KEYWORD SEARCH IN THE GLYCOMICS PORTAL

by

KI TAE MYOUNG

(Under the Direction of John A. Miller)

ABSTRACT

In this thesis, I have developed keyword search capabilities for the *GlycomicsPortal*, where a user can simply type keywords to access information. The search includes parsing of the user keywords, searching the parsed user keywords in Lucene, retrieving information from Lucene search results and the database, finding common portal objects from the retrieved information, and ranking the retrieved information. On the result page, users can see reasons for all results. Each reason is displayed indicating how the search rank algorithm works, so that the users can understand why the result was found and how the rank score was calculated. We also have the advanced search where a user can specify different options such as a phrase search, proximity search, or field restriction. In the advanced search, a query is generated based on the user's keywords and specified options. Here, the user can restrict to fields or specify Boolean operators (AND, OR, and NOT), or choose a phrase or proximity search. Since results are hyperlinks, the user can navigate through the list of ranked results to examine the details of each.

INDEX WORDS: GlycomicsPortal, Full text search in relational databases, Keyword search in relational databases

KEYWORD SEARCH IN GLYCOMICS PORTAL

by

KI TAE MYOUNG

B.A. University of Mississippi, 2009

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2012

KEYWORD SEARCH IN THE GLYCOMICS PORTAL

by

KI TAE MYOUNG

Approved:

Major Professor:	John A. Miller Ismailcem Budak Arpinar
Committee:	Krzysztof J. Kochut William York

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2012

DEDICATION

Our Father in heaven,

hallowed be Your name,

Your kingdom come, Your will be done on earth, as it is in heaven.

Give us today our daily bread.

Forgive us our debts, as we also have forgiven our debtors,

and lead us not into temptation, but deliver us from the evil one.

For Yours is the Kingdom, and the power, and the glory, forever, Amen.

ACKNOWLEDGEMENTS

Thanks to Dr. Miller for his time and devotion on my work. Without his advise, I would not be finishing my thesis. I specially thank to Rene Ranzinger for being my teacher and his time. Thanks to Dr. Kochut for his keen advise at every Friday meeting. Thanks to Dr. York for his support and advise. Last, thanks to Dr. Arpinar who always encourages me. My work would not be finished without support and help from these professors and Rene.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 Overview of the work

Over the last two decades, a large number of GlycoInformatics tools and systems have been developed [1]. They include databases, Web applications, Web services and workflows. Even though most tools and systems are accessible to users, the users are not aware of many of them. It is hard for a user to know about all of the existing tools and systems, or even the ones that the user is interested in. Also, searching for tools can be tedious and time-consuming. Most tools are available on Websites, each of which has a Web address. Typically, users will find hyperlinks that do not provide details about how to use the tools, so they will need to search for additional documentation about the tools. Unfortunately, such information is often not up-to-date, or the links are not working properly. Therefore, there is a need to address these shortcomings, in particular, for scientists working in the Glycomics domain, as there currently is little support in this domain.

To address the needs of scientists working in Glycomics, the GlycomicsPortal has been created. The *GlycomicsPortal (http://glycomics.ccrc.uga.edu/GlycomicsPortal/)* is a Web-based portal where researchers can register and find tools for GlycoInformatics. The portal contains descriptions, keywords and user comments about more than 60 GlycoInformatics tools and databases.

PostgreSQL, Hibernate, and Webwork are used for a database management system, Object Relational Mapping tool, and Web application framework, respectively in the portal. Information is stored in our portal database and is periodically updated. As the amount of information increases, it becomes difficult to search for relevant information.

In the past, to search in a relational database, a user would need to know the schema of the database and the query language. Currently, database management systems provide alternative solutions, such as *keyword search*. A user can simply type keywords to access information and get ranked results. There are two common approaches for searching in a relational database: full text search provided by the databases and third-party database search tools.

The full text search (keyword search) is implemented by relational database management systems such as IBM DB2, Microsoft SQL Server, Oracle, PostgreSQL, and MySQL. According to the PostgreSQL Website, "an index cannot span multiple tables" [5]. In other words, an index cannot be created on joins of tables. Thus, in order to search in multiple tables, an index needs to be created on each table. After the indexing, we query on each index of a table. However, it is not enough to capture relationships between the tables. To capture the relationships results from all the queries need to be joined. However, the results show only linked tuples that matched every table that was queried. In order to retrieve results that are partially matching the user search words, the OR operator and the WHERE clause are used with tables as shown in Appendix A. Unfortunately, this becomes too slow as more tables are added.

Several third-party database search tools exist such as Hibernate Search [8], Solr [9], and Compass [10], which are built on Lucene [12]. Results of full text search (e.g., full text search

query is ‘cancer lectin’) in a database are often a set of tuples, each of which is a list of information stored in the database. Also, these results can be related to each other in the database through foreign key references. Unfortunately, Lucene does not support the JOIN operator, which enables joining results to find relationships among the results. In other words, none of the third-party database search tools mentioned above are able to capture the relationships in the results. Thus, we query on each table and then join the results from the queries. Unfortunately, after the join, no ranking on the results is available.

As commonly available search mechanisms and tools are not sufficient for our particular search problem, solutions from research literature are sought. One research prototype called BANKS (Browsing ANd Keyword Searching) [3] allows searching over multiple tables utilizing relationships to provide search results. BANKS uses a directed graph for the database where each node and edge represents a tuple and foreign key to primary key relationship, respectively. First, tuples matching a user’s search keywords are retrieved from a database. Next, BANKS finds a common tuple of the results. Finding the common tuple means finding connections between tuples in the search results. Unfortunately, BANKS is not available to the public.

Thus, we used the idea of finding a common tuple connecting the results used in BANKS and created the portal-object centered database structure where a main object is connected to all other objects in the database. Using this database structure, we can find a main object as a common object based on the individual search results. First, Lucene documents (including table name and “row id”, e.g., example.hibernateClass.Keyword and row41) matching the user’s search keywords are retrieved from Lucene indexes. From the table names and row ids, we

access the corresponding tuples in the database. We then find the main object by following the relationships in the retrieved tuples.

In this thesis, I have developed keyword search capabilities for the *GlycomicsPortal*, where a user can simply type keywords to access information. The search includes parsing of the user keywords, searching the parsed user keywords in Lucene, retrieving information from Lucene search results and the database, finding common portal objects from the retrieved information, and ranking the retrieved information. On the result page, users can see rank reasons for all results. Each rank reason is displayed indicating how the search rank algorithm works, so that the users can understand why the result was found and how the rank score was calculated. We also have the advanced search where a user can specify different options such as a phrase search, proximity search, or field restriction. In the advanced search, a query is generated based on the user's keywords and specified options. Here, the user can restrict to fields or specify Boolean operators (AND, OR, and NOT), or choose a phrase or proximity search. Since results are hyperlinks, the user can navigate through the list of ranked results to examine the details of each.

The rest of the thesis is organized as follows: Chapter 2 presents background information and a brief overview of the recent approaches in the area of keyword search on databases. Chapter 3 gives a brief overview of the software we used. Chapter 4 describes the implementation of the search algorithm. Chapter 5 presents the evaluation of the results. Chapter 6 contains the conclusions of the work and suggests future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Introduction

Keyword search provides a simple way to access information in a database for a user by typing keywords. Full text search for relational databases provides keyword search and rank capabilities. It has been implemented on relational database management systems such as IBM DB2, Microsoft SQL Server, Oracle, PostgreSQL, and MySQL. Unfortunately, there are some limitations within those systems. In order to use full text search functions fully, a user needs to know the database schema and the query language. If the user's keywords are found in several database tables, then the user has to figure out which tables contain the search terms and how to join the tables to retrieve a result. Most Web search engines, for instance, Google or Bing, support keyword search. Keyword search is the most popular search method because of its simplicity, and users do not need to know additional information such as query languages or database schemas. Unfortunately, keyword search cannot be applied directly to the database. Information needed to answer the user query is often found across multiple tables because of data normalization. To overcome this problem, many advanced full text search systems for relational databases have been developed. In the following sections, I introduce a brief overview of advanced full text search for relational databases. Then, I explain limitations in query languages for relational databases, full text search in free relational database management

systems and Lucene-based search tools. Then, I will explain the current advanced full text search system in BANKS.

2.2 Advanced full text search

Traditional approaches for full text search focused on efficient ways of retrieving documents, but recent approaches (advanced full text search) focus on relationships of tuples in a relational database [1,2,3,4,13,14]. Finding connections between tuples reveals relationships of these tuples. Generally, two techniques are used to find relationships. Keywords are searched on a *schema graph* where a relational database schema is considered as a directed graph or a *data graph* where parts of relational database containing a user's keywords are mapped into a graph [11]. I then show what is available in relational databases.

2.2 Limitations in query languages for relational databases

SQL query is a common query language for searching in a relational database. Converting a user's keywords into an SQL query enables one to capture relationships between search results. However, after executing an SQL query, there is no rank score available. This is a limitation in query languages for relational databases. I then show a limitation of full text search in current relational database management systems in the following section.

2.3 Full text search in relational database management systems

Current database management systems enable full text search in a database. However, it is not easy for a user to search keywords in multiple tables. The example shown below is a full

text search query found at the MySQL Website [5]:

```
SELECT * FROM articles WHERE MATCH (title,body)
```

In order to search over multiple tables, the user has to know how to combine tables in the database and needs to know column names as well. According to the MySQL Website, "A full-text search that uses an index can name columns only from a single table in the MATCH() clause because an index cannot span multiple tables" [5]. This is a common issue with full text search systems in existing systems.

2.4 Lucene full text search

Lucene is a free Java search engine library that can be used to search in multiple documents, based on Lucene index. Lucene uses inverted indexes for a search, where each keyword maps to documents containing keyword. With inverted indexes, Lucene search is faster than the full text searches implemented in relational database management systems [4]. However, Lucene does not directly support searching databases. There are open source tools built using Lucene such as Solr, Hibernate search, and Compass, which support full text search in a database. However, the full text search in these systems do not support the JOIN operator. In other words, the full text search does not capture relationships between tuples matching the user's keywords.

2.5 Advanced full text search using a graph in BANKS

In BANKS [3], tuples containing keywords are retrieved from the database, and then an

index is created, which contains ids of the tuples and node ids, one of which represents a corresponding tuple. We assume that a directed graph as a model of the database is already created. A tree structure is created from the tuples containing the keywords. The index is used to map nodes in the tree structure to tuples. Another index is also created for mapping user keywords to ids of tuples where the user keywords are found.

BANKS uses the idea of finding a common root from tuples that contain user keywords. Finding the common root means finding connections between tuples that contain user keywords. BANKS uses a directed graph for the database where each node and edge represents a tuple and foreign key to primary key relationship, respectively.

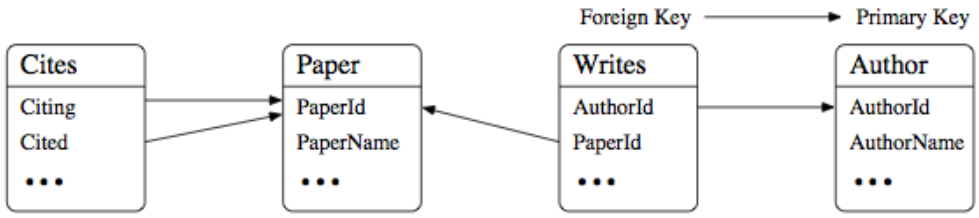


Figure 1: Example schema graph used in BANKS

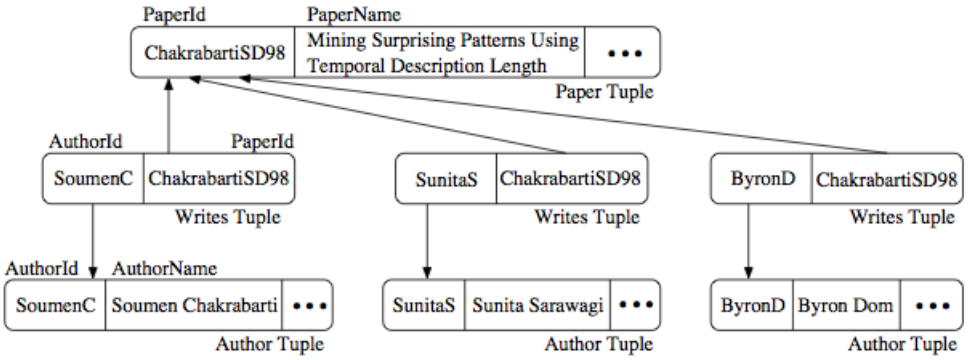


Figure 2: A fragment of the database picture in BANKS

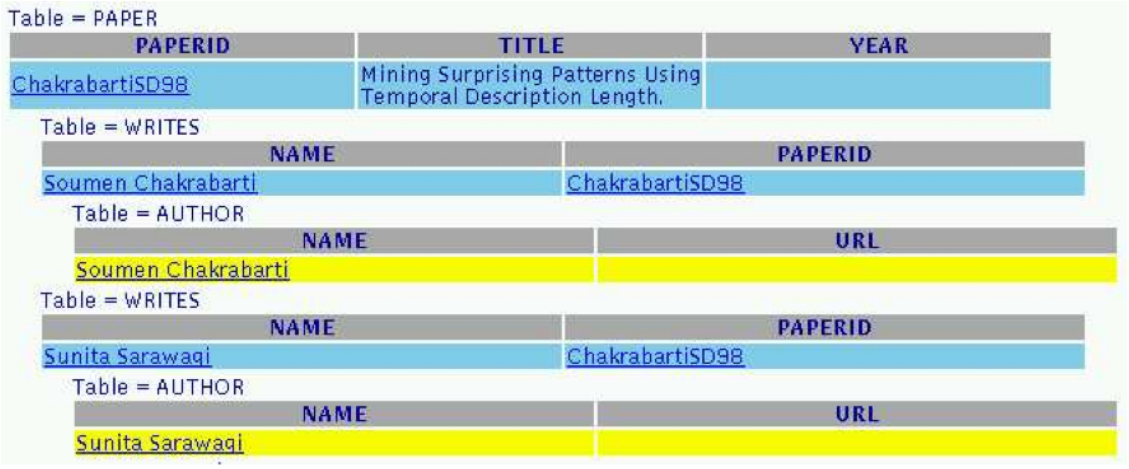


Figure 3: An example of a search result in BANKS

The above three figures describe the main algorithm in BANKS. Figure 1 shows an example of a part of the schema in BANKS. First, we explain the example in Figure 2. The Paper tuple has relationships with three Author tuples, each of which has an author name, Soumen, Sunita and Byron, respectively. In other words, the paper described in the Paper tuple is written by the three authors. Then, we explain how to find the Paper tuple for the user's keywords (Soumen, Sunita) in BANKS.

There are the following four major steps:

1) Searching the user's keywords in the database

2) Returning associated nodes for the retrieved tuples

3) Finding a common root node from the nodes

4) Showing the results

We then explain how each step works in detail:

1) Searching the user's keywords in the database

The user's keywords are searched in the database. Tuples matching the user's keywords are retrieved from the database.

2) Returning associated nodes for the retrieved tuples

The nodes corresponding to the tuples are retrieved. There are two Author tuples where the keywords are found, because we can find two Author nodes that contain keywords (nodes' ids are SoumenC and SunitaS, respectively) as shown in Figure 2.

3) Finding a common root node from the nodes

There is a relationship between an Author table and a Writes table. Two Writes nodes are discovered for the two Author nodes following the foreign key to primary key relationships. Then, the common root, the Paper tuple, is discovered. This is how to form a tree in BANKS, and this algorithm is called the *backward expanding search algorithm* [3]. The *backward expanding search algorithm* uses *Dijkstra's algorithm* to find the shortest path to the common root.

4) Showing the results

Figure 3 shows the search result for the user's keywords. The result is a tree, which keeps corresponding nodes to the joins of tuples containing the user's keywords following foreign key relationships.

BANKS uses a *schema graph* where a relational database schema is considered as a directed graph to find relationships of tuples in a database. In the following chapters, I will introduce full text search in the GlycomicsPortal where Lucene and PostgreSQL are used to find relationships between tuples in our database.

CHAPTER 3

ARCHITECTURE AND SOFTWARE COMPONENTS

This chapter shows the architecture of the GlycomicsPortal. Also, we give a brief overview of the software to develop the portal.

3.1 Architecture

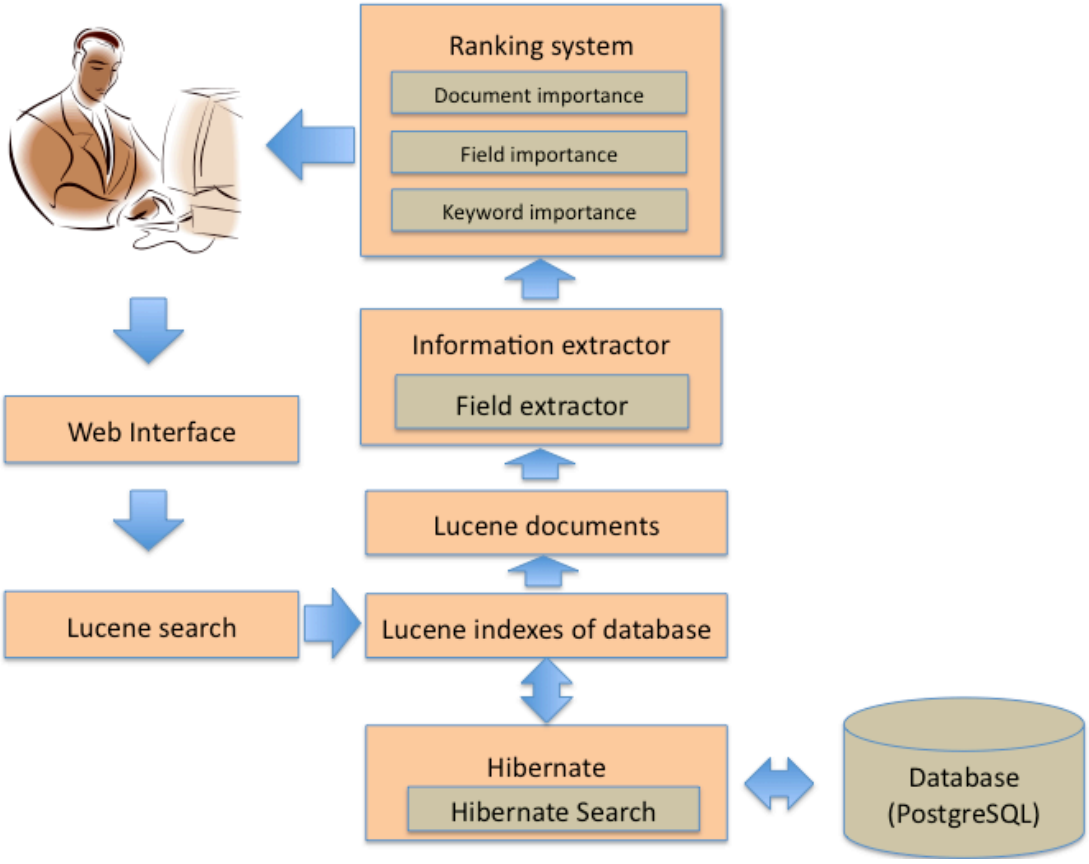


Figure 4: Architecture of *GlycomicsPortal* search

1) Lucene search

Lucene 3.0.3 is the core part of our search. It is a free and powerful Java search engine library. It supports different queries such as wildcard search and fuzzy search. A user's keywords are converted into a Lucene search string. Lucene search then uses this Lucene search string on one or more Lucene indexes and returns a set of Lucene documents. Lucene indexes are used to index various tables in the GlycomicsPortal database.

