The Faculty of Engineering and Science

Department of Computer Science

TITLE:
Improving quality of keyword query results using ESKuSE

PROJECT PERIOD:
DAT6,
01-01-2003 - 06-06-2003

AUTHOR:
Martin Christensen

PROJECT SUPERVISOR:
Heidi Gregersen

NUMBER OF COPIES: 5

TOTAL PAGE NUMBERS: 33

This report presents the ESKuSE keyword search engine: a fast and efficient keyword search engine that searches relational databases. It models database schema as graphs and uses graphs extensively in its execution of queries.

Based on hints provided at different levels in the database, it makes informed guesses about the data on a conceptual level rather than just a logical.

Martin Christensen

Contents

Chapter 1

Introduction

This report is a continuation of the work presented in [4], although the present report can be read independently of its predecessor. It describes ESKuSE, an acronym for ***ESKuSE** is a **S**tructureless **K**eyword q**u**ery **S**earch **E**ngine*, which is a fast and efficient keyword search engine for relational databases.

1.1 Motivation

Today, large amounts of data are stored in relational databases. They are almost exclusively queried in a structured fashion using query languages such as SQL. A user who wishes to issue his own queries on the database must have knowledge of its schema, and must either use SQL (or equivalent languages) or rely on e.g. graphical query design tools. Most users, however, only interface indirectly with the database through applications that hide communication with the database through predefined queries.

This imposes several restrictions on the user. First of all, it raises the bar on the qualifications needed to issue queries. Secondly, structured queries will only return the results that the user expects to get, whereas keyword queries might also find results that are unexpected. Thirdly, structured queries allow almost no vagueness, but the user may not have enough information to make an adequately precise query.

Keyword querying will not displace structured querying in its current roles, since the latter has significant benefits with regard to control of the query results and raw performance, but the two complement each other well. Perhaps most significantly, keyword query engines can dramatically reduce the effort required to publish structured data and subsequently perform queries on these data [2].

The great success Internet search engines shows that people with no special training or education take very easily to using keyword queries to find relevant information on the Internet. Keyword search engines for relational databases can present users with a similar interface, which will thus be both familiar and unintimidating to even inexpert users. This could easily prove to become a significant new source of information on corporate networks, though the general method is still very young, and has not yet seen widespread usage.

1.2 Keyword queries

Today, we know keyword queries primarily from Internet search engines. We can enter our query as a list of keywords, which are normally just words and/or numbers, and that's all the search engine needs to find documents that match our query. While a relational database is very different from a collection of web pages, queries may well look exactly the same and be expected to return results in much the same way.

Keyword queries, by their nature, offer only limited expressive power. As we know them from Internet search engines, keyword queries are normally either considered conjunctive or disjunctive. Consider a keyword query as a set of keywords $\{a, b, c\}$. A conjunctive query will require all keywords to be found, i.e. $a \wedge b \wedge c$, and a disjunctive query will require at least one keyword to be found, i.e. $a \vee b \vee c$.

Most search engines, though, allow a combination of the two, which is to say that some keywords are specified to be required, typically by prefixing the keyword with a $+$, and the remainder are optional. Given such rules, the query $\{+a, b, c\}$ would be expressed as $a \vee (a \wedge (b \vee c))$. Furthermore, exclusion of certain keywords, commonly denoted by prefixing the excluded keyword with a $-$, is also supported by most search engines, where $\{-a, b, c\}$ would be expressed $\neg a \wedge (b \vee c)$. The $+$ and $-$ prefixes add significant improvements in expressive power of a query while not adding significantly to the complexity of the query itself. Generally, exclusive-or queries are not supported.

The popularity of today's Internet search engines shows that people generally are comfortable with using keyword queries. When submitting a keyword query to a search engine, the usual way of thinking of it is as being 'as conjunctive as possible', i.e. the user expects as many keywords as possible to be found, but if not all of them can be found in any one result, fewer will suffice. Let's call these queries 'mostly conjunctive' as opposed to 'strictly conjunctive', the latter which will be a true conjunctive query. The impreciseness in this form of query can either be hidden in more complex logical expressions of the query than is shown here, or, perhaps more likely, in some ranking system. In the latter case, the query might be entirely disjunctive, but the ranking system could favour results in which many of the keywords in the query occur. Fortunately, all this complexity is hidden from the view of the user, who can usually get good search results without any knowledge of the underlying system.

1.3 Related work

The work presented in this report is the continuation of [4]. The ESKuSE system has seen considerable speed improvements since the writing of aforementioned report, and the quality of search results has been significantly increased. Speed improvements have been introduced mainly through simple mass-query optimisation, which dramatically reduces the number of SQL queries necessary to perform for each keyword query, especially on more complex queries. Furthermore, an algorithm that generates 'templates' for all possible candidate networks, which are then stored in the schema graph, thereby avoiding many redundant calculations. Quality improvements are achieved by improving the algorithms involved in executing the query, by making informed guesses about the conceptual database model based on information about the logical database model, and by adding to the expressive power of keyword queries.

ESKuSE is in many ways similar to the DISCOVER system presented in [7]. DISCOVER's method of ranking query results is implicit in its search algorithm. ESKuSE allows for different ranking techniques, but currently it only uses graph weights in its data structures for this purpose. DISCOVER, using its Master Index, is able to locate precisely the tuples containing keywords, but only uses this knowledge to generate tuple sets, where it could possibly also be used to further optimise SQL queries. DISCOVER has a practical upper limit of candidate network sizes of around $5 - 6$ on a 100 MB database; larger candidate networks take a prohibitively long time to evaluate. This limits the usefulness of the system to either small queries or small databases. For queries to take prohibitively long for ESKuSE to perform on a database of similar size, either the query must consist of very many keywords, or one or more of the keywords in the query must appear in thousands of tuples. The former problem should be considered a user error, and the latter problem will often be solved by removing 'too common' words from the index like Internet search engines most often do.

In the BANKS system, described in [2], a database is perceived as a graph in which every tuple is a node, and where relationships represent edges. Since the entire database is perceived as a graph, which is held in main memory, this graph will be larger than the database itself, setting a much lower limit to the size of database that can be practically searched than ESKuSE. ESKuSE's query graphs and candidate networks can be considered incomplete or uninstantiated subgraphs of the larger graph that BANKS uses. The authors employ a ranking system similar to that of Google [3], where references to tuples count as votes. Also they use the 'fan out' of connecting nodes in ranking: if a query is performed on the relationship between two people, and they both belong to a group of 6 people and another group of 100 people, then the smaller group will be taken to be the closer relationship. This ranking technique has been adapted for use in ESKuSE. BANKS provides a means for the user to interactively refine query results. Currently, ESKuSE only returns final, unalterable results, but it is possible that ESKuSE could be made to support something similar with fairly minor changes to the existing code base.

DBXplorer's symbol tables, described in [1], are much like the index used in ESKuSE. The paper explores different indexing strategies which could prove valuable to further work on ESKuSE's index, but which would require completely rethinking keyword nodes. A problematic limitation of DBXplorer is that it cannot connect two tuples in the same relation.

1.4 Problem definition

This project aims to finish the work started in [4] and create a fully functional keyword search engine backend for relational databases that is practically usable in production environments.

To meet this goal, two equally important subgoals must be met:

- query results must be returned quickly, and
- query results must be of high quality.

Being intended for interactive use rather than batch processing, ESKuSE should be able to return results to most queries within a matter of seconds, or the point of interactive use is soon lost. Thus the speed issue becomes important. The importance of high-quality query results is obvious, since no-one will want to use a search engine that yields mostly useless results.

1.5 Composition of the report

This report is laid out as follows.

Chapter 2 presents the basic ESKuSE system. It provides a walk-through of the ESKuSE architecture, defines the data structures used and describe the main algorithms involved in performing keyword queries. Also, it describes the most important speed optimisations.

Chapter 3 discusses the concept of quality in query results, and it presents the techniques that ESKuSE uses to improve its results.

Chapter 4 presents and discusses the tests that have been performed with ESKuSE and their results.

Chapter 5 concludes the report and discusses possible future work.

Chapter 2

ESKuSE fundamentals

This chapter describes the basic ESKuSE system. It is meant to address some of the clarity issues in [4]. Many of the unused implementation options mentioned in [4] are omitted from this paper so as to not add unnecessary complexity.

ESKuSE is a search engine that searches arbitrary relational databases. The basic units that we search for with ESKuSE are keywords. To formalise the notion from Chapter 1.2, the following definition is given:

Definition 2.1 (keyword) *A keyword kw is a tuple $(TYPE, value)$, where $TYPE$ is the data type of the keyword, and $value$ is the what is being searched for in the database. The $value$ element must be of type $TYPE$.*

From this, the definition of keyword queries follows trivially:

Definition 2.2 (keyword query) *A keyword query Q is a set of keywords $\{kw_0, kw_1, \ldots, kw_n\}$.*

As an example, $(STRING, john)$ and $(INTEGER, 42)$ are possible keywords, and they can form the keyword query $\{(STRING, john), (INTEGER, 42)\}$.

With a proper definition of keyword queries in hand, we can begin to explore how they are processed.

2.1 Architecture

This section describes ESKuSE's architecture. It serves to give the reader an understanding of how the elements introduced throughout this chapter tie together. In this section, terms are used that are not immediately explained; these terms are detailed later in this chapter. Figure 2.1 illustrates the ESKuSE architecture. The 'Caller' on Figure 2.1 is assumed to be some program that uses ESKuSE as a back-end.

