satisfy the users immediate information need. We address this problem by separating the problem of qunit identification from the problem of query evaluation and ranking. Our aim is to answer the user’s query intent with an atomic unit of information, and not to guide the user to the relevant sections in the database. Unlike Precis results, qunits results are independent of data models. The qunits paradigm first partitions the database into actual pieces of information called qunits, making it amenable to standard IR techniques.

2.5 Faceted Search

Our solution to this problem can be considered a method to organize information in a way that it is more amenable to a user’s information needs. Work has been done in organizing information along individual perspectives using faceted search. Each “facet” is a distinct attribute that users can use to search and browse records. English and Hearst [43, 61] talk about design considerations for building faceted search systems. [149] approaches the faceted search problem by picking hierarchies based on user search patterns. [130] surveys many techniques of dynamic taxonomies and combines them. However, it is clear that faceted search is ideal for homogeneous data with multiple attribute constraints, and not a database with many different kinds of information in it.

Work has been done in building search mechanisms aware of the application logic layer above the database. [42] utilize the architecture patterns in application logic to better index dynamic web applications. [54] approaches this problem by exploiting exposed web service interfaces to provide a unified concept-search interface.

2.6 Deep & Shallow Web

The World Wide Web (WWW) has conventionally been considered as a flat corpus of web documents. Recent work on the "Deep Web" [60] revisited this assumption, acknowledging the existing of websites that are essentially frontends to large databases via web forms. Due to the nature of the query interface, the underlying data in the database is not accessible to the typical web crawler, and hence not searchable via a web search engine. Work by Gravano, et al. [28, 55] address this problem by considering some of the web queries as database queries, and then evaluating them on the appropriate web-inaccessible database.

Work done in the TextRunner [15], Yago [142] and WebTables [30] projects address the "Shallow Web", i.e. the structured data available inside plain HTML documents that were considered "flat" so far. These projects derive millions of tuples of information from crawls of the World Wide Web, with huge and diverse schema. Unlike traditional databases, querying these database are exceptionally hard, due to the lack of a good system to guide the user through the immense schema.

2.7 Query Answering in Web Search

Applications of database-backed query answering are slowly appearing in mainstream search engines. These services have been successful at implementing template-based search, where a keyword query is identified and rerouted as a structured query to a database, answering questions such as "what is the population of china".

With qunits, we propose to take this idea front and center, and make this the central paradigm for all search.

2.8 Query Classification

Due to the generic nature of web search queries, there is great incentive in being able to glean schema-relevant information from the query terms. The task of query classification, and the related task of query annotation typically involves the use of trainable models that are run upon query execution to result in a richer, more restrictive search query. Classifiers are also used to identify query intent [10, 70, 86].

In the presence of sentence-like queries, deep features such as part-of-speech can be used as features in the classification. Zhang and Lee [159] discuss the use of support vector machines (SVMs) to perform query classification on natural language parsed questions, while Krishnan et al. [82] extract features from questions to classify the answer type using CRF-based models.

With smaller search queries, one is forced to augment the query information with other sources. [49, 78] evaluate the use of external web features such as link, URL text and page content from search results to classify queries. Beitzel et al. [18, 19] discuss the use of web search query logs to tackle classification recall problems due to feature sparseness.

2.9 Query Recommendation

Query recommendation is a popular feature in today's search engines. Longitudinal, cognitive and qualitative studies, have shown that query recommendations, usually delivered through an *autocompletion* interface drastically reduces errors in queries [29], promotes iterative refinement [8], and yields higher relevance [132] in the end results.

These recommendations are usually built by mining search query logs from existing users [11, 160]. Ranking of query suggestions is typically based on parameters such as query

volume and query history. Mei, Zhou and Church [106] propose a tuning parameter-free approach to query suggestion, by computing the hitting time on a large scale bipartite graph of queries and clickthrough information.

Jones et al. [76] recommendation also involves rewriting of queries by viewing it as a machine translation problem, for the purpose of simplifying them or for use in sponsored search.

2.10 Imprecise Querying

A popular paradigm to database queries is to assume that the query is inherently imprecise. Systems implementing *keyword search in databases* [123] eschew the notions of structure in the input query and assume that the query is a short list of keywords. The structure in the data is used to heuristically generate a collection of results using the keywords. The results are then ranked and presented to the user. Another approach is to annotate the keyword input with structured data using past query logs [131]. A more traditional approach is to build a layer above relational algebra that uses semantic distances to approximate exact queries[116].

2.11 Query Construction

A large variety of interesting interfaces have built to aid construction of database queries. Query-By-Example [161] and Query-By-Output [148] allow the user to communicate the query by examples of what a result might look like, or the desired output itself. Many web search engines provide query autocompletion based on past query logs. In [120], the concept of autocompletion is used to rapidly suggest predicates to the user in order to create conjunctive `SELECT-PROJECT-JOIN` queries. In a similar light, work has also been done to use mine query SQL logs to derive and suggest reusable query templates to the user[79].

2.12 Iterative Querying

In contrast to autocompletion, the concept of *find-as-you-type* allows the user to quickly iterate through the query process by providing results on each keystroke. In *Complete-search* [17], Bast et al modify the inverted index data structure to provide incrementally changing search results for document search. *TASTIER* [87] provides find-as-you-type in relational databases by partitioning the relation graph. In the information retrieval area, Anick et al.[9] achieve interactive query refinement by extracting key concepts from the results and presenting them to the user. Faceted search[155] extends this to present the user with multiple facets of the results, allowing for mixing of search and browse steps. We believe that the inclusion of *responsiveness* into faceted search will make them more responsive and intuitive.

Chapter 3
The Qunits Paradigm

Keyword search against structured databases has become a popular topic of investigation, since many users find structured queries too hard to express, and enjoy the freedom of a "Google-like" query box into which search terms can be entered. Attempts to address this problem face a fundamental dilemma. Database querying is based on the logic of predicate evaluation, with a precisely defined answer set for a given query. On the other hand, in an information retrieval approach, ranked query results have long been accepted as far superior to results based on boolean query evaluation. As a consequence, when keyword queries are attempted against databases, relatively ad-hoc ranking mechanisms are invented (if ranking is used at all), and there is little leverage from the large body of IR literature regarding how to rank query results.

Our proposal is to create a clear separation between ranking and database querying. This divides the problem into two parts, and allows us to address these separately. The first task is to represent the database, conceptually, as a collection of independent "queried units", or *qunits*, each of which represents the desired result for some query against the database. The second task is to evaluate keyword queries against a collection of qunits, which can be treated as independent documents for query purposes, thereby permitting the use of standard IR techniques. We provide insights that encourage the use of this query paradigm, and discuss preliminary investigations into the efficacy of a qunits-based framework based on a prototype implementation.

3.1 Introduction

Keyword search in databases has received great attention in the recent past, to accommodate queries from users who do not know the structure of the database or are not able to write a formal structured query. Projects such as BANKS [22], Discover [66], ObjectRank [14] and XRank [56] have all focused on providing better algorithms for ranking elements in a database given a keyword query. These algorithms use various data models for the purposes of scoring results. For example, BANKS models the database as a graph, with each tuple represented as a node, and relationships represented as edges. It then considers candidate results that form a minimal spanning tree of the keywords. ObjectRank, on the other hand, combines tuple-level PageRank from a pre-computed data graph with keyword matching. A common theme across such efforts is the great attention paid to different ranking models for search.

This paradigm is similar to one adopted by the information retrieval community who strive to improve document search. For them, the task is to create an algorithm to find the best *document* given a keyword query. Various approaches, such as TF/IDF and PageRank, have emerged as popular approaches to solving the search problem, where the user is presented with a list of *snippets* from top-ranking documents.

However, unlike document collections, large schemas do not have the luxury of a clear definition of *what* a document is, or what a search result comprises. Even if one assumes that the ranking of the results is correct, it remains unclear what information should be presented to the user, i.e. what snippets form a database search result. Records in a schema-rich database have many possible sub-records, parent records, and joinable records, each of which is a candidate for inclusion in the result, but only a few of which are potentially useful to the average user.

For example, consider the keyword query *george clooney movies*. When issued on a standard document search engine, the challenge is straightforward: pick the best document corresponding to the query and return it. However, for a database, the same task becomes

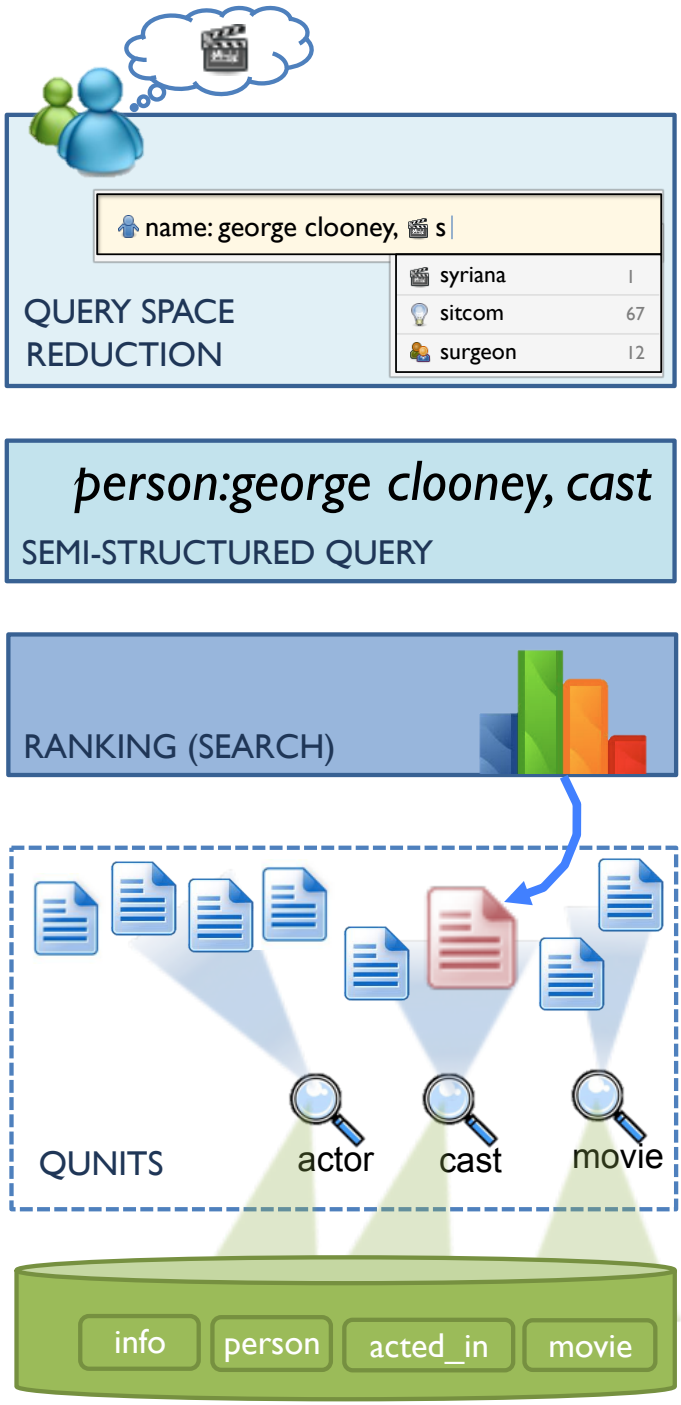


Figure 3.1: The Qunits Search Paradigm : The database is translated into a collection of independent *qunits* using *result space reduction*, discussed in Chapter 6. Given an imprecise query intent, a user interacts with the autocompletion interface discussed in Chapter 4 to construct a semi-structured query, in a process called *query space reduction*. This query is then used to rank (using standard information retrieval techniques) and pick the best *qunit instance* as the result.

nontrivial. In the case of a relational database, should we return just singular tuples that contains all the words from the query? Do we perform some joins to denormalize the result before presenting it to the user? If yes, how many tables do we include, and how much information do we present from them? How do we determine what the important attributes are? The title of a movie isn't unique due to remakes and sequels, and hence it will not be a key. How do we determine it is more useful to the user than the primary key, *movie_id*? Clearly, there are no obvious answers to these questions.

In other words, a database search system requires not just a mechanism to find matches and rank them, but also a systematic method to demarcate the extent of the results. It may appear that simply including the complete spanning tree of joins used to perform the match (in the case of relational systems, such as BANKS), or including the complete sub-tree rooted at the least common ancestor of matching nodes (in the case of XML systems, such as XSearch), will suffice. However these simple techniques are insufficient, often including both too much unwanted information and too little desired information.

One cause of too much information is the chaining of joins through unimportant references, which happens frequently in normalized databases. One reason for too little information, particularly in relational systems, is the prevalence of `id` attributes due to normalization. If this were the only issue, it could be addressed by performing a value join every time an internal "id" element is encountered in the result. However, the use of simple rules such as this also appears insufficient.

Consider, as shown in Figure 3.2, part of the example schema for the Internet Movie Database. The schema is comprised of primarily of two entities[1], *person* and *movie*. Additional tables, such as *cast*, establish relationships, while tables such as *genre* are used to normalize common strings.

Given a keyword query such as *george clooney movies*, a natural solution is to perform a join of ``name = george clooney'' from the `person` table, with the

[1] we use the term *entities* to refer to values in table columns, formatted to be queryable by the end user

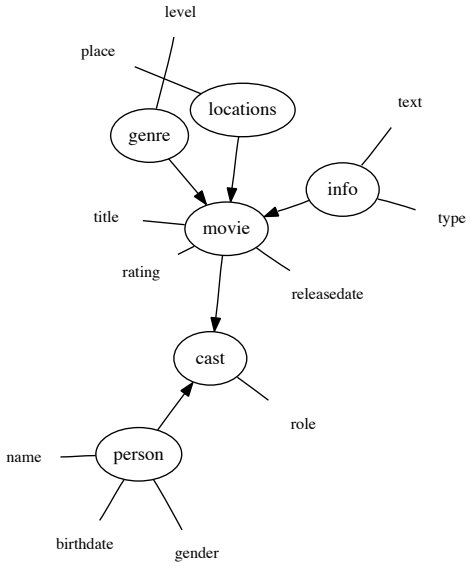


Figure 3.2: A simplified example schema.

`movies` table via `cast`. The `movies` table is normalized and contains `id` pointers to the `locations`, `genre`, and `info` table. There is nothing in terms of database structure to distinguish between these three references through `id`. Yet it is probably the case that resolving the id references to the shooting locations and genre is desired, but not to include the lengthy plot outline (located in the `info` table) in the search result.

In addition to the challenge of demarcating an actual result, the problem of identifying which result to return for a query still remains. Based on prior work in the area of search query log analysis [85, 141], and analyses described in Sections 3.2 and 6.3.1, we observe that:

- A majority of users' queries are underspecified
- The mapping between true information need and the actual query terms is frequently not one-to-one
- The challenge of mapping a query to a specific fragment of the database is nontrivial

We will study these observations in the following section.

3.2 Understanding Search

The Internet Movie Database or IMDb[2] is a well-known repository of movie-related information on the Internet. To gain a deeper understanding of search behavior, we performed a formative user study with five users, all familiar with IMDb and all with a moderate interest in movies. The subjects had a large variance in knowledge about databases. Two were graduate students specializing in databases, while the other three were non-computer science majors. The subjects were asked to consider a hypothetical movie database that could answer all movie-related queries. Given this, the users were asked to come up with five "information needs", and the corresponding keyword queries that they would use to query the database. Information needs were manually distilled from a long-form description provided by the user (1-2 sentences) to concise canonical phrases.

As expected, nearly all our subjects look for the summary page of a movie. But this information need is expressed in five different ways by the users. Similarly, the cast of a movie and finding connections between two actors are also common interests. Once again these are expressed in many different ways. Conversely, a query that only specifies the title of the movie may be specified on account of four different information needs (first column of the table). Similarly, users may specify an actor's name when they mean to look for either of two different pieces of information – the actor's filmography, or information about co-actors. Clearly, there exists a many-to-many relationship between information need and queries.

Another key observation is that 10 of the 25 queries here are single entity queries, 8 of which are *underspecified* – the query could be written better by adding on additional predicates. This happened even though we reassured users that we *could* answer all movie-related queries.

The results of our interviews are displayed in Table 3.1. Each row in this table is an information need suggested by one or more users. Each column is the query structure the

[2] http://imdb.com

info. need / *keyword query*	[title]	[title] box office	[actor]	[award] [year]	[actor] [actor]	[genre]	[title] ost	[title] cast	[title] [freetext]	movie [freetext]	[title] year	[title] posters	[title] plot	don't know
movie summary	a,c								a	d	b		d	
cast	e							c,d						
filmography			e,a											
coactorship			a, c		b									
posters												b		
related movies	e													
awards				a										
movies of period														b
charts / lists						e			d					
recommendations										b				e
soundtracks							c							
trivia	c													
box office		d												

Table 3.1: Information Needs vs Keyword Queries. Five users (a, b, c, d, e) were each asked for their movie-related information needs, and what queries they would use to search for them.

user thought to use to obtain an answer for this information need. The users themselves stated specific examples; what we are presenting is the conceptual abstraction. For example if the user said they would query for *"star wars cast"*, we abstract it to query type `[title] cast`. Notice that the unmatched portion of the query(*cast*) is still relevant to the schema structure and is hence considered an attribute. Conversely, users often issue queries with words that are non-structural details about the result, such as "movie space transponders". We consider these words free-form text in our query analysis.

Entries in this matrix identify data points from individual subjects in our experiment. Note that some users came up with multiple queries to satisfy the same information need, and hence are entered more than once in the corresponding rows.

3.3 A Principled Approach

3.4 Qunit as a conceptual unit

In this section, we introduce the notion of a "qunit", representing an atomic unit of information in response to a user's query. The search problem then becomes one of choosing the most appropriate qunit(s) to return, in ranked order – a problem that is much cleaner to address than the traditional approach of first finding a match in the database and then determining how much to return. The fundamentals of this concept are described in Section 3.4. Section 3.5 describes search inside a qunits based system, while Section 3.6 discusses methods presents different techniques for identifying qunits in a database.

We now define the notions of a *qunit* and *qunit utility*:

Qunit: A *qunit* is the basic, independent semantic unit of information in a database.

Every user has a "mental map" of how a database is organized. Qunits are an embodiment of this perceived organization. The organization may not be the most efficient representation, in terms of storage, or other computational considerations. For example, while the user may like to think of the IMDb as a collection of actor profiles and movie listings, the underlying relational database would store normalized "fact" tables and "entity" tables, for superior performance. In other words, a qunit can be considered as a view on a database, which we call the *base expression*, with an additional *conversion expression* that determines the presentation of the qunit.

Qunits do not need to be constrained by the underlying data representation model. This allows qunits to be much richer than views. For example, consider the IMDb database. We would like to create a qunit definition corresponding to the information need "cast". A cast is defined as the people associated with a movie, and rather than have the name of the movie repeated with each tuple, we may prefer to have a nested presentation with the movie title

on top and one tuple for each cast member. The base data in IMDb is relational, and against its schema, we would write the *base expression* in SQL with the *conversion expression* in XSL-like markup as follows:

```
SELECT * FROM person, cast, movie
WHERE cast.movie_id = movie.id AND
      cast.person_id = person.id AND
      movie.title = "$x"
RETURN
  <cast movie="$x">
  <foreach:tuple>
      <person>$person.name</person>
  </foreach:tuple>
  </cast>
```

The combination of these two expressions forms our *qunit definition*. On applying this definition to a database, we derive qunit *instances*, one per movie.

Qunit Utility: The *utility* of a qunit is the importance of a qunit to a user query, in the context of the overall intuitive organization of the database. This applies to both qunit definitions and qunit instances.

The number of possible views in a database can be quite large: consider all combinations of columns from all combinations of joinable tables given. For a schema graph with loops, the number becomes infinite, since tables can be repeated. Hence, the total number of *candidate* qunit definitions is an inordinately large number. The utility score of a qunit is required for selecting a top-ranking set of *useful* qunits from the large pool of candidate qunits. From a

user's perspective, this process captures the most salient and relevant concepts that should be returned for a particular query.

It should be noted that qunit utility is a subjective metric. It is dependent on the intent and nature of the user, and is not necessarily uniform for all users. For our purpose, we approximate this metric with clearly defined, objective surrogates. This is similar in spirit to measuring *document relevance* in information retrieval, where TF/IDF is a well-defined and widely accepted objective metric used to approximate document relevance.

Once qunits have been defined, we will model the database as a flat collection of independent qunits. Qunits may overlap, and in fact one qunit may even completely include another. However, these overlaps will not be given any special attention. Similarly, references (links, joins, etc) are expected to have been resolved to the extent desired at the time that qunits were defined. Thus in our movie database, a qunit such as `movie` can list its actors, but there is no explicit link between a `movie` qunit and an `actor` qunit. In subsequent querying, each qunit is treated as an independent entity.

Note, however, that context information, not part of the qunit presented to the user, may often be useful for purposes of search and ranking. For example, link structures may be used for network analysis. Our model explicitly allows for this.

3.5 Qunit-based Search

Consider the user query, *george clooney surgeon*, as shown in Figure 3.1. Queries are first processed to identify entities using standard query segmentation techniques[144].

In our case one high-ranking segmentation is "[person.name] [role.kind]" and this has a very high overlap with the qunit definition that involves a join between "person" and "role". Now, standard IR techniques can be used to evaluate this query against qunit instances of the identified type; each considered independently even if they contain elements in common. The qunit instance describing the cast of the TV show *ER* is chosen as the appropriate result.

As we can see, the qunits based approach is a far cleaner approach to model database search. In current models of keyword search in databases, several heuristics are applied to leverage the database structure to construct a result on the fly. These heuristics are often based on the assumption that the structure within the database reflects the semantics assumed by the user (though data / link cardinality is not necessarily an evidence of importance), and that all structure is actually relevant towards ranking (though internal *id* fields are never really meant for search).

The benefit of maintaining a clear separation between ranking and database content is that structured information can be considered as one source of information amongst many others. This makes our system easier to extend and enhance with additional IR methods for ranking, such as relevance feedback. Additionally, it allows us to concentrate on making the database more efficient using indices and query optimization, without having to worry about extraneous issues such as search and ranking.

It is important to note that this conceptual demarcation of rankings and results does not imply materialization of all qunits. In fact, there is no requirement that qunits be materialized, and we expect that most qunits will not be materialized in most implementations.

3.6 Qunit Derivation

In the previous section, we discussed how to execute a query in a database that already had qunits identified. One possibility is for the database creator to identify qunits manually at the time of database creation. Since the subject matter expert is likely to have the best knowledge of the data in the database, such expert human qunit identification is likely to be superior to anything that automated techniques can provide. Note that identifying qunits merely involves writing a set of view definitions for commonly expected query result types and the manual effort involved is likely to be only a small part of the total cost of database design. If a forms-based database interface has been designed, the set of possible returned

results constitute a good human-specified set of qunits.

Even though manual expert identification of qunits is the best, it may not always be feasible. Certainly, legacy systems have already been created without qunits being created. As such, automated techniques for finding qunits in a database are important. We will discuss these techniques in Chapter 6.

Chapter 4
Query Space Reduction

4.1 An instant response interface

The problem of searching for information in large databases has always been a daunting task. In current database systems, the user has to overcome a multitude of challenges. To illustrate these problems, we take the example of a user searching for the employee record of an American *"Emily Davies"* in an enterprise database. The first major challenge is that of schema complexity: large organizations may have employee records in varied schema, typical to each department. Second, the user may not be aware of the exact values of the selection predicates, and may provide only a partial or misspelled attribute value (as is the case with our example, where the correct spelling is *"Davis"*). Third, we would like the user to issue queries that are meaningful in terms of result size: a query listing all employees in the organization would not be helpful to the user, and could be expensive for the system to compute. Lastly, we do not expect the user to be proficient in any complex database query language to access the database.