2) Hibernate Search

Hibernate Search 3.3.0.Final is built on Lucene and is a sub project of Hibernate. Hibernate is a free ORM (Object Relational Mapping) tool for mapping between Java objets and a relational database. Lucene indexes are built by the Hibernate Search tool, which also controls updating the index for changes in the database.

3) Information extractor

After retrieving the set of Lucene documents from Lucene search, we extract information of fields containing the user's keywords and portal objects that are entries in the main table PortalObject. More details are explained in chapter 4.

4) Ranking system

Using the extracted information, we calculate importance of the user's keywords, fields and portal documents as shown in Chapter 4.1.3. An importance aspect of searching for portal documents is the rank score for the user's keywords search.

CHAPTER 4

KEYWORD SEARCH

In this chapter, we introduce how our database structure helps our search, and explain the implementation of regular search and advanced search. Also, we show the algorithm used to rank the search results.

4.1 Regular Search

In this section, we introduce the database structure in the GlycomicsPortal and explain the implementation of regular search.

4.1.1 Database structure of *GlycomicsPortal*

Information is spread out across several tables in the database. In the database, all information is related to the main table entries, each of which is called a *PortalObject*. *PortalObjects* have four sub-types: Web service, workflow, software and database.

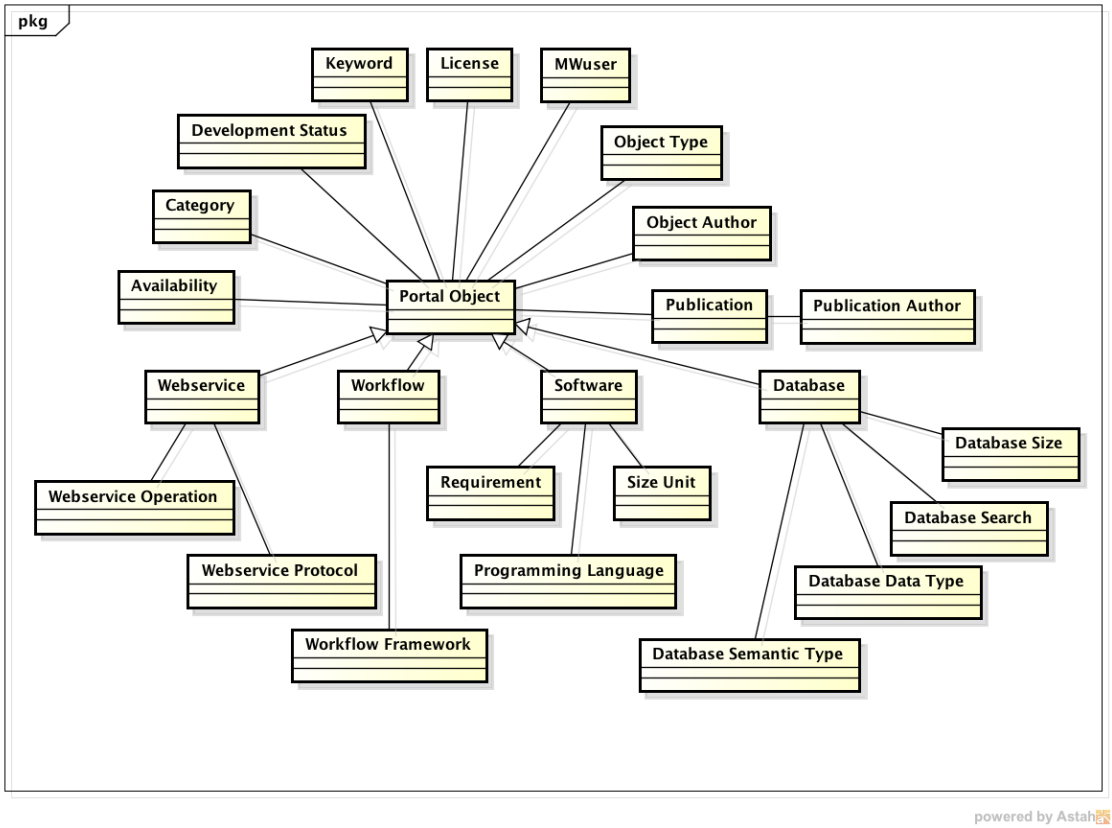


Figure 5: UML class diagram of searchable portion of the *GlycomicsPortal*

Figure 5 shows the UML class diagram of the searchable portion of the *GlycomicsPortal*. The main table, Portal Object, contains the common information about *PortalObjects*. Other tables contain related information for the *PortalObjects*. Except four tables that are sub-types of Portal Object, the other tables have 1:n or m-to-n relationships to the main table. Designing a portal-object centered database facilitates finding relationships between search results. Search results in the GlycomicsPortal are *PortalObject*s along with relevant information in related tables. We can create a PortalDocument by joining the tuples resulting from the Lucene searches following relationships indicated by foreign keys incident upon a common Portal Object.

4.1.2 Search Algorithm

The search algorithm has the following seven major steps (or 8 individual steps/activities as shown in Figure 6):

1) Processing the user's query (step 1)

2) Searching the parsed user's keywords using Lucene (step 2)

3) Retrieving information from Lucene search results (step 3)

4) Finding common *PortalObjects* from the retrieved information (steps 4 - 5)

5) Grouping the retrieved information by *PortalObjects* (step 6)

6) Ranking the retrieved information (step 7)

7) Displaying the results (step 8)

Figure 6 shows how the search algorithm works with an example. I then explain how each step works in detail.

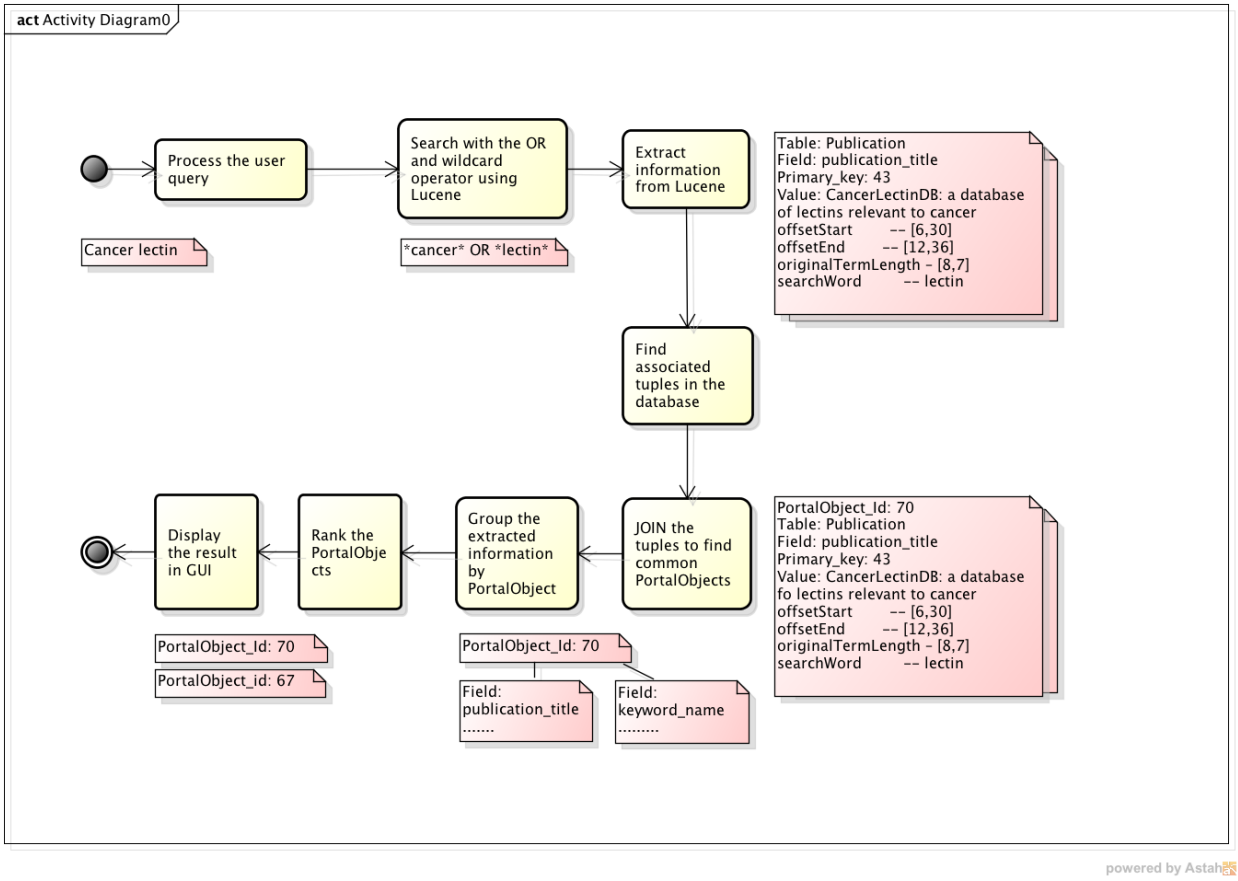


Figure 6: UML activity diagram of the search algorithm

1) Processing the user's query

For regular search, the user's query string is tokenized using `LuceneStandardAnalyzer`, which removes stop words and converts the keywords into lowercase. Then, the filtered keywords are saved into a `Set<String>` collection to remove duplicates.

For advanced search, each character in the user's query is checked using validation rules. While validating the user's query, a tree-based data structure is also built to represents the query.

2) Searching the filtered keywords in *Lucene*

The filtered keywords in the `Set<String>` collection are searched over Lucene indexes. Lucene search uses inverted indexes, which are automatically built and updated by HibernateSearch. First, the filtered keywords are searched over the Lucene indexes using the Lucene wildcard (*) and the Boolean OR operator. For instance, if the filtered keywords are 'A B C', then the Lucene search string becomes `"*A* OR *B* OR *C*"`. This string is passed to the parse method of Lucene's queryParser, which generates an expanded search string that includes all searchable fields in the database. In order to retrieve the results/hits (up to 100) based on the expanded search string, the search method of Lucene's indexSearcher is called as shown below:

```
indexSearcher.search(queryParser.parse("*A* OR *B* OR *C*"), 100)
```

Each hit corresponds to a tuple in the database that matched part of the expanded search string. Each hit contains information about the tuple, including the table name, primary key and all matching searchable fields with corresponding values for the tuple. Note, not all fields in the database are searchable, because some contain private information or information that is not useful for searching.

3) Retrieving information from Lucene search results

In this step, information about the searchable fields is retrieved from each hit. Although Lucene indicates whether a tuple matches the expanded search string, it does not indicate which of the searchable fields actually matched. Thus, I have created a field extractor that retrieves the fields that matched the expanded search string. If a field contains one of the filtered keywords,

then the field extractor retrieves the field's information and offsets of the filtered keywords. The offsets of the filtered keywords are retrieved from the `TermPositionVector` of `Lucene.index`, which also stores the frequencies of all terms in the searchable fields. All the retrieved information is stored in an object called `SearchResults` as shown in Figure 7.

```
public class SearchResults
{
    /** The primary key of the row where a user keyword is found
     */
    private Integer hbClassId = 0;
    /** The full name of the table where a user keyword is found
     */
    private Class<?> hbClassName = null;
    /** The shortened name of the field where a user keyword is found
     */
    private String fieldName = null;
    /** A user keyword that is found in the field
     */
    private String searchWord = "";
    /** List of lengths of the words matching the user keywords
     */
    private List<Integer> originalTermLength = new ArrayList<Integer>();
    /** List of start offsets matching the user keywords
     */
    private List<Integer> offsetStart = new ArrayList<Integer>();
    /** List of end offsets matching the user keywords
     */
    private List<Integer> offsetEnd = new ArrayList<Integer>();
    /** List of portalObject primary keys corresponding to tuples matching the keywords
     */
    private List<Integer> portalObjectIds = new ArrayList<Integer>();
    /** A string value of the field where a user keyword is found
     */
    private String actualFieldValue = "";
```

Figure 7: `SearchResults` Java class

4) Finding common *PortalObjects* from the retrieved information

After retrieving the information, we find associated tuples in the database using table names and primary keys. We then join each tuple with the main table to find a common

PortalObject using the Hibernate Query Language (HQL). An example of a HQL is shown below.

```
SELECT distinct O.id
FROM PortalObject AS O inner join O.objectToKeywords as OK
WHERE keyword_id = :id
```

This query retrieves the ids of PortalObjects from the join of Portal Object and its associations where the id for the keyword is given by the parameter :id (for example, in Figure 6 :id = 43).

5) Grouping the retrieved information by *PortalObjects*

After *PortalObjects* are found from the retrieved information, we group this information by *PortalObjects*, because we want to know which searchable fields stored in the database are related to which *PortalObjects.* During the grouping by *PortalObjects,* parts of the retrieved information such as offsets or search words (as shown in Figure 8) are combined. An example of combining the information is shown in Figure 8. After combining, we can see which *PortalObject* is related to which searchable fields and which user's keywords. Figure 9 shows an example of a PortalDocument that results from combining information in the common *PortalObject*'s group.

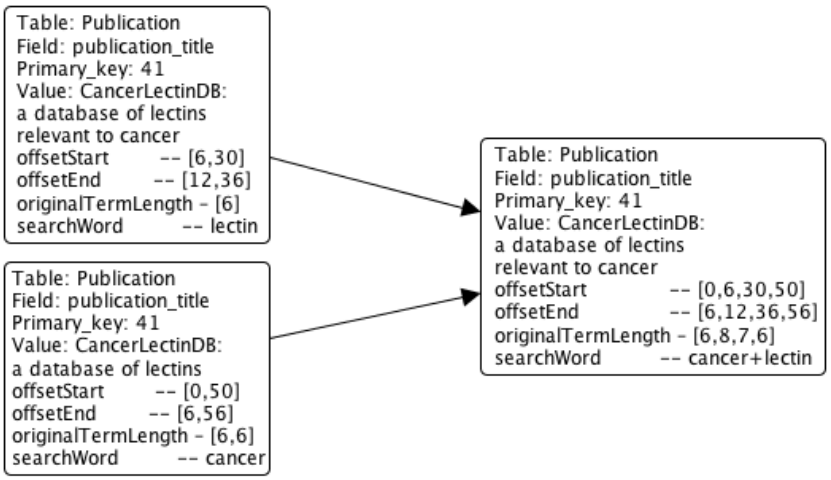


Figure 8: An example of combining searchable fields

```
Table: PortalObject
Primary_key: 70

Search Field: object_name
Value: CancerLectinDB
offsetStart          -- [0,6]
offsetEnd            -- [6,12]
originalTermLength - [6,8]
searchWord       -- cancer+lectin

Search Field: publication_title
Value: CancerLectinDB: a database of lectins relevant to cancer.
offsetStart          -- [0,6,30,50]
offsetEnd            -- [6,12,36,56]
originalTermLength - [6,8,7,6]
searchWord       -- cancer+lectin

Search Field: keyword_name
Value: Lectin
offsetStart          -- [0]
offsetEnd            -- [6]
originalTermLength - [6]
Search word: lectin
```

Figure 9: An example of a PortalDocument

6) Ranking the retrieved information

Based on the information in each PortalDocument, we rank the *PortalObject*s to indicate how well they match the keywords. The algorithm used to rank *PortalObject*s is a modified version of the original tf_idf algorithm, which is explained in the next section. In this step, we also show which fields are matched and on what keywords.

7) Displaying the results

For each ranked *PortalObject,* the name, description and rank score of the *PortalObject* are shown in the search result page. Each search result has a link to the detailed description page where more information about the *PortalObject* is shown such as keywords and publications. An example of a result is shown in Appendix C.

4.1.3 Search Rank Algorithm

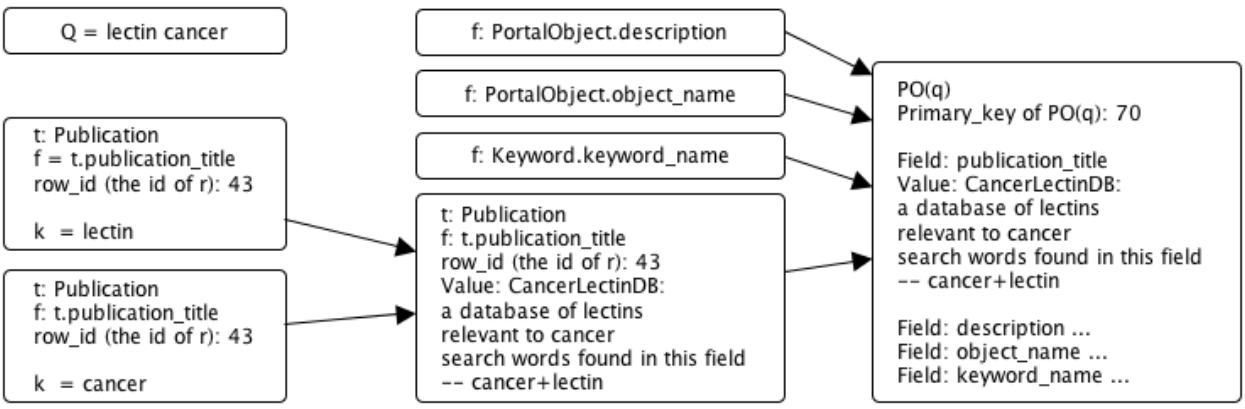
The rank algorithm is slightly modified from the original *tf-idf* (term frequency – inverse document frequency) formula [7]. We have added a function called match ratio (*mr*) that will be 1 if the keywords are found exactly in the text and will decrease as keywords are found embedded in longer words. All the key variables used in the search rank algorithm are defined in Figure 10 (a). Figure 10 (b) shows an example of how key variables are used in the search rank algorithm. More details are available in Figure 11.