Schema This is ESKuSE's representation of the schema of the database. It is read from a definition file that is written by hand. It is possible to create a valid schema definition that does not represent the database schema with total accuracy; for instance, it might be desirable that certain relations be made unavailable to ESKuSE. Individual attributes in any relation may be flagged as 'do not index', which means that ESKuSE will not attempt to find keywords in attributes with this flag.

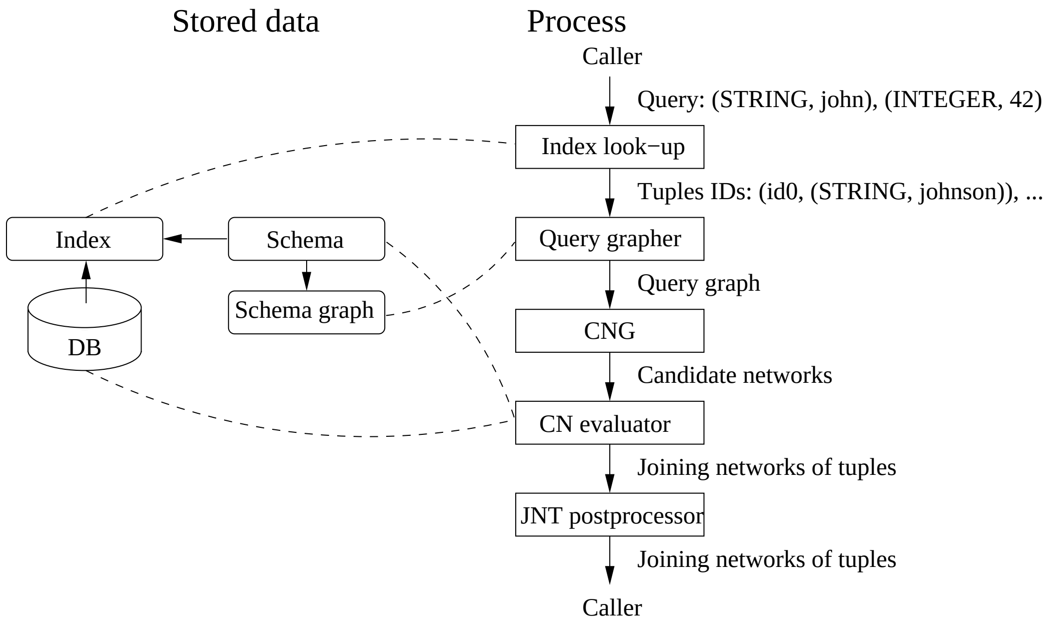


Figure 2.1: Model of ESKuSE's architecture. The rounded boxes to the left represent data resources, and the sharp-edged boxes to the right represent processes. Solid lines with arrows represent data flow, and punctured lines represent dependency on data resources.

Schema graph The schema graph is a graph representation of the logical database structure as described by the above mentioned schema description.

Index This is a database-wide index that it used to find the tuples that given keywords can be found in. It is basically an inverted file index. When given a keyword as an input, it returns identifiers all indexed tuples that the keyword can be found in. A tuple identifier has the form $(relation, pkvalue)$, where $relation$ is the relation that the tuple belongs to, and $pkvalue$ is the value of the primary key of the tuple. For this reason, only relations with primary keys may be indexed.

Index look-up This component looks up the keywords it receives from the calling program and output the tuple identifiers for the tuples that each keyword was found in.

Query grapher Based on the stored schema graph and the tuple identifiers from above, the this component creates a query graph for the current keyword query.

CNG The CNG (candidate network generator) finds candidate networks in the query graph it receives as input, eliminates any redundancy in the candidate networks and outputs them.

CN evaluator This component translates the candidate networks it receives into SQL queries, which it then executes. This yields joining networks of tuples, which it outputs. The translation from candidate network to SQL query requires knowledge of primary keys and foreign key relationships, which is obtained from ESKuSE's internal schema representation.

JNT postprocessor The joining networks of tuples received from the previous component are combined as described in Chapter 2.3 and output according to rank, best ranking results first.

2.2 Data model

Schema structure information is presented on a well-known form by representing the schema as a graph.

First, let's examine our database schema. Let $S = (R, K)$ be the schema over our database of interest, where $R = \{r_0, r_1, \ldots, r_n\}$ is the set of relations in S, and $K = \{k_0, k_1, \ldots, k_m\}$ is the set of foreign key relationships between relations in S. Each $k_i \in K$ is represented as (r_k, r_l), r_k and r_l being relations in R, where there's an attribute in r_k that is a foreign key referencing an attribute in r_l. It may be allowed that $k = l$.

Using the above, we can define our graph to represent the schema. We call such graphs schema graphs. They are directed, weighted graphs and contain only one node type: relation notes.

Definition 2.3 (relation node) *A relation node n corresponds to exactly one relation $r \in R$, where R is the set of relations in our schema.*

As a matter of terminology, n is said to represent r, or, conversely, r maps to n. The notation $r \rightarrow n$ is used to show that r maps to n.

A basic schema graph directly represents a schema without trying to make implicit information explicit. This type of graph is what was called a naïve schema graph in [4]. It serves as the basis for creating more elaborate schema graph as we'll see in Chapter 3.

Definition 2.4 (basic schema graph) *Let $G^S = \left(N^S, E^S\right)$ be the basic schema graph for S, where N^S is the set of relation nodes in the graph, and E^S is the set of edges. $N^S = \{n_0, n_1, \ldots, n_n\}$ where each element in R is represented by an element in N^S.*

Each edge $e \in E^S$ is represented as (n_k, n_l, w), where $n_k, n_l \in N^S$, and w is the weight of the edge. E^S is defined as $\forall (r_k, r_l) \in K \; \exists n_k, n_l \, (r_k \rightarrow n_k \wedge r_l \rightarrow n_l) \Rightarrow (n_k, n_l, 1), (n_l, n_k, 2) \in E^S$.

As an example of a basic schema graph, consider the simple schema $(\{r_0, r_1\}, \{(r_0, r_1)\})$. If $r_0 \rightarrow n_0$ and $r_1 \rightarrow n_1$, this schema will be represented by the basic schema graph depicted in Figure 2.2(a).

Schema graphs are not used directly in queries. We assume that the database schema only very rarely changes, and thus the schema graph that represents it need only change equally rarely. If the schema changes, the schema graph must be rebuilt.

When a query is performed, a query graph is built based on the schema graph. This schema graph can be a basic schema graph or a more advanced schema graph, but the procedure for building the query graph will be the same. A query graph is built from the schema graph, i.e. the schema graph will be a subset of the query graph.

The main thing that distinguishes a query graph from a schema graph is the presence of another node class: keyword nodes.

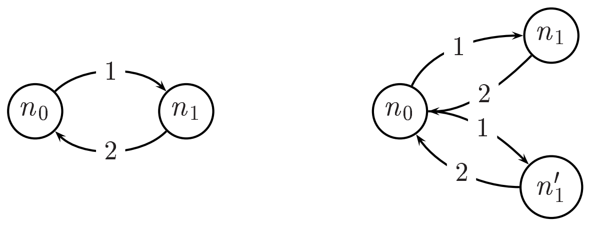
Definition 2.5 (keyword node) *A keyword node n' represents a tuple $t \in r$, where $r \in R$. Let Q be a keyword query. There exists a non-empty set of keywords $KW^t = \{kw_0, kw_1, \ldots, kw_n\}$, where $KW^t \subseteq Q$, such that each $kw_i \in KW^t$ can be found at least once in t, and no kw_j can be found in t such that $kw_j \in Q \wedge kw_j \notin KW^t$.*

As a matter of notation, a keyword node n' is said to represent both the tuple t, the set of keywords KW^t and every element of KW^t. Furthermore, if $t \in r$, then n' is said to stem from the relation node n, where $r \rightarrow n$.

When we perform a keyword query, keyword nodes represent the locations of the data we are interested in. When we insert keyword nodes into the query graph in the appropriate places, we will have sufficient information to create SQL queries to determine if and how the tuples that the keyword nodes represent are connected.

Definition 2.6 (query graph) *Let query graph G'^S be created for a query $Q = \{kw_0, kw_1, \ldots, kw_n\}$ over schema graph G^S. For every tuple in the database that contains at least one keyword, a keyword node n' is created to represent it. If n' stems from n, then n' will be given the same edges with the same weights leading to and from it as n.*

For an example of a simple query graph, consider the example basic schema graph given above. If some keyword is found in some tuple in the relation that maps to n_1, the resulting query graph will be the one depicted in Figure 2.2(b).



(a) Schema graph

(b) Query graph. n'_1 is a keyword node

Figure 2.2: Simple examples of a basic schema graph and a query graph built from it.

The two classes of graphs that have just been described are used to represent the data that queries are performed upon. Using these, we can adequately describe a keyword query. However, they are insufficient to describe the results yielded by such queries.

Consider the following facts:

- schema graphs, and by extension query graphs, accurately represent the underlying logical structure of the database,
- the information we are after can be found in the result set of some SQL query,
- SQL queries are expressed through a combination of keyword information and structural information, both of which are present in the query graph.