The instant-response interface is similar in interaction to various services such as auto-completion in word processors and mobile phones, and keyword query suggestion in search engines. The user is presented with a text box, and is expected to type in keywords, a simple interaction paradigm that is considered the standard in search systems. As soon as the user

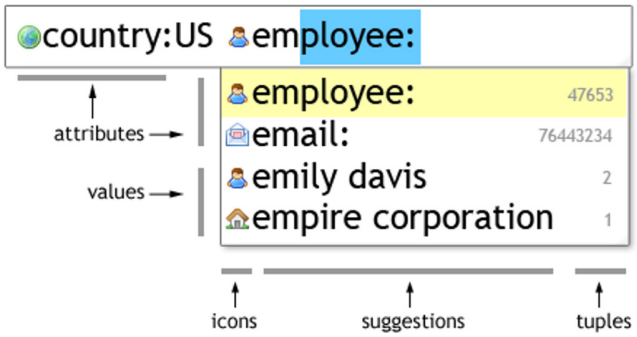


Figure 4.1: An Instant Response User Interface.

starts typing text into the textbox, the system instantly begins to provide suggestions, as shown in Figure 4.1. The first set of suggestions is a set of schema level parameters; i.e. a ranked list of keys, or entity names; which are simplified names for attributes and table names in a relational database, or element tags in an XML database. This allows the user to discover the schema as an ongoing implicit process. The next set of suggestions is a ranked list of text fragments from the data, which provides the user with an insight into valid predicate values. Each text fragment has an associated icon corresponding to the entity type it belongs to, which are expanded to a full *key:value* pair on selection. This allows the user to disambiguate between text values based on their type. Clicking on the icons in the text input box will wrap the text selection around the associated key name and hence trigger the drop down to modify the key, if needed. In addition to these suggestion sets that are integrated into a single list, the estimated number of tuples, or result objects from the resultant constructed query is provided on the right of each suggestion. If the estimated number of results becomes zero due to a value string that never occurs, or a query that violates the schema, further suggestions are stopped and the background of the text input box changes color, depicting an invalid query. The user is then expected to reformulate the query by changing the terms entered.

4.2 User Experience

In the case of our example, the user first types in *"US"*, to specify that he is considering only the United States offices of the organization. Since *"US"* came up as a valid suggestion, the user selected it from the drop down box, which expands the query to country:US. This is beneficial for the database, since it can now do an exact attribute match instead of looking through multiple attributes. Then, the user types in *"em"*; the suggestion mechanism returns a list of suggestions, with two key values, and two data values. The user notices that there are two employee records which have *"Emily Davis"* in them, and selects the respective suggestion. Since the query so far will still return a result set of cardinality two, the user decides to refine the query even further, typing in *"department:"*, which provides the suggestions *"Sales"*, and *"HR"*, one for each result tuple. By selecting *"HR"*, the user is able to locate the single employee record of interest, and directly navigate to it. It should be noted that the complete interaction occurs over just a few seconds, as all suggestions are provided in real time as the user is typing out the query.

In our system, we present to the user an interface visible as a single text box in a web browser. This javascript based interface communicates with the main Java-based backend suggestion server, which can be run either locally or on another system. We offer users a chance to pose queries to the database loaded in the server. The users are guided through the search process by the instantaneous suggestions, and are then presented with results upon query execution. We also demonstrate the "plug and play" nature of our system, where we allow any existing relational or XML database to be loaded quickly for use with the instant response interface, by simply pointing to the database location. The schema and other metadata information are automatically read from the database, and are used to load the database contents, creating the inverted index, phrase tries and schema graph. An optional step is provided to the user for annotating the schema with application-specific metadata (e.g. attribute "EMPZ" could be annotated with "Employees").

We demonstrate our application on two datasets. The first dataset is based on the IMDb

dataset. This comprises of movie information with over 100,000 records, including actors, movie titles and other film related information. For our second dataset, we use the MiMI molecular interaction database, which is an XML dataset over 1 gigabyte in size, containing over 48 million entities including data collected from external databases.

4.3 Challenges

We adopt the standard conjunctive attribute-value query language, where queries are of the form

$$Q = key1 : value1 \cdot key2 : value2 \cdot key3 : value3 \cdot ...$$

Where key$_i$ denotes the key or attribute of a particular object, and value*i* denotes a specific value for that attribute or key. This form of querying can be considered an annotated form of basic keyword querying that is simple enough to be effective and yet rich enough to efficiently query the database. It is used widely in mainstream search engines, and the biomedical community. Keyword identifiers are considered optional, so that a pure keyword query with no attribute name specifications is also acceptable. Any such value is expected to match any attribute of the selected object. Conversely, a single key identifier without a data value is also acceptable, and would be interpreted as the parent type of the expected results. Depending on the schema, the key attributes specified may not belong to the object of interest, but rather to a "closely related" object. In such a case the query is interpreted using the techniques suggested in Schema-free XQuery[88].

In contrast to visual query systems[32, 135], the idea of incrementally and instantaneously formulating queries using database schema and data brings forth its own set of problems. Unlike form-based querying where the users information need is explicit, we need to *infer* what data the user is expecting to find *while* the user is typing in his query. Also, while recent developments in completion based search systems[16] attempt to apply efficient index structures for basic keyword querying; this is only a small part of the challenges faced

in our schema and data based language query system.

Suggestion Quality

There is a subtle element of distraction introduced while proposing suggestions to the user, which may overwhelm the user[13]. Hence, we have to keep in mind that the quality of the suggestions need to be high enough such that it is clear that the suggestion mechanism is worth the negative effects of UI-based distraction. In addition to this, we acknowledge the well known human mind's limit[108] to perceive and consider choices, taken to be approximately seven. Given this fixed limit, we use a ranking strategy based on acceptance likelihood, to prioritize suggestions and display only a top-k set of suggestions.

Performance

The notion of suggesting information to the user at the time of input also depends critically on the "instantaneous" appearance of the suggestions. It has been shown[110] that a maximum delay of 100ms can be considered for user interfaces to seem instantaneous; clearly all our suggestions need to generated and displayed to the user in less time than this. This poses challenges for our system, which is required to be efficient enough to afford extremely fast querying even for large databases with diverse schema. These two challenges must each be addressed in the context of attribute name suggestions, data value suggestions, and size estimation. We describe our solutions to these challenges in the following section.

4.4 System Architecture

In contrast to currently available iterative query building systems, our system employs a continuously updating interface to provide real-time query responses, allowing the user to incorporate suggestions and modify the query in real time, as opposed to going through a time-consuming cycle of iterative query building. Beyond the visible user interface, our

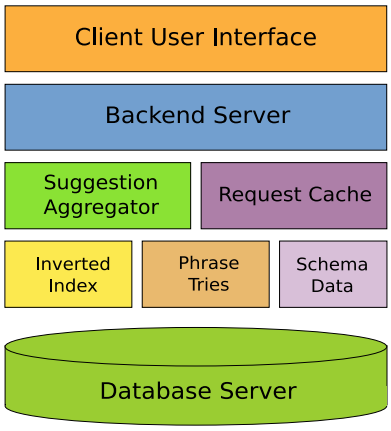


Figure 4.2: System architecture.

system involves a backend server that handles the continuous query requests from the interface, and replies to it. At the core of the system lie the three modules that provide suggestion inputs, as described in the previous section. We use Lucene[59] as our inverted index for size estimation purposes. For the fragment suggestions, we implement in-memory trie-based data structures. The third module provides all schema-related functions, such as query validation and attribute-name suggestion. The information from these modules is then aggregated and combined to be presented as suggestions, which are served off the UI backend server. We implement a caching layer as an integral part of our system, caching not only requests for suggestions, but also the underlying size estimates, etc. Beyond this, the results are then passed on to the suggestion server, which serves the update calls of the user interface.

4.5 Generating Database Suggestions

4.5.1 Schema based suggestions

The first few entries in the suggestion list are a ranked list of suggestions for possible keys. This is done by using a suffix tree to generate all possible attributes that are a completion of the current word typed in. All candidate keys are then ranked according to distances in

the schema graph, between the candidate keys and the currently selected keys. Since the user may not know actual attribute names, or the attribute names may be too cryptic to be used as keywords, we maintain a metadata map that stores all the convenient words used for the field name. We augment this metadata with a subset of the WordNet dataset for further unsupervised synonym-based expansion.

4.5.2 Schema based validation

We use the schema to construct a reachability index for detecting validity of the query. An $n \times n$ matrix is constructed, where entry(i, j) is true if element name i occurs at least as an n^{th} level descendent of element name j in any data object in the database, where n is a upper bound parameter set during database set up. For example, an attribute *"employee"* could contain attributes *"name"* and *"country"* in its data, while *"designation"*(an employee attribute) and *"buildingtype"*(an office buildings attribute) will not have any common 1^{st} or 2^{nd} level ancestors in a personnel database. The query is considered valid with respect to the schema if there exists at least one row in the matrix that contains true entries for each of the attributes in the query. To define descendents in relational data, we develop a hierarchy with attributes at the leaves, tuples as their parents, table as the grandparent, and joins amongst tables as higher levels of ancestry.

4.5.3 Data fragment suggestions

All human-readable data in the database is first fragmented into simple chunks of phrases using statistical methods. We then construct two tries; in the first, each phrase is entered with its attribute as a prefix; in the second the attribute is entered as a suffix. The first trie is used for attribute-specific completion, where the attribute name has been provided by the user. The second trie is used for generic completions, where the attributes are stored as suffixes of the phrase string, allowing us to deduce the attribute name, and hence the icon for

the suggestion. In the case of attributes with a large number of distinct phrases, we prefer to reuse the generic completion trie in place of the attribute specific trie, for efficiency reasons. The leaf node of each trie is also appended with the count information, which is used to rank the suggestions in descending order of popularity.

4.5.4 Size estimation

Each item in the database is provided a unique ID based on its ancestry, in the form $x.y.z$, where x is the parent of y, which in turn is the parent of z. Ancestry is easy to understand for XML data. For relational data, we consider the same hierarchy concepts as Section 4.5.2. We use a fielded inverted index to store these unique identifiers. At query time, we merge the query-generated attribute lists using a simple ancestry check; which allows us to come up with the number of actual items that match the query. Since the query process is incremental, both attribute lists and merged lists are cached. In addition, we do not consider large lists for the sake of efficiency and time-constraints in our query processing. This is done by ignoring merges of any lists that are beyond a certain threshold count, and reporting an estimate of size based on an incomplete independence assumption.

Chapter 5
Generating Phrase Suggestions

5.1 Text inside databases

While databases are typically considered to comprise primarily of numeric data and short strings, searchable databases often contain large amounts of text. For example, the average number of words in a column value in the Internet Movie Database (IMDb) is 2.6. While the database contains large amounts of small strings such as entity names, categorical field labels and textual attribute values, it also contains large chunks of text, sometimes multiple paragraphs long, such as movie reviews, plot details and actor biographies, as is evidenced by the long tail in the frequency distribution of column values grouped by number of words in Figure 5.1.

When generating query suggestions to the user, it is not possible to suggest entire blocks of text. Instead, one has to address the problem of generating suggestions, i.e. text autocompletion at the level of *phrases*.

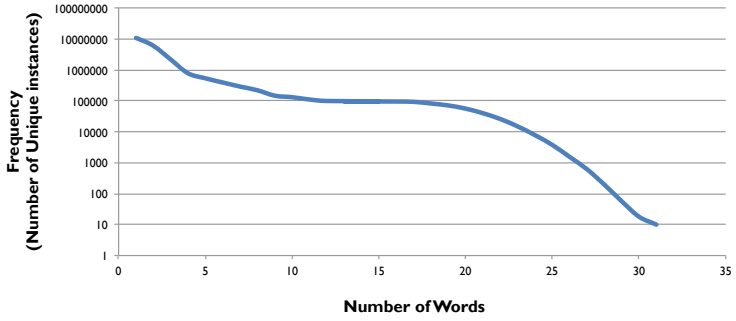


Figure 5.1: Word lengths vs number of unique instances in the IMDb movie database.

Existing work on text autocompletion has so far been restricted to the level of completing letters in single words. This usually entails the building of a trie or a suffix tree over all the words, i.e. the *v*ocabulary of the corpus; followed by a prediction system that simply looks up completions based on the built trie and the prefix.

In this chapter, we study the problem of autocompletion not just at the level of a single "word", but at the level of a multi-word "phrase". There are two main challenges: one is that the number of phrases (both the number possible and the number actually observed in a corpus) is combinatorially larger than the number of words; the second is that a "phrase", unlike a "word", does not have a well-defined boundary, so that the autocompletion system has to decide not just what to predict, but also how far.

We introduce a *FussyTree* structure to address the first challenge and the concept of a *significant* phrase to address the second. We develop a probabilistically driven multiple completion choice model, and exploit features such as frequency distributions to improve the quality of our suffix completions. We experimentally demonstrate the practicability and value of our technique for an email composition application and show that we can save approximately a fifth of the keystrokes typed.

5.2 Text Autocompletion

Text input interfaces have been undergoing a sea change in the last few years. The concept of automatic completion, or *autocompletion* has become increasingly pervasive. An autocompletion mechanism unobtrusively prompts the user with a set of suggestions, each of which is a suffix, or completion, of the user's current input. This allows the user to avoid unnecessary typing, hence saving not just time but also user cognitive burden. Autocompletion is finding applications in specialized input fields such as file location fields [118], email address fields [118] and URL address bars [103], as well as new, more aggressive applications such as dictionary-based word / phrase completion [100] and search query suggestion, available

now in mainstream web browsers [103]. With the recent increase in user interface to server communication paradigms such as AJAX [125], and as users become more receptive to the idea of autocompletion, it will find many more applications, giving rise to increased expectations from autocompletion systems.

Current autocompletion systems are typically restricted to *single-word* completion. While being a very helpful feature, single word completion does not take advantage of collocations in natural language – humans have a tendency to reuse groups of words or "phrases" to express meanings beyond the simple sum of the parts. From our observations, examples of such collocations can not only be proper noun phrases; such as *"Enron Incorporated"*, but also commonly used phrases such as *"can you please send me the"* or *"please let me know if you have any questions"*. Such phrases become much more common when attached to a personalized context such as email, or personal writing, due to the tendency of an individual to mention the same the same phrases during repetitive communication. Many current systems incorporate multiple word occurrences by encoding them as single words; for example *"New York"* is converted to *"New_York"*. However, such schemes do not scale beyond a few words, and would bloat the index (e.g. suffix-tree) sizes for applications where we desire to index a large number of phrases. It is this multi-word autocompletion problem that we seek to address in this chapter.

As noted in the previous chapter, autocompletion is only useful if it appears "instantaneous" in the human timescale, which has been observed [31, 110] to be a time upper bound of approximately 100ms. This poses a stringent time constraint on our problem. For single word completion, typical techniques involve building a dictionary of all words and possibly coding this as a trie (or suffix-tree), with each node representing one character and each root-leaf path depicting a word (or a suffix). Such techniques cannot be used directly in the multi-word case because one cannot construct a finite dictionary of multi-word phrases. Even if we limit the length of phrases we will consider to a few words at most, we still need an "alphabet" comprising all possible words, and the size of the dictionary is several orders

of magnitude larger than for the single word case.

It goes without saying that autocompletion is useful only when suggestions offered are correct (in that they are selected by the user). Offering inappropriate suggestions is worse than offering no suggestions at all, distracting the user, and increasing user anxiety [13]. As we move from word completion to phrase completion, we find that this correctness requirement becomes considerably more challenging. Given the first 5 letters of a long word, for example, it is often possible to predict which word it is: in contrast, given the first 5 words in a sentence, it is usually very hard to predict the rest of the sentence. We therefore define our problem as one of completing not the full sentence, but rather a multi-word *phrase*, going forward as many words as we can with reasonable certainty. Note that our use of the word "phrase" does not refer to a construct from English grammar, but rather an arbitrary sequence of words. At any point in the input, there is no absolute definition of the end of the current phrase (unlike word or sentence, which have well-specified endings) – rather the phrase is as long as we are able to predict, and determining the length of the phrase itself becomes a problem to address. For example, given the prefix *please let*, one possible phrase completion is *please let me know*; a longer one is *please let me know if you have any problems*. The former is more likely to be a correct prediction, but the latter is more valuable if we get it correct (in that more user keystrokes are saved). Choosing between these, and other such options (such as *please let me know when*), is one problem we address in this chapter.

Traditional methods of autocompletion have provided only a single completion per query. Current autocompletion paradigms and user interface mechanisms do allow for multiple suggestions to be given to the user for example by cyclic tab-based completions [143] in command line shells, or by drop-down completions in search engines. However, this facility is typically used to present a comprehensive list of possible completions without any notion of ranking or summarization.

There are dozens of applications that could benefit from multi-word autocompletion, and

the techniques we develop in this chapter should be applicable irrespective of the application context. Nonetheless, to keep matters concrete, we will focus on email composition as our motivating application. We all spend a significant fraction of our work day composing emails. A tool can become valuable even if it helps decrease this time only slightly. We hypothesize that the typical official email conversation is repetitive and has many standard phrases. For each user, there is typically available a long archive of sent mail, which we can use to train an autocompletion system. And finally, the high speed at which most of us type out emails makes it essential that autocompletion queries have very short response times if they are to help rather than hinder the user, leading to tight performance requirements which we will attempt to address.

5.2.1 Our contributions

We introduce a query model that takes into account the multiplicity of completion choice, and propose a way to provide *top-k, ranked suggestions*, paying attention to the nature of results returned.

We introduce the concept of a *significant phrase*, which is used to demarcate frequent phrase boundaries. It is possible for significant phrases to overlap. It is also possible for there to be two (or more) significant phrases of differing lengths for any completion point. To evaluate systems that provide multiple ranked autocompletion results, we define a novel total profit metric for ranked autocompletion results, that takes into account the cost of distraction due to incorrect suggestions.

At the physical level, we propose the construction of a word-based suffix tree structure we call the *FussyTree*, where every node in the tree corresponds to a word instance in the corpus, and root-leaf paths depict phrases. Queries are executed upon receipt of a prefix phrase from the user-interface, and the suffix-tree is then traversed to provide the results. We develop techniques to minimize resource requirements in this context, such as tree size and construction time.

The next section discusses the challenges and motivations for our work, and details some of the existing solutions to the problems at hand. In section 3, we describe our data model and the notion of *significance*. The FussyTree data structure, its construction and querying are defined in section 4. This is followed by section 5, where we discuss evaluation strategies and metrics for our experiments. We report our experiments and findings in section 6, and suggests extensions to our work in section 7. We conclude with a discussion of the problem at hand in section 8.

5.3 Motivation and Challenges

5.3.1 Pervasiveness of Autocompletion

The concept of autocompletion has become prevalent in current user interfaces. It has found its way into mobile phones [138], OS-level file and web browsers [103], desktop search engines such as Google Desktop [99], GNOME Beagle [98] and Apple Spotlight [97], various integrated development environments, email clients such as Microsoft Outlook [100], and even word processors such as Microsoft Word [100] and OpenOffice [20]. The latest versions of Microsoft Internet Explorer [101] and Mozilla Firefox [103] implement autocompletion to suggest search queries to the user. This widespread adoption of autocompletion demands more from the backend mechanisms that support it in terms of fast response times with large amounts of suggestion data. While improvements in underlying computer hardware and software do allow for implementation of more responsive user interfaces, the basic computational challenges involving autocompletion still need to be solved.

5.3.2 Related Work

The concept of autocompletion is not new. Implementations such as the "Reactive Keyboard" developed by Darragh and Witten [39] have been available since the advent of modern

user interfaces, where the interface attempts to predict future keystrokes based on past interactions. Learning-based assistive technologies have been very successfully used to help users with writing disabilities [92], and this technology is now being extended to accelerate text input for the average user.

There have been successful implementations for autocompletion in controlled language environments such as command line shells developed by Motoda and Yoshida [115], and by Davison and Hirsh [40]. Jacobs and Blockeel have addressed the problem of Unix command prediction using variable memory Markov models. Integrated Development environments such as Microsoft Visual Studio also exploit the limited nature of controlled languages to deliver accurate completion mechanisms that speed up user input.

As in these controlled languages, there is high predictability in natural human language text, as studied by Shannon [137], making a great case for the invention of automated natural language input. Phrase prediction and disambiguation for natural language have been active areas of research in the speech recognition and signal processing community [50], providing reliable high-level domain knowledge to increase the accuracy of low-level audio processing. Phrase level prediction and disambiguation have also found great use in the area of machine translation to improve result quality [151]. Similar ideas involving language modeling have been used for spelling correction [83]. The exact task of sentence completion has been tackled using different approaches. Grabski et al. [53] have developed an information retrieval based set of techniques, using cosine similarity as a metric to retrieve sentences similar to the query. Bickel et al. [23] take on a more language model-based approach, by estimating parameters in linearly interpolated n-gram models.

While there have been significant advances in the level of phrase prediction for natural language, there appears to be very little attention paid to the *efficiency* of the prediction itself. Most models and systems are built for offline use, and are not designed for real time use at the speed of human typing - several queries a second. Also, the data models implemented by most are agnostic to actual implementation issues such as system memory. We attempt

to bridge this gap, using a data-structures approach to solving this problem. We conceive the phrase completion problem as a suffix tree implementation, and institute methods and metrics to ensure result quality and fast query response times.

In addition to the nature of the queries, we recognize parameters that have not been considered before in traditional approaches to the problem setting. While the concept of contextual user interaction is quite prevalent [134], the *context* of the query phrase has been ignored in surveyed text-prediction literature. This is an important feature, since most human textual input is highly contextual in nature. In the following sections, we evaluate the ability of improving result quality using the query context.

The problem of frequent phrase detection has also been addressed in the context of "phrase browsing" in which documents are indexed according to a hierarchy of constituent phrases. The Sequitur algorithm used in these techniques [112, 122, 126] suffers from the problem of aggressive phrase creation and is not ideal for cases where we wish to index only the n-most frequent phrases, as opposed to all the repeated phrases in the document. Additionally the in-memory data structure created is hence extremely large and infeasible for large datasets.

Efficient data structures for prefix-text indexing have been studied. For example, trie variants such as burst tries [62, 153] have been shown to be effective for indexing word sequences in large corpora. However, these techniques still require memory resident structures and furthermore do not consider phrase boundaries or phrase frequencies, and hence cannot be used for our application.

5.3.3 Vocabulary and size

There has been considerable work [127, 145, 146] that considers the problem of frequent phrase storage using suffix trees for both small and large data sets. However, traditional techniques that study efficient suffix tree construction make an assumption that the vocabulary (tree alphabet) involved is small. Since we consider natural language phrase autocompletion

as our target application, our vocabulary is that of all the possible words in the text, which is a very large number. Hence, the average fanout of each node in the suffix tree is expected to be quite high, that is, each word will have a high number of distinct words following it. In addition, the trees are expected to be sparse and extremely varied in their structure. This high fanout and sparse, varied structure challenges current scalable construction algorithms since sub-tasks in these algorithms, such as the identification of common substrings.

Such flat, wide suffix-trees also make querying difficult due to the increased costs in searching through the nodes at each level. A possible solution is to maintain each list of children in lexicographic order, which affects the lower bounds on suffix tree construction taking it from $O(n)$ to $O(n \log \sigma)$, where σ is the size of the vocabulary and n the size of our input text. Farach et al. [46] address this very problem and propose a linear-time algorithm for suffix tree construction. However, this algorithm involves multiple scans of the input string to construct and splice together parts of the tree, which makes it impossible to implement applications that are incremental or stream-like in nature, or difficult where the corpus size is very large.

In other related work, the estimation of phrase frequencies has been looked into as an index estimation problem. Krishnan et al. [81] discuss the estimation techniques in the presence of wildcards, while Jagadish et al. [69] use various properties such as the short-memory property of text to estimate frequency for substrings. However, both papers consider only low vocabulary applications. For example, the construction of pruned count suffix trees, as suggested in these papers, is infeasible for phrases due to the large intermediate size of the trees, even with in-construction pruning.

5.4 Data Model

In the context of the problem setting described above, we now formalize our autocompletion problem.

Let a document be represented as a sequence of words, $w_1, w_2, \ldots, w_N$. A phrase r in the document is an occurrence of consecutive words, $w_i, w_{i+1}, \ldots, w_{i+x-1}$, for any starting position i in $[1,N]$. We call x the *length* of phrase r, and write it as $len(r) = x$.

The autocompletion problem is, given $w_1, w_2, \ldots w_{i-1}$, to predict completions of the form: $w_i, w_{i+1}, \ldots, w_{i+c-1}$, where we would like the likelihood of this completion being correct to be as high as possible, and for the length of the completion to be as large as possible.

Of course, the entire document preceding word i can be very long, and most of it may have low relevance to the predicted completion. Therefore, we will consider a *phrase* to comprise a prefix of length p and a completion of length c. Values for both p and c will be determined experimentally.