Variables:
k = a keyword, which is a string consisting of letters and numbers
Q = $\{k\}$representing a user's query
t = a table in the Portal database
c = a column in the Portal database
f = $t.c$ such that column c within table t is chosen to be a searchable field

F = $\{f\}$ representing all the searchable fields
R = $\{r\}$ representing all the rows containing k

Functions:
$R(q)$ = **select * from** t **where** c LIKE '%k%'
for any k in Q and t.c in F
ResultsSet of rows matching a user's keywords in their searchable fields
$PO(q)$ = $R(q) \cap$ PortalObjects $\cup\{$ r in PortalObject $| \exists$ r' in $R(q)$ where r' joins with $r\}$
Set of PortalObjects directly or indirectly related to the ResultsSet
$PD(q)$ = a merge of $PO(q)$ and $R(q)$ forming PortalDocuments
For each PortalObject, all rows in the ResultSet referencing to this PortalObject are joined with it.
$fs(f)$ =a field score, which is a numerical value indicating the importance of f and is in the range (0,1]
$ncw(k,f,r)$ = numbers of characters in a word containing keyword k in row r of field f
$nw(k,f,r)$ = numbers of words containing keyword k in row r of field f
$npd(k,f)$ = number of PortalObjects referenced by field f, when the field contains keyword k.

(a)



(b)

Figure 10. Key variables and functions in search rank algorithm

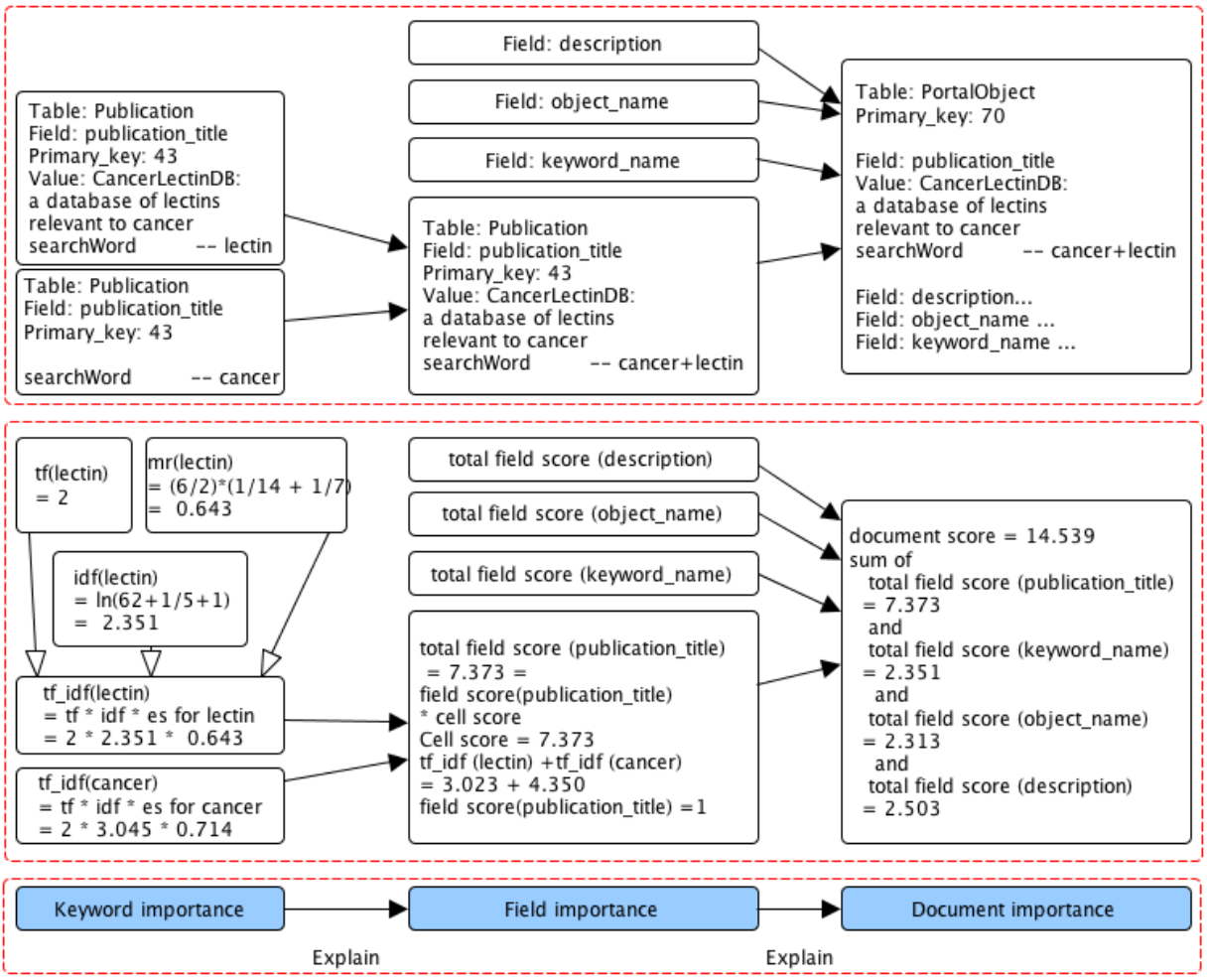


Figure 11. Search rank algorithm

First, I explain the main idea of the search rank algorithm. As shown in Figure 11, from the left hand side to the right hand side, I show the logic of the search rank algorithm. The importance of each component (a keyword, field or document) is a numerical value representing its importance. A modified *tf_idf* score indicates the importance of a keyword (*k*). A total field score indicates the importance of all keywords in that field. A document score indicates the importance of all fields containing the keywords in that document.

Next, we explain how the search rank algorithm works in detail. We will show all the formulas used in the rank algorithm. For every formula, we give an example and description.

Match Ratio:

$$mr(k,f,r) = \frac{len(k)}{nw(k,f,r)} \cdot \sum_{j=1}^{nw(k,f,r)} \frac{1}{ncw_j(k,f,r)}$$

$$mr(k,f,r) \in \mathbb{Z}^*$$

mr (lectin, publication_title, row43) = (6/2) · (1/14 + 1/7) = 0.643

The match ratio score shows how much the keywords are matched in words. For example, the keyword 'lectin' is found two times in the words "CancerLectinDB" and "lectins" in the publication_title field of the row43 in Figure 11. The length of each word containing the user's keyword is 14 and 7, respectively. Note that each occurrence of the word becomes a term in the *mr* formula (e.g., if "lectins" appeared twice, the result would be (6/3) · (1/14+1/7+1/7) = 0.714).

Term Frequency:

$tf(k,f,r)$ = frequency of keyword k in row r of field f

$$tf(k,f,r) \in \mathbb{Z}^*$$

tf (lectin, publication_title, row43) = 2

This *tf* formula is an application of *tf* to PortalDocuments. A common variant of the *tf* formula is shown below.

$tf(k,d)$ = the frequency of keyword/term k in document d

This *tf* formula is used to weigh frequently occurring keywords more highly. In the example, the number of occurrences of the keyword lectin in the publication.publication_title field is 2.

Inverse Document Frequency:

$$idf(k,PD) = \ln\left(\frac{|PD|+1}{|\{pd \in PD \mid k \text{ in } pd\}|+1}\right) \qquad idf(k,f) \in \mathbb{R}^*$$

$$idf(\text{lectin, PD}) = \ln\left(\frac{62+1}{5+1}\right) = 2.351$$

This formula is an application of *idf* to PortalDocuments. A common variant of the *idf* formula is shown below.

$$idf(k,D) = \ln\left(\frac{|D|+1}{|\{d \in D \mid k \text{ in } d\}|+1}\right)$$

This *idf* formula is used to weigh uncommon (or less frequently occurring) keywords more highly. In the example, the total number of PortalDocuments is 62, while the number of occurrences of the keyword 'lectin' in all the PortalDocuments is 5.

Modified tf_idf:

$$tf\_idf(k,f,r) = tf(k,f,r) \cdot idf(k,PD) \cdot mr(k,f,r)$$
$$tf\_idf(k,f,r) \in \mathbb{R}^*$$

$$tf\_idf(\text{lectin, publication\_title, row43}) = 2 \cdot 2.351 \cdot 0.643 = 3.023$$

The *tf_idf* shows the importance of the keyword. In each cell, the keyword either fully matched or partially matched. Thus, we multiply the match ratio score of the keyword by the *tf_idf* score.

Cell Score:

$$cs(f,r) = \sum_{k \in K} tf\_idf(k,f,r)$$
$$cs(f,r) \in \mathbb{R}^*$$

$$tf\_idf(\text{lectin, publication\_title, row43}) \cdot 2 \cdot 2.351 \cdot 0.643 + tf\_idf(\text{cancer, publication\_title, row43}) \cdot 2 \cdot 3.045 \cdot 0.714 + = 7.373$$

The cell score show the importance of the keyword in that field in a given row. Several keywords can be found in a cell defined by a particular field and row. Thus, we sum the *tf_idf* scores of the keywords found in the cell. In the example, two keywords (lectin, cancer) are found in the field (publication_title) in the row43.

Total Field Score:

$$tfs(pd,f) = fs(f) \cdot \sum_{r \in R(pd)} cs(f,r)$$

$$tfs(pd,f) \in \mathbb{R}^*$$

$$tfs\ (70,\ \text{publication\_title})\ =\ 1 \cdot\ 7.373\ =\ 7.373$$

A total field score indicates the importance of all keywords in that field. We multiply the sum of cell scores by the field score of publication_title, which is manually assigned and stored in the database. The function *R(pd)* represents a set of rows that contain keywords and reference a particular PortalDocument. In the example, only one cell in the publication_title that contains keywords and references PortalDocument 70.

Document Score:

$$ds(pd) = \sum_{f \in F} tfs(pd,f)$$

$$ds(pd) \in \mathbb{R}^*$$

$$\begin{aligned} ds(70)\ =\ & tfs\ (70,\ \text{publication\_title})\ + \\ & tfs\ (70,\ \text{description})\ + \\ & tfs\ (70,\ \text{object\_name})\ + \\ & tfs\ (70,\ \text{keyword\_name}) \end{aligned}$$

$$\begin{aligned} ds(70) = 7.373\ & + \\ 2.503\ & + \\ 2.313\ & + \\ 2.351\ & \\ = 14.539\ & \end{aligned}$$

The document score (*ds*) is the sum of all total field scores. Thus, a document score indicates the importance of all fields containing the keywords in that document. We consider a higher document score is more relevant to the keywords.

4.2 Advanced Search

4.2.1 Introduction

In this section, we introduce our advanced search. Advanced search is inspired by the *Pubmed's* advanced search where a query is generated based on a user's inputs such as a field name and keyword. Similarly, in our advanced search, a query is generated based on the user's inputs. An example of a query in advanced search is shown below.

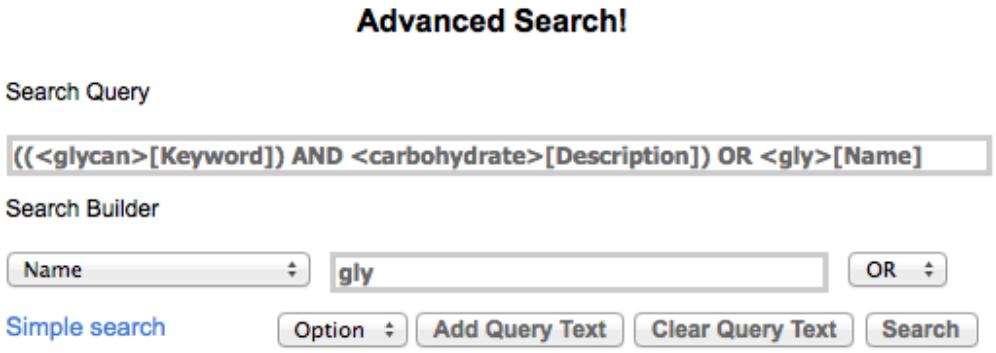


Figure 12: The advanced search query with an example

The search query shown in Figure 12 is generated based on a user's inputs. More details are explained in Appendix B.

We explain how an advanced search query is performed in our system. Each keyword is written between "<" and ">", and each searchable field name is written between "[" and "]". First, a keyword and field are captured from an advanced search query. Next, the keyword and

field are passed into the function, which performs a search over Lucene indexes to retrieve related information. A search is performed over Lucene indexes for each keyword with a field. After retrieving the information, we find common *PortalObjects* and group the retrieved information by *PortalObjects*. We then process Boolean Operators in the user query with all the retrieved information.

4.2.2 Different search options

We show different options in advanced search in this section.

Proximity Search: Search for the exact matching words and different orders of a user's keywords can be found.

Phrase Search: Search for the words that are exactly matched.

Field Restriction: A user can restrict to fields.

Boolean Search: OR, AND or combination of OR and AND operators are available.

CHAPTER 5

EVALUATION OF WORK

There are free full text search systems available, such as Lucene, Hibernate Search and PostgreSQL, but there are some drawbacks in these systems. We have attempted to make those systems comparable to our full text search system. We explain why these full text search systems are not comparable with our search system, and we describe our evaluation with human experts.

In order to perform a comparison, first, we need to create joins of each table containing the searchable fields with the main table. These joins enable finding *PortalObjects* in the main table from the tables that are queried and contain user's keywords. Next, we create a big table that contains all the joins using the OUTER UNION operator. The purpose of creating the big table is to be able to capture the results partially matching the user's keywords. Since a query on joins of tables reveals only linked tuples that matched every table that was queried, we need to create the big table. Then, we can create indexes of columns on the big table to perform full text search that supports rank scores.

We then explain the drawbacks in these systems. The drawback of Lucene is that it does not support the JOIN operator. Thus, we join each result returned from Lucene search with the main table to find *PortalObjects* using PostgreSQL. However, after the join, there is no rank score available. In other words, we do not have any criteria for the comparison.

Hibernate Search takes a Hibernate query and converts it to a Lucene query for full text search. However, Hibernate Search does not support the JOIN operator when creating a full text search query [8], we face the same issue that we have with Lucene.

For PostgreSQL, which supports the JOIN operator, there are different issues. In order to create the big table, we need to use the OUTER UNION operator. Unfortunately, PostgreSQL does not support the OUTER UNION operator yet. We also have tried to build joins of tables containing all the searchable fields and save the result of the joins using the MATERIALIZED VIEW operator. Using the MATERIALIZED VIEW, we can reuse the result of the joins. Then, we can create an index on fields in the result to perform full text search. However, PostgreSQL does not support the MATERIALIZED VIEW operator. Next, we tried to use the full text search with *tsvectors*, which is described in the PostgreSQL Website [6]. The *tsvectors* becomes less effective when there are many characters in a text. In other words, it becomes fairly slow. We have also tried indexes of *tsvectors* to enhance the speed of a query. Even though we found it faster than *tsvectors*, it is still not fast enough as shown in Appendix B.

Finally, we decided to compare our search results with results from two experts. We show the precision and recall of our search for user queries and which search options give more impact to our search. We created 8 queries written in English, each of which is converted to an advanced search query that can be used in our search. The 8 queries are shown in Table 1, and Table 2 shows the 8 advanced search queries with Boolean operators.

Table 1: 8 human queries used in the evaluation

Query 1: Find tools for processing Mass spectrometry
Query 2: Find tools for processing Glycosylation site prediction
Query 3: Find tools related to Carbohydrate
Query 4: Find databases related to Carbohydrate structure
Query 5: Find databases related to Enzyme OR Lectin
Query 6: Find carbohydrate structure databases that provide a substructure search
Query 7: Find all software and databases related to NMR
Query 8: Find all software and databases related to 3D structure of Glycan

Table 2: 8 advanced search queries with Boolean operators used in the evaluation

Query 1: Find tools for processing Mass spectrometry ="mass spectrometry[ALL] AND software[ALL]"
Query 2: Find tools for processing Glycosylation site prediction = "Glycosylation site prediction[ALL] AND software[ALL]"
Query 3: Find tools related to Carbohydrate ="carbohydrate[ALL] AND software[ALL]"
Query 4: Find databases related to Carbohydrate structure ="carbohydrate structure[ALL] AND Data source[ALL]"
Query 5: Find databases related to Enzyme or lectin ="Enzyme[ALL] OR lectin[ALL] AND Data source[ALL]"
Query 6: Find databases that provide a substructure search ="sub structure search[ALL] AND Data source[ALL]"
Query 7: Find all software and databases related to NMR ="nmr[ALL]
Query 8: Find all software and databases related to 3D structure of Glycan ="3D structure[ALL]"

For each query, we applied 6 different search options to find out which search options give more impact to the search as shown in Table 3. For each search query with a search option, we generate 3 queries, each of which uses only the OR, AND or both Boolean operators. Thus,

we have 8 advanced queries with 6 options, each of which has 3 queries. In other words, for each query, there are 18 different queries are used as shown in Table 3.

Table 3: 18 different advanced search queries for each human query used in the evaluation

Search Options	Description	Boolean Operators		
		OR	AND	OR + AND
Regular	No restrictions	1st	2nd	3rd
Phrase	Only search for an exact user input	4th	5th	6th
Proximity	Same as Phrase, but not ordered	7th	8th	9th
Field Restriction	Can restrict to fields	10th	11th	12th
Phrase + Field	Combination of Phrase and Field Restriction	13th	14th	15th
Proximity + Field	Combination of Proximity and Field Restriction	16th	17th	18th

We have calculated precision, recall, and $F_{0.5}$ scores for each query. We then calculate the average of the $F_{0.5}$ scores. $F_{0.5}$ score weights precision more than recall. This is why we have used $F_{0.5}$ instead of F_1, which considers precision and recall equally. Table 4 shows how to calculate precision, recall and $F_{0.5}$ scores.

Table 4: A precision, recall and $F_{0.5}$ score

	Retrieved	Not retrieved
Number of relevant results	a	b
Number of irrelevant results	c	d
Precision	a / (a+c)	
Recall	a / (a+b)	
$F_{0.5}$ score	$\frac{(1+0.5^2) * \text{Precision} * \text{Recall}}{0.5^2 * \text{Precision} + \text{Recall}}$	

Precision is the number of relevant results retrieved divided by the total number of results that are retrieved. Recall is the number of relevant results retrieved divided by the total number of relevant results.