Given these facts, we now know that it is possible to represent the eventual result of a keyword query as a subgraph of the query graph, because it can be transformed into an SQL query whose result set is relevant to the keyword query. Because of the limited expressive power of keyword queries as discussed in Chapter 1.2, it is not necessary to use more complicated SQL queries than can adequately be expressed by ESKuSE's data structures.

The result of a keyword query is the result set of one or more SQL join queries, just as it quite often is when we use SQL directly. These results will be joining networks of tuples, which are, as the name implies, tuples that join, i.e. the result of a joining SQL query. When working with query graphs, we can't tell which tuples join and which do not, just as we can't tell what the result set of an SQL query will be (most of the time). When writing SQL queries, we specify what we want our result set to look like and how we want to narrow it. It is exactly the same thing we do when working with query graphs. To represent the possible joining networks of tuples on the graph level, we have candidate networks. We might say that candidate networks are templates for our query results.

In [7], the authors describe two properties of their Joining Networks of Tuple Sets, namely that they can be *total* and *minimal*. These properties are also useful for ESKuSE, and they are given the following meanings:

Let a query graph G'^S be created over schema graph G^S based on query Q. Consider a subgraph T of G'^S that is a tree. T is *total* if all keywords in Q are represented by at least one keyword node in T, and T is *minimal* when the root node and every leaf node is a keyword node.

With these definitions in hand, the definition of candidate networks follows simply:

Definition 2.7 (candidate network (CN)) *Let T be a subgraph of query graph G'^S. If T is a tree and it is minimal, T is a candidate network (CN).*

With CNs being minimal, we help prevent superfluous data from 'polluting' the results. Unlike DISCOVER, ESKuSE does not require CNs to also be total, meaning that keyword queries are not strictly conjunctive.

So far, we have only dealt with data descriptions and pointers to data. As mentioned previously, CNs can be translated into SQL queries. This makes bridging the gap between data descriptions and concrete data, as represented by joining networks of tuples, easy.

Definition 2.8 (joining network of tuples (JNT)) *A joining network of tuples (JNT) is an instantiation of a CN.*

A set of JNTs is the final result of a query.

This concludes the definition of all ESKuSE's central data structures, from the input received from the caller to the final output that it will be given in return.

2.3 Basic algorithms

This section describes the algorithms used in the current CNG and JNT postprocessor.

First, please note that there is a restriction on how the CNG that ESKuSE currently uses generates CNs, which is simpler than the definition of CNs given. Currently, CNs do not branch (DISCOVER calls non-branching CNs *sequences*), and they only contain two keyword nodes, one at either end. As such, the JNTs that the CNs generate can be thought of as pairs of relevant and related tuples.

The 'relatedness' of data is defined by the important constant $MaxPathLen$, whose role will presently be described. As we already know, CNs are created from query graphs, and query graphs are weighted, directed graphs. Consider a CN as a path through the query graph. The length of the path will be the sum of weight of the edges in the CN in the direction from the root to the leaf. If the length of the path is greater than $MaxPathLen$, then the tuples at either end are not considered to be related, and thus the path is not accepted as a CN, since the user will not want the search results to contain unrelated data. $MaxPathLen$ should be set with some care: if it has too small a value, the CNG will ignore data that should be considered related, and if it has too large a value, data may be included that should not be considered related.

Consider the following example. If two tuples join without any intermediate tuples (e.g. `SELECT * FROM r1, r2 WHERE r1.pk = r2.fk`), there is obviously bound to be a close relationship between the two tuples, but if there are many intermediate tuples (e.g. `SELECT * FROM r1, r2, ..., rn WHERE r1.pk = r2.fk AND ...AND r(n-1).pk = rn.fk`), the data in the tuples at either extreme are not likely to be closely related.

$MaxPathLen$ also impacts performance. Since it indirectly affects the largest allowed size of CNs, it also affects the size of the largest SQL join queries that the DBMS must perform.

Now that $MaxPathLen$ has been accounted for, we can look at how the two components work.

Initially, the CNG receives a query graph. Algorithm 2.1 shows the Python code use in the previous implementation of the CNG algorithm. Due to optimisations described in Chapter 2.5, these computations have been moved elsewhere, but the algorithm shows how the CNG works conceptually. It employs a depth first search-like algorithm to search from each keyword node in the query graph. Whenever another keyword node is reached from the start node that does not make the sum of the weight of all the edges between the start node and the current node larger than $MaxPathLen$, the subgraph on the current path is appended to the list of CNs.

When all possible CNs from each node have been calculated, redundancies will be found and eliminated. Consider two CNs a, b, c and c, b, a. They represent the same information, and thus they are redundant. Whichever of them has the larger sum of edge weights is removed, or if they have the same weight, one is chosen arbitrarily for removal.

The now redundancy-free list is output from the CNG and passed to the CN evaluator, which creates JNTs from the CNs. The evaluation process itself is very straightforward: an example of its use is given in Chapter 2.4, and more information can be found in Chapter 2.5. The list of JNTs is passed to the JNT postprocessor.

First, here's a short overview of how the JNT postprocessor works conceptually: It creates composite JNTs from the smaller JNTs it receives. It does this by first creating an undirected graph from the received JNTs: its nodes represent the same tuples that were represented by keyword nodes in the query graph, and its edges are the JNTs that join the different tuples. The resulting graph may not be connected. The composite JNTs will each span an entire connected component within the graph. Because of the way that the ESKuSE implementation handles JNTs internally, it would be impractical to use actual code snippets to illustrate the algorithms like above, so instead pseudo-code algorithms are given, though using Python syntax.

The algorithm for creating the JNT graph is given in Algorithm 2.2. Conceptually, this graph can be considered an instantiation of a query graph. Consider a basic JNT $t_0, t_1, \ldots, t_n$ as output by the CN evaluator, where $1 \leq n \leq MaxPathLen$. We know that t_0 and t_n are tuples that were represented by keyword nodes, or the network that the JNT was created from would not be minimal and thus not a CN. We also know that t_0 and t_n may be shared with other JNTs because the same keyword nodes may appear in several CNs. It then makes sense to think of t_0 and t_n as nodes in a graph, and $t_1, \ldots, t_{n-1}$ as an undirected edge connecting them. This is how JNT graphs are created.

Consider the JNT $t_0, t_1, \ldots, t_n$ from before and another JNT $t_n, t_{n+1}, \ldots, t_{n+m}$. Because they have the tuple t_n in common, $t_0, t_1, \ldots, t_{n+m}$ will also be a JNT. In this way, we can continue merging JNTs that share tuples into composite JNTs. This is the principle behind Algorithm 2.3. It starts with

```
MPL = 3 # Constant MaxPathLen. It may be different from 3.

def crawl(g, s): # Takes a query graph and a start node. Initialise the crawl.

    path = []
    candidatepaths = []
    pathlength = 0

    candidatepaths = __crawl(g, s, s, path, pathlength, candidatepaths)

    return candidatepaths


def __crawl(g, s, node, path, pathlength, candidatepaths):

    newpath = path[:] # Avoid Python’s reference semantics, so do manual copy-by-value.
    newpath.append(node)

    for child in node.children.keys():
        if not child.name == s.name: # We don’t want to go back to s.
            newpathlength = pathlength + node.children[child]

            if newpathlength <= MPL:
                # Do not continue crawling if it’s a keyword node.
                if type(child) == keyword_node:
                    newpath.append(child)
                    candidatepaths.append((newpath, newpathlength))
                else:
                    candidatepaths = __crawl(g, s, child, newpath, newpathlength, candidatepaths)

    return candidatepaths
```

Algorithm 2.1: CNG algorithm used to find paths of size $MaxPathLen$ or less.

```
def mk_jntgraph(keywordnodes, jnts): # Takes a list of all keyword nodes from the query
                                     # graph and all JNTs from the CN evaluator.
    jntgraph = graph()

    for kwn in keywordnodes: # Create a node for each tuple represented by a keyword node.
        jntgraph.add_node(kwn)

    # At the head and tail of each JNT from the CN evaluator is a tuple that is represented
    # by a keyword node. For each JNT, create an edge between the nodes representing its
    # head and tail.
    for jnt in jnts:
        kwn_u = head(jnt)
        kwn_v = tail(jnt)
        kntgraph.add_edge(kwn_u, kwn_v)

    return jntgraph
```

Algorithm 2.2: Pseudo-code algorithm to create a graph from the JNTs produced by the CN evaluator.

an arbitrary node in the JNT graph and, using a breadth first algorithm, grows the JNT to span the entire connected component of the graph, which may be the entire graph. The exception to this rule is when a node has no edges: that means that the tuple does not participate in any JNTs, and thus it is considered irrelevant by this algorithm. Ultimately, the algorithm will output one JNT per connected component with more than one node.

```
def jntgraph_search(jntgraph): # Takes a JNT graph as created by mk_jntgraph().

    nodes = jntgraph.get_nodes()
    jnts = [] # The list of JNTs that we create from the graph. Recall that JNTs are trees.

    # Breadth first search that removes visited nodes from the graph and appends them to
    # the current JNT.
    while nodes:
        t = tree()
        a = nodes.pop()
        if not a.children(): # Disregard unconnected nodes: they do not participatey in any JNTs.
            continue
        tree.append_leaf(a, None) # Make 'a' a leaf node of None, i.e. the root.

        for i in tree.leaves():
            for j in i.children():
                tree.append_leaf(j, i)
                nodes.remove(j)

        jnts.append(tree)

    return jnts
```

Algorithm 2.3: Pseudo-code algorithm to make larger JNTs from the graph produced by Algorithm 2.2.