We now introduce an example to help explain concepts throughout the rest of this chapter, a sample of a multi document text stream for email as shown in Table 5.1. We also present the n-gram frequency table, in this example choosing a training sentence size $N = 2$. In other words, we determine frequencies for words and word pairs but not for word triples or longer sequences. We are not interested in phrases, or phrase components, that occur infrequently. In this example, we have set a threshold parameter $\tau = 2$. The italicized (last three) phrases have frequency below this threshold, and are hence ignored.

5.4.1 Significance

A phrase for us is any sequence of words, not necessarily a grammatical construct. What this means is that there are no explicit phrase boundaries[1]. Determining such boundaries is a first requirement. In practice, what this means is that we have to decide how many words ahead we wish to predict in making a suggestion to the user.

On the one hand we could err in providing suggestions that are too specific; e.g. a certain

[1]Phrases are not expected to go across sentence boundaries, so the end of a sentence is an upper bound on the length of a phrase, but note that we have not yet seen the end of the current sentence at the time we perform phrase prediction, so we do not know what this upper bound is.

Doc 1	please call me asap
Doc 2	please call if you
Doc 3	please call asap
Doc 4	if you call me asap

phrase	**freq**	**phrase**	**freq**
please	3	please call	3
call	4	call me	2
me	2	if you	2
if	2	me asap	2
you	2	*call if*	*1*
asap	3	*call asap*	*1*
		you call	*1*

Table 5.1: An example of a multi-document collection.

prefix of the sentence is a valid completion and has a very high probability of being correct. However the entire suggestion in its completeness has a lower chance of being accepted. Conversely, the suggestions maybe too conservative, losing an opportunity to autocomplete a longer phrase.

We use the following definition to balance these requirements:

Definition A phrase "*AB*" is said to be ***significant*** if it satisfies the following four conditions:

- *frequency* : The phrase *AB* occurs with a threshold frequency of at least τ in the corpus.
- *co-occurrence* : "*AB*" provides *additional information* over "*A*", i.e. its observed joint probability is higher than that of independent occurrence.

$$P(\text{“}AB\text{”}) > P(\text{“}A\text{”}) \cdot P(\text{“}B\text{”})$$

- *comparability* : "*AB*" has likelihood of occurrence that is "comparable" to "*A*". Let $z \geq 1$ be a comparability factor. We write this formally as:

$$P(\text{“}AB\text{”}) \geq \frac{1}{z} P(\text{“}A\text{”})$$

- *uniqueness* : For every choice of "*C*", "*AB*" is much more likely than "*ABC*". Let $y \geq 1$ be a *uniqueness* factor such that for all C,

$$P(\text{“}AB\text{”}) \geq yP(\text{“}ABC\text{”})$$

z and y are considered tuning parameters. Probabilities within a factor of z are considered comparable, and probabilities more than a factor of y apart are considered to be very different.

In our example, we set the frequency threshold $\tau = 2$, the comparability factor $z = 2$, and the uniqueness factor $y = 3$. Consider Doc. 1 in the document table. Notice that the phrase *"please call"* meets all three conditions of co-occurrence, comparability, and uniqueness and is a *significant* phrase. On the other hand, *"please call me"* fails to meet the uniqueness requirement, since *"please call me asap"* has the same frequency. In essence, we are trying to locate phrases which represent sequences of words that occur more frequently together than by chance, and cannot be extended to longer sequences without lowering the probability substantially. It is possible for multiple significant phrases to share the same prefix. E.g. both *"call me"* and *"call me asap"* are significant in the example above. This notion of significant phrases is central to providing effective suggestions for autocompletion.

5.5 The FussyTree

Suffix trees are widely used, and are ideal data structures to determine completions of given strings of characters. Since our concern is to find multi-word phrases, we propose to define a suffix tree data structure over an alphabet of words. Thus, each node in the tree represents a word. Such a *phrase completion suffix tree* is complementary with respect to a standard character-based suffix tree, which could still be used for intra-word completions using standard techniques. We refer to our data structure as a *FussyTree*, in that it is fussy about the strings added to it.

In this section we introduce the FussyTree as a variant of the pruned count suffix tree, particularly suited for phrase completion.

5.5.1 Data Structure

Since suffix trees can grow very large, a *pruned count suffix tree* [81] (PCST) is often suggested for applications such as ours. In such a tree, a count is maintained with each node, representing the number of times the phrase corresponding to the node occurs in the corpus. Only nodes with sufficiently high counts are retained, to obtain a significant savings in tree size by removing low count nodes that are not likely to matter for the results produced. We use a PCST data structure, where every node is the dictionary hash of the word and each path from the root to a leaf represents a frequent phrase. The depth of the tree is bounded by the size of the largest frequent phrase h, according to the h^{th} order Markov assumption that constrains w_t to be dependent on at most words $w_{t+1} \cdots w_{t+h}$. Since there are no given demarcations to signify the beginning and end of frequent phrases, we are required to store every possible frequent substring, which is clearly possible with suffix trees. Also, the structure allows for fast, constant-time retrieval of phrase completions, given fixed bounds on the maximum frequent phrase length and the number of suggestions per query.

Hence, a constructed tree has a single root, with all paths leading to a leaf being considered as frequent phrases. In the *Significance Fussytree* variant, we additionally add a marker to denote that the node is *significant*, to denote its importance in the tree. By setting the pruning count threshold to τ, we can assume that all nodes pruned are not significant in that they do not satisfy the frequency condition for significance. However, not all retained nodes are significant. Since we are only interested in significant phrases, we can prune any leaf nodes of the ordinary PCST that are not significant. (We note that this is infrequent. By definition, a leaf node u has count greater than the pruning threshold and each of its children a count less than threshold. As such, it is likely that the uniqueness condition is satisfied. Also, since the count is high enough for any node not pruned, the comparability test is also likely to be satisfied. The co-occurrence test is satisfied by most pairs of consecutive words in natural language).

5.5.2 Querying: The *Probe* Function

The standard way to query a suffix tree is to begin at the root and traverse down, matching one query character in turn at each step. Let node u be the current node in this traversal at the point that the query is exhausted in that all of its characters have been matched. The sub-tree rooted at node u comprises all possible completions of the given query string.

For the FussyTree, we have to make a few modifications to the above procedure to take into account the lack of well-defined phrase boundaries. There are precisely two points of concern – we have to choose the beginning of the prefix phrase for our query as well as the end point of the completions.

Recall that no query is explicitly issued to our system – the user is typing away, and we have to decide how far back we would like to go to define the prefix of the current phrase, and use as the query to the suffix tree. This is a balancing act. If we choose too long a prefix, we will needlessly constrain our search by including irrelevant words from the "distant" past into the current phrase. If we choose too short a prefix, we may lose crucial information that should impact likely completions. Previous studies [69] have demonstrated a "short memory property" in natural language, suggesting that the probability distribution of a word is conditioned on the preceding 2-3 words, but not much more. In our scenario, we experimentally found 2 to be the optimum choice, as seen in section 5.7.5. So we always use the last two words typed in as the query for the suffix tree.

Once we have traversed to the node corresponding to the prefix, all descendants of this node are possible phrase completions. In fact, there are many additional phrase completions corresponding to nodes that have been pruned away in the threshold-based pruning. We wish to find not all possible completions, but only the significant ones. Since we have already marked nodes significant, this is a simple question of traversing the sub-tree and returning all significant nodes(which represent phrases) found. In general there will be more than one of them.

5.5.3 Tree Construction

5.5.4 Naive algorithm

The construction of suffix trees typically uses a *sliding window* approach. The corpus is considered a stream of word-level tokens in a sliding window of size N, where N is the order of the Markovian assumption. We consider each window instance as a separate string, and attempt to insert *all its prefixes* into the PCST. For every node that already exists in the tree, the node count is incremented by one. The tree is then *pruned* of infrequent items by deleting all nodes (and subsequent descendants) whose `count` is lower than the frequency threshold τ.

```
ADD-PHRASE(P)
1  while P ≠ {}
2  do
3     if IS-FREQUENT(P)
4       then
5            APPEND-TO-TREE(P)
6            return
7     REMOVE-RIGHTMOST-WORD(P)
```

```
IS-FREQUENT(P)
1  for i ← 1 to maxfreq
2  do
3     //slide through with a window of size i
4     for each f in SLIDING-WINDOW(P,i)
5     do
6        if FREQUENT-TABLE-CONTAINS(f) = false
7           then
```

8 **return false**

9 **return true**

APPEND-TO-TREE(P)

1 *//align phrase to relevant node in existing suffix tree*

2 *//add all the remaining words as a new path*

5.5.5 Simple FussyTree Construction algorithm

The naive algorithm for PCST construction described above does not scale well since the entire suffix tree including infrequent phrases is constructed as an intermediate result, effectively storing a multiple of the corpus size in the tree. [81] suggests calling the `prune()` function at frequent intervals during the construction of the suffix tree using a scaled threshold parameter. However, our experiments showed that this strategy either produced highly inaccurate trees in the case of aggressive thresholding, or even the intermediate trees simply grew too large to be handled by the system with less aggressive thresholding.

To remedy this, we propose a basic *Simple FussyTree Construction* algorithm in which we separate the tree construction step into two parts. In the first step we use a sliding window to create an n-gram frequency table for the corpus, where $n = 1, ..., N$, such that N is the training sentence size. We then prune all phrases below the threshold parameter and store the rest. In our second step, we add phrases to the tree in a *fussy* manner, using another sliding window pass to scan through the corpus. This step stores all the prefixes of the window, given that the phrase exists in the frequency table generated in the first step. The testing of the phrase for frequency is done in an incremental manner, using the property that for any bigram to be frequent, both its constituent unigrams need to be frequent, and so on. Since the constituent n-grams are queried more than once for a given region due to the sliding window property of our insertion, we implement an LRU cache to optimize the

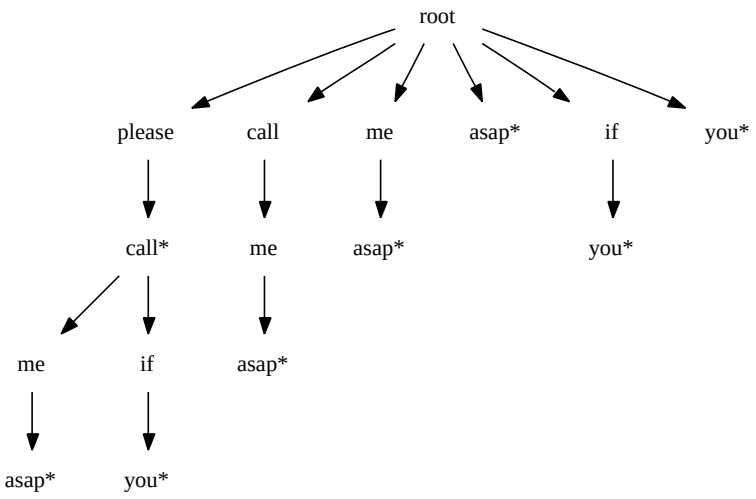


Figure 5.2: An example of a constructed suffix-tree.

`isFrequent()` function.

An example illustrating the FussyTree algorithm

We use the data from the example provided in Table 5.1 to construct our aggregate tokenized stream:

(please, call, me, asap, -:END:-, please, call, if, you, -:END:-, ⋯)

where the -:END:- marker implies a frequency of zero for any phrase containing it. We now begin the construction of our tree using a sliding window of 4. The first phrase to be added is *(please, call, me, asap)* followed by its prefixes, *(please, call, me)*, *(please, call)* and *(please)*. We then shift the window to *(call, me, asap, -:END:-)*, and its prefixes *(call, me, asap)*, *(call, me)* and *(call)*, and so on. All phrases apart from those that contain either of *(call, if), (call, asap), (you, call)* or *(-:END:-)* will meet the frequency requirements, resulting in Figure 5.2. Significant phrases are marked with a $*$. Note that all leaves are significant (due to the *uniqueness* clause in our definition for significance), but some internal nodes are significant too.

5.5.6 Analysis

We now provide a brief complexity analysis of our tree construction algorithm. It uses a sliding window approach over the corpus, adding the windowed phrases to the tree. Given a corpus of size S and a window of fixed size h, we perform frequency checks on each of the phrase incrementally. Hence, the worst case number of actual frequency checks performed is $S \times 2^h$. Note that h is a small number (usually under 8), and hence does not pose any cause for concern. Additionally, the LRU cache for frequency checks brings down this number greatly, guaranteeing that our corpus scanning is done in $O(S)$. Further, each phrase can in the worst case create a completely new branch for itself which can have a maximum depth of h, the size of the window. Thus, the number of tree operations is also linear with the corpus size, indicating that our construction algorithm time will grow linearly with the size of the corpus size. The constant-time lookup for the querying stage is explained thus: a query can at worst force us to walk down the longest branch in the tree, which is of maximum depth h, a fixed number.

We now look into the correctness of our tree construction with respect to our definition of significance. In our implementation, the FussyTree is constructed with only those phrases possessing a frequency greater than τ, ensuring that all nodes marked significant in the tree represent frequent phrases. This asserts correctness the *frequency* rule in the notion of significance. The *co-occurrence* and *comparability* are also rendered correctly, since we possess all the count information to mark them. However, to assert *uniqueness* in the leaf nodes of our tree, we note the loss of count information for the leaf nodes' children. Obviously, the frequency for the children of the leaf nodes is lesser than the threshold, and hence is most likely low enough to satisfy the uniqueness clause. However, there may be a case in which the children may have just missed the threshold, making the leaf node ineligible for the *uniqueness* clause. In this case, we conservatively mark all leaf nodes as significant.

5.5.7 Training sentence size determination

The choice of the size of the training sentence size N, is a significant parameter choice. The cost of tree construction goes up with N, as longer frequent phrases are considered during the construction. This predicates the need to store n-gram frequency tables (where $n = 1 \ldots N$) take up considerable resources, both in terms of storage and query times to check for frequency. The cost of phrase look-up in the resultant tree also increases because of this reason. Hence, in the interest of efficiency, we would like to minimize the value of N. Yet, using too small a value of N will induce errors in our frequency estimates, by ignoring crucial dependence information, and hence lead to poor choices of completions. Keeping this in mind, we experimentally derive the value of N in Section 5.7.5.

We note similar ideas in [146], where frequency information is recorded only if there is a significant divergence from the inferred conditional probabilities. However we point out that the frequency counts there are considered on a per-item basis, as opposed to our approach, which is much faster to execute since there are no requirements to do a recurring frequency check during the actual creation of the FussyTree structure.

5.5.8 Telescoping

Telescoping [104] is a very effective space compression method in suffix trees (and tries), and involves collapsing any single-child node (which has no siblings) into its parent node. In our case, since each node possesses a unique count, telescoping would result in a loss of information, and so cannot be performed directly, without loss of information. Given the large amount of storage required to keep all these counts, we seek techniques to reduce this, through mechanisms akin to telescoping.

To achieve this, we supplant the multi-byte `count` element by a single-bit flag to mark if a node is "significant" in each node at depth greater than one. All linear paths between significant nodes are considered safe to telescope. In other words, phrases that are not signif-

icant get ignored, when determining whether to telescope, and we do permit the collapsing together of parent and child nodes with somewhat different counts as long as the parent has no other significant children. To estimate the frequency of each phrase, we do not discard the count information from the significant nodes directly adjacent to the root node. By the *comparability* rule in the definition of significance, this provides an upper bound for the frequency of all nodes, which we use in place of the actual frequency for the `probe` function. We call the resulting telescoped structure a *Significance FussyTree with (offline) significance marking*.

Online significance marking

An obvious method to mark the significance of nodes would be to traverse the constructed suffix tree in a postprocessing step, testing & marking each node. However, this method requires an additional pass over the entire trie. To avoid them, we propose a heuristic to perform an on-the-fly marking of significance, which enables us to perform a significance based tree-compaction without having to traverse the entire tree again.

Given the addition of phrase *ABCDE*, only *E* is considered to be promoted of its flagged status using the current frequency estimates, and *D* is considered for demotion of its status. This ensures at least 2-word pipelining. In the case of the addition of *ABCXY* to a path *ABCDE* where *E* is significant, the branch point *C* is considered for flag promotion, and the immediate descendant significant nodes are considered for demotion. This ensures considering common parts of various phrases to be considered significant.

APPEND-TO-TREE(P)

1 *//align phrase to relevant node in existing suffix tree*
2 *using cursor* **c**, *which will be last point of alignment*
3 CONSIDER-PROMOTION(c)
4 *//add all the remaining words as a new path*

```
5    the cursor c is again the last point of alignment
6  CONSIDER-PROMOTION(c)
7  CONSIDER-DEMOTION(c → parent)
8  for each child in c → children
9      do
10         CONSIDER-DEMOTION(child)
```

We call the resulting structure a *Significance FussyTree with online significance marking*.

5.6 Evaluation Metrics

Current phrase / sentence completion algorithms discussed in related work only consider single completions, hence the algorithms are evaluated using the following metrics:

$$Precision = \frac{\text{n(accepted completions)}}{\text{n(predicted completions)}}$$

$$Recall = \frac{\text{n(accepted completions)}}{\text{n(queries, i.e. initial word sequences)}}$$

where n(accepted completions) = number of accepted completions, and so on. In the light of multiple suggestions per query, the idea of an *accepted completion* is not boolean any more, and hence needs to be quantified. Since our results are a ranked list, we use a scoring metric based on the inverse rank of our results, similar to the idea of Mean and Total Reciprocal Rank scores described in [128], which are used widely in evaluation for information retrieval systems with ranked results. Also, if we envisage the interface as a drop-down list of suggestions, the *value* of each suggestion is inversely proportional to it's rank; since it requires 1 keypress to select the top rank option, 2 keypresses to select the second ranked one, and so on. Hence our precision and recall measures are redefined as:

$$Precision = \frac{\sum(1/\text{rank of accepted completion})}{\text{n(predicted completions)}}$$

$$Recall = \frac{\sum(1/\text{rank of accepted completion})}{\text{n(queries, i.e. initial word sequences)}}$$

To this end we propose an additional metric based on a "profit" model to quantify the number of keystrokes saved by the autocompletion suggestion. We define the income as a function of the length of the correct suggestion for a query, and the cost as a function of a *distraction cost d* and the rank of the suggestion. Hence, we define the **Total Profit Metric**(**TPM**) as:

$$TPM(d) = \frac{\sum(\text{sug. length} \times \text{isCorrect}) - (\text{d} + \text{rank})}{\text{length of document}}$$

Where *isCorrect* is a boolean value in our sliding window test, and d is the value of the distraction parameter. TPM metric measures the *effectiveness* of our suggestion mechanism while the precision and recall metrics refer to the quality of the suggestions themselves. Note that the distraction caused by the suggestion is expected to be completely unobtrusive to user input : the value given to the distraction parameter is subjective, and depends solely on how much a user is affected by having an autocompletion suggestion pop up while she is typing. The metric TPM(0) corresponds to a user who does not mind the distraction at all, and computes exactly the fraction of keystrokes saved as a result of the autocompletion. The ratio of TPM(0) to the total typed document length tells us what fraction of the human typing effort was eliminated as a result of autocompletion. TPM(1) is an extreme case where we consider every suggestion(right or wrong) to be a blocking factor that costs us one keystroke. We conjecture that the average, real-world user distraction value would be closer to 0 than 1.

We hence consider three main algorithms for comparison: (1) We take the naive pruned count suffix tree algorithm*(Basic PCST)* as our baseline algorithm. (2) We consider the basic version of the FussyTree construction algorithm called the *Simple FussyTree Construction*

as an initial comparison, where we use frequency counts to construct our suffix tree. (3) Our third variant, *Significance FussyTree Construction* is used to create a significance-based FussyTree with significance marked using the online heuristic.

5.7 Experiments

We use three different datasets for the evaluation with increasing degrees of size and language heterogeneity. The first is a subset of emails from the Enron corpus [80] related to emails sent by a single person which spans 366 emails or 250K characters. We use an email collection of multiple Enron employees, with 20,842 emails / 16M characters, as our second collection. We use a more heterogeneous random subset of Wikipedia[2] for our third dataset comprising 40,000 documents / 53M characters. All documents were preprocessed and transformed from their native formats into lowercase plaintext ASCII. Special invalid tokens (invalid Unicode transformations, base64 fragments from email) were removed, as was all punctuation, so that we can concentrate on simple words for our analysis. All experiments were implemented in the Java programming language and run on a 3GHz x86 based computer with 2 gigabytes of memory and 7200RPM, 8MB cache hard disk drive, running the Ubuntu Linux operating system.

To evaluate the phrase completion algorithms, we employ a sliding window based test-train strategy using a partitioned dataset. We consider multi-document corpora, aggregated and processed into a tokenized word stream. The sliding window approach works in the following manner: We assume a fixed window of size n, larger than or equal to the size of the training sentence size. We then scan through the corpus, using the first α words as the *query*, and the first β words preceding the window as our *context*. We then retrieve a ranked list of suggestions using the suggestion algorithm we are testing, and compare the predicted phrases against the remaining $n - \alpha$ words in the window, which is considered the "true"

[2]English Wikipedia: `http://en.wikipedia.org`

Algorithm	Small	Large Enron	Wikipedia
Basic PCST	13181	1592153	–
Simple FussyTree	16287	1024560	2073373
Offline Significance	19295	1143115	2342171
Online Significance	17210	1038358	2118618

Table 5.2: Construction Times (ms).

completion. A suggested completion is accepted if it is a prefix of the "true" completion.

5.7.1 Tree Construction

We consider the three algorithms described at the end of Section 5.6 for analysis. We ran the three algorithms over all three candidate corpora, testing for construction time, and tree size. We used the standard PCST construction algorithm [69] as a base line. We found the values of comparability parameter $z = 2$ and uniqueness parameter $y = 2$ in computing significance to be optimum in all cases we tested. Therefore, we use these values in all experiments reported. Additionally, the default threshold values for the three corpora were kept at 1.5×10^{-5} of corpus size; while the value for the training sentence size N was kept at 8, and the trees were queried with a prefix size of 2. The threshold, training sentence size and prefix size parameters were experimentally derived, as shown in the following sections.

We present the construction times for all three datasets in Table 5.2. The PCST data structure does not perform well for large sized data – the construction process failed to terminate after an overnight run for the Wikipedia dataset. The FussyTree algorithms scale much better with respect to data size, taking just over half an hour to construct the tree for the Wikipedia dataset. As we can see from Table 5.2, the Significance FussyTree algorithm is faster than offline significance marking on the Simple FussyTree, and is only slightly slower to construct than the Simple FussyTree algorithm.

5.7.2 Prediction Quality

We now evaluate the the prediction quality of our algorithms in the following manner. We simulate the typing of a user by using a sliding window of size 10 over the test corpus, using the first three words of the window as the context, the next two words as the prefix argument, and the remaining five as the *true completion*. We then call the probe function for each prefix, evaluating the suggestions against the true completion. If a suggestion is accepted, the sliding window is transposed accordingly to the last completed word, akin to standard typing behavior.

In Table 5.4 we present the TPM(0) and TPM(1) scores we obtained for three datasets along with the recall and precision of the suggestions. We compare our two tree construction algorithms, the Simple FussyTree, and the Significance FussyTree. The Basic PCST algorithm is not compared since it does not scale, and because the resultant tree would any way be similar to the tree in the Simple FussyTree algorithm. We see (from the TPM(0) score) that across all the techniques and all the data sets, we save approximately a fifth of key strokes typed. Given how much time each of us puts into to composing text every day, this is a really huge saving.

To put the numbers in perspective, we consider modern word processors such as Microsoft Word [100], where autocompletion is done on a single-suggestion basis using a dictionary of named entities. We use the main text of the *Lord of The Rings* page on the English Wikipedia as our token corpus, spanning 43888 characters. In a very optimistic and unrealistic scenario, let us assume that *all* named entities on that page (for our problem, all text that is linked to another Wikipedia entry) are part of this dictionary; this could be automated using a named entity tagger, and also assume that we have a **perfect** prediction quality given a unit prefix size. Given this scenario, there are 226 multiple word named entity instances, which, given our optimistic assumptions, would result in **an ideal-case completion profit** of 2631 characters, giving us a $TPM(0)$ of **5.99**%. Our phrase completion techniques can do better than this by a factor of 3.

Algorithm	Small	Large Enron	Wikipedia
Simple FussyTree	8299	12498	25536
Sigf. FussyTree	4159	8035	14503

Table 5.3: Tree Sizes (in nodes) with default threshold values.