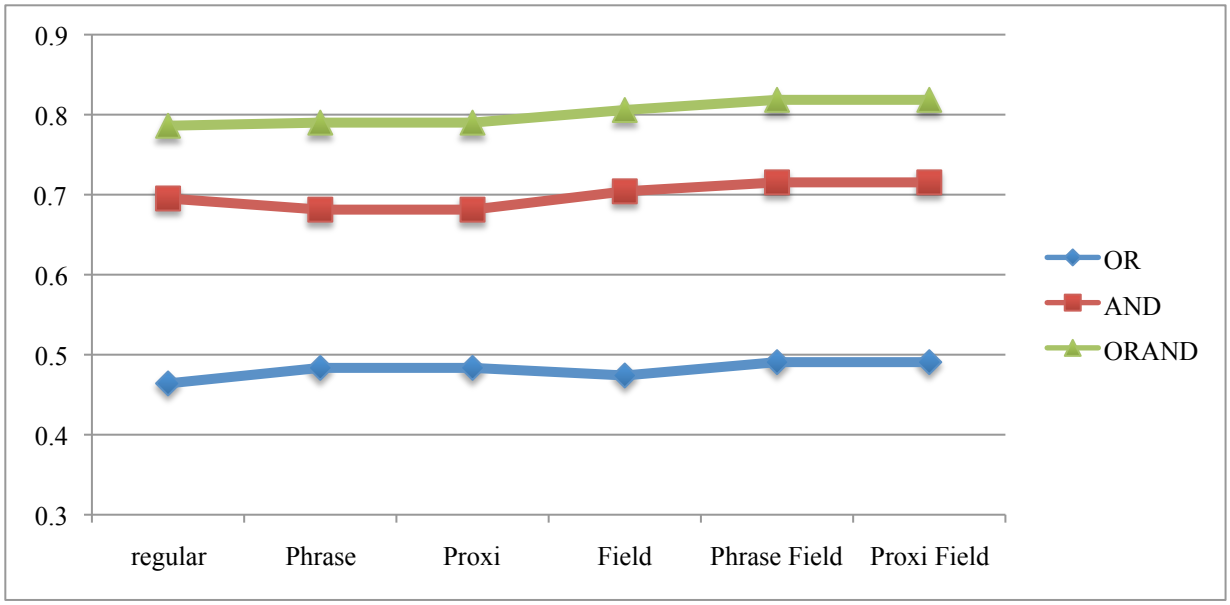


Figure 13. Average $F_{0.5}$ scores of 8 search queries compared to Expert1's answer.

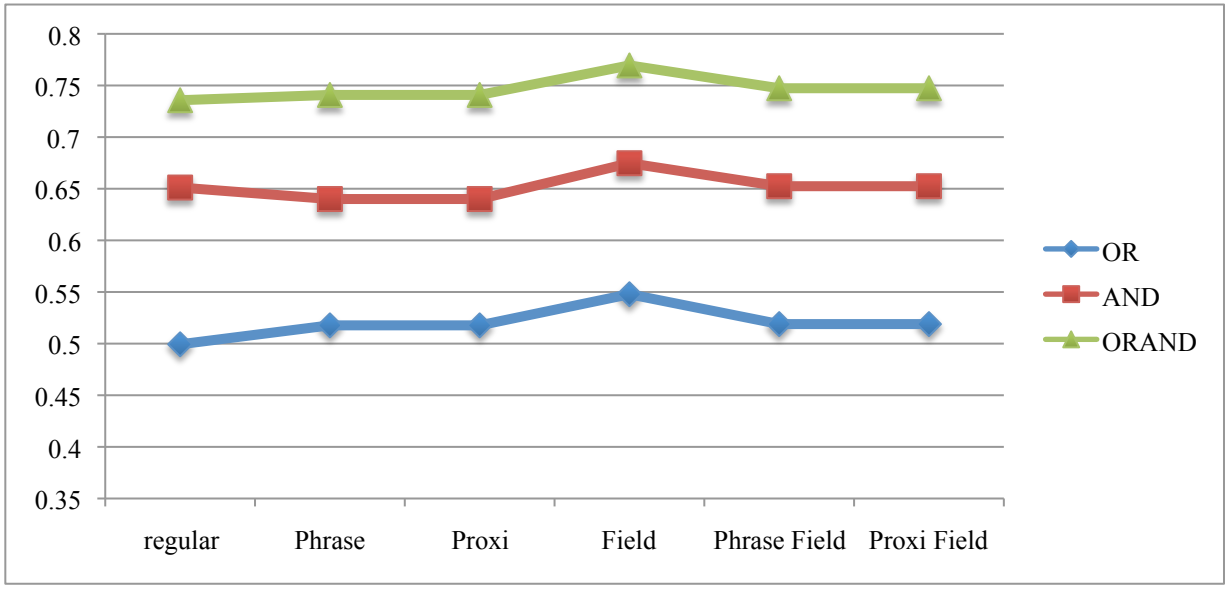


Figure 14. Average $F_{0.5}$ scores of 8 search queries compared to Expert2's answer.

We take two examples from the results performed (mean -1.3 * standard deviation used as a cut-off threshold) as shown in Figure 13 and 14. Based on all the results in Appendix A, we found that precision scores tend to get higher as we apply more options; combinations of Boolean operators (as shown ORAND) show higher $F_{0.5}$ scores than using only a Boolean operator (as shown OR or AND) in Figures 13 and 14. It shows that it is hard to express a query written in English using only the OR Boolean operator or AND Boolean operator. Also, searching with the AND Boolean operator shows higher $F_{0.5}$ scores than using the OR Boolean operator. The reason, we found, is that most of the advanced search queries use the AND Boolean operator. We also found that when fields are restricted, $F_{0.5}$ scores and precision scores get higher than when the other search options are applied. When fields are restricted, the user's keywords are found in the restricted fields. As a result of applying the field restrictions, precision scores will be increased, because keywords found in other fields will not be retrieved. For example, when a user searches for all *PortalObject*s of type software without applying field restrictions, the word, 'software,' in a description field may also be retrieved. The word 'software' may also be found with databases or workflows. In this case, the search results contain irrelevant results to the keyword 'software', which lowers the precision score of the search results.

We then discuss what lowers the $F_{0.5}$ scores. We observed that some user's keywords were not found by our search. In fact, all *PortalObjects* are related to the keyword, 'carbohydrate', but only some *PortalObjects* were retrieved. The reason is that this keyword is similar in meaning to other words such as 'glycan'. Without ontology or dictionaries, our search

system is not able to find synonyms or related words of the keyword. This would be a good example that shows the limitation of text search without ontology.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter contains the conclusions about this work and suggests future work.

6.1 Conclusions

This thesis demonstrates a different approach to full text search in relational databases using Lucene. We have developed a full text search, where a user can simply type keywords to access information. The contribution of our work is as follows. Our keyword search is able to search multiple tables in a relational database with rank capability. Next, we have built the Lucene field extractor, which can retrieve the information about fields where the user's keywords are found. In section 6.2, I provide some ideas for future work.

6.2 Future Work

1. User search histories

The field scores can be generated based on the popularity scores of *PortalObjects* and user search histories. The popularity scores of *PortalObjects* are not used in the *GlycomicsPortal* yet, and the popularity score is the number of user clicks on each *PortalObject*. The user search histories may reveal additional information such as the users' interests, which can be used to find out popular keywords among users and popular *PortalObjects*. Search histories may also reveal how many active users there are, which can be used to show the popularity of the portal. The

portal makes use of the information to learn more about the users. This feature will improve the rank for the search, because the field scores will be determined by the user search histories. Last, the user search histories can be also used in the search bar to show similar search words that were used before.

2.Stemming

Stemming extracts a root form from a word. Using stemming, we can find the plural of a word or different words having the same root form. For instance, two words (‘swimmer’ and ‘swimming’) have the same root form ‘swim’. Using this technique, we can find similar words having the same meaning or derived from the same root.

3. Synonyms

Wordnet synonym sets can be used for finding synonyms. The synonyms are useful for finding words similar to a user’s keyword when the keyword is not found in the search results. The user keyword can be replaced with those similar words to retrieve relevant information. The synonyms will be helpful to obtain desired results in the search.

REFERENCES

[1] Von der Lieth, C.-W.; Lütteke, T. and Frank, M. : The role of informatics in glycobiology research with special emphasis on automatic interpretation of MS spectra. *Biochim Biophys Acta,* 2006*, 1760*, 568-577

[2] Agrawal, S., S. Chaudhuri, et al. (2002). DBXplorer: A System for Keyword-Based Search over Relational Databases. Proceedings of the 18th International Conference on Data Engineering, IEEE Computer Society: 5.

[3] Hulgeri, A. and C. Nakhe (2002). Keyword Searching and Browsing in Databases using BANKS. Proceedings of the 18th International Conference on Data Engineering, IEEE Computer Society: 431.

[4] A. Arslan and O. Yilmazel, "A comparison of relational databases and information retrieval libraries on turkish text retrieval," in Natural Language Processing and Knowledge Engineering, 2008. NLP-KE '08. International Conference on, 19-22 2008, pp. 1 –8.

[5] MySQL; http://dev.mysql.com/doc/refman/5.0/en/fulltext-natural-language.html, accessed April 2012

[6] PostgreSQL; http://www.postgresql.org/docs/9.1/static/textsearch-controls.html, accessed April 2012

[7] G. Salton, A. Wong, and C. S. Yang (1975), "A Vector Space Model for Automatic Indexing," Communications of the ACM, vol. 18, nr. 11, pages 613–620.

[8] HibernateSearch; http://docs.jboss.org/hibernate/search/3.3/reference/en-US/html/search-query.html, accessed April 2012

[9] Solr; http://lucene.apache.org/solr/, accessed April 2012

[10] Compass; http://compass-project.org, accessed April 2012

[11] Yu, Jeffrey Xu, Lu Qin, and Lijun Chang. "Keyword Search in Relational Databases: A Survey." *IEEE Data Eng. Bull.* 1st ser. 33 (2010): 67-78. Web.

[12] Lucene; http://lucene.apache.org/core/, accessed April 2012

[13] Fang Liu, Clement Yu, Weiyi Meng, Abdur Chowdhury, Effective keyword search in relational databases, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, June 27-29, 2006, Chicago, IL, USA

[14] A. Baid, I. Rae, J. Li, A. Doan, and J. Naughton, “Toward Scalable Keyword Search over Relational Data,” Proceedings of the VLDB Endowment, vol. 3, no. 1, pp. 140–149, 2010.

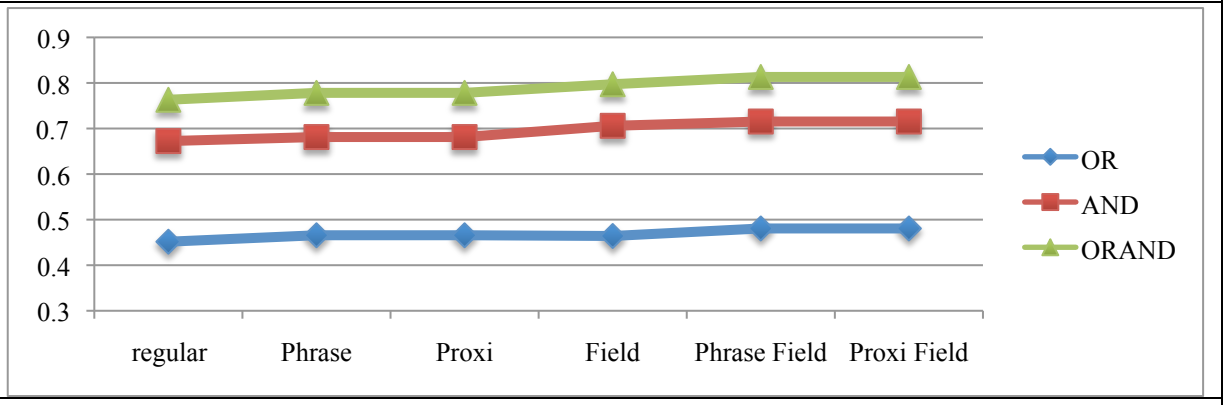
APPENDIX A

RESULT SCORES WITH DIFFERENT STANDARD DEVIATIONS

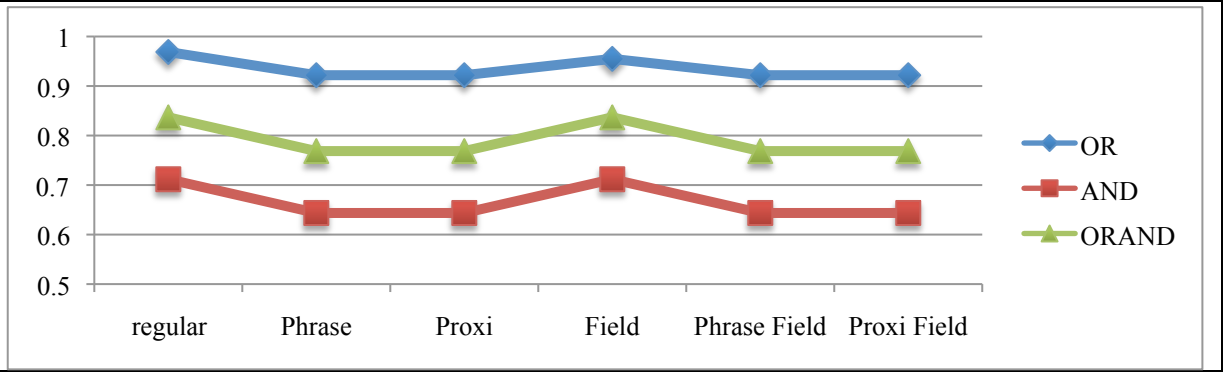
This Appendix contains all the F0.5 scores with different standard deviations. We tested our search results with 6 different standard deviation thresholds to measure, which give higher F0.5 scores. The 6 different standard deviation thresholds are used (0, 1, 1.3, 1.5, 1.7 and 2.0). For instance, the number "0" means the mean of rank scores of the results - 0 * standard deviation used as a cute-off threshold.