The order in which the JNT postprocessor outputs JNTs depends on the ranking of the JNTs. A low ranking value is considered better than a high one. The rank of a JNT depends on two things: the number of keywords in the JNT (keyword rank kwr), and the sum of the weights of the edges in the CN that can generate the JNT (edge weight rank ewr). A JNT will have a rank of $ewr - kwr$.

We will want as many of the keywords in the keyword query to be represented in our results. We will also want the same keyword to be represented several times, if possible, but the more times the same keyword appears, the less interesting another occurrence will be. For each keyword kw_i in a query that occurs k times in the JNT, we compute the partial keyword rank $kwr_i = \sum_{j=0}^{k} \frac{1}{j}$, and the collected keyword rank will be the sum of the partial keyword ranks, i.e. $kwn = \sum_i kwn_i$. As an example, consider keyword query $Q = \{kw_0, kw_1\}$, where kw_0 occurs 3 times in the JNT under our consideration, and kw_1 occurs 2 times. We then have that $kwr_0 = \frac{1}{1} + \frac{1}{2} + \frac{1}{3}$, and $kwr_1 = \frac{1}{1} + \frac{1}{2}$, and by summation, $kwn = \frac{10}{3} \approx 3.3$.

Arguments have been given above for how JNTs can be merged to form larger JNTs. The same must necessarily also be true for CNs, since CNs are what generate JNTs. The method for calculating the combined edge weight, or path length, remains the same. This combined edge weight is set to equal ewr. Since we can now compute both ewr and kwr, we can determine a JNTs rank.

The JNT postprocessor, as it works now, has several drawbacks. Growing the composite JNTs to span the entire connected component of the JNT graph can easily create JNTs that are significantly larger than the user expects as a result. It's unlikely that the user will very often have results in mind that span more than a handful of relations or so. Smaller JNTs could be just as relevant, but much more manageable. Also, Algoritm 2.3 discards arbitrary JNTs, meaning that they will not be used in the composite JNTs: if we consider a strongly connected component with the set of nodes $\{a, b, c\}$, then, if we start the breadth first search from node a, the JNT represented by the edge (b, c) will never be considered. Despite its drawbacks, the JNT postprocessor is very fast, running in $\mathrm{O}(n)$ with respect to the number of keyword nodes in the query graph (even if, technically, it never actually sees the query graph). An algorithm that addresses the above mentioned drawbacks, but which conversely is slower, is under development. Chapter 5.2 gives further mention of this new algorithm.

2.4 Example of use

This section gives a complete example of how ESKuSE would execute a query on a database. This example does not consider any optimisations made to ESKuSE, nor any of the refinements presented in Chapter 3.

For this example, a small library database has been created. Its schema is depicted in Figure 2.3, and its data contents can be seen in Table 2.1. It contains information about books and their authors, borrowers and what books they have checked out in the past.

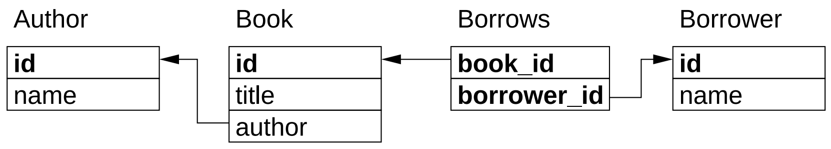


Figure 2.3: Schema for the example database. Attributes in bold are primary keys. Arrows represent foreign key relationships.

First, we create a schema graph, depicted in Figure 2.4 for the database. For each relation in the schema, we create a relation node. Figure 2.3 shows that the Book relation has a foreign key pointing to the Author relation. Thus we create an edge with weight 1 from the $book$ node to the $author$ node and an edge with weight 2 from the $author$ node to the $book$ node. We do the same for every other foreign key relationship, and the result is the schema graph in Figure 2.4.

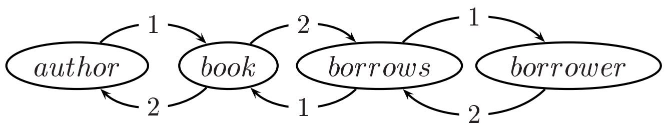


Figure 2.4: Schema graph for the example database.

We need to set $MaxPathLen$ to a reasonable value. To start with, let's set it to 3. Later we'll see the consequences of higher and lower values.

Now we're ready to perform queries on the database. Let's say that we wish to know what is know about Alice and Complete Works books. To find relationships between these in the database, we can issue the query $\{(STRING, alice), (STRING, complete), (STRING, works)\}$. Since the database is small enough that we don't need a global index, we can find keyword nodes manually. The only place we find the keyword 'alice' is in the Borrower relation in the tuple where the value

Author	
id	name
1	William Shakespeare
2	Oscar Wilde
3	Charles Dickens

Book		
id	title	author
1	Complete Works	1
2	Complete Works	2
3	The Picture of Dorian Gray	2
4	David Copperfield	3
5	The Tragedy of Hamlet	1

Borrows	
book_id	borrower_id
1	1
2	1
1	2
5	2
2	3
3	3
4	3

Borrower	
id	name
1	Alice
2	Bob
3	Charles

Table 2.1: Data for the example database.

of the primary key (id) is 1, so we create a keyword node $(borrower, 1)$ to represent this tuple. This keyword node stems from the $borrower$ relation node, so it gets the same edges as this relation node: an edge with weight 2 leading to $borrows$, and an edge with weight 1 leading from $borrows$. The keywords $(STRING, complete)$ and $(STRING, works)\}$ can be found in the same two tuples in the Book relation, so for each of the two tuples, we create a keyword node. The resulting query graph is depicted in Figure 2.5.

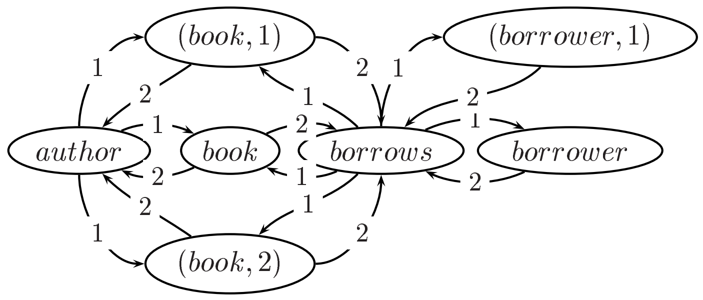


Figure 2.5: Query graph for the example query.

Now the CNG takes over. From each keyword node, it attempts to find paths to other keyword nodes whose collected edge weights are no greater than $MaxPathLen$. Again, we can start with the $(borrower, 1)$ keyword node. To begin with, the current path length will be 0, since we have not gone anywhere yet. From $(borrower, 1)$, we can only go to one other node, $borrows$. Since the weight of the edge leading to $borrows$ is 2, the current path length is set to 2. Also, we find that the node is not a keyword node, so we continue our search. From $borrows$, we can go to every adjacent node except the one we came from. Every edge leading from $borrows$ has a weight of 1, so we can still 'afford' to go anywhere. When we go to the $borrower$ and $book$ nodes, in each case we find that they are neither keyword nodes, nor can we go any further, because by going to either node, our current path length has been increased to $3 = MaxPathLen$, so we abandon the search along those paths. When from $borrows$ we go to $(book, 1)$ and $(book, 2)$, in both cases we find that we've come to a keyword node, so we record each path as a CN and abandon the search along these paths. So in our search from $(borrower, 1)$, we finish our search having found the following CNs:

- $(borrower, 1)\,, borrows, (book, 1)$
- $(borrower, 1)\,, borrows, (book, 2)$

We do the same for the two remaining keyword nodes, and in the end, we obtain the following CNs:

- $(borrower, 1)\,, borrows, (book, 1)$
- $(borrower, 1)\,, borrows, (book, 2)$
- $(book, 1)\,, borrows, (borrower, 1)$
- $(book, 1)\,, borrows, (book, 2)$
- $(book, 2)\,, author, (borrower, 1)$
- $(book, 2)\,, author, (book, 1)$

Some of these CNs are redundant. For instance $(book, 2)\,, author, (book1, 1)$ represents the same information as $(book, 1)\,, author, (book, 2)$; one CN is the other with the nodes in the reverse order. To reduce redundancy, the CN with the highest collected edge weight is removed. In this case, the two CNs have the same collected edge weight, so either may be removed. Ultimately, our list of CNs is reduced to:

- $(borrower, 1)\,, borrows, (book, 1)$
- $(borrower, 1)\,, borrows, (book, 2)$
- $(book, 1)\,, borrows, (book, 2)$
- $(book, 2)\,, author, (book, 1)$

These CNs are then passed to the CN evaluator. Here, each CN is translated to an SQL query, which is subsequently executed. The CN $(borrower, 1)\,, borrows, (book, 1)$ will be translated into the SQL query `SELECT * FROM Borrower AS R1, Borrows AS R2, Book AS R3 WHERE R1.id = R2.borrower_id AND R2.book_id = R3.id AND R1.id = 1 AND R3.id = 1`. It becomes necessary to rename relations when the same relation appears in the query more than once, such as would be the case when the two latter CNs are translated. The two former CNs yield each a JNT, the two latter yield none, so we get the JNTs:

- $(1, \text{Alice})\,, (1, 1)\,, (1, \text{Complete Works}, 1)$
- $(1, \text{Alice})\,, (2, 1)\,, (2, \text{Complete Works}, 2)$

These JNTs are then passed to the JNT postprocessor where another graph is generated. This takes the tuples represented from keyword nodes before, i.e. $(1, \text{Alice})$, $(1, \text{Complete Works}, 1)$ and $(2, \text{Complete Works}, 2)$ creates a node for each. Then it inserts an edge in this graph for every JNT. Since the JNT $(1, \text{Alice}), (1, 1), (1, \text{Complete Works}, 1)$ has the tuple $(1, \text{Alice})$ in one end and $(1, \text{Complete Works}, 1)$ in the other, this JNT creates an edge between the two. The resulting graph is depicted in Figure 2.6. Since this graph is connected, when the JNT postprocessor runs its breadth first search to combine JNTs, only one JNT will be the final result. If the search started in the node to the far left, the JNT would be $(1, \text{Complete Works}, 1), (1, 1), (1, \text{Alice}), (2, 1), (2, \text{Complete Works}, 2)$.