Corpus: Enron Small

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple FussyTree	26.67%	80.17%	22.37%	18.09%
Sigf. FussyTree	43.24%	86.74%	21.66%	16.64%

Corpus: Enron Large

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple FussyTree	16.59%	83.10%	13.77%	8.03%
Sigf. FussyTree	26.58%	86.86%	11.75%	5.98%

Corpus: Wikipedia

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple FussyTree	28.71%	91.08%	17.26%	14.78%
Sigf. FussyTree	41.16%	93.19%	8.90%	4.6%

Table 5.4: Quality Metrics.

Despite the discarding of information through significance marking and telescoping, we observe how the *Significance FussyTree* algorithm results in a much smaller tree size (as seen in Table 5.3), and can still perform comparably with the baseline *Simple FussyTree* in terms of result quality. We observe an overall increase in recall and precision on adding significance, and that it causes a slight reduction in TPM metrics especially in high-variance corpora (as opposed to single author text). We consider the TPM drop to be acceptable in light of the reduction in tree size.

Looking across the data sets, we observe a few weak trends worth noticing. If we just look at the simple FussyTree scores for recall and precision, we find that there is a slight improvement as the size of the corpus increases. This is to be expected – the larger the corpus, the more robust our statistics. However, the TPM scores do not follow suit, and in fact show an inverse trend. We note that the difference between TPM and recall/precision is that the latter additionally takes into account the length of suggestions made (and accepted), whereas the former only considers their number and rank. What this suggests is that the average length of accepted suggestion is highest for the more focused corpus, with emails

Algorithm	Small	Large Enron	Wikipedia
Simple FussyTree	0.020	0.02	0.02
Sigf. FussyTree	0.021	0.22	0.20
Sigf. + POS	0.30	0.23	0.20

Table 5.5: Average Query Times (ms).

of a single individual, and that this length is lowest for the heterogeneous encyclopedia. Finally, the recall and precision trends are mixed for the significance algorithms. This is because these algorithms already take significance and length of suggestion into account, merging the two effects described above. As expected, the TPM trends are clear, and all in the same direction, for all the algorithms.

5.7.3 Response Time

We measure the average query response times for various algorithms / datasets. As per Table 5.5, it turns out that we are well within the 100ms time limit for "instantaneous" response, for all algorithms.

5.7.4 Online Significance Marking

We test the accuracy of the online significance marking heuristic by comparing the tree it generates against the offline-constructed gold standard. The quality metrics for all three datasets, as shown in Table 5.6, show that this method is near perfect in terms of accuracy[3], yielding excellent precision[4] and recall[5] scores in a shorter tree construction time.

[3] percentage predictions that are correct

[4] percentage significant marked nodes that are correct

[5] percentage of truly significant nodes correctly predicted

Dataset	Precision	Recall	Accuracy
Enron Small	99.62%	97.86%	98.30%
Enron Large	99.57%	99.78%	99.39%
Wikipedia	100%	100%	100%

Table 5.6: Online Significance Marking heuristic quality, against offline baseline.

5.7.5 Tuning Parameters

Varying training sentence size

As described in Section 5.5.7, the choice of the training sentence size N is crucial to the quality of the predictions. We vary the value of this training sentence size N while constructing the Significance FussyTree for the Small Enron dataset, and report the TPM scores in Figures 5.3 and 5.4. We infer that the ideal value for N for this dataset is 8.

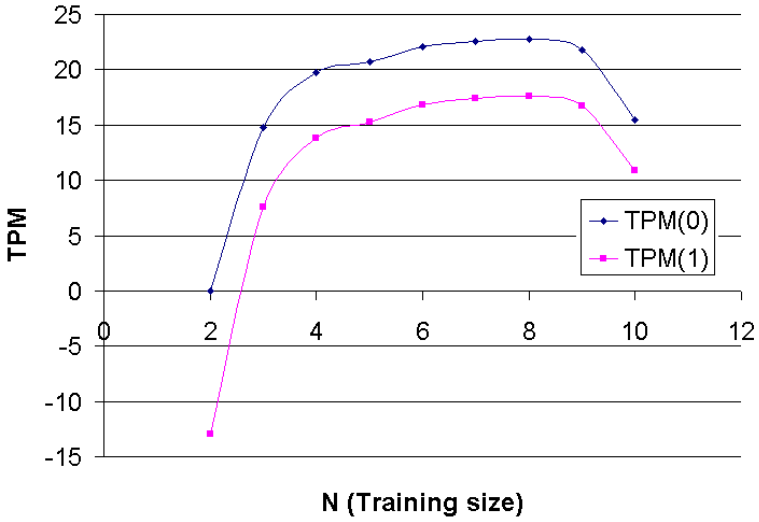


Figure 5.3: Effect of varying training sentence size in Small Enron dataset.

Varying prefix length

The prefix length in the probe function is a tuning parameter. We study the effects of suggestion quality across varying lengths of query prefixes, and present them in Figures 5.3 and 5.4. We see that the quality of results is maximized at length = 2.

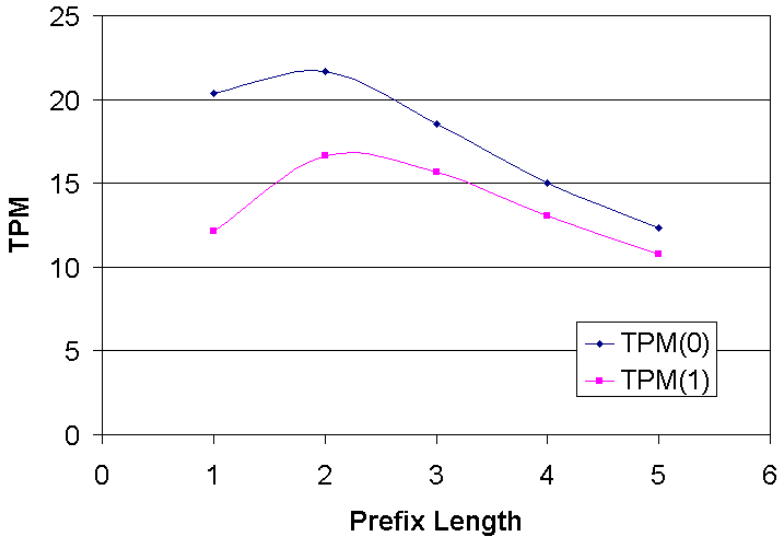


Figure 5.4: Effect of varying prefix length on TPM in Small Enron dataset.

Varying frequency thresholds

To provide an insight into the size of the FussyTree, in nodes, as we vary the frequency threshold during the construction of the FussyTree for the Small Enron and Large Enron datasets; and observe the change in tree size, as shown in Figures 5.5 and 5.6.

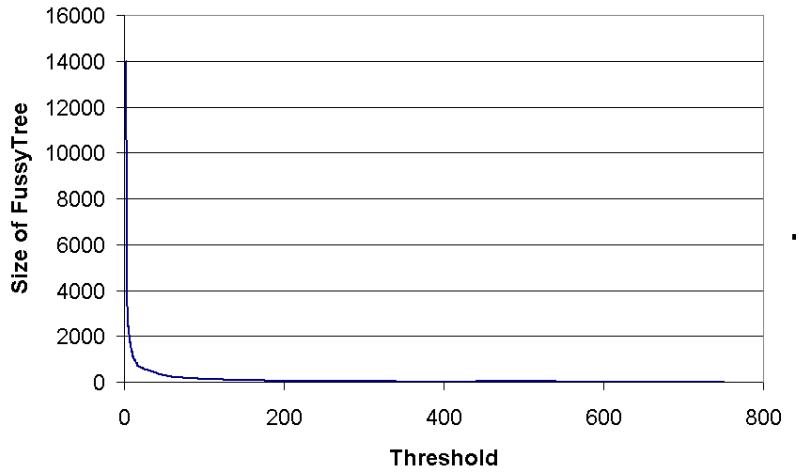


Figure 5.5: Effect of varying threshold on FussyTree size in Small Enron dataset.

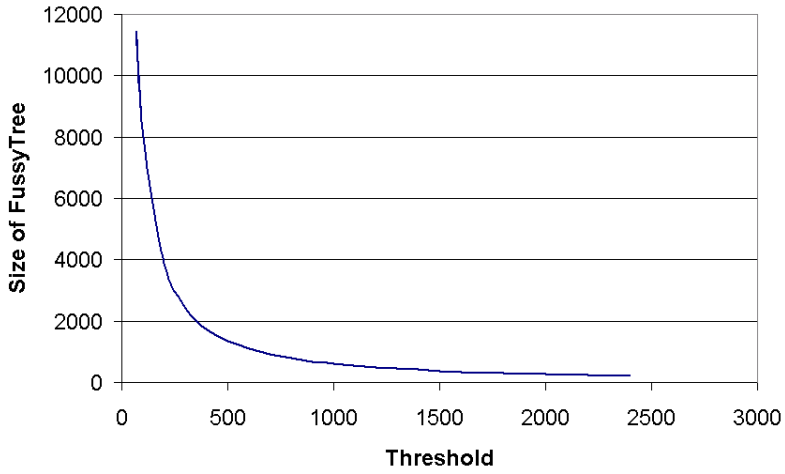


Figure 5.6: Effect of varying threshold on FussyTree size in Large Enron dataset.

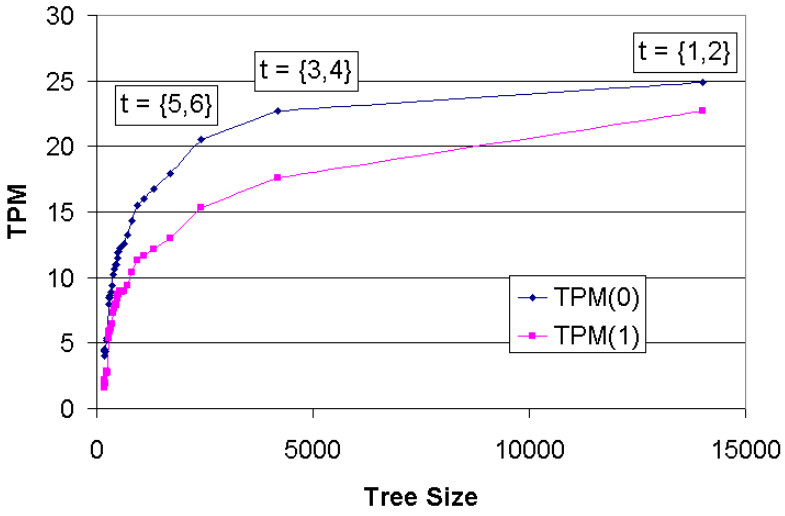


Figure 5.7: Effect of varying FussyTree size on TPM for Small Enron dataset, *t* labels show threshold values for the respective tree sizes.

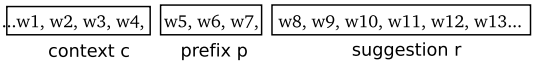
Varying tree size

We now use the frequency threshold as a way to scale up the size of the tree for the Significance FussyTree. We plot the TPM values(0 and 1) against the tree size for the Small Enron dataset. As can be seen, since the decrease in tree size initially causes a very gradual decrease in TPM, but soon begins dropping quite sharply at around threshold = 5 and above. Hence, we prefer to use threshold = 4 in this case.

5.8 Possible Extensions

In experiments over our test corpora in Section 5.7.3, we observe that the `probe` function takes less than a hundredth of a millisecond on average to execute and deliver results. Studies by Card [31] and Miller [110] have shown that the appearance of "instantaneous" results is achieved within a response time span of 100ms. Hence, this validates our premise and shows that there is a clear window of opportunity in terms of query time, to include mechanisms that improve result quality using the additional information. We discuss some methods to improve quality in this section.

We base our extensions on the premise that the *context* of a phrase affects the choice of suitable phrase completions. The *context* of a phrase is defined as the set of words $w_{i-y}, w_{i-y+1}, \ldots, w_{i-1}$ preceding the phrase $w_i, w_{i+1}, \ldots w_{i+x}$.



The *context* of a prefix can be used to determine additional semantic properties, such as semantic relatedness of the context to the suggestion, and the grammatical appropriateness based on a part-of-speech determination. We conjecture that these features abstract the language model in a way so as to efficiently rank suggestions by their validity in light of the context. This context of the query is used to *rerank* the suggestions provided by the FussyTree's `probe` function. The reranking takes place as a postprocessing step after the query evaluation.

5.8.1 Reranking using part-of-speech

We use a hash map based dictionary based on the Brill part-of-speech*(POS)* tagger [25] to map words to their part-of-speech. A part of speech automata is trained on the entire text during the sliding window tree construction, where each node is a part of speech, and each *n*-gram (to a certain length, say 5) is a distinct path, with the edge weights proportional to the frequency. Hence the phrase *"submit the annual reports"* would be considered as an

Corpus: Enron Small

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple + POS	25.0%	77.09%	22.22%	17.93%
Sigf. + POS	40.44%	81.13%	21.25%	16.23%

Corpus: Enron Large

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple + POS	16.13%	85.32%	15.05%	9.01%
Sigf. + POS	23.65%	84.99%	12.88%	6.16%

Corpus: Wikipedia

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple + POS	22.87%	87.66%	17.44%	14.96%
Sigf. + POS	41.16%	95.3%	8.88%	4.6%

Table 5.7: Quality Metrics, querying with reranking.

increment to the *"verb-det-adj-noun"* path weights. The reasoning is that part-of-speech *n*-grams are assumed to be a good approximation for judging a good completion in running text. During the query, the *probe* function is modified to perform a successive `rerank` step, where the suggestions R are ranked in descending order of $prob(POS_c, POS_p, POS_{r_i})$, where POS_c, POS_p and POS_{r_i} are the POS of the context, prefix and suggestion, respectively. We then evaluate the precision, recall, TPM and execution time of the new reranking-based `probe` function, as reported in Table 5.7. We observe that the reranking has little average affect on Recall, Precision and TPM. We note however that on any individual query, the input can be significantly positive or negative. Isolating the cases where the effect is positive, we are working towards a characterization of the effect.

5.8.2 Reranking using semantics

Another source of contextual information is the set of meanings of the prefix's word neighborhood. We reason that some word meanings would have a higher probability to co-occur with certain others. For example, for any sentence mentioning *"hammer"* in its context and *"the"* as prefix, it would more probable to have *"nail properly"* as a possible completion than *"documents attached"*, even if the latter is a much more frequent phrase overall, disregarding context. A simple bipartite classifier is used to consider $prob(A|B)$, where A is the set of

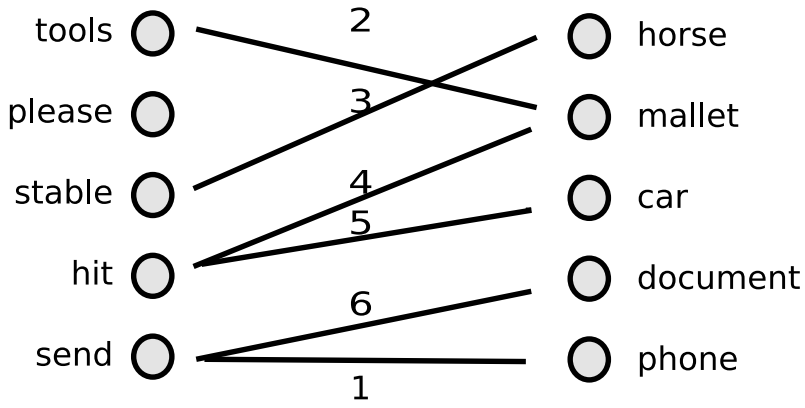


Figure 5.8: Bipartite graph - synset co-occurrences.

WordNet [109] synsets, or word meanings in the result set, and B is the set of synsets in the query, as shown in Figure 5.8. The classifier returns a collection of synsets for every synset set provided in the context and prefix. Suggestions are then ranked based on the number of synsets mapped from each suggestion. However, experiments show that the benefit due to semantic reranking was statistically insignificant, and no greater than due to POS reranking. In addition, it has resource requirements due to the large amount of memory required to store the synset classes[1], and the high amount of computation required for each bipartite classification of synsets. We believe that this is because co-occurrence frequency computed with the prefix already contains most of the semantic information we seek. We thus do not consider semantic reranking as a possible improvement to our system.

5.8.3 One-pass algorithm

The FussyTree algorithm involves a frequency counting step before the actual construction of the data structure. This preprocessing step can be incorporated into the tree construction phase, using on-line frequency estimation techniques, such as those in [45, 94] to determine frequent phrases with good accuracy. A single pass algorithm also allows us to extend our application domain to stream-like text data, where all indexing occurs incrementally. We

[1] WordNet 2.1 has 117597 synsets, and 207016 word-sense pairs

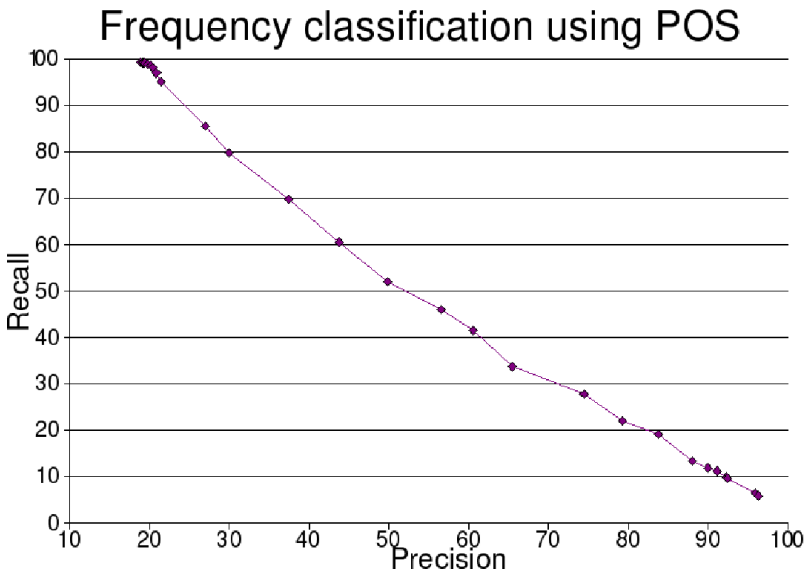


Figure 5.9: Recall vs precision for POS-based frequency classifier.

analyze various features of frequent phrases, such as part-of-speech, capitalization, and semantics. We show how the part-of-speech information of these phrases is a good feature for predicting frequent phrases with near 100% recall, by building a simple naive Bayesian classifier to predict frequency. Since the part-of-speech features of frequent strings are likely to be much less sensitive to changes in corpus than the strings themselves, we reason that it is viable to create a POS-based classifier using an initial bootstrap corpus. The classifier can then be used to build suffix trees, while at the same time train itself. This is a very convenient technique to use for frequent re-indexing of data, for example an overnight re-index of all personal email that improves the autocompletion for the next day. Our experiment proceeds as follows. We train a Naive Bayes classifier to classify a phrase as *frequent* or *infrequent* based on the POS *n*-grams, which are generated using the process described in section 7.1. We then test the classifier on the training data, which in our case is the online trading emails from the Enron dataset. We also report numbers from tests on other datasets; first, a separate sample of emails from the Enron collection, and secondly the sample of text from the Wikipedia.

On the conservative side of the precision-recall curve, experiments on the large sized

Enron collection report precision scores of 30%, 27% and 20%, for 80%, 85% and 99% recall respectively. We focus on this part of the recall-precision curve, since it is acceptable to have a large number of prospective frequent phrases, which are pruned later when updated with real counts.

5.9 Conclusion

In this chapter, we observed that large strings are an important part of real-world databases, and hence phrase-based querying methods need to be explored. Given this requirement, we have introduced an effective technique for multi-word autocompletion. To do so, we have overcome multiple challenges. First, there is no fixed end-point for a multi-word "phrase". We have introduced the notion of significance to address this. Second, there often is more than one reasonable phrase completion at a point. We have established the need for ranking in completion results, and have introduced a framework for suggesting a ranked list of autocompletions, and developed efficient techniques to take the query context into account for this purpose. Third, there is no dictionary of possible multi-word phrases, and the number of possible combinations is extremely large. To this end, we have devised a novel FussyTree data structure, defined as a variant of a count suffix tree, along with a memory-efficient tree construction algorithm for this purpose. Finally, we have introduced a new evaluation metric, TPM, which measures the net benefit provided by an autocompletion system much better than the traditional measures of precision and recall. Our experiments show that the techniques presented in this chapter can decrease the number of keystrokes typed by up to 20% for email composition and for developing an encyclopedia entry.

Today, there are widely used *word completion* algorithms, such as T9 [138]. We have shown that phrase completion can save at least as many keystrokes as word completion. However, word completion and phrase completion are complementary rather than competing. In terms of actual implementation, phrase completion can be triggered at every word

boundary(e.g. when the user types a space), while word completion can be queried all other times.

Chapter 6
Result Space Reduction

6.1 Qunit Derivation Techniques

So far, we discussed how to navigate the query space in a database that already had qunits identified. One possibility is for the database creator to identify qunits manually at the time of database creation. Since the subject matter expert is likely to have the best knowledge of the data in the database, such expert human qunit identification is likely to be superior to anything that automated techniques can provide. Note that identifying qunits merely involves writing a set of view definitions for commonly expected query result types and the manual effort involved is likely to be only a small part of the total cost of database design.

Even though manual expert identification of qunits is the best, it may not always be feasible. Certainly, legacy systems are already created without qunits being created. As such, automated techniques for finding qunits in a database are important. In this section, we discuss the automated derivation of qunits from a database.

Given a database, there are several possible sources of information that can be used to infer qunits. Of course, there is the database schema. Since identifying qunits requires us to write base expressions defining them, and writing these expressions requires schema knowledge, the use of schema knowledge is a starting point. In addition, there are three independent possible sources of information worth considering. The first and most obvious is the data contained in the database itself. A second source of information is the history of keyword queries posed to the system from previous instances of search on the database, also

known as a keyword search query log. The final source of information about the database are sources of evidence outside the database, such as published results and reports that could be based on information from the database in question, or a similar data source. We consider each of these three sources in turn.

6.1.1 Using schema and data

Inspired by previous keyword search efforts [14, 22], we model the database as a graph, with foreign key-primary key relationships as edges connecting tuple nodes. The schema is also considered a graph in a similar manner. We use the link structure of the data to infer important regions in the schema that can be considered qunits. We utilize the concept of *queriability* of a schema to infer the important schema entities and attributes in a database. Queriability [72] is defined as the likelihood of a schema element to be used in a query, and is computed using the cardinality of the data that the schema represents.

Qunit base expressions are generated by looking at the top-k schema entities based on descending queriability score. Each of the top-k_1 schema entities is then expanded to include the top-k_2 neighboring entities as a join, where k_1 and k_2 are tunable parameters.

We run the PageRank algorithm on the data graph with initial values set to the indegree of each node. Upon attaining convergence, the value of each node is then aggregated (summed) by the corresponding schema node; for example if there were three tuples in the table `person` with scores 0.1, 0.3 and 0.4, then they would be summed such that the label `person` = 0.8.

We then consider the top α nodes in the schema graph as the basis of our qunits and their top schema neighbors as value joins. The GROW function assumes the argument T is in descending order of PageRank, enabling it to provide the most likely join path for each node. The data-based qunit generation algorithm when applied to the IMDb schema chose *person* and *movie* as its two highest pageranked nodes. We thus consider these as the starting types for the qunits for this database, as would happen at line 8 of the DATA-QUNITS algorithm.

We would then go over their most important (i.e. high PageRank) neighbors in the schema graph, and consider them for inclusion into our qunit using a join. Take for example the table *person*; it would join with *cast*, and in turn GROW to *movie*, but not uncontrollably to *genre*, as it is a low PageRank element and is not part of X at line 8.

```
DATA-QUNITS(database D, schema S)
 1  D' ← SAMPLE(D)
 2  qunit U[ ] ← {}
 3  histogram P ← PAGERANK(D')
 4  histogram X ← {}
 5  for i ← 1 to |D'|
 6      do T ← LOOKUP-TYPE(D'_i)
 7          X[T] += P[D'_i]
 8  for each type t in FREQ-TOP(X)
 9      do U ← U ∪ GROW(t, NEIGHBORS(t,S), S)
10  return U
```

```
GROW(type t, type[ ] T, schema S)
1  for each type r in T
2      do if S → IS-JOINABLE(t, r)
3             then q ← COMPOSE-JOIN(t, GROW(r, T − t − r, S))
4                  return q
5  return t
```

```
UTILITY(qunit U, clause[ ] T)
1  score s ← 0
2  for each clause c_u in CLAUSES(U)
```

```
3      do for each clause c_t in T
4           do if IS-COMPATIBLE(c_t, c_u)
5                then s ← s + (c_u → PageRank)
6                     T ← T − c_t
7  return s
```

The UTILITY function for the data based derivation determines the relevance of a qunit to a user query. In this case, the utility score for a qunit is the sum of the PageRank scores of its schema members that are compatible with the query: *george clooney movies* maps to *[name] movies*, which maps to `person.name` and `movies`. The LOOKUP-TYPE function looks up the data-to-type index to infer which part of the schema a piece of data belongs to. In the case that a data element belongs to multiple types (e.g. "Marylin" is a person, and the name of a documentary) we disambiguate to the most probable type. The FREQ-TOP function returns the most frequent items in the histogram, while the NEIGHBORS(t,S) function returns the list of schema elements that have a join relationship with t as per schema S.