Expert1 with no standard deviation

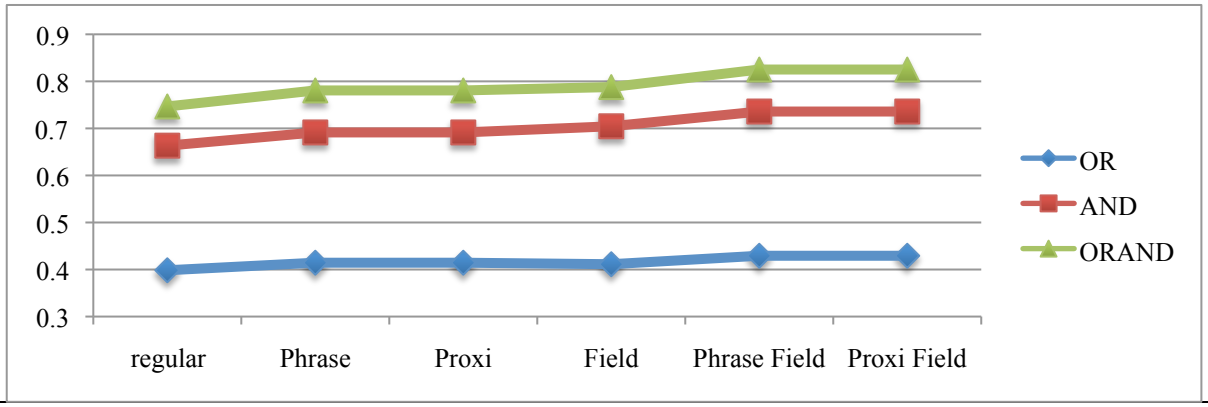
F0.5 score



Recall

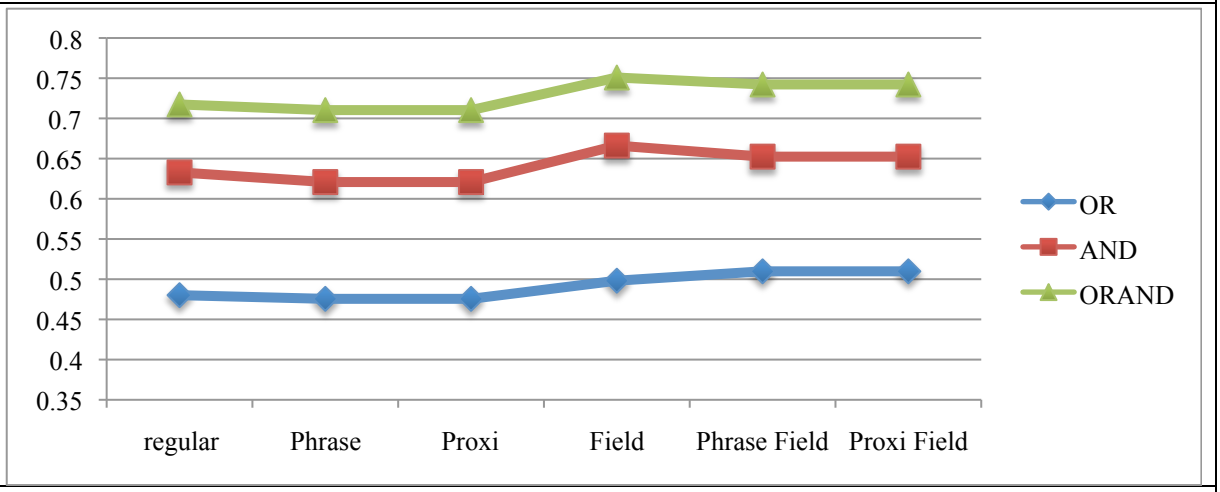


Precision

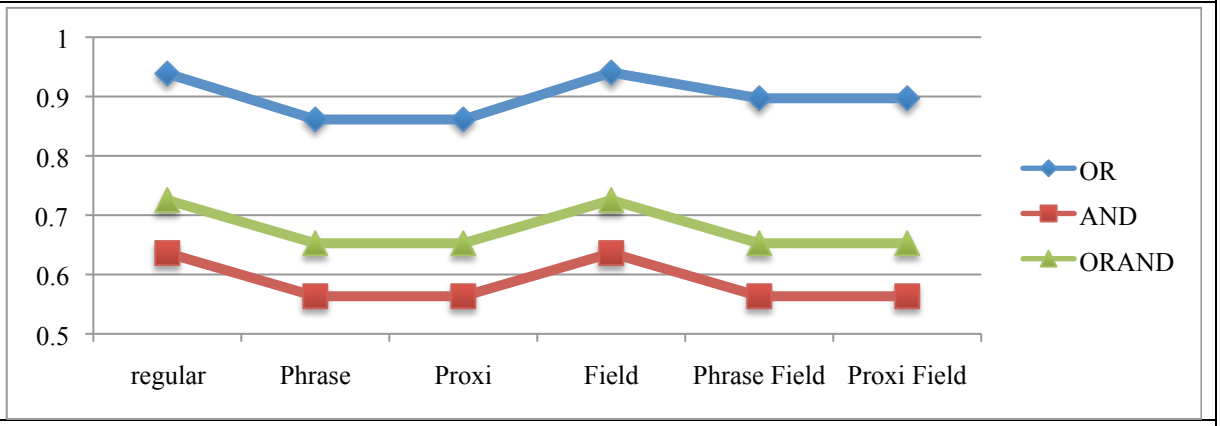


Expert2 with no standard deviation

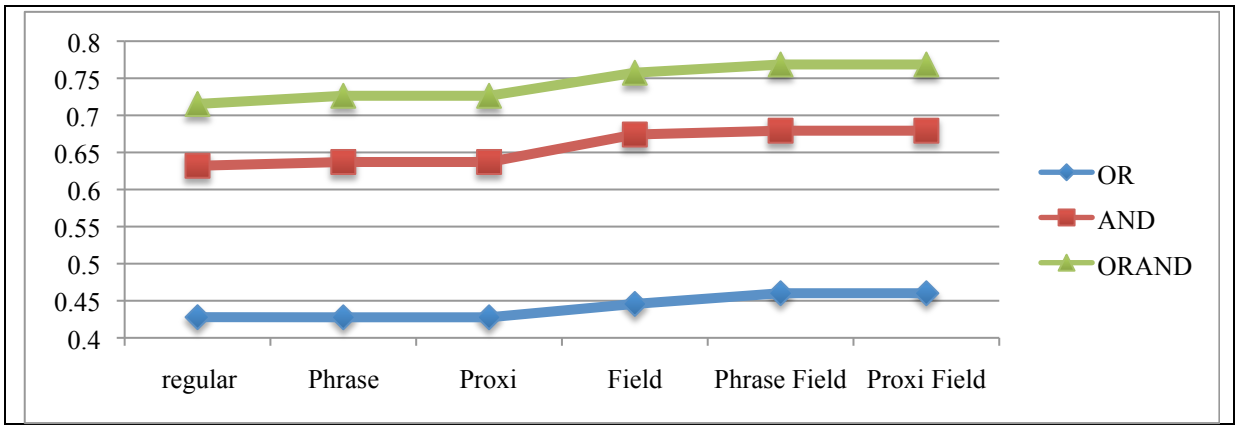
F0.5 score



Recall

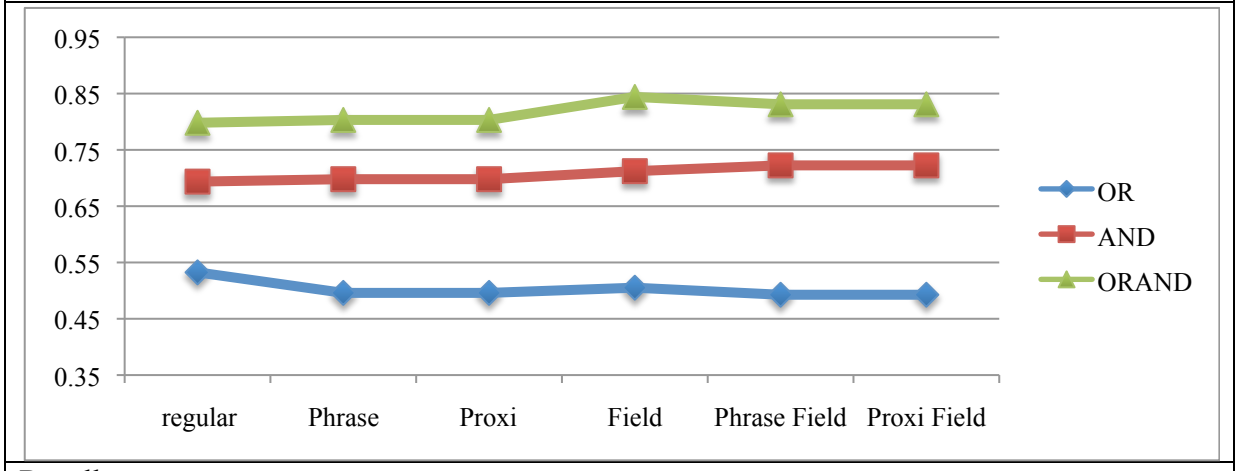


Precision

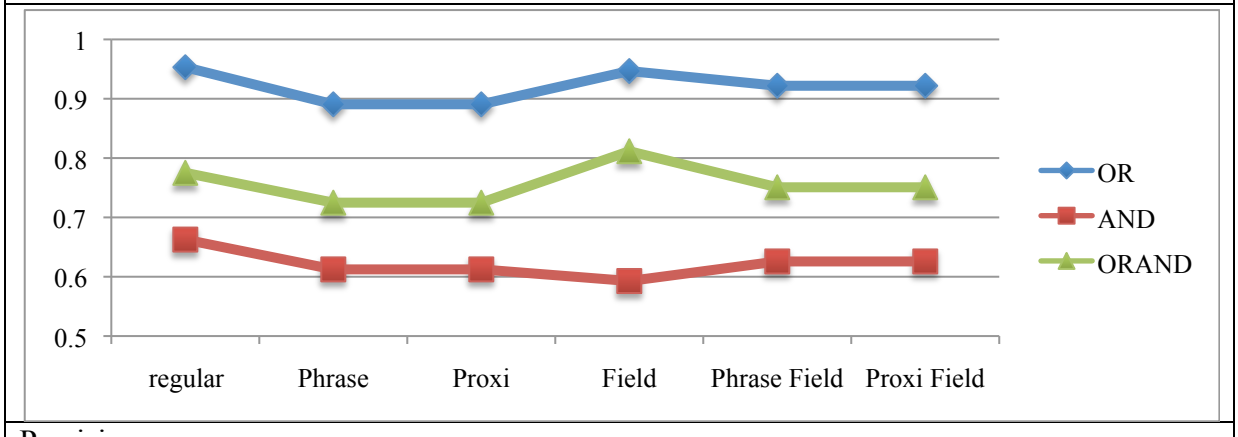


Expert1 with one standard deviation from the mean

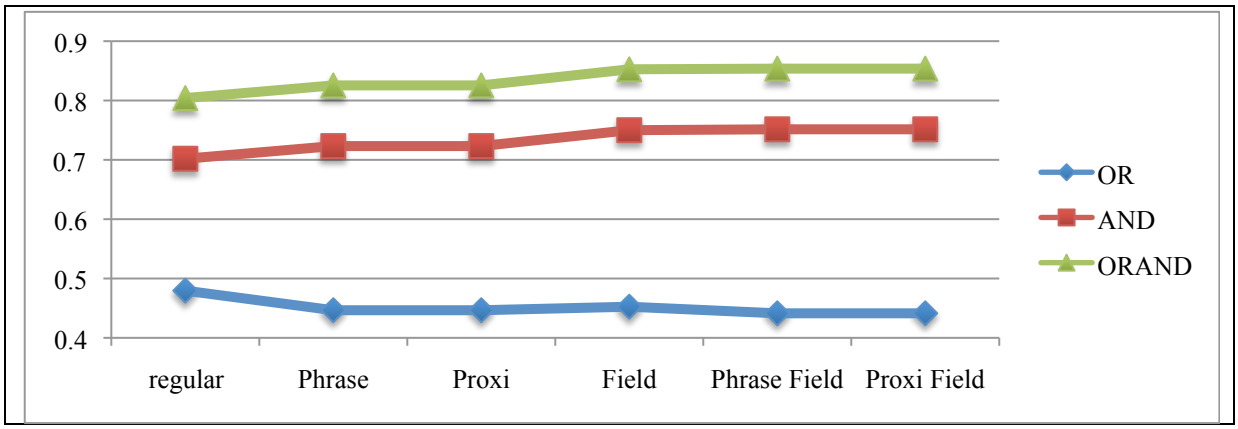
F0.5 score



Recall

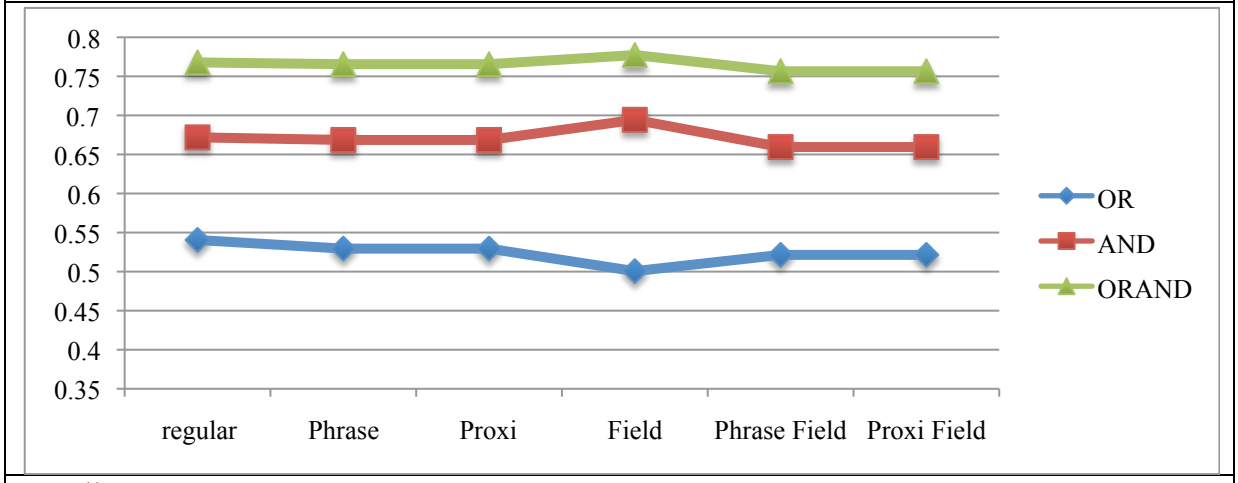


Precision

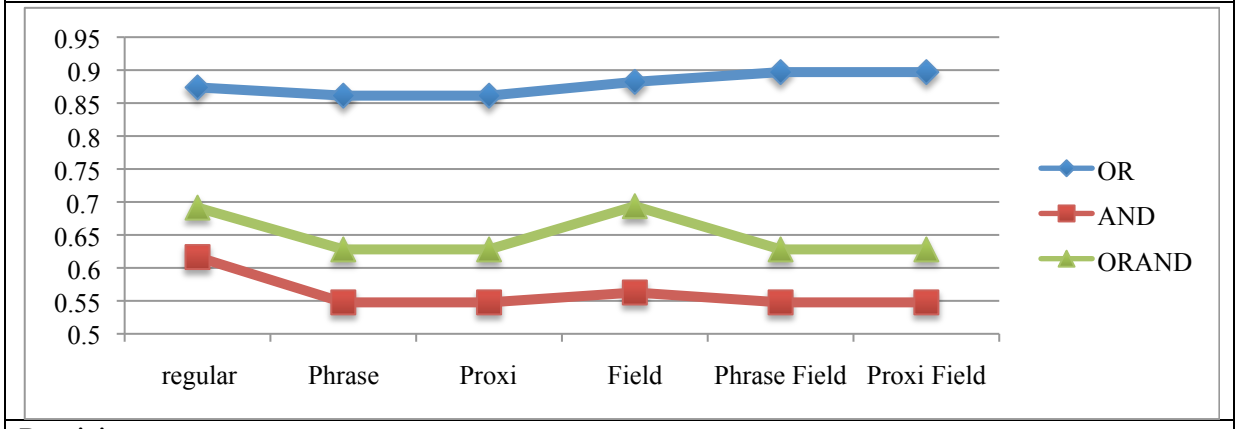


Expert2 with one standard deviation from the mean

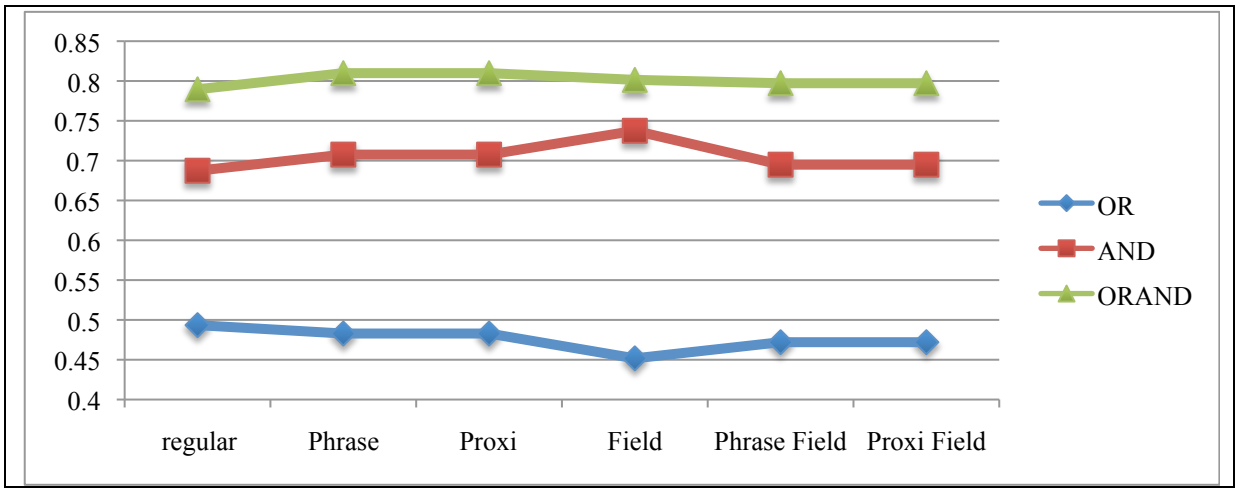
F0.5 score



Recall

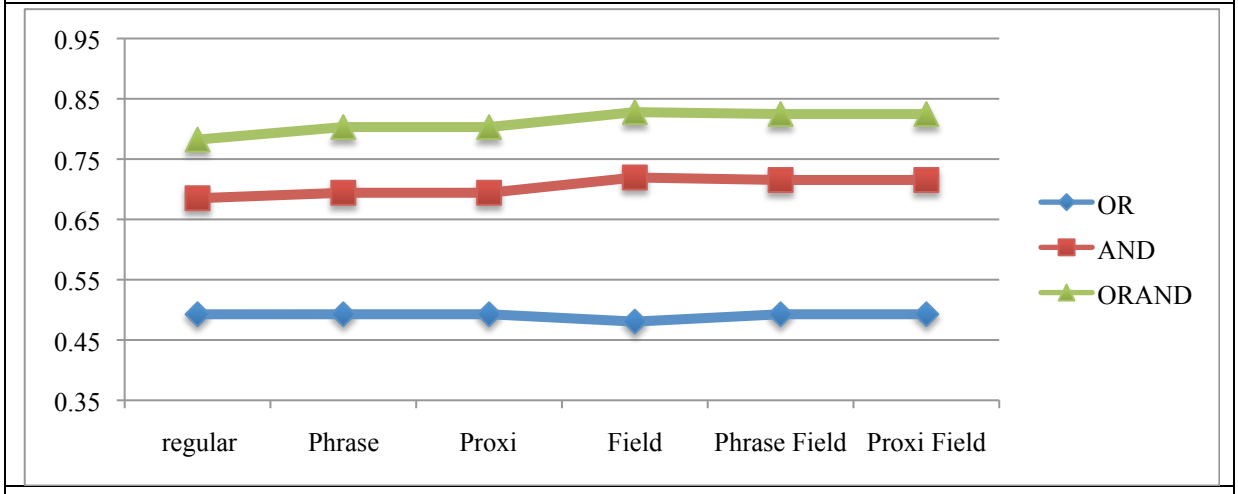


Precision

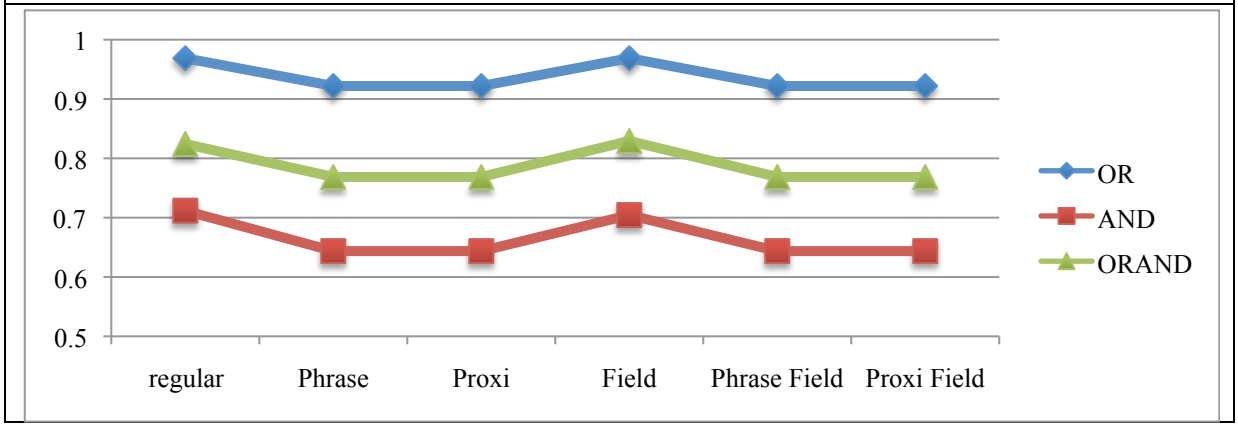


Expert1 with 1.3 standard deviation from the mean

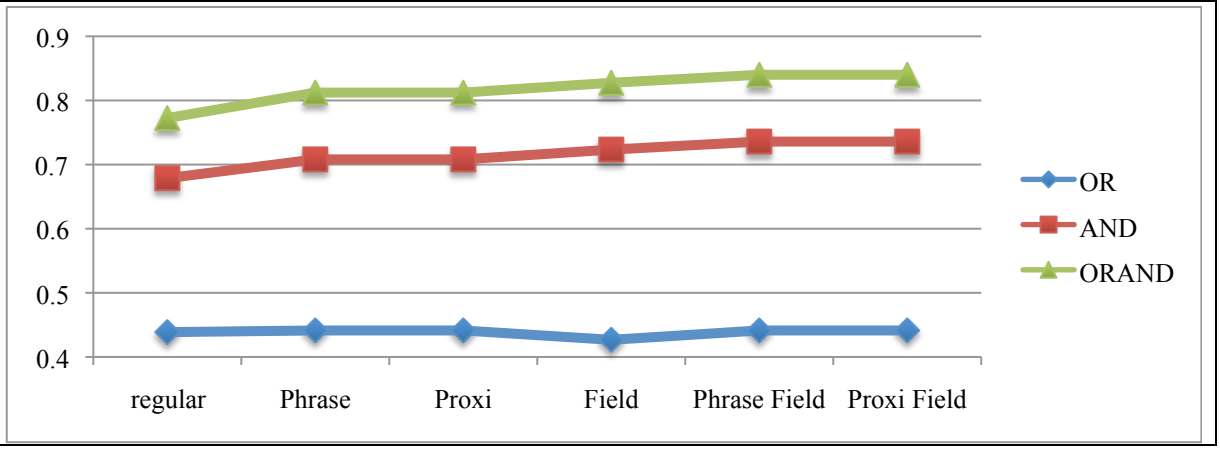
F0.5 score



Recall