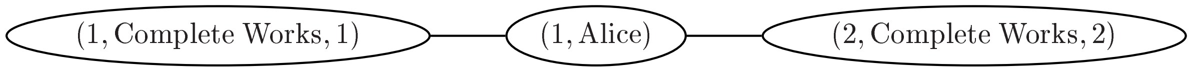


Figure 2.6: JNT graph for the example query.

Since there will be only one JNT in the result, ranking is not strictly necessary, but for the sake of completeness, let's compute it just the same. Assume the resulting JNT is the one given above. The keyword rank kwn depends on how many different keywords are found, and how many times the same keywords were found. The keyword $alice$ was found once, and the keywords $collected$ and $works$ were found twice each. Thus we can compute that $kwn = \frac{1}{1} + 2\left(\frac{1}{1} + \frac{1}{2}\right) = 4$. The edge weight rank will be the length of the path $(book, 1), borrows, (borrower, 1), borrows, (book, 2)$, i.e. $2 + 1 + 2 + 1 = 6$. That makes the rank of the JNT $6 - 4 = 2$.

Now that we have seen how ESKuSE processes a query from start to finish, let's consider a few thought experiments. Say we issue the query $\{(STRING, bob), (STRING, charles)\}$, expecting to find information about Bob and Charles' common reading interests. With $MaxPathLen$ set to 3 as it is, we will generate no CNs on this query. If we had specified the title of a book in our query, we would have been able to see if they had both borrowed the book, because the path length from $borrower$ to $book$, and vice versa, is 3. However, the CN that we expect to generate, namely $borrower, borrows, book, borrows, borrower$, has a path length of 6. This example database is so small that no relation can truly be said to be unrelated to any other relation so increasing $MaxPathLen$ to this large value may be justified. If, however, we consider the TPC-H schema shown in Figure 4.1, we see that if we set $MaxPathLen = 6$, we would be in a situation where the ORDERS relation would be considered directly related to the REGION relation, which is hardly reasonable to assume. If we look at Figure 2.4, we see that the real problem is that by our definition, borrowers are not considered very closely related to the books that they borrow. Conceptually, they are as closely related to the books as the authors of the books are, but this is not reflected in ESKuSE's model. Chapter 3 addresses this type of problems.

2.5 Speed optimisations

This section describes two optimisations that allow ESKuSE to resolve queries significantly faster than by using the basic ESKuSE algorithms as described above. For simple queries, or for small databases, the speed improvements are modest, but more difficult queries are resolved several orders of magnitude more efficiently.

The primary speed improvement comes from a simple means of mass-query optimisation. Normally, ESKuSE would attempt to evaluate each CN independently, i.e. translate it to its own SQL query and execute that query. This optimisation finds CNs that are similar and evaluates them with one query. Consider the CNs output by the CNG in the example given in Chapter 2.4. An SQL query for evaluating the topmost CN is given below the list of CNs. The reader will notice that the second CN from the top can be evaluated with an almost identical SQL query. In fact, we can evaluate both CNs with one query: `SELECT * FROM Borrower AS R1, Borrows AS R2, Book AS R3 WHERE R1.id = R2.borrower_id AND R2.book_id = R3.id AND R1.id IN (1) AND R3.id IN (1, 2)`.

Consider a CN $C_0 = a', b, c'$, where keyword nodes a' and c' stem from relation nodes a and c respectively. We then call the network a, b, c the *template* for C_0. If we have some $C_1 = a'', b, c''$, we can use the same template for this CN. Also, for all non-branching CNs, it makes no practical difference if we reverse the order of the nodes, so if we have a $C_2 = c', b, a'$, then $C_0 \equiv C_2$, and thus we can use the same template for C_2. The mass-query optimiser finds the minimum set of templates necessary to cover all CNs. As illustrated above, we only need to execute one SQL query per CN template.

Early experiments with the mass-query optimiser on a 10 MB TPC-H database showed that, for a wide range of queries with between 2 and 130 keyword nodes in the query graph, the average number of SQL queries was reduced from nearly 1400 to just 4, and that the average time to perform these queries was reduced from 185 seconds to 1 second. More recently, owing to a redesign of ESKuSE's index and the way it identifies tuples represented by keyword nodes, further speed improvements have been made, though similar comparative experiments have not been run.

After the above described mass-query optimisation technique has been applied, for queries that generate query graphs with hundreds of keyword nodes, the majority of the time it takes to execute a keyword query is spent in the CNG. Again, consider the CN templates described above. Because keyword nodes inherit the edges of the relation node that they stem from, the path lengths for a CN will be the same as the path lengths for its template.

As before, consider the CNs output by the CNG in the example in Chapter 2.4, where the two topmost CNs have the same template: $borrower, borrows, book$. If we have the template beforehand, we can generate these two CNs without having to go through the depth first search-like algorithm of the CNG described in Algorithm 2.1 for each individual keyword node. We can divide the template into three parts: the head, the body and the tail, where the head is the first node, the tail is the last node, and the body is everything between the two. The body may be empty. In our example, $borrower$ is the head, $borrows$ is the body and $books$ is the tail. As described in Algorithm 2.4, for each keyword node stemming from the head, we can create a CN with each keyword node stemming from the tail by connecting the two with the body of the template.

```
h: set of all keyword nodes stemming from the head
t: set of all keyword nodes stemming from the tail

for i in h:
    for j in t:
        make_cn(i, body, j)
```

Algorithm 2.4: Pseudo-code algorithm to generate CNs from CN templates.

What this optimisation does is that it generates all possible CN templates by using a slightly modified version of Algorithm 2.1. It can do this based on the schema graph, so there is even no need to make these computations for every query. All these possible CN templates are then stored in the relation node objects in the schema graph. Each relation node has a hash map of all CN templates that it is head of and whose path lengths do not exceed $MaxPathLen$. Generating CNs then becomes simple: for each relation node , consider all other relation nodes that is the tail in one of the templates in which the current relation node is the head, then apply Algorithm 2.4.

The exact complexity of the unmodified CNG based on Algorithm 2.3 has not been analysed, but since it behaves very similarly to a regular depth first search, a safe guess is that it runs in $\mathrm{O}(nb^m)$ [5], where n is the number of keyword nodes, b is the average branch factor in the query graph and m is the maximum search depth. This optimisation has complexity $\mathrm{O}(m^2n^2)$, where m is the number of relation nodes and n is the number of keyword nodes. Since m is constant for any one database, it may be more appropriate to give the complexity $\mathrm{O}(n^2)$. For all practical purposes, however, complexity becomes a moot point with the optimised version, since the time it takes to generate CNs becomes negligible compared to SQL query execution times for both small and large numbers of keyword nodes.

Chapter 3

Improving quality of search results

This chapter discusses the nature of high-quality search results and describes the means by which ESKuSE attempts to yield them.

3.1 Discussion of quality

In order to make a search engine that yields search results of high quality, it is first necessary to make clear what makes a search result good. Anyone who has used a search engine on the web, for instance, has an intuition of what a good search result is, which, simply put, boils down to 'that which best corresponds to what I am looking for.' By its nature, this is subjective and thus not easily quantified. One way to do this would be by popular vote, i.e. the best search engine is the one that most people agree is best, and by implication, it must be the one that often yields good search results.

It is not fair to compare two methods of searching independently of the means by which a query is specified. Examine, for instance, the difference between a keyword query and an SQL query: formulating an SQL query requires knowledge of the database schema, the insight (or foresight) to specify exactly what the query result should be and not least familiarity with SQL. By the above definition, an SQL query is guaranteed to yield very high quality search results. On the other hand, formulating a keyword query is trivial, but it becomes much more difficult to guarantee the quality of search results. The most important difference between SQL and keyword queries is their respective expressive power: while an SQL query can express many things about the exact form of the desired result, logical relationships etc., a keyword query can only express keywords to include or exclude. However, the fact that it is difficult to guarantee search results of high quality for keyword queries does not mean that it is not possible to generally get good results.

Attempts have been made to quantify the notion of good search results, but with little success. Ultimately, the judgement of what is good and what is bad in the methods described in this report and specifically in this chapter has defaulted to the author's subjective opinion, likely to be biased qua his paternal role.