While this method is intuitive and self-contained within the database, there are many instances where using just the data as an arbiter for qunit derivation is subobtimal. Specifically, in the case of underspecified queries such as "george clooney", with the example schema in Figure 3.2, creating a qunit for "person" would result in the inclusion of important movie "genre" and the unimportant movie "location" tables, since every movie has a genre and location. However, while there are many users who are interested in "george clooney" as an actor in romantic comedies, there is very little interest in the locations he has been filmed at.

6.1.2 Using external evidence

Thus far, we have looked at the schema and data to discover qunits. However there also is a wealth of useful information that exists in the form of *external evidence*. We now propose our third algorithm that uses external evidence to create qunits for the database.

External evidence can be in the form of existing "reports" – published results of queries to the database, or relevant web pages that present parts of the data. Such evidence is common in an enterprise setting where such reports and web pages may be published and exchanged but the queries to the data are not published. By considering each piece of evidence as a qunit instance, the goal is to learn qunit definitions.

In our running example, our movie search engine is aware of the large amount of relevant organized information on the Internet. Movie information from sources such as Wikipedia [1] are well organized and popular. Since the actual information in Wikipedia will have a great overlap with that from IMDb, our goal is thus to learn the organization of this overlapped data from Wikipedia.

Consider a collection of relevant evidence $\boldsymbol{E}$, such that E_i represents a document containing a set of matched entities e in the database. Our intention is to derive a *type signature* of each document that represents the distribution of entities in the document. These signatures are then clustered, and the most frequent signatures are considered for conversion into qunits. Each signature is first broken up into a list of entity types, which is used to infer a list of POSSIBLE-JOIN-PLANS. This is done by first looking in the evidence for an instance of the signature, i.e. a sample of the evidence that yielded this signature. We then consider all possible permutations of joins using the schema graph to come up with a list of candidate queries. Each query is then computed against the database. We select the candidate query whose TYPE-SIGNATURE is most similar to the signature considered, and add it to our qunit list. We utilize the same utility function as that of the data-based derivation algorithm discussed in Sec. 6.1.1.

EXTERNAL-EVIDENCE(database D, evidence [] E)

1 qunit $U[] \leftarrow \{\}$

2 histogram $S \leftarrow \{\}$

[1] `http://wikipedia.org`

```
3  for i ← 1 to |E|
4      do signature s_e ← TYPE-SIGNATURE(D, E_i)
5          S[s_e]++
6  S ← FREQ-TOP(S)
7  for i ← 1 to |S|
8      do int d_min ← ∞
9          query u ← {}
10         for each query q in POSSIBLE-JOIN-PLANS(S_i)
11             do evidence r = EXECUTE-QUERY(q)
12                signature S_r = TYPE-SIGNATURE(D, r)
13                if (SIG-DIFF(S_i, S_r) < d_min)
14                  then d_min ← SIG-DIFF(S_i, S_r)
15                       u ← q
16         U ← U ∪ u
17 return U
```

```
TYPE-SIGNATURE(database D, evidence E_i)
1  Signature S ← {}
2  for each entity e in E_i
3      do Type T ← LOOKUP-TYPE(e, D)
4         S[T]++
5
6  FREQ-NORMALIZE(S)
7  return S
```

In this algorithm, the SIG-DIFF function takes two type signatures and computes the number of types that do not match. The LOOKUP-TYPE is the same function from the

algorithm in Sec. 5.1 and infers the type of an item. The FREQ-NORMALIZE function normalizes the frequency distribution into a "softer" version that is more amenable to clustering. For our experiments, we use the logarithmic function and where the base is derived empirically.

Our algorithm depends on the quality of entity extraction from documents and its correlation with entities in the database. The former has received great attention [38] by the information extraction community, and the latter can be achieved with high levels of accuracy [41]. Another factor to consider is the relevance of the documents to the data. We solve this by first creating a sample of entities from our database, and then searching for them in the given external evidence for documents. The matching documents are considered relevant if they contain another entity from the database. This reduces the number of false positives in the effective training set for our algorithm.

Note that qunit derivation is performed once, at the time of database build. None of the work described above in this section is performed at query time. As such, the time required for qunit derivation is not of primary importance – the cost should be compared to the cost of database and index build. As it turns out, none of the techniques described above are that expensive, and their costs depend on various factors, such as training sample size. For example, for the data sets described in Section 6.3, the query-rollup technique required only a few seconds, the runtime varying linearly with the size of the query log. We do not present a full analysis of this time cost here since it is so small in all cases as not to be worth serious optimization.

6.1.3 Using query logs

While using the structure of the data gives us great advantages in qunit discovery, we ask ourselves if there are other sources of information *outside* the database that can be used to gain insights about the data. Search query logs are a great example of such an information source, as they contain invaluable information about user needs.

We use a *query rollup* strategy for query logs, inspired by the observation that keyword queries are inherently under-specified, and hence the qunit definition for an under-specified query is an aggregation of the qunit definitions of its specializations. For example, if the expected qunit for the query "george clooney" is a personality profile about the actor George Clooney, it can be constructed by considering the popular specialized variations of this query, such as "george clooney actor", "george clooney movies", and so on.

To discover qunits using this strategy, we use a query log from the existing (simple) keyword search engine on the database. Note that it is also possible to use a structured-query log from traditional database engine on similar data for this purpose as well. We begin by sampling the database for entities, and look them up in our structured-query log. For each query, we map the query plans on the schema, and then consider the popular plan fragments for the qunit definitions.

QUERY-ROLLUP()

```
1  qunit U[ ] ← {}
2  histogram Utility[ ] ← {}{0}
3  QTrie ← BUILD-TRIE(QueryLog)
4  for each query L in QueryLog
5      do T_L ← TYPIFY(L_i)
6         S ← LOOKUP-TRIE(QTrie, L_i + " ")
7         histogram X ← {}
8         for j ← 1 to |S|
9             do T_S ← TYPIFY(S_j)
10               if (T_S − T_L) < τ
11                  then X[T_S] += FREQ(S_j)
12        histogram U_L ← {}
13        for each T_S in FREQ-TOP(X)
```

14 **do** $U_L \leftarrow U_L \cup T_S$

15 $U \leftarrow U \cup U_L$

16 $Utility[T_L][U_L] += \text{FREQ}(L_i)$

17 OUTPUT(UTILITY $\leftarrow Utility[\,][\,]$)

18 **return** FREQ-TOP(U_{final})

We walk through a single iteration of our algorithm next. Consider the entity "george clooney" of type `person.name`. We first look up the query log for all queries containing this entity, let us say the top query is "george clooney", a single entity query. We then look for all queries that are supersets of this query, the topmost are "george clooney movies", "george clooney bio" and "george clooney age". Thus, we construct clauses for each of them, and create a qunit that is the union of them. We then increment the 2-d histogram UTILITY to note that the type `person.name` maps to the qunit that is the union of "george clooney movies", "george clooney bio" and "george clooney age". This histogram is later used for the purpose of returning utility values on a per query basis for the UTILITY function.

6.2 Query Evaluation

We now present the algorithm for querying the database using qunits. The general idea is, given a set of qunits, the result of a query can be converted into better results. For each result, try to fit it to the closest qunit. Then, expand/shrink outwards to get the closest qunit, and materialize.

Consider a collection of qunits F, a query Q and the database D. The base expression of each qunit F_i is denoted by F_i^B, and the conversion expression is denoted by F_i^C. The algorithm is as follows:

QUNIT-QUERY(Q, F, D)

1 $F_x \leftarrow$ SELECT-QUNIT(Q, F)

2 $Query \leftarrow$ EXPAND-TO-QUNIT-QUERY(Q, F_x)

3 $Result \leftarrow$ EXECUTE-QUERY$(Query, D)$

4 $Presentation \leftarrow$ CONVERT$(Result, F_x^C)$

SELECT-QUNIT(Q, F)

1 $T_Q \leftarrow$ CANONICAL-TREE(Q)

2 **for** $i \leftarrow 1$ **to** $|F|$

3 **do** $T_F \leftarrow$ CANONICAL-TREE(F_i^B)

4 **if** $T_Q \subset T_F$

5 **then** $Score \leftarrow |$TREE-DIFF$(T_Q, T_F)| \times$ UTILITY(F_i)

6 $Candidates \leftarrow Candidates \cup (F_i, Score)$

7 **return** TOP$(Candidates)$

EXPAND-TO-QUNIT-QUERY(Q, F_x)

1 $T_Q \leftarrow$ CANONICAL-TREE(Q)

2 $T_F \leftarrow$ CANONICAL-TREE(F_x^B)

3 **for each** $clausediff$ **in** TREE-DIFF(T_Q, T_F)

4 **do if** IS-UNDERSPECIFIED$(clausediff)$

5 **then** SET-PARAMS$(T_F, clausediff)$

6 **return** MAKE-QUERY(T_F)

With the use of qunits, we attempt the infer from the database that there is a notion or *qunit*, denoting the "availability of books" in a library, which can be defined as *Find all copy.ids for books where the title contains the search terms, and copy.available equals "yes"*.. Using this qunit, the search term be expanded into this qunit query, providing the

exact copy id for the available book, as desired.

6.3 Experiments and Evaluation

In this section, we begin by first discussing a brief pre-experiment to explore the nature of keyword searches that real users posed against a structured database. The following subsections describe the use of a real-world querylog to evaluate the efficacy of qunit based methods.

6.3.1 Movie Querylog Benchmark

To construct a typical workload, we use a real world dataset of web search engine query logs [124] spanning 650K users and 20M queries. All query strings are first aggregated to combine all identities into a single anonymous crowd, and only queries that resulted in a navigation to the `www.imdb.com` domain are considered, resulting in 98,549 queries, or 46,901 unique queries. We consider this to be our *base query* log for the IMDb dataset. Additionally, we were able to identify movie-related terms in about 93% of the unique queries (calculated by sampling).

We then construct a benchmark query log by first classifying the base query log into various types. Tokens in the query log are first replaced with schema "types" by looking for the largest possible string overlaps with entities in the database. This leaves us with typed templates, such as "[name] movies" for "george clooney movies". We then randomly pick two queries that match each of the top (by frequency) 14 templates, giving us 28 queries that we use as a workload for qualitative assessment.

We observed that our dataset reflects properties consistent with previous reports on query logs. At least 36% of the distinct queries to the search engine were "single entity queries" that were just the name of an actor, or the title of a movie, while 20% were "entity attribute queries", such as "terminator cast". Approximately 2% of the queries contained more than

one entity such as “angelina jolie tombraider”, while less than 2% of the queries contained a complex query structure involving aggregate functions such as “highest box office revenue”.

6.3.2 Evaluating Result Quality

The result quality of a search system is measured by its ability to satisfy a user’s information need. This metric is subjective due to diversity of user intent and cannot be evaluated against a single hand-crafted gold standard. We conducted a result relevance study using a real-world search query log as described in the following subsection, against the Internet Movie Database. We asked 20 users to compare the results returned by each search algorithm, for 25 different search queries, rating each result between 1 (result is correct and relevant) and 0 (result is wrong or irrelevant).

For our experiments, we created a survey using 25 of the 28 queries from the movie querylog benchmark. The workload generated using the query log is first issued on each of the competing algorithms and their results are collected. For our algorithms mentioned in Section 6.1, we implement a prototype in Java, using the imdb.com database (converted using IMDbPy(`http://imdbpy.sf.net`) to 15 tables, 34M tuples) and the base query log as our derivation data. To avoid the influence of a presentation format on our results, all information was converted by hand into a paragraph in a simplified natural English language with short phrases. To remove bias, the phrases were collated from two independent sources. 20 users were then sourced using the Amazon Mechanical Turk (`http://mturk.com`) service, all being moderate to advanced users of search engines, with moderate to high interest in movies. Users were then primed with a sample “information need” and “query” combination : *need to find out more about julio iglesias* being the need, and *“julio iglesias”* being the search query term. Users were then presented with a set of possible answers from a search engine, and were asked to rate the answers presented with one of the options listed in Table 2.

Users were then asked to repeat this task for the 25 search queries mentioned above. The table also shows the score we internally assigned for each option. If the answer is

score	*rating*
0	provides incorrect information
0	provides no information above the query
.5	provides correct, but incomplete information
.5	provides correct, but excessive information
1.0	provides correct information

Table 6.1: Survey Options. Evaluators were only presented with the *rating* column, and asked to pick one of the classes for each query-result pair. Scores were then assigned post-hoc.

incorrect or uninformative it obviously should be scored 0. If it is the correct answer, it obviously should be scored 1. Where an answer is partially correct(incomplete or excessive), we should give it a score between 0 and 1 depending on how correct it is. An average value for this is 0.5.

To provide an objective example of a qunit-based system, we utilize the structure and layout of the `imdb.com` website as an expert-determined qunit set. Each page on the website is considered a unique qunit instance, identified by a unique URL format. A list of the qunits is generated by performing a breadth-first crawl starting at the homepage, of 100,000 pages of the website and clustering the different types of URLs. Qunit definitions were then created by hand based on each type of URL, and queried against the test workload. Users were observed to be in agreement with each other, with a third of the questions having an 80% or higher of majority for the winning answer.

We now compare the performance of currently available approaches against the qunits described in the derivation section, using a prototype based on ideas from Section 3.5. To do this, we first ran all the queries on the BANKS[22] online demonstration. A crawl of the `imdb.com` website was converted to XML to retrieve the LCA(Lowest Common Ancestor) and MLCA [88] (Meaningful Lowest Common Ancestor). The MLCA operator is unique in that it ensures that the LCA derived is unique to the combination of queried nodes that connect to it, improving result relevance. In addition to these algorithms, we also include a data point for the theoretical maximum performance in keyword search, where the user

rates every search result from that algorithm as a perfect match.

Results are presented in Figure 6.1 by considering the average relevance score for each algorithm across the query workload. As we can see, we are still quite far away from reaching the theoretical maximum for result quality. Yet, qunit-based querying clearly outperforms existing methods.

Result Quality by Query Type

In addition to looking at the overall quality of each algorithm against the overall workload, we provide a drill-down comparison against each class of queries. To achieve this, we hand-classified each of the queries in our workload into five types and present the performance of each query type in Figure 6.2. First, we selected the *single entity* queries, where the user simply mentions the name of an actor or movie, such as *"julia roberts"*. Notably, both the MLCA and LCA algorithms fail at this approach since they returned the smallest possible element, which in this case was the name of the entity itself, providing no useful information to the user. In the case of *entity join* queries such as *"steven segal movies"*, where the system is expected to return an implicit join of the name steven segal against the movie table, BANKS and LCA suffer due to the fact that they provide the closest movie appearance as a single result, as opposed to providing a list of movies. MLCA works well in disambiguation queries such as *"clark gable actor"*, where it was correctly able to identify the overall entity element based on the meaningfulness criterion. Note that the entire entity profile (such as the actor page) was more satisfactory to our users than the concise human-created response. For *entity attribute* queries such as *"diane keaton bio"*, LCA suffers poorly since it only returned the title of the page as a single element, because it said "Biography of Diane Keaton". However, when asked for *entity join* results (e.g. *"mel gibson quotes"* – a join of the person *mel gibson* against all his movie's quotes), the current methods beat the qunit-based methods, as they are better at performing joins across multiple tables / entities, which is contrary to the notion of qunits, where we identify individual entities beforehand.

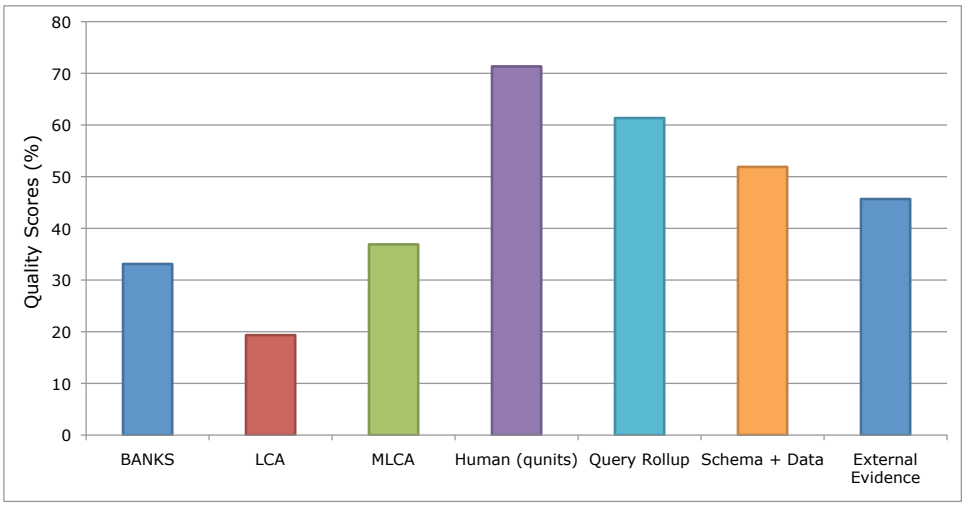


Figure 6.1: Comparing Result Quality against Traditional Methods : "Human" indicates hand-generated qunits.

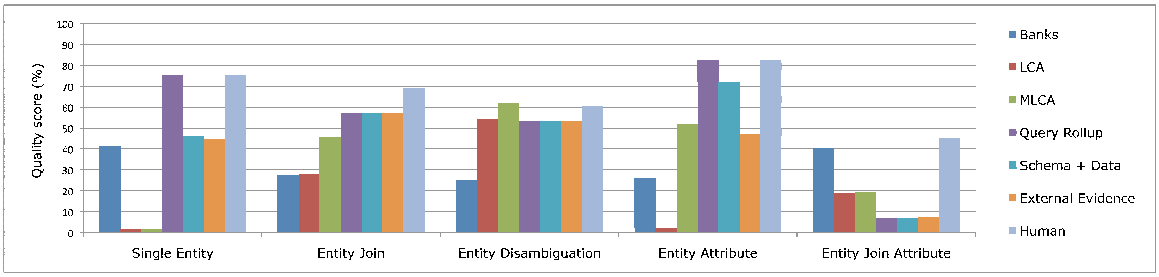


Figure 6.2: Result Quality by Query Type.

6.4 Conclusion

Keyword queries against structured data is a recognized hard problem. Many researchers in both the database and information retrieval communities have attacked this problem, and made considerable progress. Nonetheless, the incompatibility of two paradigms has remained: IR techniques are not designed to deal with structured inter-linked data, and database techniques are not designed to produce results for queries that are under-specified and vague. This chapter has elegantly bridged this gap through the concept of a qunit.

Additionally, this paradigm allows both techniques to co-exist under the same ranking model. Search engines have started to provide interesting ways [52, 154] to simultaneously search over a heterogenous collection of documents and structured data, with interesting

implications for search relevance and ranking. Qunits provide a clean approach to solving this problem.

For the keyword query front end, the structured database is nothing more than a collection of independent qunits; so standard information retrieval techniques can be applied to choose the appropriate qunits, rank them, and return them to the user. For the structured database backend, each qunit is nothing more than a view definition, with specific instance tuples in the view being computed on demand; there is no issue of underspecified or vague queries.

In this chapter, we presented multiple techniques to derive the qunits for a structured database and determine their utility for various query types; we presented an algorithm to evaluate keyword queries against such a database of qunits, based on typifying the query; we experimentally evaluated these algorithms, and showed that users find qunit-based query results substantially superior to results from the best keyword-based database query systems available today.

Chapter 7

Qunit Derivation using Keyword Search Query Logs

7.1 Introduction

In the last chapter, we defined the process of qunit derivation and observed how quints can be generated using various unsupervised methods, such as using the data and schema, by using query logs, or by using external evidence information. The advantage to using these methods is that they do not have the manual overhead of human-generated qunits, and hence allow search engines to derive qunits at scale. When used as part of an overall search pipeline, these unsupervised methods for deriving qunits were found to be superior to traditional methods of keyword search in databases.

In particular, we observed that the query log based method outperformed all other methods in our search satisfaction study. In this chapter, we take a closer look at deriving qunits using keyword search query logs. We provide a formal definition of the qunit derivation problem in the context of keyword search query logs and the relational database model. We demonstrate that the definition lends to that of an optimization problem of selecting the k-best qunit definitions from a potentially infinite space of structured query results, and assert that this problem is related to the Maximum Coverage problem. As a solution, we propose an updated version of the greedy algorithm presented in the previous chapter, *Query Rollup*, that successfully derives qunits by clustering SQL query templates using keyword search query logs as training information. We show that the qunits derived from

our algorithm are close to the optimal solution, and walk through the derivation of qunits for a real-world database and keyword search query log.

7.2 Preliminaries

We now define the various terms used in the following sections:

Keyword Search Query

Our training data is considered to be a set of keyword search query strings. Each query string is a set of words. As mentioned in Section 7.6.2, while we use a click log for our experiments, we only use the queries themselves, all user, click and time information ignored.

An example of a keyword search query is *"george clooney actor"*. While this may have been asked by many different people multiple times in different query sessions, we ignore all this information and only use the string, and its *support*, i.e. the number of people who have queried for it.

Semi-structured Queries

Using the typification process described in Section 7.7, each query in the keyword search query log is converted to a set of *type:value* pairs, where the *type* represents a `table.column` name in the database schema, and the value represents the segment of the search query that corresponds to a valid value in the database, belonging to the respective `table.column`. Semantically, these queries follow the conjunctive model, i.e. a qunit instance that matches the query will need to possess all the *type:value* pairs to be considered a candidate result.

An example of the semi-structured query form of the keyword search query *george clooney actor* is *name.name:george clooney, role_type.role:actor*. In this example, `name.name` and `role_type.role` are table.column labels from the database schema.

If should be noted that for our algorithm, there is exactly one typification, i.e. one semi-structured query for each keyword search query.

Specializations

A semi-structured query p can be a *specialization* of another semi-structured search query q if p is a superset of q, and is not identical to q. Conversely, q is a *generalization* of p. Note that each semi-structured query is a set of *type:value* pairs, and hence in a conjunctive model, a specialized query will be more restrictive than its generalization.

Two examples of specializations for the semi-structured query *name.name:george clooney, role_type.role:actor* are *name.name:george clooney, role_type.role:actor, title.title:ocean's eleven* and *name.name:george clooney, role_type.role:actor, info_type.info:birthdate*. Both specializations represent longer keyword queries, and in the conjunctive model of database queries, they lead to more specific queries over the database.

Query Model

Semi-structured queries are then mapped to the relational query model. We consider a subset of SQL for our query model, specifically looking at **conjunctive, parameterized SELECT-PROJECT-JOIN queries with equality predicates**, i.e. queries are allowed only the `SELECT, FROM` and `WHERE` keywords with all `WHERE` clauses being connected with `AND`s. Each `WHERE` clause uses only the *equals* operator (`''='`') and matches a column either against another column reference (`''t1.c1 = t2.c2''`), an explicit value (`''t1.c1 = "ABC"''`) or a parameter (`''t1.c1 = $''`). A parameterized clause is expanded out to a full clause by using a matching typed segment from the incoming search query.

Projected columns reflect all columns in the participating tables, while `JOIN`s are inferred as natural joins on the shortest interconnect between the schema elements of matched `table.column`s in the schema graph.