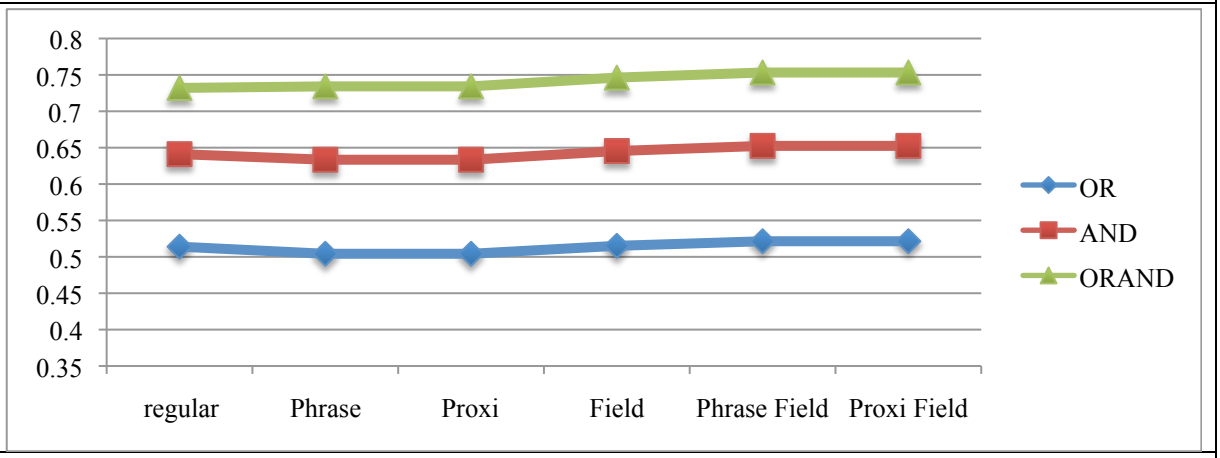


Precision

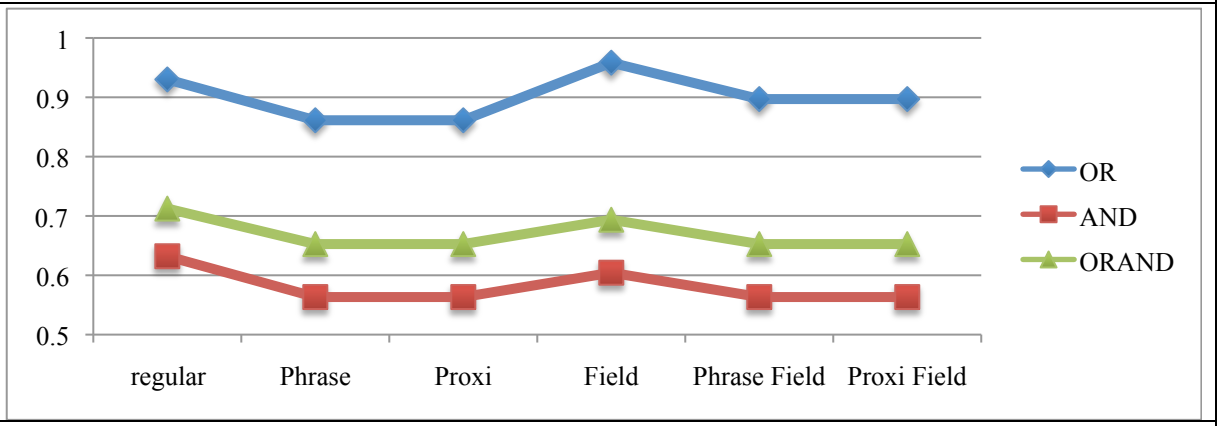


Expert2 with 1.3 standard deviation from the mean

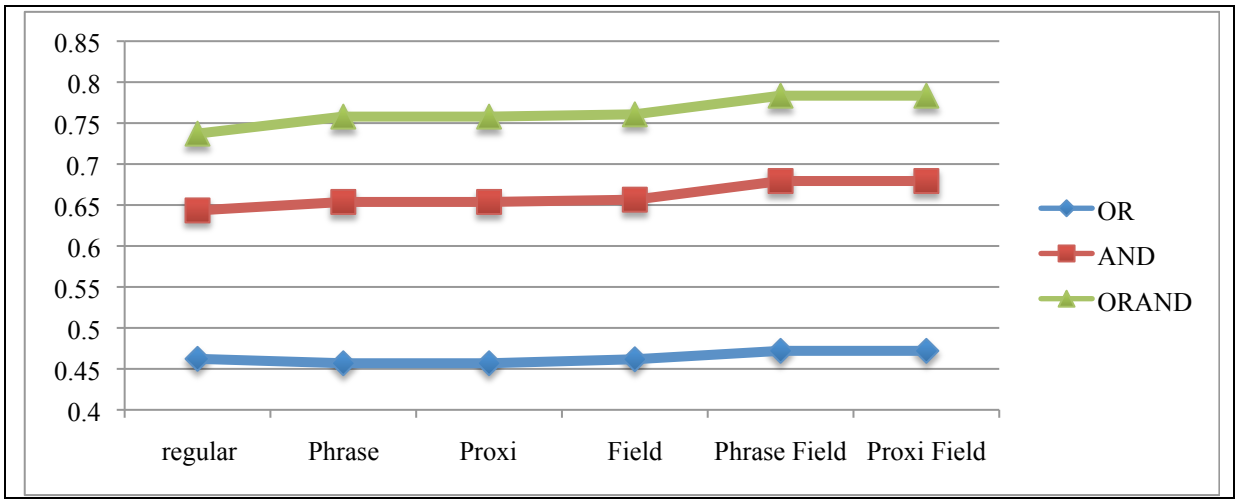
F0.5 score



Recall

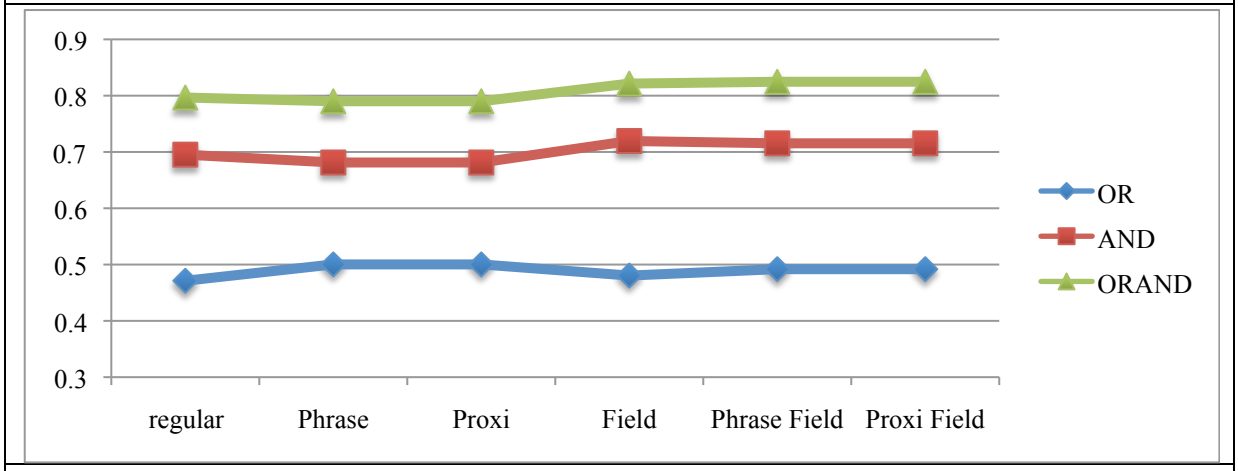


Precision

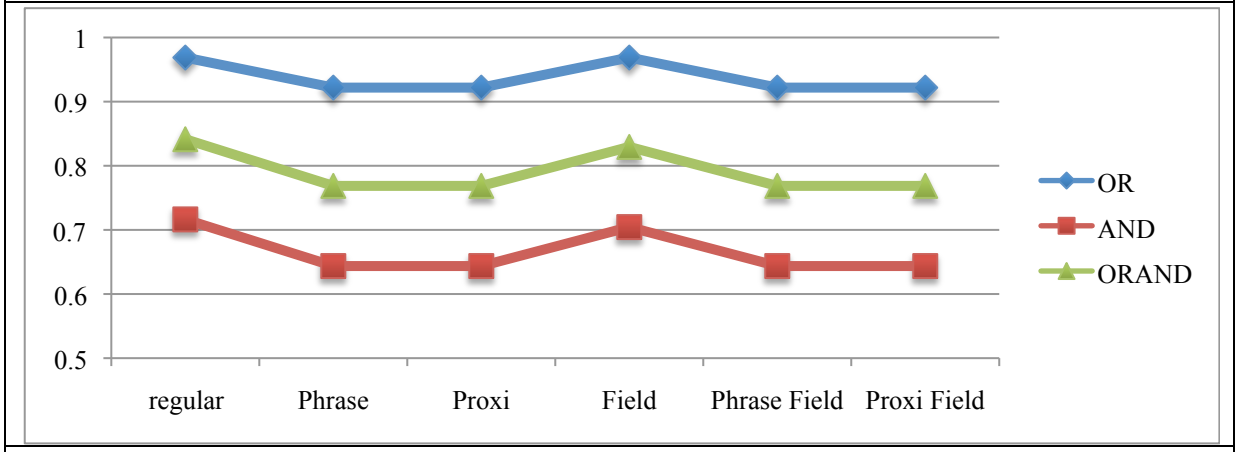


Expert1 with 1.5 standard deviation from the mean

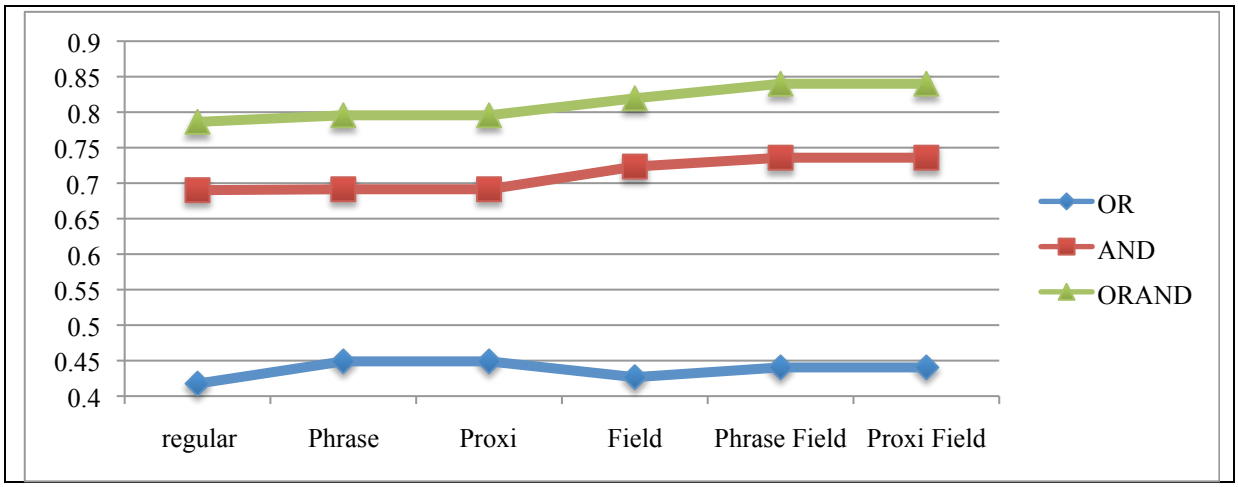
F0.5 score



Recall

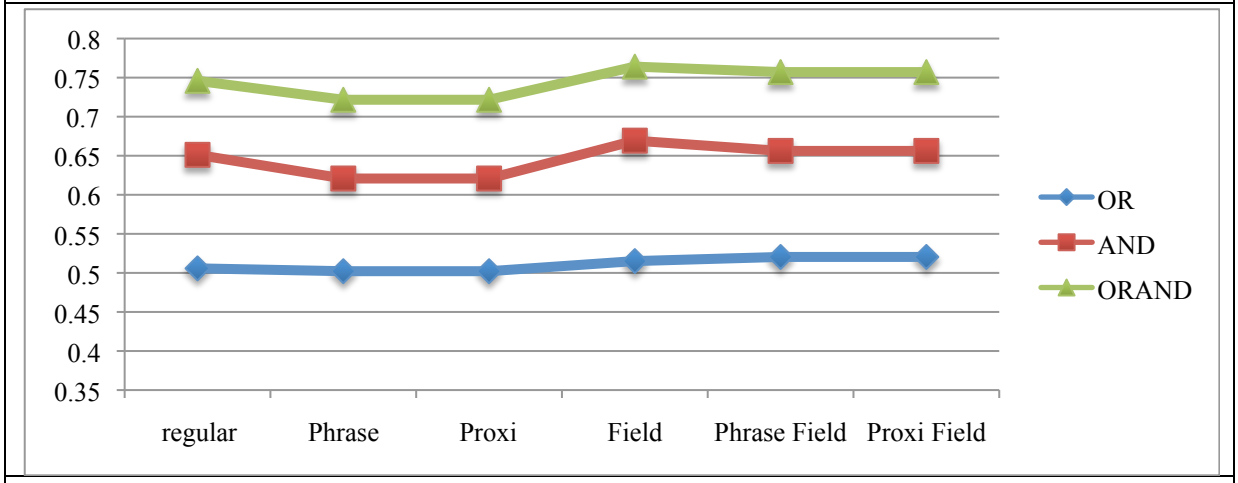


Precision

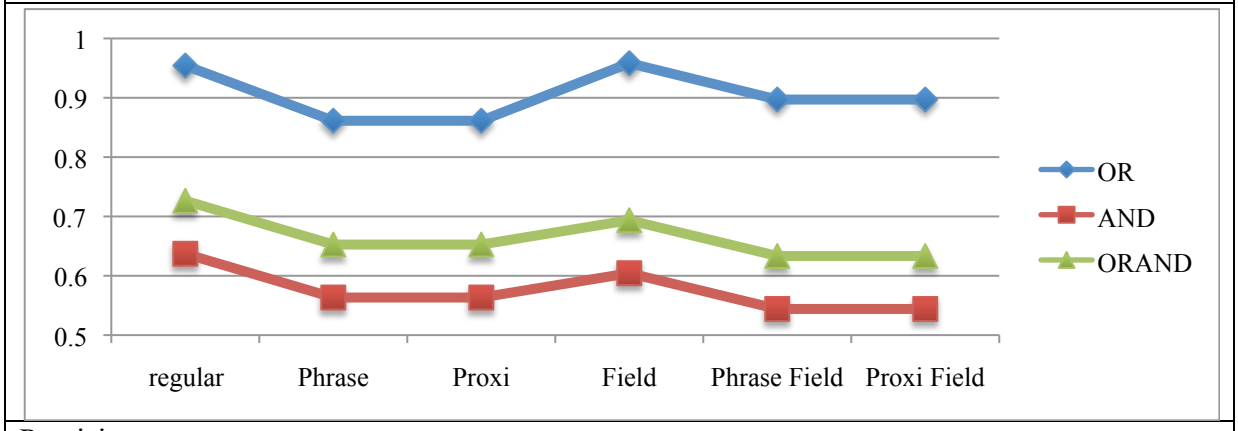


Expert2 with 1.5 standard deviation from the mean

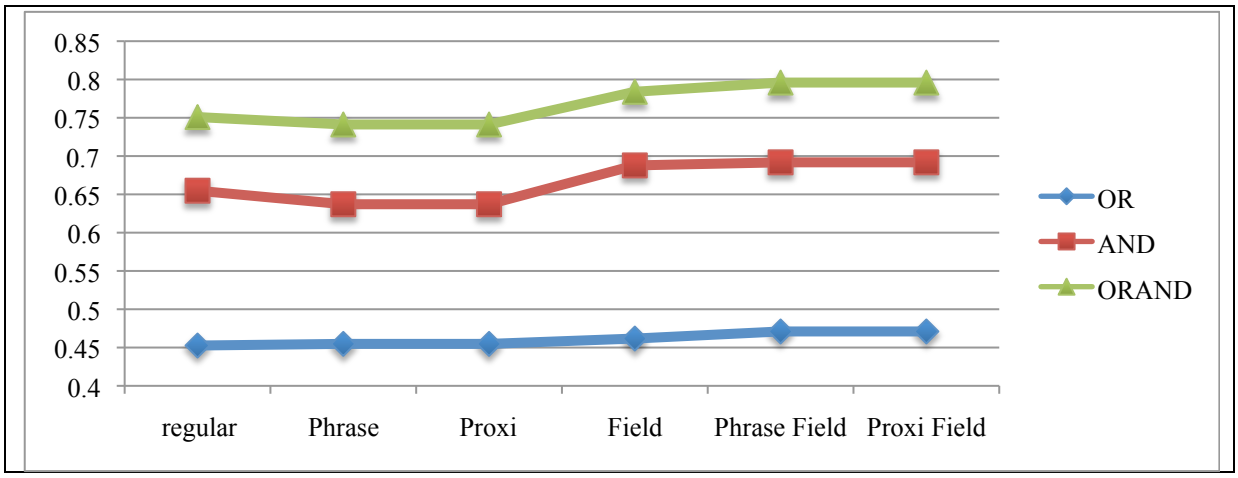
$F_{0.5}$ score



Recall

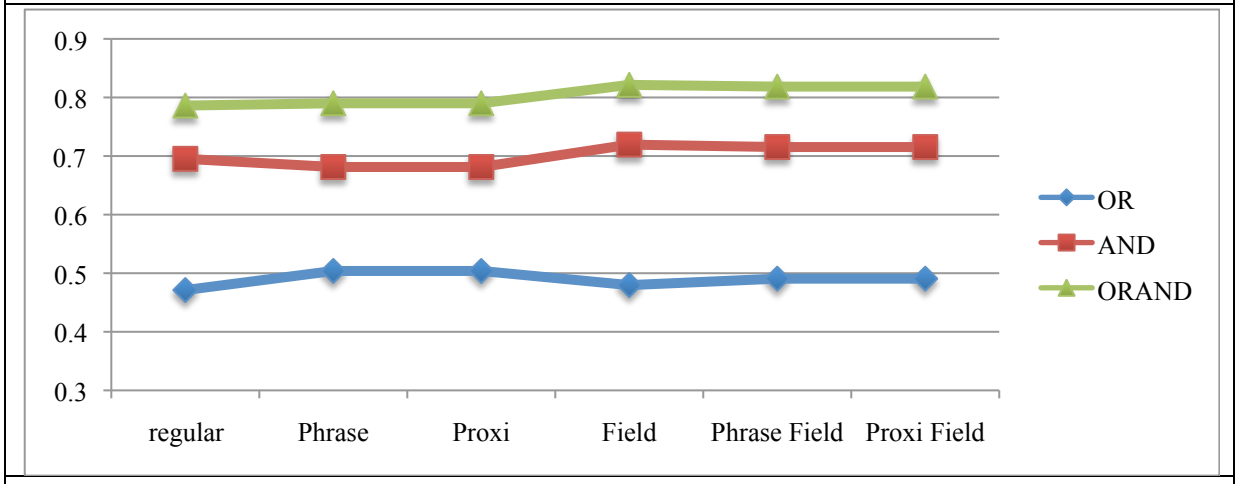


Precision

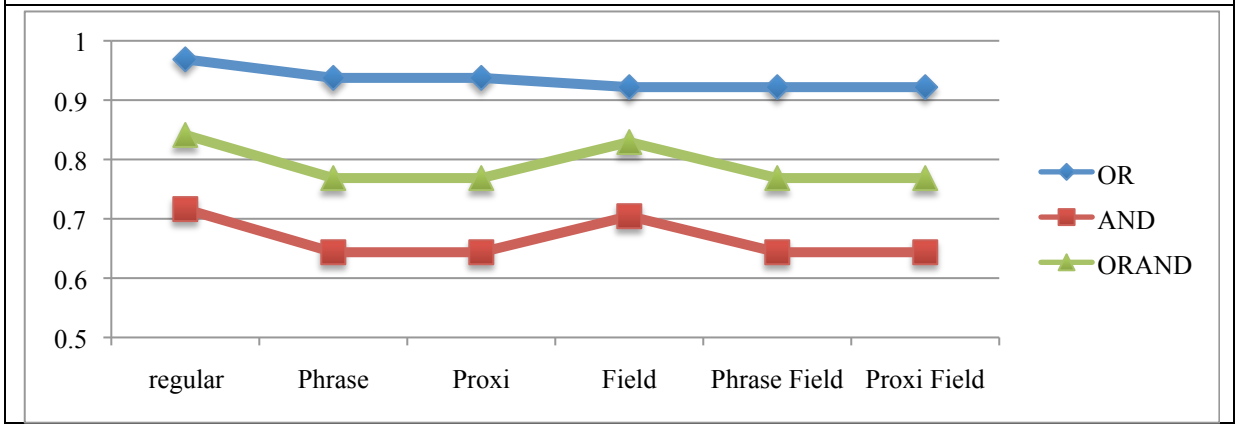


Expert1 with 1.7 standard deviation from the mean

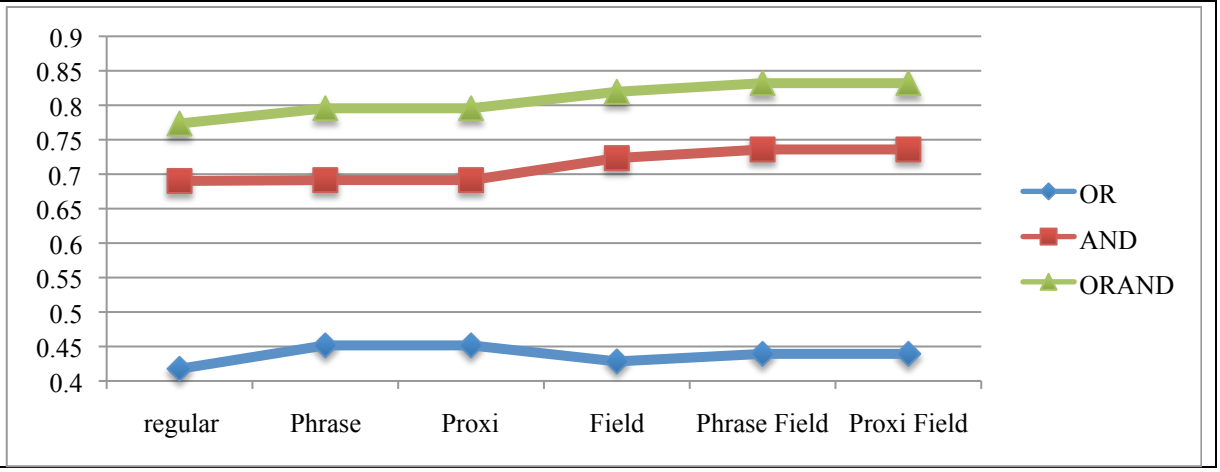
F0.5 score



Recall

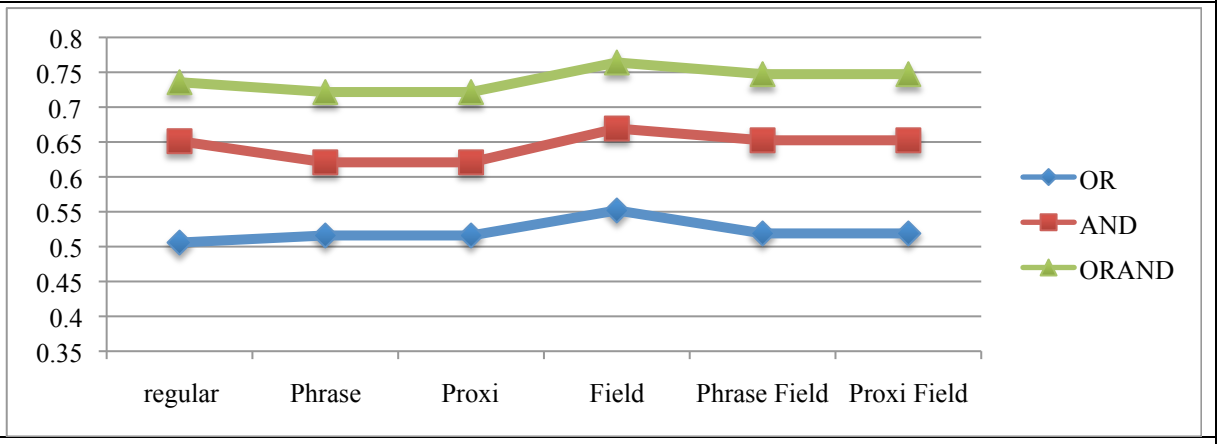


Precision

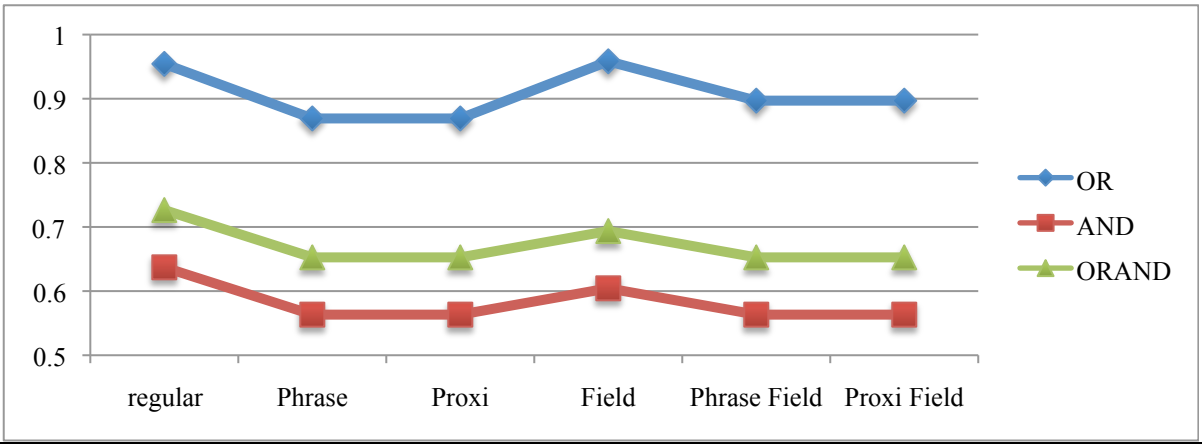