3.2 Schema structural hints

Databases are typically modelled as having three levels: the conceptual, logical and physical levels. The conceptual level is fairly abstract, and represents the user's view of the data; the logical level is the one that we can query with SQL, where we deal with relations, tuples, etc.; and the physical level is where the details regarding storage on disk are handled. ESKuSE knows only about the logical level, but it will be able to yield higher quality search results if it can make educated guesses about how the conceptual level looks, since that would bring it closer to the user, so to speak. This section describes how such educated guesses are made and how they're reflected in ESKuSE's data model.

In [8] and [6], instructions are given for how to reduce Entity-Relation schema to relations. From those instructions it quickly becomes clear that for the way that many Entity-Relation constructs are mapped to relations, there is no easy or reliable way of reversing the process. Perhaps the simplest, and certainly very worthwhile, constructs to identify is one-to-many and many-to-many relationships. This report does not attempt to describe how other constructs can be taken advantage of.

In Definition 2.4 we see how a foreign key relationship between two relations result in a schema graph where the weight of the edge in the direction of the foreign key is lower than in the opposite direction, and as such, moving in the direction of the foreign key is considered an indicator of closer relationship, as described in Chapter 2.3, and it is also favoured in ranking.

The reasoning is the same as given in [2]: a foreign key can only reference a single tuple, and thus in that direction the relationship is unique. In the other direction, a primary key may have many references to it, and thus the relationship may not be unique. The authors reason that a rare relationship indicates a closer connection between data than a less rare relationship.

The consequence of Definition 2.4 and the above reasoning is that an edge that leads to a unique tuple (a one-to-one or many-to-one relationship) is attributed a weight of 1, while an edge that possibly leads to multiple tuples (one-to-many or many-to-many relationship) is attributed a weight of 2. Binary many-to-many relationships and ternary or greater one-to-many or many-to-many relationships are mapped to relations using a *relationship set relation*, i.e. a relation containing information about the relationship set. Recall the discussion at the end of Chapter 2.4. When thinking on the conceptual level, we are more interested in relationships between data than in actual relations in the database. Since the schema graph represents the way we think about relationships in the database, we will want it to reflect our conceptual model of the database. We do this by locating relation nodes that represent relationship set relations and altering the edges to and from these nodes to adapt them to our conceptual model.

A simple rule is used to identify relationship set relations. A relation is identified as a relationship set relation if all participants of its primary key are foreign keys. This strategy will fail if the creator of the database for some reason does not choose a primary key for the relationship set relation. Such a reason could be there being no practical benefit of indexing a particular relationship set relation. Since ESKuSE's internal representation of the database schema can differ from the actual schema, it is still possible to tell ESKuSE what the primary key is, even when it's not reflected in the actual schema.

When a relationship set relation has been identified, the edges leading to and from the node that represents the relation in the schema graph will have their weights altered to reflect their cardinality.

Let $G = (N, E)$ be a basic schema graph for our schema. Let $n_c \in N$ be a node that represents a relation that we have identified as a relationship set relation. We alter all edges that lead to n_c to have weight 0.5, i.e. $\forall n_i \, (n_i, n_c, w) \in E \Rightarrow w = 0.5$, as shown in Figure 3.1. This makes it always cheap to go to n_c; when going away from n_c again, we can determine if it's a 'to-one' or 'to-many' relationship and set the weight of outgoing nodes accordingly.

Conceptually, determining the cardinality of a relationship represented by a relationship set relation is simple. Let three relations r_0, r_1, r_2 be connected by relationship set relation r_c, where r_c has attributes k_0, k_1, k_2 that are foreign keys pointing to r_0, r_1 and r_2 respectively. Consider the following from r_0's point of view. Say that attribute k_1 is not unique and that k_2 is unique. This means that r_0 has a to-many relationship with r_1 and a to-one relationship with r_2. This is illustrated in Figure 3.2(a).

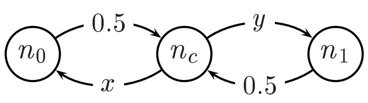


Figure 3.1: Edges leading to a relation node that represents a relationship set relation all receive the weight 0.5.

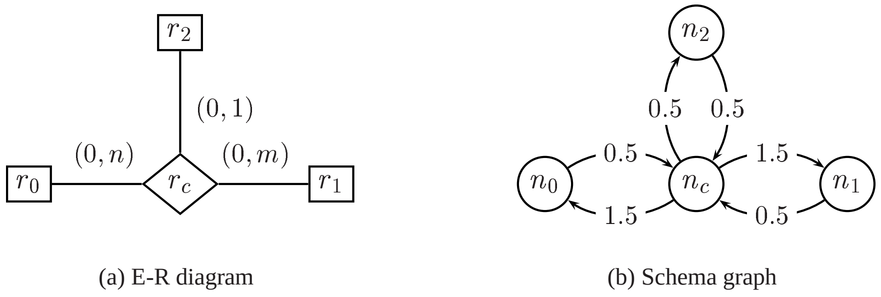


Figure 3.2: The relationship c between r_0, r_1 and r_2 in the E-R diagram to the left is mediated by the relationship set relation represented by n_c to the right. The E-R model to the left is then translated to the schema graph to the right.

We create a schema graph to represent these relations, which is depicted in Figure 3.2(b). As mentioned before, all edges leading to n_c will have weight 0.5. Above it was explained that to-one relationships get an edge weight of 1, and to-many relationships get an edge weight of 2. To get to n_c will cost 0.5, so the remainder is left to the outgoing edge weight, so we get edges $(n_c, n_1, 1.5)$ and $(n_c, n_2, 0.5)$. To put it formally, if $r_i \to n_i$ and $k_i \in r_c$ is an attribute referencing r_i, then $\forall n_i\, (n_c, n_i, w) \in E \wedge (k_i \text{ is unique}) \Rightarrow w = 0.5$, and $\forall n_i\, (n_c, n_i, w) \in E \wedge (k_i \text{ is not unique}) \Rightarrow w = 1.5$.

So long as we know which attributes in r_c are unique and which are not, all is well. However, we do not always know this. There are two ways to be certain that an attribute is unique:

- it is the only participant in the primary key, or
- it was created with the SQL `UNIQUE` constraint.

We can also try to determine experimentally whether an attribute is unique or not. If the SQL queries `SELECT COUNT(<attribute>) FROM <relation>` and `SELECT COUNT(DISTINCT <attribute>) FROM <relation>` return different values, then we can be sure that the attribute is not unique. If they return the same value, then right now, the attribute is unique, but we don't know if that is by coincidence or by design. In conclusion, the most reliable solution is for a knowledgeable human to indicate uniqueness where positive automatic identification cannot be made. ESKuSE does not attempt to experimentally determine uniqueness.

Having presented this method for automatically tuning the schema graph, it should be noted that it is, of course, entirely possible to manually adjust the schema graph if one so desires. This way, one can suggest relatedness of data exactly as one wishes.

This section has presented a very black and white view on relationship cardinality; *either* it's a to-one *or* a to-many relationship, and consequently the combined edge weights in crossing a node representing a relationship set relation are *either* 1 *or* 2. There is room for more nuances. For instance, by the reasoning given at the start of this section, there are plenty of reasons why a one-to-two relationship should be treated differently than a one-to-five hundred relationship. This path has not yet been explored, but is left for future work.

3.3 Database meta-information in ranking

The BANKS system, described in [2], employs a notion that the authors call *prestige* in ranking search results. The principle is the same as that of Google's [3] PageRank algorithm. This section describes how BANKS' ranking strategy can be adapted for use in ESKuSE, although this has not yet been done.

First, let's clarify what prestige means. Prestige is determined by the number of foreign keys that point to a tuple; the more foreign key references, the higher prestige the tuple has. A high prestige implies that the prestigious tuple is more important than a less prestigious tuple, and therefore it is statistically more likely to be relevant to any given query.

For ESKuSE to employ such a technique, its ranking system must be expanded to include a prestige rank, which would be very similar to the current keyword rank, since it binds itself to a particular tuple.

The primary challenge in adapting prestige-based ranking is that we do not know how many references a particular tuple has. The most precise way to determine this is, of course, by counting them. Say that relation r1 has an attribute fk that references relation r2's primary key pk. If we want to know how many references the tuple in r2 where fk = 1 has, we can issue the query `SELECT COUNT(*) FROM r1, r2 WHERE r1.fk = r2.pk AND r2.pk = 1`. If we had to do this for every keyword node in a query graph, the number of queries itself rather than their individual complexity could quickly become quite expensive. If we wanted to check the number of references for every tuple in r2 with a pk value of 1-n, we can fortunately do it with just one query, `SELECT COUNT(*), r2.pk FROM r1, r2 WHERE r1.fk = r2.pk AND r2.pk IN (1,2,...,`n`) GROUP BY r2.pk`. The valid values of pk are given as a set rather than as an integer range ($1 \leq \mathrm{pk} \leq n$) because pk will not likely always be integers, let alone a contiguous range of them.

In the worst case, one such query must be issued for every foreign key relationship. They are likely to involve full-table scans and therefore be relatively expensive, since foreign keys are rarely indexed outside of relationship set relations as described in Chapter 3.2. A choice will be have to made whether the extra ranking information will be worth the necessary performance hit.