For example, given an incoming semi-structured query *name.name:george clooney, role_type.role:actor*, the SQL queries are:

```
SELECT name.name, role_type.role
FROM name ⋈ cast ⋈ role_type
WHERE name.name="george clooney" AND role_type="actor"

SELECT name.name, role_type.role
FROM name ⋈ cast ⋈ role_type
WHERE name.name="george clooney" AND role_type="$"

SELECT name.name, role_type.role
FROM name ⋈ cast ⋈ role_type
WHERE name.name="$" AND role_type="actor"

SELECT name.name, role_type.role
FROM name ⋈ cast ⋈ role_type
WHERE name.name="$" AND role_type="$"
```

Note that the `name` and `role_type` tables join through the `cast` table using natural joins. The four queries represent the different variations on parameterization.

In accordance to the definition of *specialization* for semi-structured queries, a SQL query q_a is a specialization of another query q_b if the semi-structured queries they originated from had a specialization relationship, i.e. a is a specialization of b. Thus at the SQL level, the sets of q_a's selections, projections and joins will each be respective supersets of the sets of q_b's selections, projections and joins.

Result Model

Each Qunit definition is a set of conjunctive, parameterized SELECT-PROJECT-JOIN queries, grouped together. This grouping is unlike the traditional `UNION` operator in that each query query's result may have different columns and hence this is a simple bag-of-queries, and does not represent a single relation. The normalization of heterogenous result relations before presentation to the user is not covered; we refer to this issue as part of the *presentation model* [68]. All members in a grouping have the same parameterization; i.e. if a column c is parameterized in query q_i in qunit definition Q, then all other queries $q_j \in \mathbf{Q}$ also have c as a parameterized `WHERE` clause.

The following is an example of an *actor* qunit that shows the roles the actor has acted in, and his birthdate.

$Q_{actor} :=$

- ```
  SELECT name.name, role_type.role
  FROM name ⋈ cast ⋈ role_type
  WHERE name.name="$" AND role_type="actor"
  ```
- ```
  SELECT name.name, info.info, info_type
  FROM name ⋈ info ⋈ info_type
  WHERE name.name="$" AND info_type="birthdate"
  ```

Satisfiability

As described in the previous section, each qunit definition can be applied to a database, in conjunction with a semistructured query, to produce a qunit instance. A qunit instance Q is known to satisfy a semi-structured query s if:

- The *type:value* pairs are a superset of the parameterized columns in the qunit definition of Q.
- Each column value in the *exact SQL query*'s result (i.e. query intent) that the incoming semi-structured query represents is present in the qunit instance.

The boolean nature of satisfiability does not lend well to qunit derivation algorithms. Hence, we propose a fractional variant of satisfiability as well. Given v_r being the column values present in the resulting qunit instance, and v_i being the column values in the result of running the exact query intent on the database, the satisfiability of v_r for v_i is the Jaccard of the two, namely:

$$S = \frac{|v_r \cap v_i|}{|v_r \cup v_i|}$$

This formulation encourages answers to be as correct as possible, without providing extraneous facts. For example, one possible qunit for the information need of an actor profile expressed with a semi-structured query *"name.name:George Clooney"* is the `name.name` projection of the tuple for *George Clooney*, which will contain just the string, "George Clooney". Thus the resulting qunit instance $v_r = \{$name.name:*George Clooney*$\}$ However, v_i (the information need) contains a lot more information, such as the birth date of the actor, the genres he acts in, etc. Hence, the numerator will be rather small, leading to a low satisfiability score. Conversely, one other possible resulting qunit instance could be the universal relation, i.e. the entire database. While it does correctly contain all the facts (i.e. column values) that the user is looking for, it also contains every other fact in the database, and hence the satisfiability will not be that high, due to a large denominator. We will use this version of satisfiability as our objective metric in the optimization problem in the next section.

7.3 Qunit Derivation Problem

In this section, we define the qunit derivation problem in context to our setting, i.e. the use of keyword search query logs as training data.

7.3.1 Problem Definition

The qunit derivation problem in the context of keyword search query logs is defined as an optimization problem follows.

Definition Given a database D and a workload of semi-structured queries W, generate k groups of SELECT-PROJECT-JOIN SQL queries that *best satisfy* the expected information need for each query in the workload.

This problem statement takes the database and the keyword search query log, typified into semi-structured queries as an input, and produces k qunit definitions as output.

One question to answer is, how does a set of qunits *best satisfy* a workload? For this, we use the Jaccard metric mentioned before to define *"best satisfy"* over a workload of queries, we take the sum of satisfaction for each query in the workload as our objective metric.

A second concern is about the space of optimization. In a database with loops in the schema graph, the space of possible groups of SELECT-PROJECT-JOIN SQL queries is infinite. Even without loops, this space is exponential to the number of columns in each table and the joins permitted, and is quite large even for small databases. Clearly, this is intractable, and there is need to bound this space. To do this, we limit the space of possible SELECT-PROJECT-JOIN SQL queries to only those that can be inferred from the semi-structured queries, the process of of which is detailed in the Algorithm section.

Based on this, our revised definition of the problem is:

Definition Given the space of SELECT-PROJECT-JOIN SQL queries inferred from an input workload of semi-structured queries **w**, form k groups of SQL queries such that $\sum_w S(w_i, q_i)$ is maximized, where S is the satisfiability function over each workload information need, and the answer qunit for that workload query.

7.3.2 Computational Complexity

The revised definition of the problem falls into the space of set cover problems. We liken our problem our optimization problem to that of Maximum Coverage [3], which is known to be NP-Hard. The Maximum Coverage problem is defined as follows:

Definition Given a set of items $E = e_i, e_2, \ldots, e_n$, a parameter k and a collection of sets $S = \{s_1, s_2, \ldots, s_m\}$ where $s_i \subset E$:

Find a subset $S' \subset S$ such that $|S'| \leq k$ and the number of covered elements $|\bigcup_{s_j \in S'} s_i|$ is maximized.

Given this definition, consider the space of all SQL queries generated from the semi-structured query log to be set of items E, and the space of all possible qunits to be S. Thus, given a parameter k, and a threshold τ such that the *satisfiability* of the qunit instance is greater than τ, we can now write qunit satisfiability as set membership, making the problem of qunit derivation equivalent to Maximum Coverage. Since Maximum Coverage is known to be NP-Hard [3] and have an approximation lower bound close to that of a greedy approach, we consider a greedy approach to our qunit derivation algorithm.

7.4 Algorithm

7.4.1 Overview

Based on the greedy strategy, we devise the following algorithm outline:

1. For each keyword search query in the query log, produce its corresponding semi-structured query using *typification*.
2. For each semi-structured query produced, use the schema to map the query to all possible parameterized SELECT-PROJECT-JOIN SQL queries.
3. With the top-k parameterized SELECT-PROJECT-JOIN SQL queries as the starting points for k qunit definitions, grow out the clusters by adding the SQL queries if they are a *specialization* of that query.

Based on this outline, we now detail each step of our algorithm in the following subsections.

7.4.2 Mapping keyword search queries to semi-structured queries

A core part of our algorithm is the mapping of keyword searches to SQL queries, which is performed as follows. Keyword queries are first *typified* to semi-structured queries. Typification is the process of converting a free-form string of words to a set of `attribute:value` pairs, where each `attribute` is a `table.column_name` in the database, and each `value` is a column value in the database.

Performing an accurate typification is in itself a challenging task, and has been discussed in detail in recent works by Sarkas et al. [131] and Fagin et al. [44]. While independent of their work, our implementation follows a similar maximum information approach to that of [131] and provides typifications of a quality good enough for the overall qunit derivation process. Our typification algorithm involves two underlying steps. The first step is the *segmentation* of the query $q = \{w_1, w_2, w_3, \dots\}$ into segments $\mathbf{s} = \{\{w_1, w_2, \dots, w_i\}, \{w_{i+1}, w_{i+2,\dots}\}, \dots\}$ The second step is *tagging*, wherein each segment s_i is assigned a *table:column* label. The steps are described as follows:

Segmentation: We use a segmentation model that prefers the most likely segments:

$$\mathbf{s} = \arg\max_{s}(\sum P(s_i))$$

Where the probability of a segment s_i is its frequency of occurrence in the database, normalize by the number of values in the database.

Tagging: Once a query is segmented, each segment is tagged with a *table:column* schema label. This is done first generating a histogram of the values in the database with the corresponding table and column label used as the class label in the form *table:column*. This histogram is then used to determine the table and column label for each query segment by picking the most popular table and column label in the database for that value. Both query segments and database values are normalized for correct reconciliation.

7.4.3 Mapping semi-structured queries to SQL queries

Each semi-structured query is now a set of *table.column:value* pairs, which we translate into all possible SQL queries that can be interpreted from it. To do this, we first locate all the `table.columns` on the schema graph, and take the shortest interconnect as the join path. This process is identical to Steiner-tree discovery, and heuristics for finding near-optimal approximations are well-documented [105, 129]. Once the join path is decided, we are left with enumerating the columns to place in `SELECT` and the conditions in the `WHERE` clause. For the `SELECT`, we consider all possible combinations of columns from the tables participating in the join. For the `WHERE` clause, we consider all possible combinations from the original semi-structured query, such that each column can either be unbounded (i.e. `column = *`, meaning that we can drop that clause), bounded (i.e. `column = v`, where `v` is the value for the column in the *table.column:value* pair from the semi-structured query), or parameterized (i.e. `column = $`). It should be noted that this process produces an exponential number of SQL queries, specifically $2^{\textit{columns from the JOINing tables}} \times 3^{\textit{attribute-value pairs in the semi-structured query}}$. While being exponential to the schema and the query sizes, it is practical to compute this since the number of attribute-value pairs is usually quite small (averaging less than 3) and the number of participating columns is bounded by the schema size.

7.4.4 Query Rollup

Once the collection of SQL queries is generated, we now perform a greedy agglomeration as follows:

QUERY-ROLLUP(SQL Log L)

1 *Queries* $\leftarrow$ HISTOGRAM(L)

2 *Queries* $\leftarrow$ FILTER-QUERIES(*Queries*, supports)

3 *OrderedQueries* $\leftarrow$ REVERSE-SORT(*Queries*, frequency)

```
4  topKQueries ← TOP-K(OrderedQueries)
5  qunits ← {}
6  Candidates ← OrderedQueries − topKQueries
7  for each query q in QueryLog
8      do for each query p in topKQueries
9          do if (q is a specialization of p AND q not in topKQueries)
10             then qunits[p] += q
11 return FREQ-TOP(qunits)
```

7.5 Evaluating Qunit Derivation

The primary challenge with the evaluation of qunit derivation is the lack of a gold standard. Specifically, the process of selecting the top-*k* qunits that best represent a database depends on the query intents of the user, and their notions of whether the information provided is sufficient. While a good solution for this is the search quality analysis provided in the previous section, it was done in the context of search quality. This adds concepts of ranking and qunit evaluation into the mix.

In order to evaluate qunit derivation in isolation, we need to evaluate the immediate output of a qunit derivation, which is a *set of qunit definitions*, against a gold standard, i.e. the ideal set of qunit definitions for a database. It is however hard to define what an *ideal output* is. We argue that finding a metric for goodness for this output is not only hard, but impossible by definition, since qunits reflect the mental model of the database users, and can not be ascertained.

In light of this, we consider a two-pronged approach. First, we model the problem of qunit derivation as an optimization problem. We define the objective function as the satisfaction of query intent by a set of qunits, given a workload. By comparing the output

of our greedy algorithm against the global optimum, we can discern the **result quality** of the qunits derived from our qunit derivation method. While this evaluation is objective, it does not convey the insights into qunit derivation in real-world scenarios. Thus, we consider a subjective evaluation by considering real-world database and keyword search query log. We walk through the qunit derivation process, sharing anecdotal results about the **impact of query log size** on derived qunits.

Objective evaluation of result quality: The identification of a global optimum set of qunits for a given database is, however, a challenge due to the size of the search space. Given a database schema, the number of possible SQL queries is exponential to the number of tables, and the number of columns for a schema with all pairwise joins possible. For a schema with join loops (i.e. there exists a cycle in a graph representation of the schema), the number of possible queries is infinite, since joins can be repeated infinitely. Further, since qunits are groups of queries, the number of possible qunits is a subset of the power set of the set of SQL queries, and hence introduces at worst another exponential increase in search space. Even further, since our final goal is to pick k qunits from the space of all qunits, the number of possible outputs is $\binom{n}{k}$, where n is the number of possible qunits.

Due to the size of our search space, finding a globally optimal result by exhaustive search is impractical for real-world databases with large number of tables and columns. Thus, we construct a small synthetic schema to perform an exhaustive search on. The details of this schema are provided in the following section, along with other experimental details and results.

7.6 Experiments

In order to get a clearer picture about qunit derivation itself, we perform two separate experiments. The first is an objective experiment that evaluates the efficacy of the greedy heuristic by measuring result quality. In the second experiment, we visualize the derivation

of qunits for various values of the query log.

7.6.1 Result Quality

Synthetic Dataset

As explained in the previous section, we perform objective evaluations of our greedy heuristic using a smaller synthetic schema. The comprises of 6 tables, each with 24 columns. For simplicity, we consider projections to be part of the *presentation layer* [68] and hence all columns are projected from each table in a query. The schema has 2 primary-foreign key relationships, yielding a total of 34 possible SQL queries which lead to 77 valid qunits, yielding 77 qunit sets where $k = 1$ (i.e. possible selections of k qunits from all valid qunits), 2,926 qunit sets where $k = 2$, 73,150 qunit sets where $k = 3$ and 13,53,275 qunit sets where $k = 4$.

We use 5 different workloads A, B, C, D and E, randomly sampled from the space of possible SQL queries. Each workload contains 7 SQL queries. For the purposes of our query rollup algorithm, we pick k queries randomly from each workload during the derivation process as the top-k queries that form the seeds of our derived qunits.

Metrics

When comparing a set of k qunits against another, we use a workload of queries to evaluate them. To avoid errors from the typification process, we use a test workload of SQL queries. For each query in the test workload, we pick the *best* qunit in our set of k qunits. Based each query in the workload, we can calculate the average recall and precision of the algorithm. We define qunit *recall* and *precision* in the following manner:

Qunit Precision: Qunit precision is defined the fraction of *relevant* query results returned for a given query, averaged over a workload. We specify *Qunit Precision* of an answer qunit

as:

$$QunitPrecision = \frac{\sum_{w \in \text{workload}} \frac{n(\text{relevant queries in answer qunit})}{n(\text{total queries in answer qunit})}}{n}$$

It should be noted that for our experiments, *answer qunit* refers to the most relevant qunit for the incoming workload query. Additionally, each query in our test workload seeks exactly one relevant query, hence $n(\text{relevant queries in answer qunit})$ will always be 1.

Recall of a qunit workload: Qunit recall is defined as the fraction of queries that contained the original query intent in the answer qunit:

$$QunitRecall = \frac{\left|\text{workload} \cap \bigcup_{qunits}(\text{derived qunits})\right|}{|\text{workload}|}$$

Results

We evaluate the quality of our qunit derivation process in the following manner. For a workload and the parameter value for k, we identify k qunits as our output. For each query in the workload, the best of k qunits is chosen as a result qunit for that query. To determine a gold standard, we pick a set of k qunits that contain *all* queries in the workload while minimizing the number of non-answer queries in the result qunit for each workload query. The gold standard is picked by an exhaustive enumeration of all possible k combinations of qunits by minimizing for the objective function:

$$f(qs) = \sum_{w \in \text{workload}} \left| \begin{array}{c} \arg\min \\ q \in \text{ qunits} \end{array} f(q) := \{|q| \ \ \textbf{if } w \in q \ \ \textbf{else } \infty\} \right|$$

Note that the objective function is related but distinct from the *satisfaction* definition in Section 7.3.1, due to the exclusion of projected columns in the synthetic result quality experiment. For workload A, we pick a gold standard qunit set where $k = 3$. For the other workloads, we set $k = 3$.

Intuitively, the qunit recall for the gold standard is always 100%, while the mean preci-

sion is maximized. Figures 7.1 and 7.2 present the Qunit Recall and Qunit Precision numbers for each of the 5 workloads, comparing the gold standard qunit sets and the corresponding qunit sets from our query rollup algorithm.

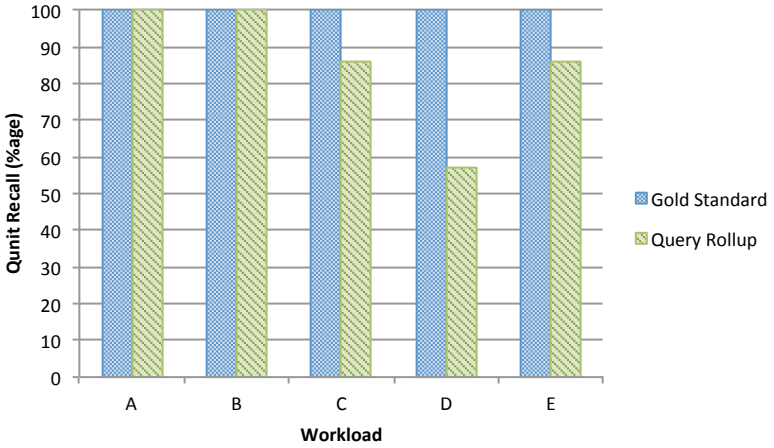


Figure 7.1: Qunit Recall across 5 workloads. Qunit Recall is equal for workloads A and B since the derived qunit sets are identical. For workloads C, D and E, Qunit Recall for the Query Rollup algorithm is lower due to missing queries.

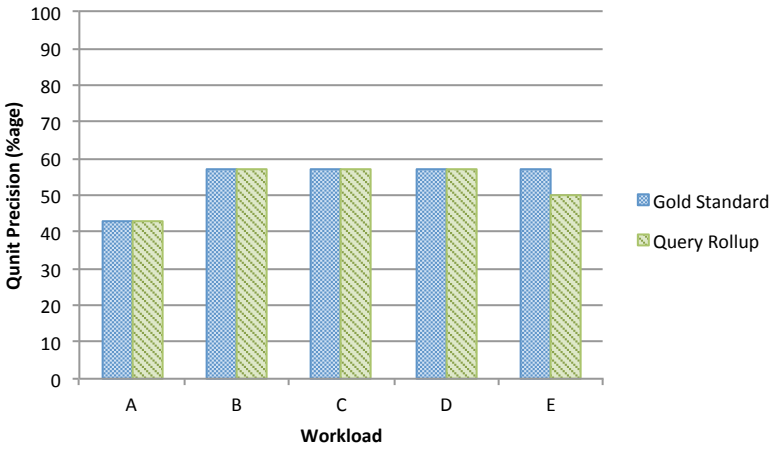


Figure 7.2: Qunit Precision across 5 workloads. Qunit Precision is equal for workloads A and B since the derived qunit sets are identical. For workloads C and D, Qunit Precision is equal to optimal precision, but at the expense of Qunit Recall (i.e. some queries are not covered by the derived qunits). For workload E, Qunit Precision is lower due to both missing answers, and agglomeration of specializations into multiple qunits.

The experimental results surface some interesting insights. First, it can be seen that our greedy *Query Rollup* algorithm is quite competitive with the gold standard results. Second, *Query Rollup* tends to tradeoff Qunit Recall in favor of Qunit Precision. This is due to the

greedy nature of our algorithm : In workloads C and D, certain longer queries surfaced as the top-k and these queries did not have any specializations in the workload. This led to creation of single-query qunits that failed to cover the entire workload, creating a loss in Qunit Recall. However, since the qunits comprised exactly one query, this increased Qunit Precision, making the Qunit Precision comparable to the gold standard. The third insight, present in workload E, is that longer queries show up as specializations of multiple queries in the top-k. This leads to their inclusion in multiple qunits (each of the top-k qunit leads to a different qunit), making answers more verbose, thereby decreasing the Qunit Precision, as can be seen in Figure 7.2.

7.6.2 Impact of Query Log Size

In this experiment, we varied the size of the input query log provided to the Query-Rollup algorithm over a real-world *movie database search* use case. We begin by defining the datasets involved.

Datasets

Keyword Query Log For our real-world experiments, we reuse the benchmark click log from the prior chapter: the AOL Search click log dataset [124] spanning 650K users and 20M queries over 3 months. Again, only queries that resulted in a clicks to the `www.imdb.com` domain are considered, resulting in 46K distinct queries. Specializations of these queries were also considered from the entire click log (i.e. queries leading to all domains, not just IMDb), leading to a total of 250K queries. Queries were then *typified* into semi-structured queries using the *typification* process previously.

IMDb Dataset Similarly, we use an updated IMDb dataset from the previous section, with 35M tuples spanning 18 tables, 77 columns and 18 primary-foreign key relationships. A visualization of the schema graph is shown in Figures 7.3 and 7.4.

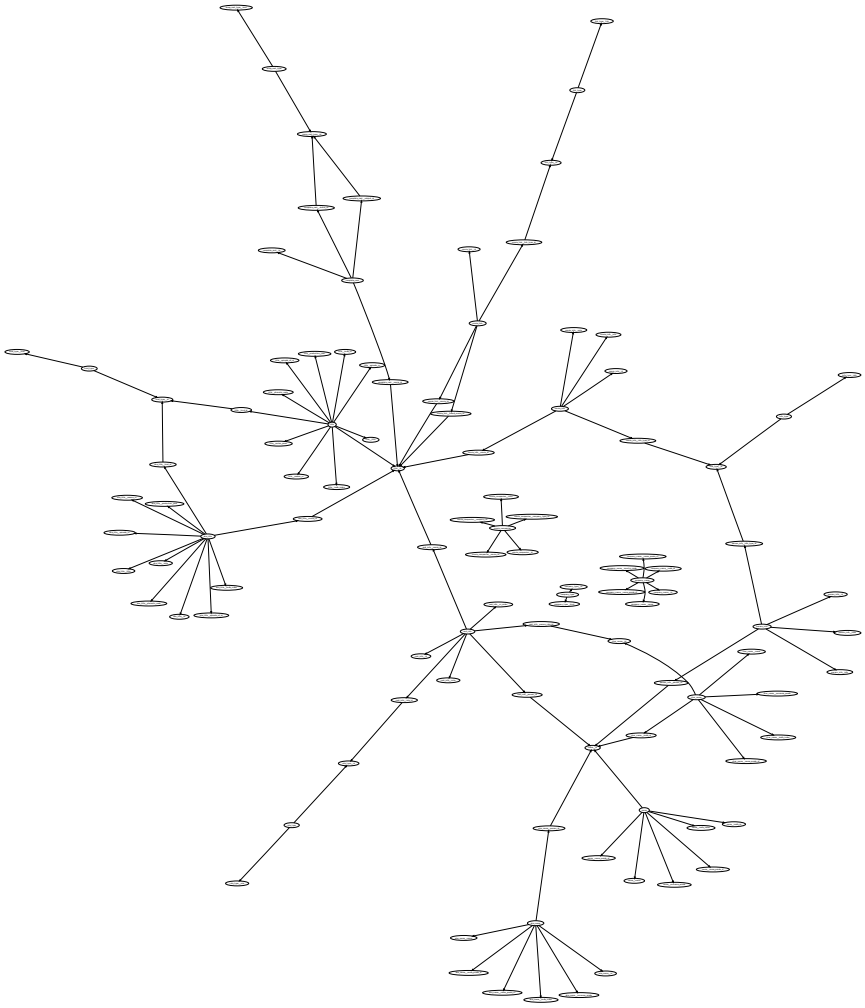


Figure 7.3: A visualization of the entire schema graph of the IMDb database. The schema comprises 18 tables, 77 columns and 18 primary-foreign key relationships. Tables and columns are represented as nodes. Table-column relationships and primary-foreign key relationships are represented as edges.

All data (i.e. queries in the query log, and column values in the database) were normalized by removing punctuations, and coalescing non-ASCII characters such as accented characters to their nearest ASCII equivalents. Some columns in the database were also reformatted to a version more amenable to querying, e.g. values in the `title.title` column listed movie titles with the form *"Terminator, The"*, which were converted to *"The Terminator"*.