Expert2 with 1.7 standard deviation from the mean

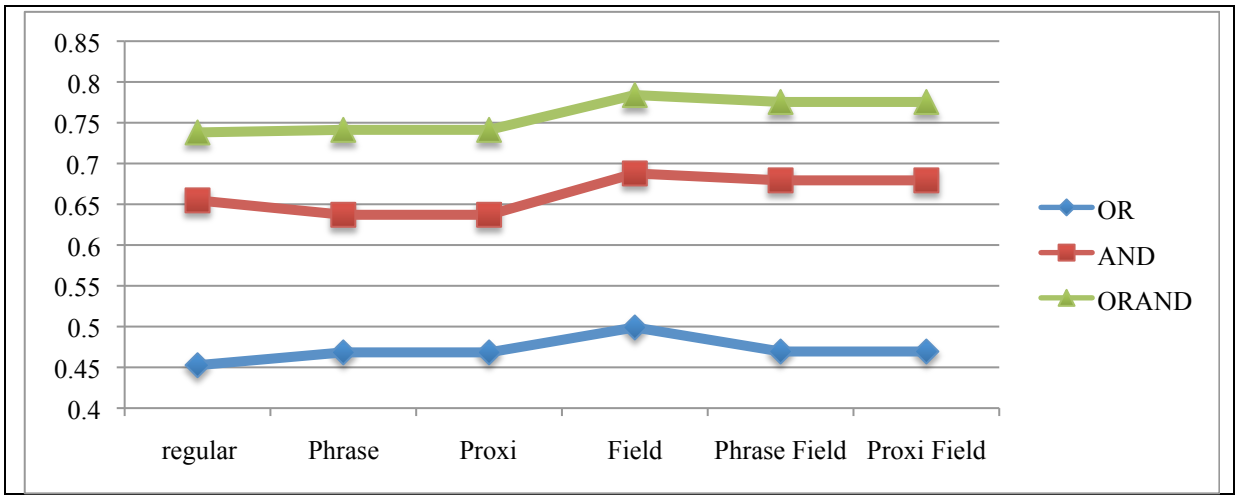
F0.5 score

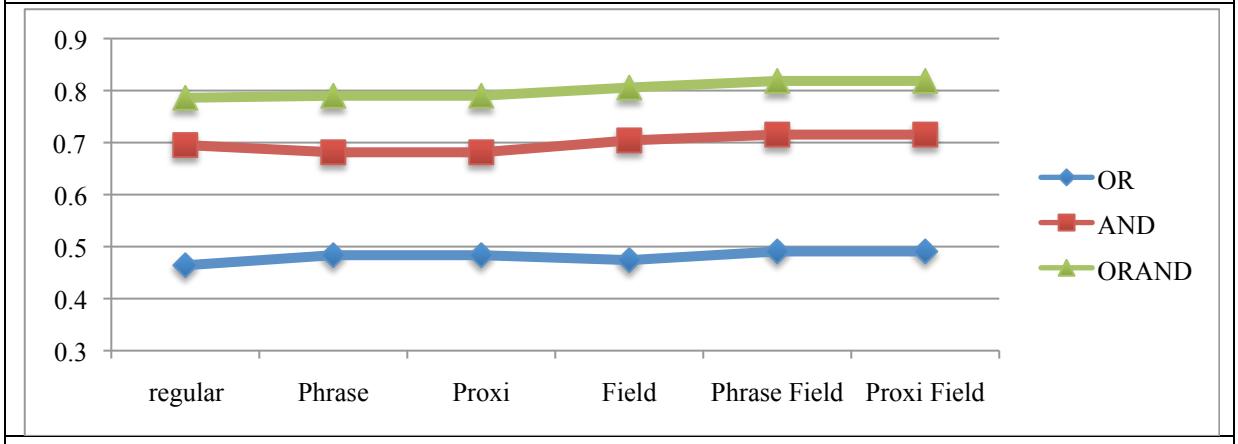
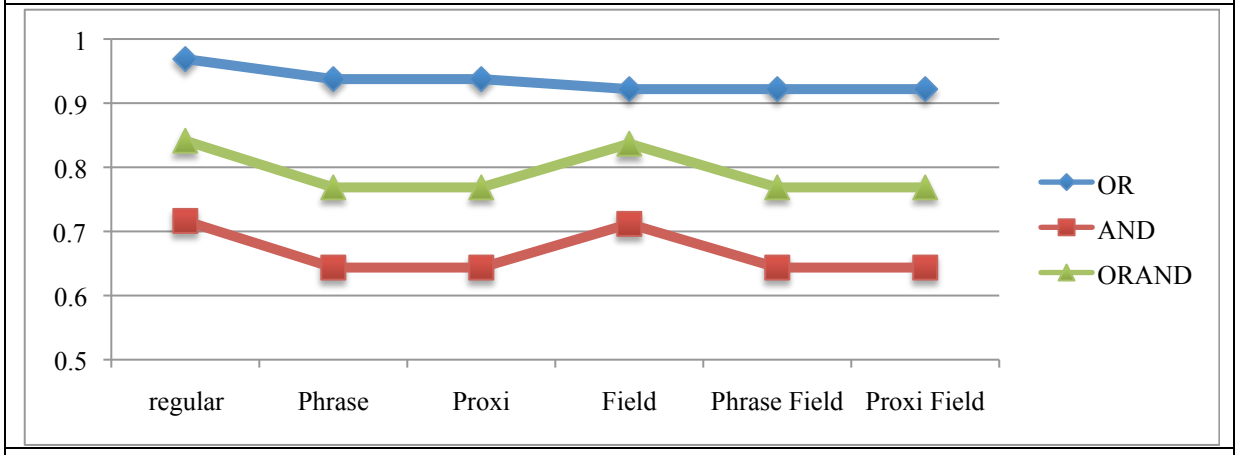


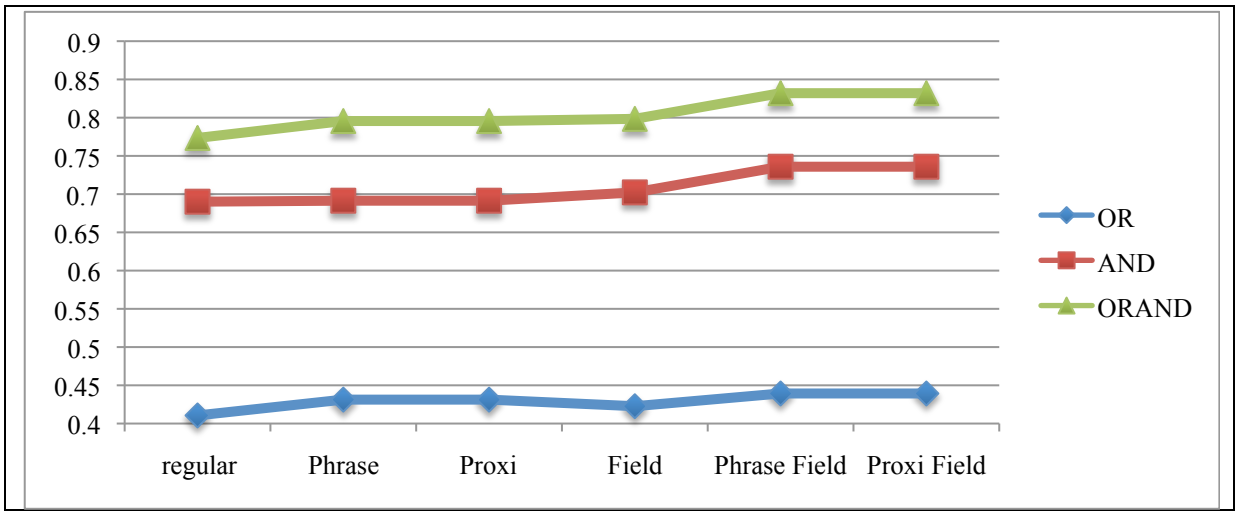
Recall



Precision

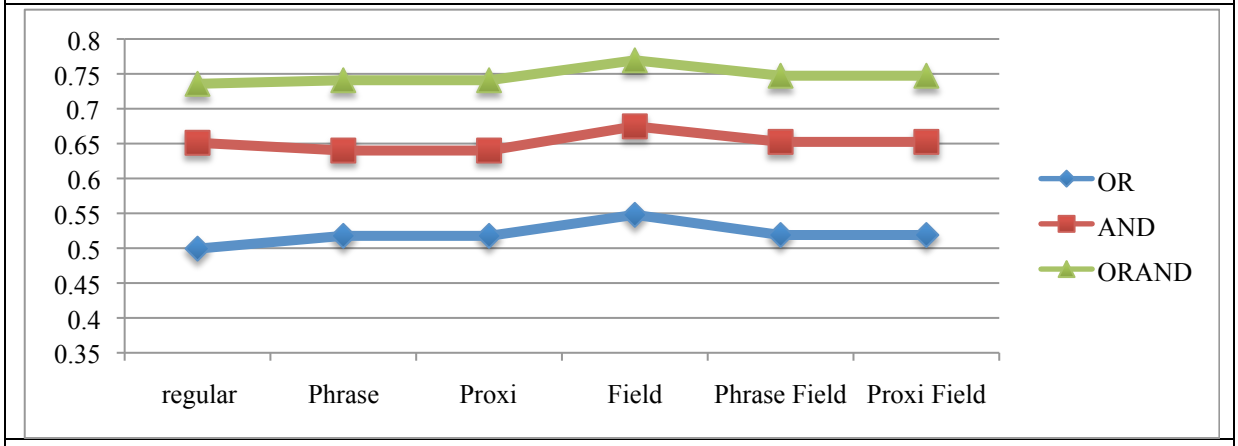


Expert1 with 2.0 standard deviation from the mean
F0.5 score

Recall

Precision

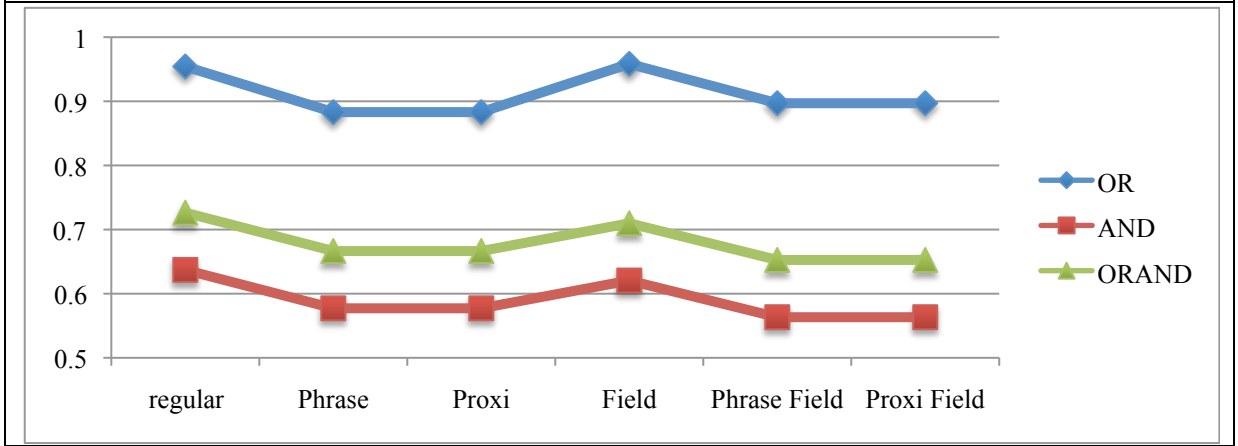


Expert2 with 2.0 standard deviation from the mean

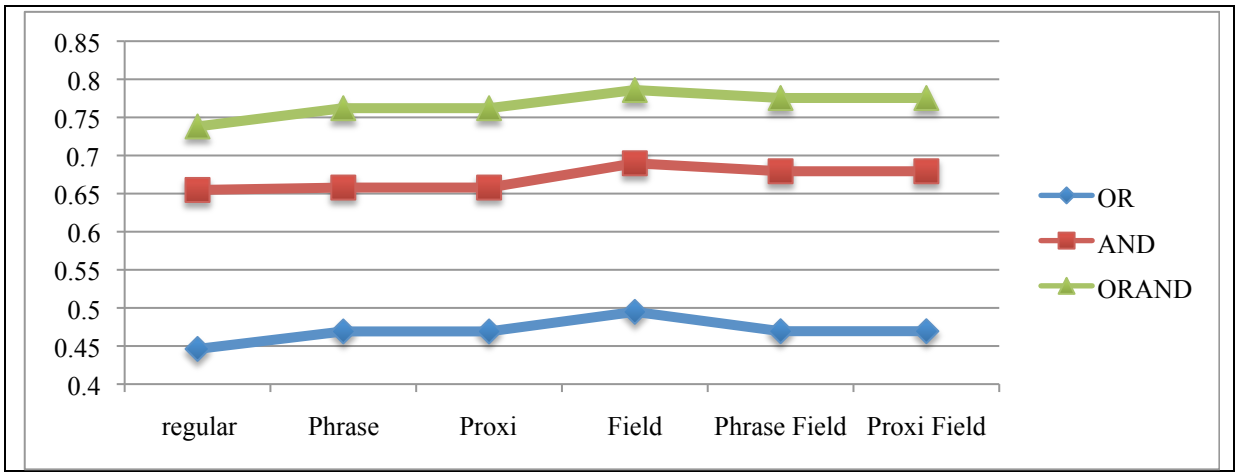
F0.5 score



Recall



Precision


0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
regular
Phrase
Proxi
Field
Phrase Field
Proxi Field
OR
AND
ORAND

APPENDIX B

QUERIES

This appendix shows the queries used to build the joins of tables containing the searchable fields in the database. In order to enable full text search and create the big table, we have tried to build the big table using PostgreSQL queries. We show each query and its run time.