A much faster, but also less precise, method of determining prestige would be to use the statistical information that most DBMS' gather and make available in special system relations. If r1 references r2 and we know that r1 has 500 tuples and r2 has 100, then on average, every tuple in r2 will have 5 references from r1. The danger is, of course, that these statistics can be greatly skewed.

3.4 Increasing expressive power

As discussed in Chapter 1.2, most search engines on the web support keyword queries with the $+$ and $-$ operators, that is, mandatory inclusion and exclusion of keywords respectively. While they have not yet been implemented for ESKuSE, doing so would not be difficult. The $+$ and $-$ operators have a very predictable effect on queries: they define demands that *must* be met for a search result to be accepted to the exclusion of results that do not meet these demands. This aids the user in writing queries that will return results of high quality.

The added expressive power cannot not be compensated for by a better ranking system: while it would be possible to give bottom rank to any result that would not contain all keywords, there would be no way of expressing the desire to exclude a word from search results. The part of the keyword query that is not prefixed with a $+/-$, however, would be treated like a normal query, and for that part of the query, normal ranking still applies.

For these reasons, it is clear that the $+$ and $-$ operators are very useful. Since they are a very new addition to ESKuSE's capabilities, their use has not been thoroughly tested.

Mandatory inclusion of a keyword is implemented in the JNT postprocessor. Every JNT is examined, and JNTs that do no contain the keyword to be included are deleted.

Mandatory exclusion of a keyword is handled primarily in the query grapher. Initially, the excluded keyword is treated like any other keyword when building the query graph. When the query graph has been built, any keyword node representing the keyword to be excluded is deleted from the graph. This method is not bulletproof, though; it only guarantees that the tuples represented by the keyword nodes in the query graph. Say that tuples t_0, t_1, t_2 form a JNT, and that t_0 and t_2 were represented by keyword nodes. There's no easy way of checking that t_1 does not contain a keyword that we will want to exclude from the query without examining the tuple for that keyword, so this must be done.

Chapter 4

Testing

This chapter describes the empirical results that have been obtained with the ESKuSE system and how they were obtained. A series of performance tests are presented that measure ESKuSE's raw speed, as well as quality tests, which are anecdotal.

4.1 Test environment

All tests have been run from a common desktop PC, an Athlon XP 1800+ with 512 MB SDRAM. The operating system is Debian GNU/Linux running kernel version 2.4.20. The RDBMS used is PostgreSQL v. 7.3.2, and Python v. 2.2.2 was used to execute the ESKuSE code. The Python module pyPgSQL has been used to interface with PostgreSQL. No tuning particular to these tests has been done to the used software.

In all speed tests, a 10 MB TPC-H database was used. Due mostly to the randomness of its data, querying a TPC-H database can be considered a torture test for keyword search engines. It is fair to assume that performance tests run against such a database can be considered a sort of worst case tests. The schema for the TPC-H database is given in Figure 4.1.

To measure the quality of results, TPC-H databases were found to be inadequate. It is very difficult for a human to determine if a search result is good or bad if it consists of only randomly generated data. For the purpose of testing quality, a document database was created containing 50 papers taken from the CiteSeer (`http://citeseer.nj.nec.com/cs`) for the documents themselves, and the rest of the data set is fictional. The schema for this database is shown in Figure 4.2. The comparatively small number of tuples in this database makes it poor for speed measurements.

4.2 Performance tests

Compared to the performance results presented previously in [4], ESKuSE has seen significant performance improvements, the most important of which are described in Chapter 2.5.

A series of tests have been run on the TPC-H database. A list of arbitrary keywords was taken from the index, all of which appear at most in 50 different tuples, on average appearing in just above 10 different tuples. A test run executes queries of 2, 5, 10 and 25 different keywords, picked at random from the list, and repeats this 10 times. Such test runs were made where $MaxPathLen$ was set to values 2, 3, 4 and 5.

Currently, processing time is spent mostly in two places, both of which belong to the CN evaluator. One is in pyPgSQL, the database interface module that ESKuSE uses. Data is received from

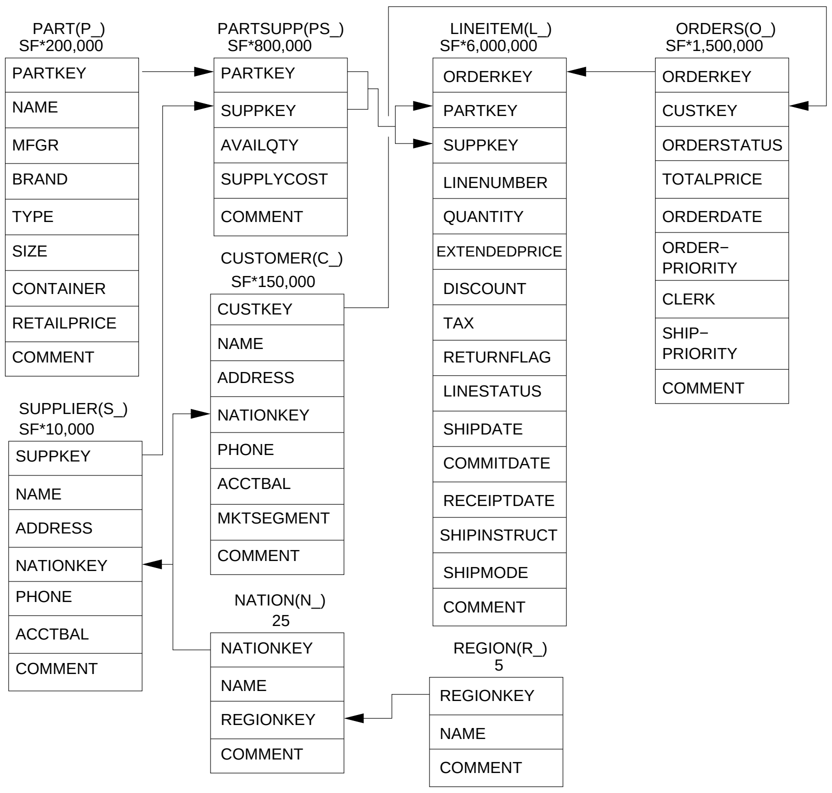


Figure 4.1: The TPC-H schema. Figure taken from [9]. Opposite to convention in this report, arrows point from the referenced attribute to the referencing attribute.

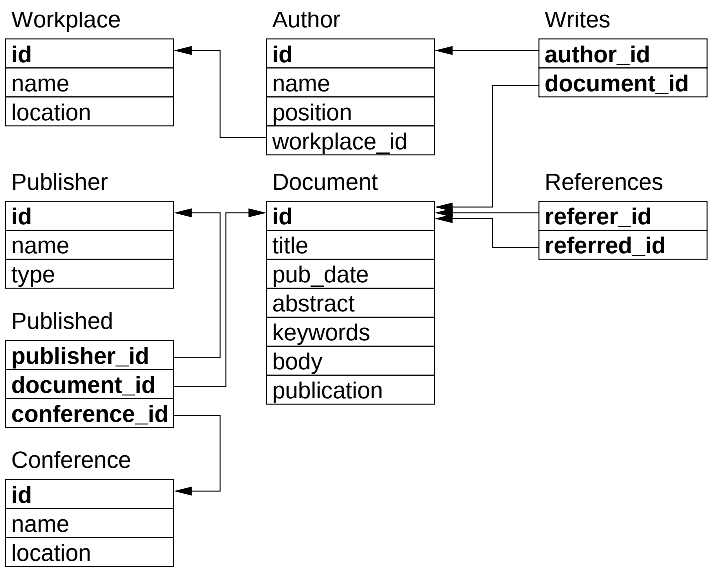


Figure 4.2: Schema for the article database. Unlike the TPC-H schema, but according to the convetion of this report, arrows point from referencing attributes to the referenced attribute. Attributes in bold are primary keys.

PostgreSQL as strings, and considerable resources are used to convert these strings to different native Python types (integers, floats etc.). This will not be so much of an issue in a better suited database using queries that are not random, since the number of generated JNTs will then typically be smaller.

The second and primary place that processing time is spent is in the DBMS, executing SQL queries.

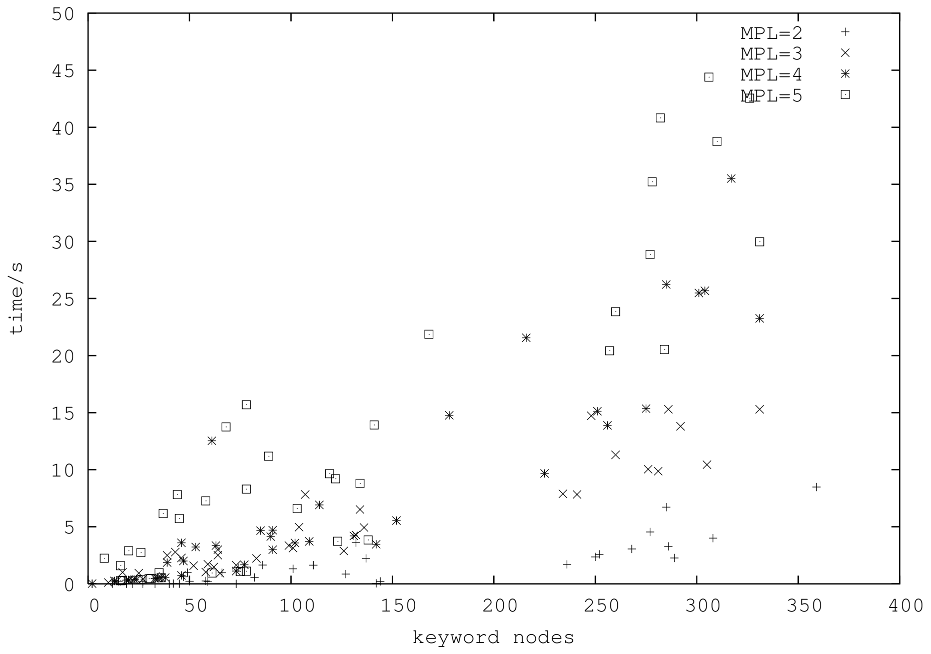


Figure 4.3: Performance tests measuring query execution time using different values of $MaxPathLen$.