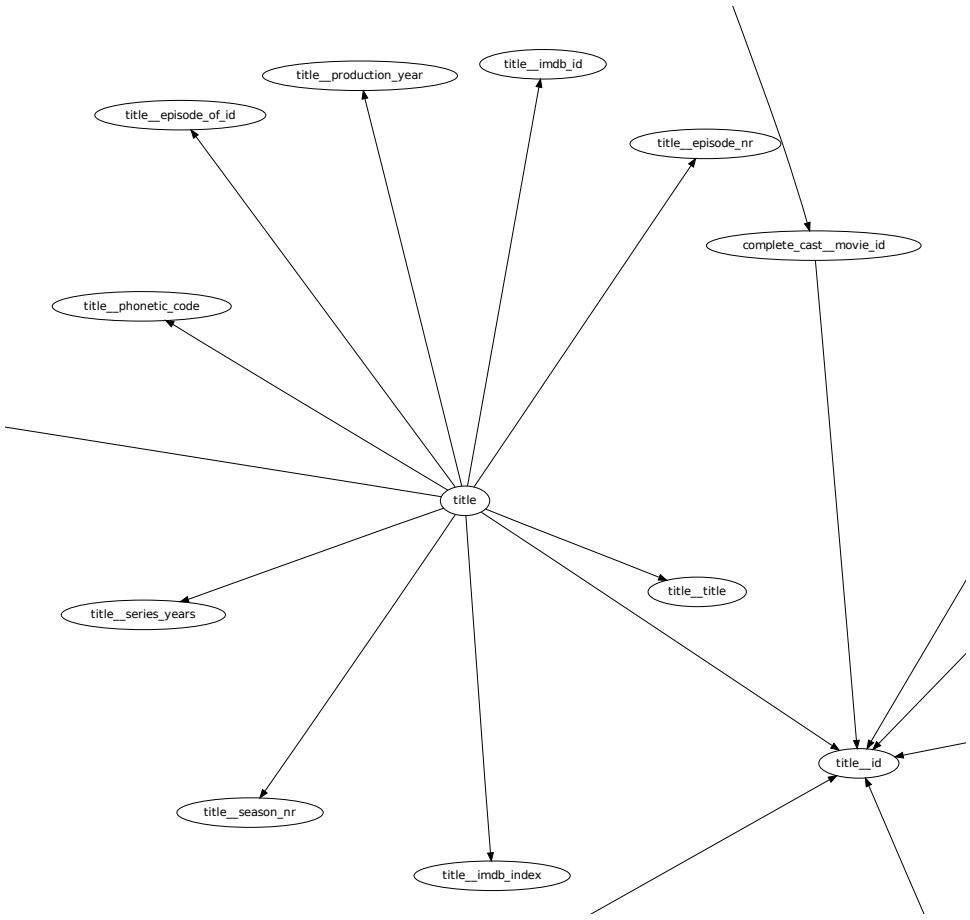


Figure 7.4: Zoomed-in subset of the IMDb schema graph. The `title` table joins with the `complete_cast` table using the `title.id = complete_cast.movie_id` primary-foreign key relationship.

Results

We set k (number of qunit definitions to be formed) to 10, and visualized the variations in qunit derivation in the following manner. Each parameterized SELECT-PROJECT-JOIN SQL query is represented as a node in a graph. The size of the node represents the support (i.e. frequency of unique keyword search queries) of that query in the underlying keyword search query log. Edges represent specializations, and top-K nodes are represented in the color blue.

A number of interesting observations can be made when varying the query log size. At $1,000$ queries, we observe a small number of queries, and the top-k do not have any

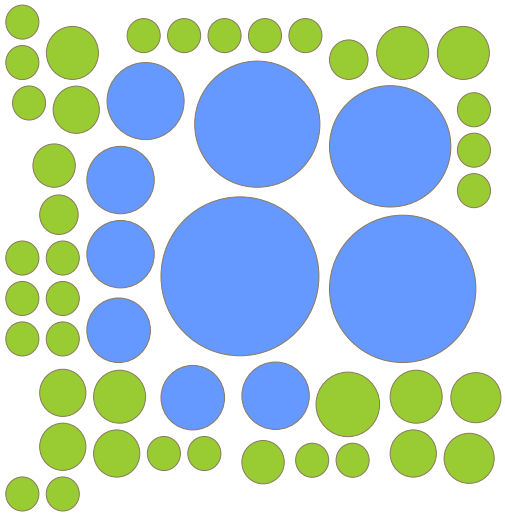


Figure 7.5: Visualization of qunit derivation with a 1,000 query log. Specializations are absent due to small size of query log.

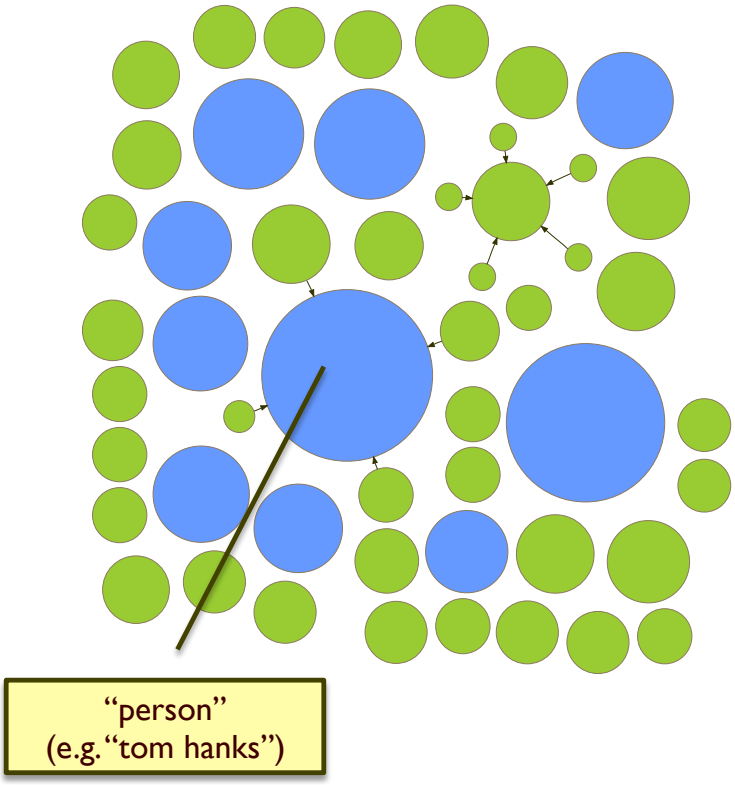


Figure 7.6: Visualization of qunit derivation with a 5,000 query log. Specializations appear as query log size increases.

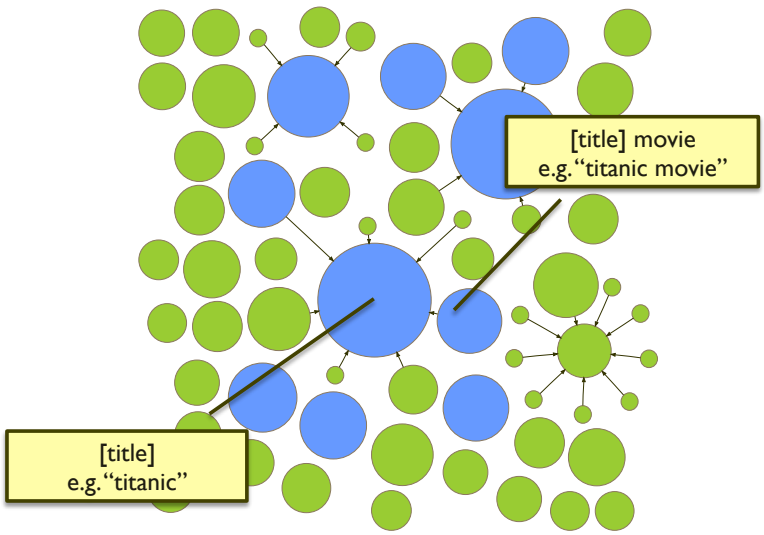


Figure 7.7: Visualization of qunit derivation with a 10,000 query log. Some specializations themselves are in the top-k.

specializations that are above the support threshold. At $n = 5,000$, specializations appear. In Figure 7.6, we notice that specializations for non-top-k queries also show up. At $n = 10,000$ some specializations themselves are in the top-k. At $50,000$ queries, we note that the diversity of queries increases, each top-k node now has a wide variety of specializations. At $100,000$ queries, we notice an interesting phenomena – queries with multiple clauses are specializations of multiple different parent queries. This occurs if there are more than two segments in the specialization, allowing the specialization to have more than 1 parent. At $250,000$ queries, the diversity of the qunits are not that different from the $100,000$ mark, and the ranking of qunits has stabilized. We then visualize the 10 qunits generated as separate colors, each representing a set of parameterized SELECT-PROJECT-JOIN SQL queries.

7.7 Implementation Considerations

We now include some additional implementation considerations that were part of the qunit derivation algorithm.

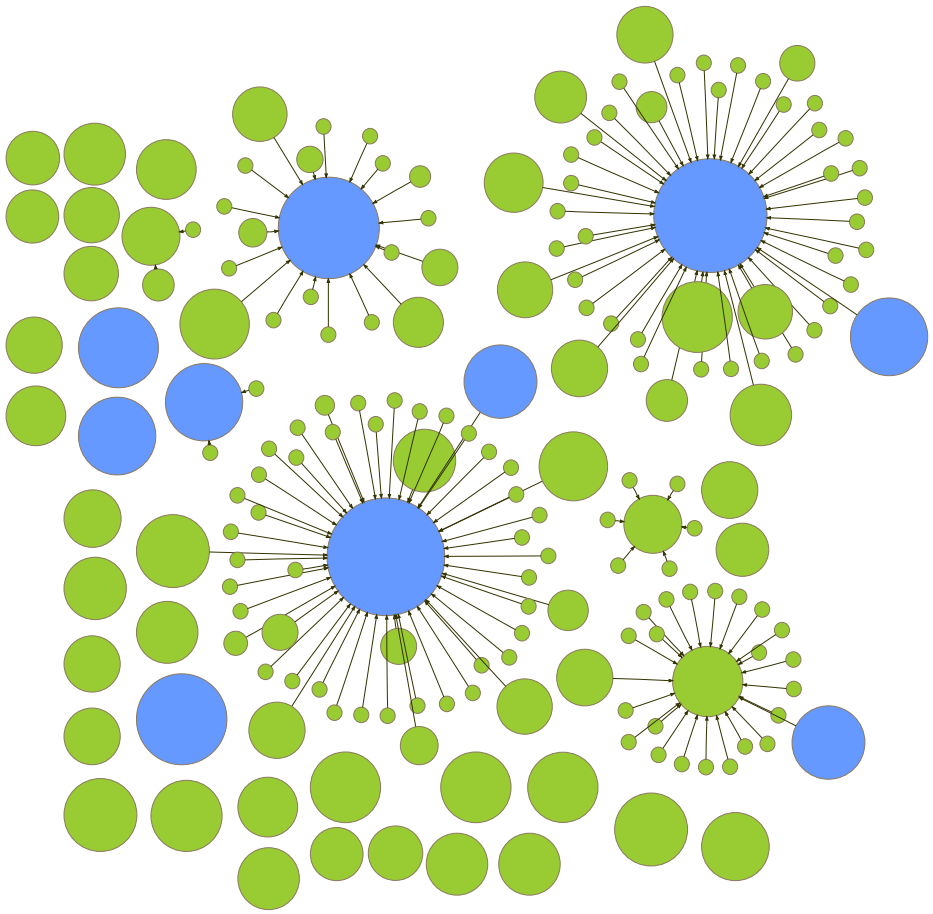


Figure 7.8: Visualization of qunit derivation with a 50,000 query log. As the size of the query log increases, the number of specializations and diversity of the qunits increases.

7.7.1 Sourcing Query Logs

A critical issue in the approach of using query logs to infer qunits is that we encounter a sourcing problem: Query logs are derived from an active search system with a significant number of users. If the goal is to use this method to create a qunits-driven search system, where do the query logs come from? We provide two solutions to this problem:

Bootstrapping

One possible solution is to bootstrap the logs from prior, simpler version of the search engine. Initially, given just the database, a fulltext search engine is set up to search and

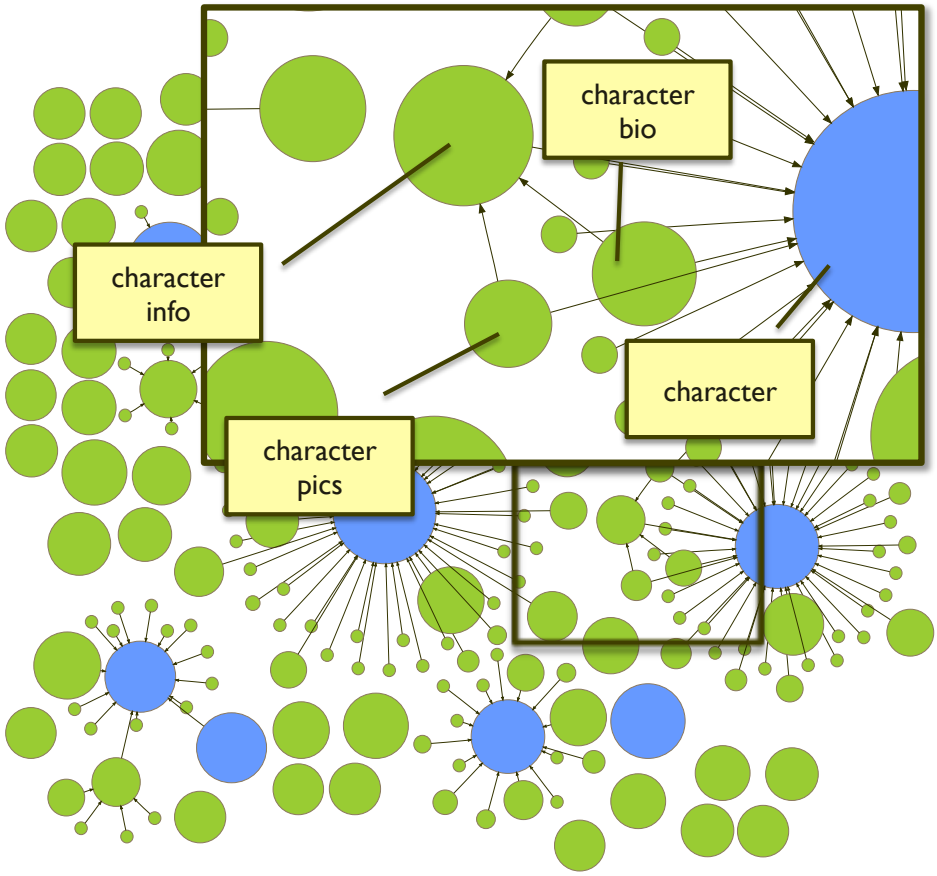


Figure 7.9: Visualization of qunit derivation with a 100,000 query log. Two top-*k* nodes, "character" and "character info" have the same specializations, "character bio" and "character pics".

return individual column values (with their corresponding tuples). This is a viable barebones search engine that can be deployed to the user base. The query logs from this barebones search engine can then be used to derive qunits and launch the first version of the qunits-based search engine. Logs from this system can then train successive versions, until the qunit definitions reach convergence.

Surrogate Query Logs

Another approach towards sourcing logs is to use query logs from a keyword search engine that searches over content that is similar (as opposed to identical) to the content in our database. This approach, described in HAMSTER [119] uses a third-party query log as a

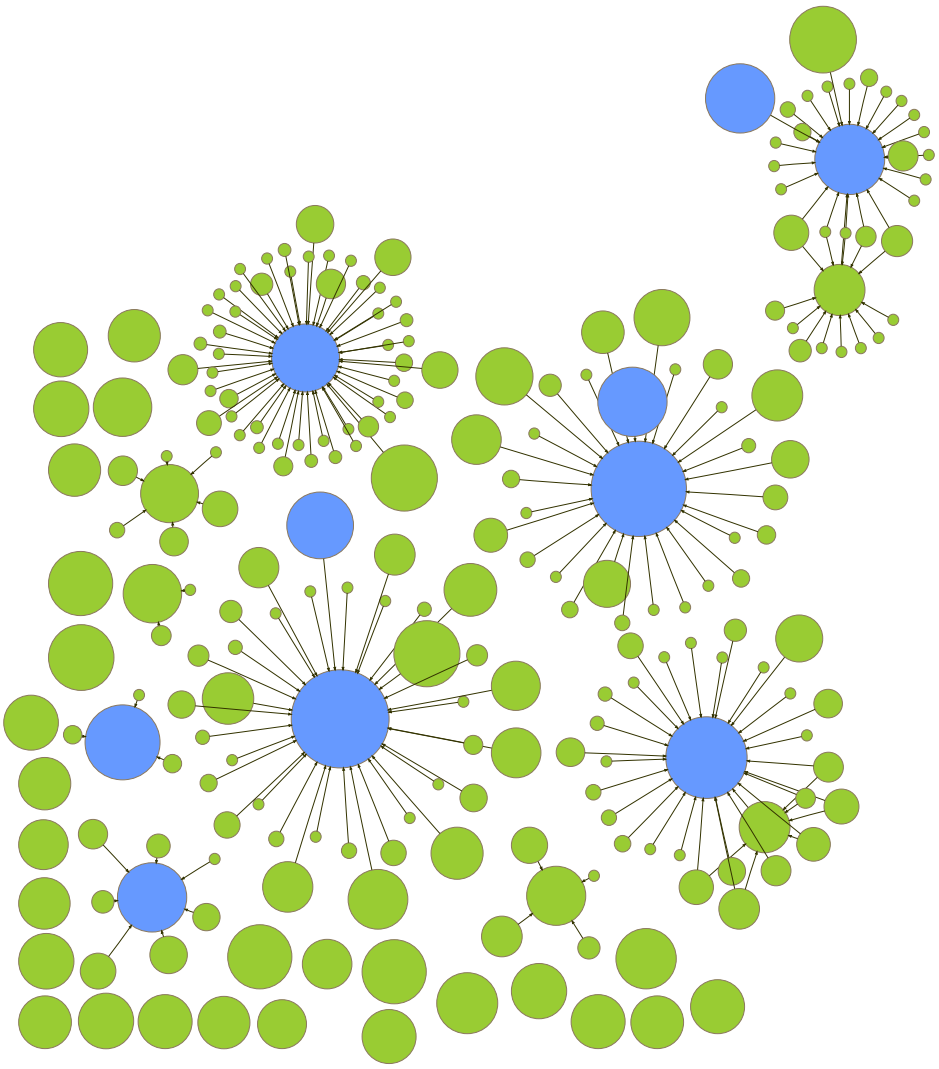


Figure 7.10: Visualization of qunit derivation with a 250,000 query log. The composition of the qunits does not change significantly compared to the qunits from the 100,000 query log.

surrogate. Finding an appropriate surrogate can be done either in a supervised manner by an administrator, or by using a similarity metric (e.g. Jaccard over bag of words) to identify similar content, and then to use searches that led to that content as our training data. This method is particularly applicable with the availability of open-domain search engine query logs [71, 124], subsets of which can be used as surrogate logs for a wide variety of domains.

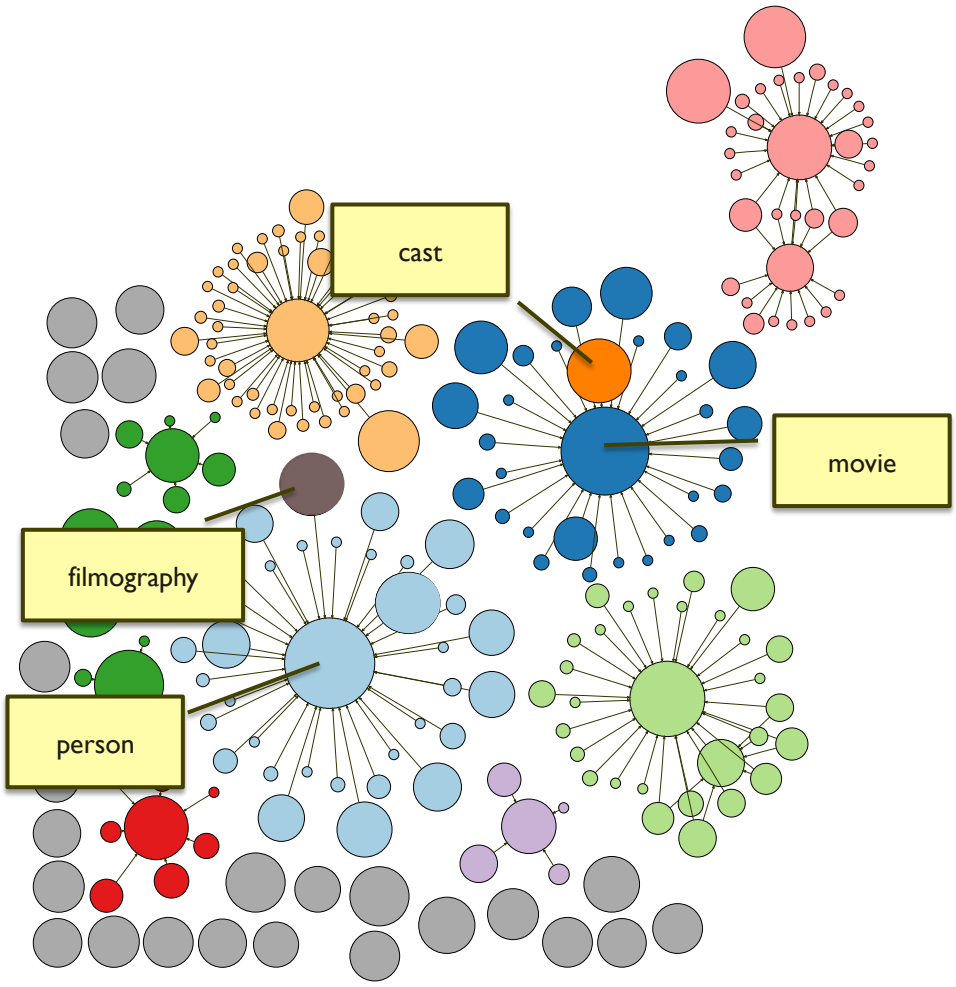


Figure 7.11: Visualization of qunit derivation with a 250,000 query log, grouped into qunits by color.

7.8 Conclusion

The derivation of qunits using keyword search query logs is clearly a practical and achievable task. In this chapter, we formalized the problem of qunit derivation as an optimization problem. In the context of query logs, we bounded the space of possible queries to those inferred by the query log itself, and described the use of workload satisfiability as an objective function. Given this formulation, we then proposed *Query Rollup*, a greedy heuristic to approximate ideal qunits.

As discussed in this chapter, the *Query Rollup* algorithm is a viable heuristic for deriving

qunits. It is modeled on the core insight that qunits can be defined by targeting the most imprecise queries, and that imprecise queries can be answered by their *specializations*.

As demonstrated using both synthetic and real data, this derivation can be performed in a completely unsupervised manner, and requires relatively small amounts of training data. It is competitive with optimal methods of derivation and yet scales well with larger schemas. *Query Rollup* trades off Qunit Recall in favor of Qunit Precision, which is acceptable in the case of search engines where serving popular queries with terse answers is preferable. Qunits derived on the IMDb dataset cover more overall query intent than human-generated qunits, and are anecdotally quite acceptable upon manual inspection.

Chapter 8
Conclusion & Future Work

8.1 Conclusion

Recent years have seen an unprecedented increase of activity on the World Wide Web in the form of data-rich web sites, social networks and web applications. This, along with developments in scientific fields such as astronomy and bioinformatics, has created an explosion of structured data made available to us. In contrast to the conventional notion of consuming data through application layers, end-users increasingly interact with data directly from the source. While advancements in hardware make it feasible to collect, store and process this information, being able to effectively search this structured data is still a daunting task.

Keyword search against structured databases has become an important and popular area of research. Users find structured queries too hard to express, and enjoy the simplicity of a single text input box into which search terms can be entered. However, current solutions are dependent on the quality of user input: we observe that user keyword queries are too succinct and imprecise, and lack enough information to be translated directly into a structured query.

Even when user queries have enough information to communicate the original query intent, we come across a further problem. Database querying is based on the logic of predicate evaluation, with a precisely defined answer set for a given query. On the other hand, in an information retrieval approach, ranked query results have long been accepted as far superior to results based on boolean query evaluation. As a consequence, when keyword

queries are attempted against databases, relatively ad-hoc ranking mechanisms are invented (if ranking is used at all), and there is little leverage from the large body of IR literature regarding how to rank query results.

This dissertation presents a principled approach to searching structured data with imprecise queries. In Chapter 3, we propose the *Qunits Paradigm*, where we divided the search process into four key stages: a vague query intent, a semi-structured query representing the *query space*, a fully structured query (and hence its corresponding result set) representing the *result space*, and the database. We posited that the database was composed of a finite collection of units of queried information: *qunits*. This approach boils the challenge of structured search down to two sub-problems – that of aiding the user to go from the vague query intent to a rich semi-structured query: *query space reduction*, and of enumerating a smaller, finite set of results from a large and potentially infinite set of results in the database: *result space reduction*.