Query	Query run time	
	No index	With Index
Query 1	0.055 seconds	0.024 seconds
Query 2	6.405 seconds	2.35 seconds
Query 3	2.052 hours	1.445 hours

Query 1

```
SELECT one.object_name, two.name,
     (one.rank1 + two.rank2)/2 AS rank
FROM portal.object_to_keyword OK,
--portal.object(description,object_name)
(SELECT object_id, object_name, description, ts_rank_cd (to_tsvector('english', object_name || ' ' || description),
query) AS rank1
FROM portal.object, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query
WHERE query @@ to_tsvector('english', object_name || ' ' || description) ) AS one,
--portal.keyword(name)
(SELECT name, keyword_id, ts_rank_cd(to_tsvector('english', name), query2) AS rank2
FROM portal.keyword, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query2
WHERE query2 @@ to_tsvector('english', name) ) AS two
WHERE (one.object_id = OK.object_id AND
   two.keyword_id = OK.keyword_id) ;
```

Query 1 with indexes

```
SELECT one.object_name, two.name,
     (one.rank1 + two.rank2)/2 AS rank
FROM portal.object_to_keyword OK,
--portal.object(description,object_name)
(SELECT object_id, object_name, description, ts_rank_cd (poindex, query) AS rank1
FROM portal.object, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query
WHERE query @@ poindex ) AS one,
--portal.keyword(name)
(SELECT name, keyword_id, ts_rank_cd (pkindex, query2 ) AS rank2
FROM portal.keyword, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query2
WHERE query2 @@ pkindex ) AS two
WHERE (one.object_id = OK.object_id AND
   two.keyword_id = OK.keyword_id) ;
```

Query 2

```
SELECT one.object_name, (one.rank1 + two.rank2 + three.rank3)/3 AS rank
FROM portal.object_to_keyword OK,
--portal.object(description,object_name)
(SELECT object_id, object_type_id, object_name, development_status_id, availibility_id, description, ts_rank_cd
(to_tsvector('english', object_name || ' ' || description), query) AS rank1
FROM portal.object, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query
WHERE query @@ to_tsvector('english', object_name || ' ' || description) ) AS one,
--portal.keyword(name)
(SELECT name, keyword_id, ts_rank_cd(to_tsvector('english', name), query2) AS rank2
FROM portal.keyword, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query2
WHERE query2 @@ to_tsvector('english', name) ) AS two,
--portal.object_type(object_type_name)
(SELECT object_type_id, object_type_name, ts_rank_cd(to_tsvector('english', object_type_name), query3) AS
rank3
FROM portal.object_type, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query3
WHERE query3 @@ to_tsvector('english', object_type_name) ) AS three
WHERE (one.object_id = OK.object_id AND
   two.keyword_id = OK.keyword_id) OR
   (one.object_type_id = three.object_type_id)
GROUP BY one.object_name, one.rank1, two.rank2, three.rank3
```

Query 2 with indexes

```
SELECT one.object_name, (one.rank1 + two.rank2 + three.rank3)/3 AS rank
FROM portal.object_to_keyword OK,
--portal.object(description,object_name)
(SELECT object_id, object_type_id, object_name, development_status_id, availibility_id, description, ts_rank_cd
(poindex, query) AS rank1
FROM portal.object, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query
WHERE query @@ poindex ) AS one,
--portal.keyword(name)
(SELECT name, keyword_id, ts_rank_cd(pkindex, query2) AS rank2
FROM portal.keyword, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query2
WHERE query2 @@ pkindex ) AS two,
--portal.object_type(object_type_name)
(SELECT object_type_id, object_type_name, ts_rank_cd(potindex, query3) AS rank3
FROM portal.object_type, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query3
WHERE query3 @@ potindex) AS three
WHERE (one.object_id = OK.object_id AND
   two.keyword_id = OK.keyword_id) OR
   (one.object_type_id = three.object_type_id)
GROUP BY one.object_name, one.rank1, two.rank2, three.rank3;
```

Query 3

```
SELECT one.object_name, (one.rank1 + two.rank2 + three.rank3 + four.rank4)/4 AS rank
FROM portal.object_to_keyword OK,
   portal.object_to_publication PUB,
--portal.object(description,object_name)
(SELECT object_id, object_type_id, object_name, development_status_id, availibility_id, description, ts_rank_cd
(to_tsvector('english', object_name || ' ' || description), query) AS rank1
FROM portal.object, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query
WHERE query @@ to_tsvector('english', object_name || ' ' || description) ) AS one,
--portal.keyword(name)
(SELECT name, keyword_id, ts_rank_cd(to_tsvector('english', name), query2) AS rank2
FROM portal.keyword, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query2
WHERE query2 @@ to_tsvector('english', name) ) AS two,
--portal.object_type(object_type_name)
(SELECT object_type_id, object_type_name, ts_rank_cd(to_tsvector('english', object_type_name), query3) AS
rank3
FROM portal.object_type, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query3
WHERE query3 @@ to_tsvector('english', object_type_name) ) AS three,
```

```
--portal.publication(title,journal)
(SELECT publication_id, title, journal, ts_rank_cd(to_tsvector('english', title || ' ' || journal), query4) AS rank4
FROM portal.publication, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query4
WHERE query4 @@ to_tsvector('english', title || ' ' || journal) ) AS four
WHERE (one.object_id = OK.object_id AND
   two.keyword_id = OK.keyword_id) OR
   (one.object_type_id = three.object_type_id) OR
   (one.object_id = PUB.object_id AND
   four.publication_id = PUB.publication_id)
GROUP BY one.object_name, one.rank1, two.rank2, three.rank3, four.rank4;
```

Query 3 with indexes

```
SELECT one.object_name, (one.rank1 + two.rank2 + three.rank3 + four.rank4)/4 AS rank
FROM portal.object_to_keyword OK,
   portal.object_to_publication PUB,
--portal.object(description,object_name)
(SELECT object_id, object_type_id, object_name, development_status_id, availibility_id, description, ts_rank_cd
(poindex, query) AS rank1
FROM portal.object, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query
WHERE query @@ poindex) AS one,
--portal.keyword(name)
(SELECT name, keyword_id, ts_rank_cd(to_tsvector('english', name), query2) AS rank2
FROM portal.keyword, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query2
WHERE query2 @@ pkindex) AS two,
--portal.object_type(object_type_name)
(SELECT object_type_id, object_type_name, ts_rank_cd(potindex, query3) AS rank3
FROM portal.object_type, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query3
WHERE query3 @@ potindex ) AS three,
--portal.publication(title,journal)
(SELECT publication_id, title, journal, ts_rank_cd(popindex, query4) AS rank4
FROM portal.publication, to_tsquery('english','lectin:* | cancer:* | data:* | Damodaran:* | Jeyakani:* | stable:* |
open:*') query4
WHERE query4 @@ popindex ) AS four
WHERE (one.object_id = OK.object_id AND
   two.keyword_id = OK.keyword_id) OR
   (one.object_type_id = three.object_type_id) OR
   (one.object_id = PUB.object_id AND
   four.publication_id = PUB.publication_id)
GROUP BY one.object_name, one.rank1, two.rank2, three.rank3, four.rank4;
```

APPENDIX C

USERS GUIDE

This chapter contains the instructions for performing regular search and advanced search in the GlycomicsPortal, and different search options.

Regular search

In the regular search, a user can simply type keywords in the search bar. An example is shown below.

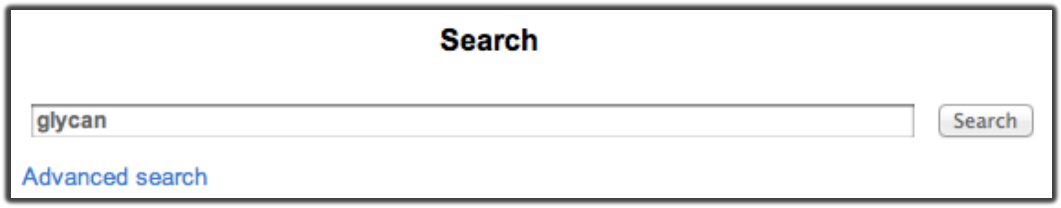


Figure 15: An example of regular search

The size of the user keyword is from 2 to 100 characters. If it is not, then an error message will be shown on the search page.

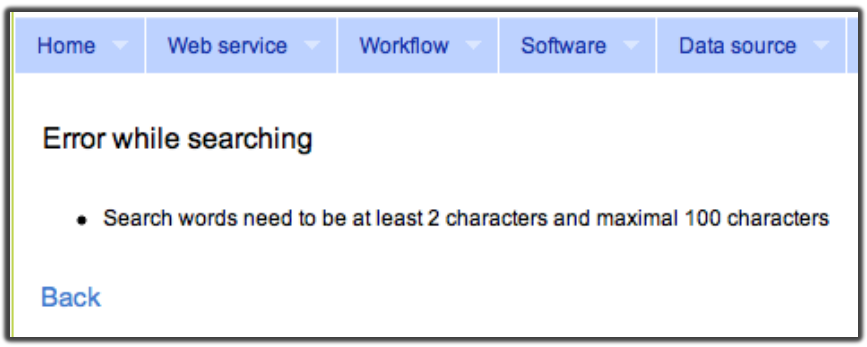


Figure 16: An error page in regular search

If there is no error when the user clicks the search button as shown below, simply search results will be shown on the same page.

Figure 17: Search button in regular search

Next, the user may see some results for the keywords or may encounter the empty page with no results. If there are no results, then please try to search with different keywords.

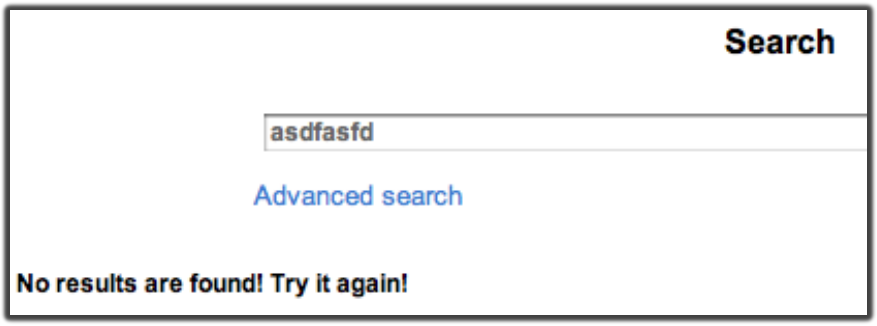


Figure 18: No results page in regular search

If not, then the user will see some results. An example is shown below.

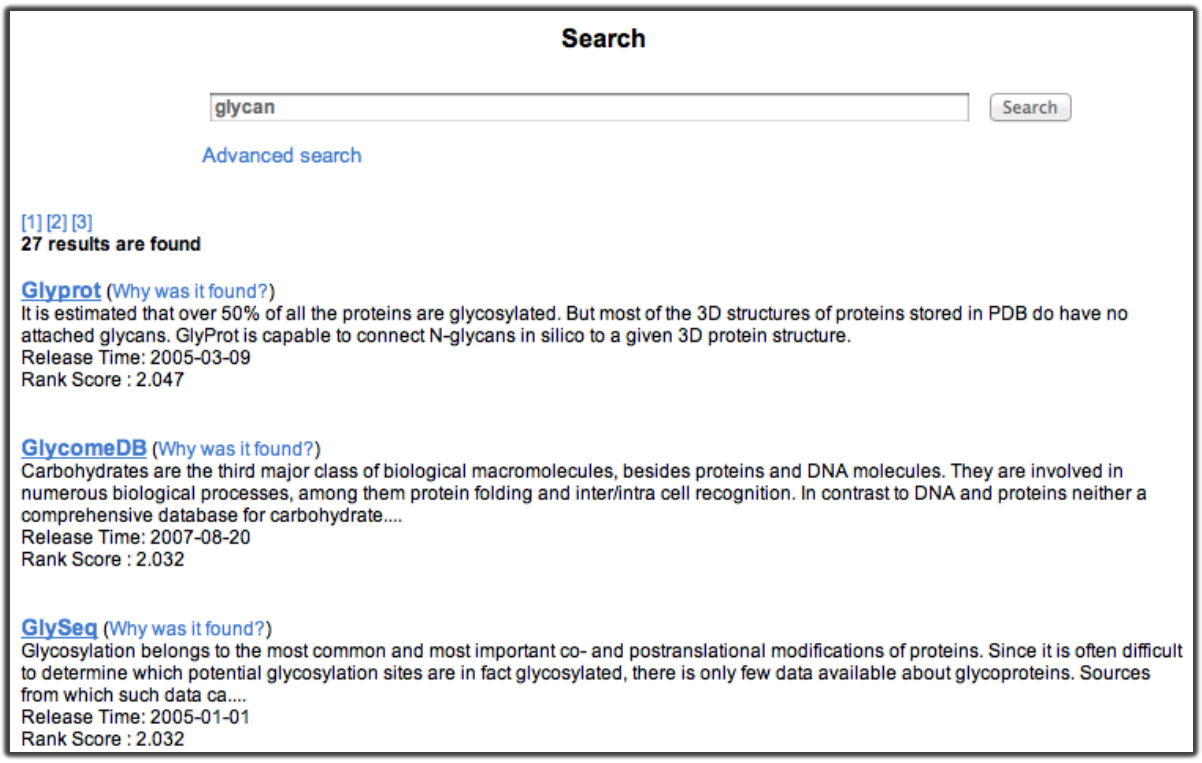
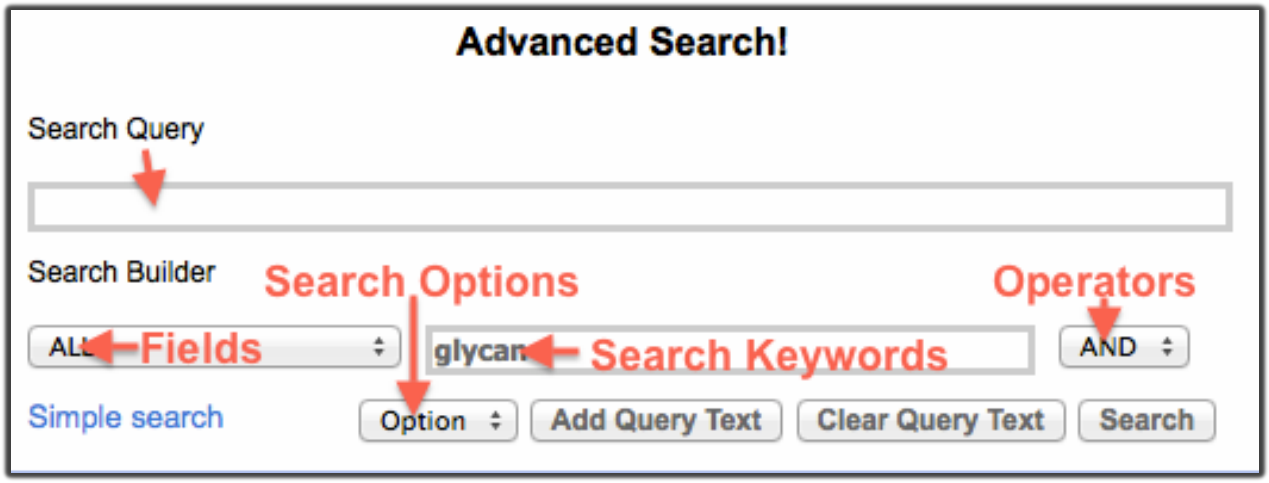


Figure 19: Results in regular search

Advanced search

In the advanced search, a user can specify fields, keywords, and operators. An example is shown below.



Search Query: where a query is shown based on a field, search keywords, and an operator.

Fields: where all fields' names are shown in the drop down menu.

Search keywords: user search keywords.

Options: Phrase search, exact search or no options

Operators: AND, OR, or NOT.

Add query text button: button to create a search query.

Clear query test button: button to clear current search query.

Search button: performs a search with a given query.

Figure 20: An explanation of advanced search

How to create a search query

1. Choose a field name and type search keywords. Operators are not used for the first search query.
2. And click the 'Add query text' button.
3. A new query is shown in the search query bar. An example is shown below.

Figure 21: A search query in advanced search

4. If you would like to add more queries, then repeat the number 1 through 3 again. An example is shown below.

Before the second query

After adding the second query

Figure 22: An example of expanding a search query in advanced search

Different search options

Phrase search

Phrase search can find an exact user input. It is an exact and ordered search.

Proximity search

Proximity search can find an exact user input, but it is not ordered.