Figure 4.3 shows the time ESKuSE takes to execute queries for different values of $MaxPathLen$, and Figure 4.4 shows the resulting number of SQL queries that must be executed per keyword query. This illustrates the discussion of this constant's impact on performance. As $MaxPathLen$ grows, so does both the number and the complexity of the SQL queries that must be executed. In this case, complexity refers to the number of joins in a single query.

One would expect queries to take less time for $MaxPathLen = 2$ for two connected reasons. Firstly, by using indices on primary keys, the number of necessary full-table scans should be kept down, and secondly, the largest relation has no foreign key references to it, making full-table scans of it unnecessary.

The test results show that even with hundreds of keyword nodes and a $MaxPathLen$ that is impractically large, all queries have been executed in less than one minute. From the performance results presented in [7] we can see that DISCOVER runs into a performance barrier when it searches for more than a certain number of keywords and allows its Candidate Networks to grow beyond a certain size. For ESKuSE, the corresponding values are significantly higher before query execution takes impractically long. Also, the ESKuSE system itself runs in only a few MB of memory, a large amount of which is used by the Python interpreter itself.

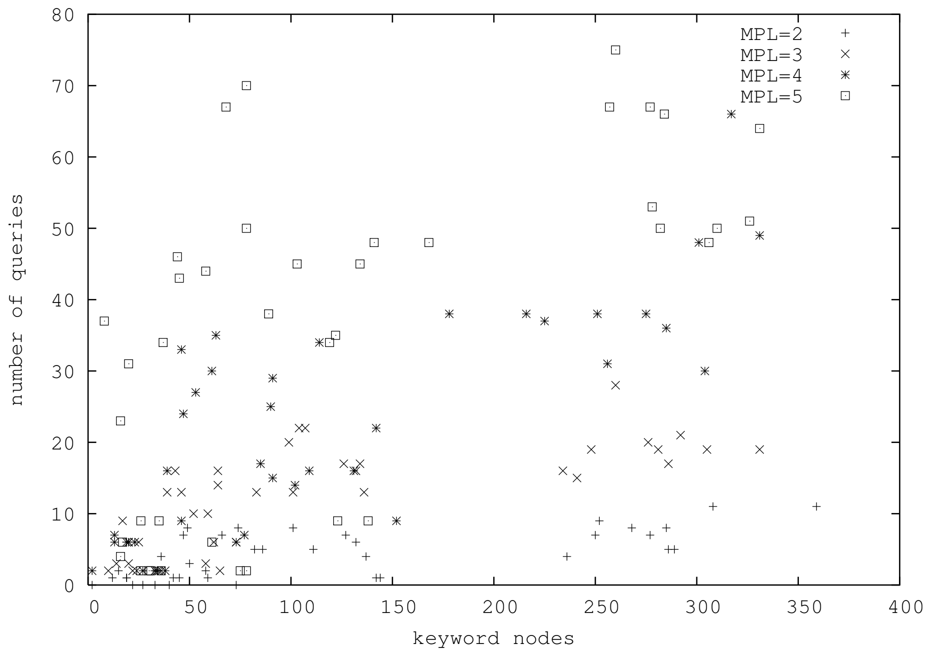


Figure 4.4: Performance test measuring the number of SQL queries necessary to execute in the CN evaluator for different values of $MaxPathLen$.

4.3 Quality tests

Attempts have been made to measure the quality improvements of the methods proposed in Chapter 3 as well as the impact of altering $MaxPathLen$ in objective ways, but they have been unsuccessful. Thus the only evidence is anecdotal by necessity, relying on the author's judgement.

As mentioned in Chapter 4.1, quality tests have been run upon a document database created specifically for the purpose. It was created as a typical example of a database for which keyword querying would be a good way of accessing its data.

In this database, a high value of $MaxPathLen$ has a much smaller negative impact than for the TPC-H database. The names of people, companies, places and document content are fairly cleanly separated, whereas in the TPC-H database, keywords are more or less just as likely to be found in one relation as in another. Since the relationships between the different relations are quite simple, we get very few 'unexpected' results, even for large values of $MaxPathLen$. $MaxPathLen = 4$ was found to be a good value, since it allows authors to be connected directly to publishers of their papers, the conferences they present their papers on as well as the other papers that either they cite or are cited by. If the schema had been more complex, this might have been too large a value.

It was found that the effect of applying the method for manipulating nodes representing relationship set relations described in Chapter 3.2 to this schema was not noticeably different from merely increasing $MaxPathLen$ to 6. The reason for this becomes clear when we notice that all relations with 'interesting' data except the Workplace relation are connected by relationship set relations. In a schema where fewer, but still some, relationships are mediated through relationship set relations, the effect of applying this method is bound to be more pronounced, and an increased $MaxPathLen$ may be less appealing than for this test database.

The added expressive power of mandatory keyword inclusion and exclusion, particularly exclusion, shows its value when searching for many different keywords. Incidentally, this is consistent with the author's own experiences when using Internet search engines. Such uses are typically searching for related papers, common keywords and such, i.e. mostly where the Document relation is concerned. In the same kind of searches, the prestige-based addition to the current ranking system presented in Chapter 3.3 would likely be a boon.

The only annoyance, but this is a big annoyance, is that the JNT postprocessor always generates as large JNTs as it can. Sometimes, JNTs of monstrous proportions will be returned where many smaller JNTs would have been preferable. These large JNTs are almost invariably focused on the Document relation. The author feels that once a replacement to the current JNT postprocessor is in place, ESKuSE can be relied upon to yield generally good query results, though there is yet room for improvement in the ranking algorithm.

Chapter 5

Conclusion

This chapter concludes upon the work presented in this report and gives the author's thoughts on future work.

5.1 Project conclusion

This report has presented the ESKuSE system, a continuation and improvement to the work begun in [4], a both time and memory efficient keyword search engine.

ESKuSE models database schema and executes queries using graphs. Methods for greatly improving query execution speed by taking advantage of these graph structures were presented, as well as several methods for using and manipulating these graphs to rank and improve the quality of search results.

Experimental results have shown that the current JNT postprocessor is often too aggressive in creating large JNTs. A replacement is under development to target this problem.

5.2 Future and ongoing work

To target mainly the annoyance of the overly large JNTs sometimes generated by the JNT postprocessor, a replacement algorithm is under development. Like its predecessor, it starts from a JNT graph, where it attempts to grow composite JNTs from the smaller JNTs it receives from the CN evaluator. Unlike its predecessor, it includes a hill-climbing algorithm where at each expansive step it will consider whether including another JNT into the composite JNT will increase or decrease its rank.

An alternative or supplement to this method would be to attempt to adopt the BANKS system's *backwards expanding search algorithm*. This maybe be possible without too much redesign of ESKuSE, since the JNT graphs created in the JNT postprocessor are similar to the graph BANKS uses to represent a database.

For ESKuSE to be adapted for practical use, it will need some form of frontend. Providing an interface through which to accept input is simple, but query result visualisation is a project unto itself. Fortunately, there is existing research to draw upon in this domain.

There are also plenty of tuning challenges left, such as making improved ranking algorithms based on the current functionality, trying to guess more of the conceptual database model from the database schema, increasing sensitivity to relationship cardinality and so on. While none of these by themselves would advance ESKuSE by leaps and bounds, they would help to create an overall better system.

5.3 Acknowledgements

The author would like to thank the people who have contributed directly or indirectly to the creation of this report and the work it represents. First and foremost, supervisor Heidi Gregersen deserves much gratitude. Had she not saved the author from himself on a nearly weekly basis, as well as provided always good advice and criticism, this work never would have been. Many thanks also go out to the people behind the DISCOVER system, which initially provided a source of inspiration, as is evident in ESKuSE's data model, and the authors of BANKS, from which many ideas have more recently been adapted. Furthermore, the author thanks all the people who have at odd moments acted as sounding boards for his ideas and have provided a great many small epsilons toward the greater goal.

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] Gaurav Bhalotia, Charuta Nakhe, Arvind Hulgeri, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[4] Martin Christensen. Eskuse: a keyword query framework for relational databases. January 2003.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[6] Elmasri and Navathe. *Fundamentals of Database Systems*. Addison-Wesley, third edition, 2000.

[7] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Searching in Relational Databases. In *VLDB*, 2002.

[8] Silberschatz, Korth, and Sudarshan. *Database System Concepts*. McGraw-Hill, 4th edition, 2002.

[9] Transaction Processing Performance Council (TPC). *TPC Benchmark*$^{\text{TM}}$ *H*, version 1.5 edition.