In Chapter 4, we solved the problem of reducing the query space – by urging the user to construct more precise queries using intuitive user interactions over a novel autocompletion-based instant-response interface. In Chapter 5, we identified the task of predicting phrases, as opposed to simply words or entire value strings, as a necessary sub-problem, and discuss the design and implementation of an effective phrase prediction system.

In Chapter 6, we focused on the problem of reducing the result space – by analyzing a database to reduce the space of possible search results into a collection of *qunits*, or queried units that can be searched for using semi-structured queries. We explored various signals that can be used in the derivation of these qunits, including the data, external evidence and keyword search query logs. In the following chapter, we dove deeper into the derivation of qunits using keyword search query logs, demonstrating the ability to infer query intent in a structured database using prior queries.

8.2 Future Work

While this dissertation provides an end-to-end solution towards answering imprecise structured search queries, there are several interesting related problems. In the following two sections, we look at two possible extensions: going beyond conventional search queries, and dealing with problems of query diversity.

8.2.1 Beyond Search

In this dissertation, we focus specifically on search queries, qualified as conjunctive SELECT-PROJECT-JOIN queries over relational data or data that is compatible with the relational model. This class of queries was chosen due to its likeness with web search and that it presents the same conjunctive assumptions, e.g. addition of terms narrow down the search and the result space, search is purely information-seeking, etc. As next steps, it would be of great interest to see the incorporation of three more classes of queries: aggregate queries and disjunction clauses.

Aggregate queries such as *"How many employees are in the Sales department"* are currently not supported in the qunits model as they involve post-processing (in this case, the aggregate is COUNT) on the results. Two possible ways to introduce aggregates into our system are to first consider aggregates as schema-level elements, for suggestion in the autocompletion-based query interface. A second way to adapt aggregation is to introduce them into the qunit derivation process by preprocessing phrases such as *"how many"* or *"average"* to corresponding available aggregate functions in the database.

Disjunctive clauses such as *"Employees in the Sales OR Marketing departments"* are quite useful when dealing with unfamiliar structured data. However, our multi-token autocompletion-based query interface is implicitly conjunctive. To include disjunctions, there are two possible paths. First, we can assume the entire query is disjunctive. This option is the easiest to implement, since this removes the need for context-sensitive suggestions,

but makes it hard to filter out content. A second option is to consider disjunctions with some notion of scoping or nesting. This requires us to introduce the concept of subclauses to both the interface and the underlying search engine. A simple implementation of this is to enable the user to autocomplete multiple queries one at a time, the disjunction of which can be issued to the search engine.

8.2.2 Query Diversity

To aid the ease of use, search engines are providing the ability for users to have their queries enriched by the search engine through techniques such as autocompletion. This feature is typically powered by the analysis of past query logs, providing the user with a ranked list of suffixes and suggestions based on previous query volume. This is one way to enrich the autocompletion-based semi-structured query interface described in Chapter 4. In Chapter 7, we study the use of keyword search query logs to infer qunits. Given the reliance on query logs to shape search infrastructure, care needs to be taken when incorporating these signals, to avoid reinforcement and overfitting issues arising from the feedback loop. A primary concern is that users will tend to converge to queries that are already popular. Secondly, the suggestions might cause the user to abandon her original information need and pursue the suggested query. As these suggestion mechanisms become more pervasive and popular, all queries will be composed using autocompletion-based interfaces. As a consequence, the mechanism that powers these features will eventually run out of unbiased query log data, impacting the quality of the feature itself.

Clearly, there is need to identify possible classes of query log bias, and discuss the implications of reduction in *query diversity*. We propose a mechanism that compensates for the decline in diversity by cross-testing, weighting queries that disagree with the query suggestions, and study its long-term effects on multiple train / log cycles.

Bibliography

[1] S. Acharya, P. Gibbons, and V. Poosala. Aqua: A Fast Decision Support Systems Using Approximate Query Answers. In *Proceedings of the International Conference on Very Large Data Bases*, 1999.

[2] E. Adar. GUESS: A Language and Interface for Graph Exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006.

[3] A. Ageev and M. Sviridenko. Approximation Algorithms for Maximum Coverage and Max Cut With Given Sizes of Parts. *Integer Programming and Combinatorial Optimization*, 1999.

[4] E. Agichtein, S. Lawrence, and L. Gravano. Learning Search Engine Specific Query Transformations for Question Answering. In *Proceedings of the International Conference on World Wide Web*, 2001.

[5] R. Agrawal, R. Bayardo, and R. Srikant. Athena: Mining-based Interactive Management of Text Databases. In *Proceedings of the International Conference on Extending Database Technology*, 2000.

[6] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings of the IEEE International Conference on Data Engineering*, 2002.

[7] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A Graphical Query Language With Recursion. *IEEE Transactions on Software Engineering*, 1990.

[8] P. Anick and R. Kantamneni. A Longitudinal Study of Real-Time Search Assistance Adoption. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2008.

[9] P. Anick and S. Tipirneni. The Paraphrase Search Assistant: Terminological Feedback for Iterative Information Seeking. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 1999.

[10] R. Baeza-Yates, L. Calderón-Benavides, and C. González-Caro. The Intention Behind Web Queries. *Lecture Notes in Computer Science*, 2006.

[11] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query Recommendation Using Query Logs in Search Engines. *EDBT Workshops*, 2004.

[12] A. Baid, I. Rae, A. Doan, and J. Naughton. Toward Industrial-Strength Keyword Search Systems over Relational Data. In *Proceedings of the IEEE International Conference on Data Engineering*, 2010.

[13] B. Bailey, J. Konstan, and J. Carlis. The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface. In *Proceedings of Human-Computer Interaction – INTERACT*, 2001.

[14] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-Based Keyword Search in Databases. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[15] M. Banko, M. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open Information Extraction from the Web. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2007.

[16] H. Bast and I. Weber. Type Less, Find More: Fast Autocompletion Search With a Succinct Index. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2006.

[17] H. Bast and I. Weber. CompleteSearch: Interactive, Efficient, & Towards IR/DB Integration. In *Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[18] S. Beitzel, E. Jensen, O. Frieder, D. Grossman, D. Lewis, A. Chowdhury, and A. Kolcz. Automatic Web Query Classification Using Labeled and Unlabeled Training Data. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2005.

[19] S. Beitzel, E. Jensen, D. Lewis, A. Chowdhury, and O. Frieder. Automatic Classification of Web Queries Using Very Large Unlabeled Query Logs. *ACM Transactions on Information Systems*, 2007.

[20] C. Benson, M. Muller-Prove, and J. Mzourek. Professional Usability in Open Source Projects. *Conference on Human Factors in Computing Systems*, 2004.

[21] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBE. *ACM SIGMOD Record*, 1999.

[22] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases Using BANKS. In *Proceedings of the IEEE International Conference on Data Engineering*, 2002.

[23] S. Bickel, P. Haider, and T. Scheffer. Learning to Complete Sentences. In *Proceedings of the European Conference on Machine Learning*, 2005.

[24] J. Borges and M. Levene. Data Mining of User Navigation Patterns. In *Proceedings of the Workshop on Web Usage Analysis and User Profiling*, 1999.

[25] E. Brill. A Simple Rule-Based Part of Speech Tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, 1992.

[26] B. Britton and A. Tesser. Effects of Prior Knowledge on Use of Cognitive Capacity. *Journal of Verbal Learning and Verbal Behavior*, 1982.

[27] P. Brown, R. Mercer, V. Della Pietra, and J. Lai. Class-Based N-Gram Models of Natural Language. *Computational Linguistics*, 1992.

[28] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-K Queries over Web-Accessible Databases. In *Proceedings of the International Conference on Data Engineering*, 2002.

[29] P. Bruza, R. McArthur, and S. Dennis. Interactive Internet Search: Keyword, Directory and Query Reformulation Mechanisms Compared. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2000.

[30] M. Cafarella, A. Halevy, D. Wang, and Y. Zhang. Webtables: Exploring the Power of Tables on the Web. In *Proceedings of the VLDB Endowment*, 2008.

[31] S. Card, G. Robertson, and J. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1991.

[32] T. Catarci, M. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: a Survey. *Journal of Visual Languages and Computing*, 1997.

[33] S. Chakrabarti, S. Sarawagi, and S. Sudarshan. Enhancing Search with Structure. *IEEE Data Engineering Bulletin*, 2010.

[34] K. Church. Empirical Estimates of Adaptation: the Chance of Two Noriegas is Closer to $p/2$ than p^2. In *Proceedings of the Conference on Computational Linguistics*, 2000.

[35] A. Cockburn and S. Brewster. Multimodal Feedback for the Acquisition of Small Targets. *Journal of Ergonomics*, 2005.

[36] E. Codd. Relational Completeness of Data Base Sublanguages. *Computer*, 1972.

[37] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary. In *Proceedings of the Latin American Symposium on Theoretical Informatics*, 2004.

[38] J. Cowie. Information Extraction. *Handbook of Natural Language Processing*, 2000.

[39] J. Darragh, I. Witten, and M. James. The Reactive Keyboard: a Predictive Typing Aid. *Computer*, 1990.

[40] B. Davison and H. Hirsh. Predicting Sequences of User Actions. *Notes of the AAAI/ICML Workshop on Predicting the Future*, 1998.

[41] A. Doan and A. Halevy. Semantic Integration Research in the Database Community: a Brief Survey. *AI Magazine*, 2005.

[42] C. Duda, D. Graf, and D. Kossmann. Predicate-based Indexing of Enterprise Web Applications. In *Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[43] J. English, M. Hearst, R. Sinha, K. Swearingen, and K. Yee. Hierarchical Faceted Metadata in Site Search Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2002.

[44] R. Fagin, B. Kimelfeld, Y. Li, S. Raghavan, and S. Vaithyanathan. Understanding Queries in a Search Database System. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2010.

[45] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2001.

[46] M. Farach. Optimal Suffix Tree Construction With Large Alphabets. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1997.

[47] U. Fayyad, G. Grinstein, and A. Wierse. *Information Visualization in Data Mining and Knowledge Discovery*. Morgan Kaufmann, 2002.

[48] M. Ferreira de Oliveira and H. Levkowitz. From Visual Data Exploration to Visual Data Mining. *IEEE Transactions on Visualization & Computer Graphics*, 2003.

[49] E. Gabrilovich, A. Broder, M. Fontoura, V. Josifovski, L. Riedel, and T. Zhang. Classifying Search Queries Using the Web As a Source of Knowledge. *ACM Transactions on The Web*, 2009.

[50] E. Giachin and T. CSELT. Phrase Bigrams for Continuous Speech Recognition. *Transactions of the International Conference on Acoustics, Speech, and Signal Processing*, 1995.

[51] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *Proceedings of the International Conference on Very Large Data Bases*, 1998.

[52] Google, Inc. Google Universal Search.

[53] K. Grabski and T. Scheffer. Sentence Completion. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2004.

[54] J. Graupmann and G. Weikum. The Role of Web Services in Information Search. *IEEE Data Engineering Bulletin*, 2002.

[55] L. Gravano, P. Ipeirotis, and M. Sahami. QProber: a System for Automatic Classification of Hidden-Web Databases. *ACM Transactions on Information Systems*, 2003.

[56] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

[57] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proceedings of the International Conference on Database Technology*, 1997.

[58] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[59] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning, 2005.

[60] B. He, M. Patel, Z. Zhang, and K. Chang. Accessing the Deep Web: a Survey. *Communications of the ACM*, 2007.

[61] M. Hearst. Design Recommendations for Hierarchical Faceted Search Interfaces. In *Proceedings of the ACM SIGIR Workshop on Faceted Search*, 2006.

[62] S. Heinz, J. Zobel, and H. Williams. Burst Tries: a Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems*, 2002.

[63] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. Haas. Interactive Data Analysis: the Control Project. *Computer*, 1999.

[64] I. W. Holger Bast. Type Less, Find More: Fast Autocompletion Search with a Succinct Index. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2006.

[65] E. Horvitz. Principles of Mixed-Initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1999.

[66] V. Hristidis and Y. Papakonstantinou. Discover: Keyword Search in Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[67] T. Imieliński and A. Virmani. MSQL: A Query Language for Database Mining. *Journal of Data Mining and Knowledge Discovery*, 1999.

[68] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making Database Systems Usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2007.

[69] H. V. Jagadish, R. Ng, and D. Srivastava. Substring Selectivity Estimation. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 1999.

[70] B. Jansen, D. Booth, and A. Spink. Determining the User Intent of Web Search Engine Queries. In *Proceedings of the International Conference on World Wide Web*, 2007.

[71] B. Jansen, A. Goodrum, and A. Spink. Searching for Multimedia: Analysis of Audio, Video and Image Web Queries. In *Proceedings of the International Conference on World Wide Web*, 2000.

[72] M. Jayapandian and H. V. Jagadish. Automated Creation of a Forms-based Database Query Interface. In *Proceedings of the International Conference on Very Large Data Bases*, 2008.

[73] M. Jayapandian and H. V. Jagadish. Automating the Design and Construction of Query Forms. In *Proceedings of the IEEE International Conference on Data Engineering*, 2008.

[74] M. Jayapandian and H. V. Jagadish. Expressive Query Specification through Form Customization. In *Proceedings of the International Conference on Extending Database Technology*, 2008.

[75] T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski, and G. Gay. Evaluating the Accuracy of Implicit Feedback from Clicks and Query Reformulations in Web Search. *ACM Transactions on Information Systems*, 2007.

[76] R. Jones, B. Rey, O. Madani, and W. Greiner. Generating Query Substitutions. In *Proceedings of the International Conference on World Wide Web*, 2006.

[77] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion for Keyword Search on Graph Databases. In *Proceedings of the International Conference on Very Large Data Bases*, 2005.

[78] I. Kang and G. Kim. Query Type Classification for Web Document Retrieval. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2003.

[79] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-Aware Autocompletion for SQL. In *Proceedings of the VLDB Endowment*, 2010.

[80] B. Klimt and Y. Yang. The Enron Corpus: A New Dataset for Email Classification Research. In *Proceedings of the European Conference on Machine Learning*, 2004.

[81] P. Krishnan, J. Vitter, and B. Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[82] V. Krishnan, S. Das, and S. Chakrabarti. Enhanced Answer Type Inference from Questions Using Sequential Models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2005.

[83] K. Kukich. Technique for Automatically Correcting Words in Text. *ACM Computing Surveys*, 1992.

[84] M. Lacroix and A. Pirotte. Domain-oriented Relational Languages. In *Proceedings of the International Conference on Very Large Data Bases*, 1977.

[85] T. Lau and E. Horvitz. Patterns of Search: Analyzing and Modeling Web Query Refinement. In *Proceedings of the International Conference on User Modeling*, 1999.

[86] U. Lee, Z. Liu, and J. Cho. Automatic Identification of User Goals in Web Search. In *Proceedings of the International Conference on World Wide Web*, 2005.

[87] G. Li, S. Ji, C. Li, and J. Feng. Efficient Type-Ahead Search on Relational Data: a Tastier Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[88] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[89] L. Lim, H. Wang, and M. Wang. Unifying Data and Domain Knowledge Using Virtual Views. In *Proceedings of the International Conference on Very Large Data Bases*, 2007.

[90] B. Liu and H. Jagadish. A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. In *Proceedings of the IEEE International Conference on Data Engineering*, 2009.

[91] Z. Liu and Y. Chen. Identifying Meaningful Return Information for Xml Keyword Search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2007.

[92] C. MacArthur. Word Processing with Speech Synthesis and Word Prediction: Effects on the Dialogue Journal Writing of Students with Learning Disabilities. *Learning Disability Quarterly*, 1998.

[93] J. Mackinlay, P. Hanrahan, and C. Stolte. Automatic Presentation for Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[94] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[95] H. Mannila. Data Mining: Machine Learning, Statistics, and Databases. In *Proceedings of the Scientific and Statistical Database Management Conference*, 1996.

[96] A9.com Inc. Opensearch `http://www.opensearch.org`.

[97] Apple Inc. Spotlight Search
http://www.apple.com/macosx/features/spotlight.

[98] Gnome Foundation. Beagle Desktop Search
http://www.gnome.org/projects/beagle/.

[99] Google Inc. Google Desktop Search
http://desktop.google.com.

[100] Microsoft Inc. Microsoft Office Suite http://www.microsoft.com/office.

[101] Microsoft Inc. Windows Explorer http://www.microsoft.com/windows.

[102] Microsoft Inc. Windows Internet Explorer http://www.microsoft.com/ie.

[103] Mozilla Foundation. Mozilla Firefox version 2 http://www.mozilla.com/firefox/.

[104] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 1976.

[105] K. Mehlhorn. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Letters on Information Processing*, 1988.

[106] Q. Mei, D. Zhou, and K. Church. Query Suggestion Using Hitting Time. In *Proceedings of the ACM Conference on Information and Knowledge Management*, 2008.

[107] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing Out in a Crowd: Selecting Attributes for Maximum Visibility. In *Proceedings of the IEEE International Conference on Data Engineering*, 2008.

[108] G. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 1956.

[109] G. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 1995.

[110] R. Miller. Response Time in Man-Computer Conversational Transactions. In *Proceedings of the Fall Joint Computer Conference*, 1968.

[111] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. *ACM SIGMOD Record*, 2001.

[112] A. Moffat and R. Wan. Re-Store: A System for Compressing, Browsing, and Searching Large Documents. In *Proceedings of the IEEE International Symposium on String Processing and Information Retrieval*, 2001.

[113] L. Mohan and R. Kashyap. A Visual Query Language for Graphical Interaction With Schema-Intensive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 1993.

[114] D. Moldovan, M. Paşca, S. Harabagiu, and M. Surdeanu. Performance Issues and Error Analysis in an Open-Domain Question Answering System. *ACM Transactions on Information Systems*, 2003.

[115] H. Motoda and K. Yoshida. Machine Learning Techniques to Make Computers Easier to Use. *Artificial Intelligence*, 1998.

[116] A. Motro. VAGUE: A User Interface to Relational Databases That Permits Vague Queries. *ACM Transactions on Information Systems*, 1988.

[117] N. Murray, N. Paton, and C. Goble. Kaleidoquery: A Visual Query Language for Object Databases. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, 1998.

[118] B. Myers, S. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 2000.

[119] A. Nandi and P. Bernstein. HAMSTER: Using Search Clicklogs for Schema and Taxonomy Matching. In *Proceedings of the VLDB Endowment*, 2009.

[120] A. Nandi and H. V. Jagadish. Assisted Querying Using Instant-Response Interfaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2007.

[121] A. Nandi and H. V. Jagadish. Effective Phrase Prediction. In *Proceedings of the International Conference on Very Large Data Bases*, 2007.

[122] C. Nevill-Manning, I. Witten, and G. Paynter. Browsing in Digital Libraries: a Phrase-Based Approach. In *Proceedings of the ACM International Conference on Digital Libraries*, 1997.

[123] M. T. Ozsu, L. Chang, and J. Yu. *Keyword Search in Databases*. Morgan & Claypool, 2010.

[124] G. Pass, A. Chowdhury, and C. Torgeson. A Picture of Search. In *Proceedings of the International Conference on Scalable Information Systems*, 2006.

[125] L. Paulson. Building Rich Web Applications With Ajax. *Computer*, 2005.

[126] G. Paynter, I. Witten, S. Cunningham, and G. Buchanan. Scalable Browsing for Large Collections: A Case Study. In *Proceedings of the ACM International Conference on Digital Libraries*, 2000.

[127] A. Pienimaki. Indexing Music Databases Using Automatic Extraction of Frequent Phrases. In *Proceedings of the International Conference on Music Information Retrieval*, 2002.

[128] D. Radev, H. Qi, H. Wu, and W. Fan. Evaluating Web-based Question Answering Systems. *Transactions of the International Conference on Language Resources and Evaluation*, 2002.

[129] G. Robins and A. Zelikovsky. Tighter Bounds for Graph Steiner Tree Approximation. *SIAM Journal on Discrete Mathematics*, 2006.

[130] G. Sacco. Some Research Results in Dynamic Taxonomy and Faceted Search Systems. In *Proceedings of the ACM SIGIR Workshop on Faceted Search*, 2006.

[131] N. Sarkas, S. Paparizos, and P. Tsaparas. Structured Annotations of Web Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[132] A. Schaefer, M. Jordan, C. Klas, and N. Fuhr. Active Support for Query Formulation in Virtual Digital Libraries: a Case Study with DAFFODIL. *Lecture Notes in Computer Science*, 2005.

[133] P. Selfridge, D. Srivastava, and L. Wilson. Idea: Interactive Data Exploration and Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[134] T. Selker and W. Burleson. Context-Aware Design and Interaction in Computer Systems. *IBM Systems Journal*, 2000.

[135] A. Sengupta and A. Dillon. Query by Templates: a Generalized Approach for Visual Query Formulation for Text Dominated Databases. In *Proceedings of the Symposium on Advanced Digital Libraries*, 1997.

[136] M. Shah, M. Franklin, S. Madden, and J. Hellerstein. Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System. *ACM SIGMOD Record*, 2001.

[137] C. Shannon. Prediction and Entropy of Printed English. *Bell System Technical Journal*, 1951.

[138] M. Silfverberg, I. S. MacKenzie, and P. Korhonen. Predicting Text Entry Speed on Mobile Phones. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2000.

[139] A. Simitsis, G. Koutrika, and Y. Ioannidis. Précis: From Unstructured Keywords As Queries to Structured Databases As Answers. *VLDB Journal*, 2008.

[140] G. Smith, M. Czerwinski, B. Meyers, D. Robbins, G. Robertson, and D. Tan. Facetmap: A Scalable Search and Browse Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[141] R. Song, Z. Luo, J. Wen, Y. Yu, and H. Hon. Identifying Ambiguous Queries in Web Search. In *Proceedings of the International Conference on World Wide Web*, 2007.

[142] F. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *Proceedings of the International Conference on World Wide Web*, 2007.

[143] A. Taddei. Shell Choice: A Shell Comparison. Technical report, Guide CN/DCI/162, CERN, Geneva, September, 1994.

[144] B. Tan and F. Peng. Unsupervised Query Segmentation Using Generative Language Models and Wikipedia. In *Proceedings of the International Conference on World Wide Web*, 2008.

[145] S. Tata, R. Hankins, and J. Patel. Practical Suffix Tree Construction. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[146] S. Tata, J. Patel, J. Friedman, and A. Swaroop. Towards Declarative Querying for Biological Sequences. Technical report, CSE-TR-508-05, University of Michigan, 2005.

[147] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the IEEE International Conference on Data Mining*, 2010.

[148] Q. Tran, C. Chan, and S. Parthasarathy. Query by Output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[149] D. Tunkelang. Dynamic Category Sets: an Approach for Faceted Search. In *Proceedings of the SIGIR Workshop on Faceted Search*, 2006.

[150] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 1995.

[151] D. Vickrey, L. Biewald, M. Teyssier, and D. Koller. Word-Sense Disambiguation for Machine Translation. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, 2005.

[152] D. Waltz. An English Language Question Answering System for a Large Relational Database. *Communications of the ACM*, 1978.

[153] H. E. Williams, J. Zobel, and D. Bahle. Fast Phrase Querying With Combined Indexes. *ACM Transactions on Information Systems*, 2004.

[154] Yahoo! Inc. Yahoo Glue.

[155] K. Yee, K. Swearingen, K. Li, and M. Hearst. Faceted Metadata for Image Search and Browsing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2003.

[156] C. Yu and H. V. Jagadish. Schema Summarization. In *Proceedings of the International Conference on Very Large Data Bases*, 2006.

[157] C. Yu and H. V. Jagadish. Querying Complex Structured Databases . In *Proceedings of the International Conference on Very Large Data Bases*, 2007.

[158] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *Proceedings of the International Conference on Very Large Data Bases*, 2005.

[159] D. Zhang and W. Lee. Question Classification Using Support Vector Machines. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval*, 2003.

[160] Z. Zhang and O. Nasraoui. Mining Search Engine Query Logs for Query Recommendation. In *Proceedings of the International Conference on World Wide Web*, 2006.

[161] M. Zloof. Query-by-example: A Data Base Language. *IBM Systems Journal*, 1977.

[162] M. Zloof. Office-by-Example: A Business Language That Unifies Data and Word Processing and Electronic Mail. *IBM Systems Journal*, 1982.