RILCA: Collecting and Analyzing User-Behavior Information in Instant Search Using Relational DBMS

Taewoo Kim and Chen Li

University of California, Irvine, CA 92697, USA
{taewookim, chenli}@ics.uci.edu

Abstract. An instant-search engine computes answers immediately as a user types a query character by character. In this paper, we study how to systematically collect information about user behaviors when they interact with an instant search engine, especially in a real-time environment. We present a solution, called RILCA, which uses front-end techniques to keep track of rich information about user activities. This information provides more insights than methods based on traditional Web servers such as Apache. We store the log records in a relational DBMS system, and leverage the existing powerful capabilities of the DBMS system to analyze the log records efficiently. We study how to use a dashboard to monitor and analyze log records in real time. We conducted experiments on real data sets collected from two live systems to show the benefits and efficiency of these techniques.

Key words: Instant search, User-behavior collection, Log analysis

1 INTRODUCTION

Instant search is different from traditional search in that it immediately presents results as a user types in a query. In contrast, traditional search systems show results only when a user explicitly sends a complete query. For example, if a user wants to find information about *Michael Jackson* in an instant-search engine, as the user types in characters, say "`michael`", results will be returned even if the user does not finish typing the entire query. This search paradigm saves users not only typing efforts but also time. Google claims that their instant search saves users an average of two to five seconds per search [1].

It is well known that query log can be analyzed to not only monitor the query workload on the server, but also obtain rich information about user behaviors and intentions. In this paper we study how to collect information about user queries in instant search, and analyze the information effectively and efficiently. Compared to log records of traditional search engines, the records of instant search have several unique characteristics. First, in traditional search, each query received by the server has completed keywords. In instant search, every keystroke can generate a query and a log record. When doing log analysis, we cannot treat each log record as a complete query since it can have a query prefix. For example, suppose a user types in the query "`michael`" character by character. The process can generate multiple log records, namely "`m`", "`mi`", "`mic`", "`mich`", "`micha`", "`michae`", and "`michael`". As a consequence, log records of instant search have a larger volume, with many records corresponding to keyword prefixes. Second, due to unique characteristic above, it becomes necessary to detect the "boundaries" between different information needs. In other words, we want to answer the following question: which of these log records can be regarded as a complete query that a user intended to type in? Along this line, we want to detect sessions in the log records corresponding to different information needs of users.

In this paper we study how to tackle these problems with a solution called "RILCA," which stands for "Real-time Instant-search Log Collector and Analyzer." Fig. 1 shows the architecture of this approach. In RILCA, we have a separate server next to the search engine, and the purpose of this server is to collect rich information about user activities on the results from the search engine. The server is also used to analyze and visualize the collected log data.

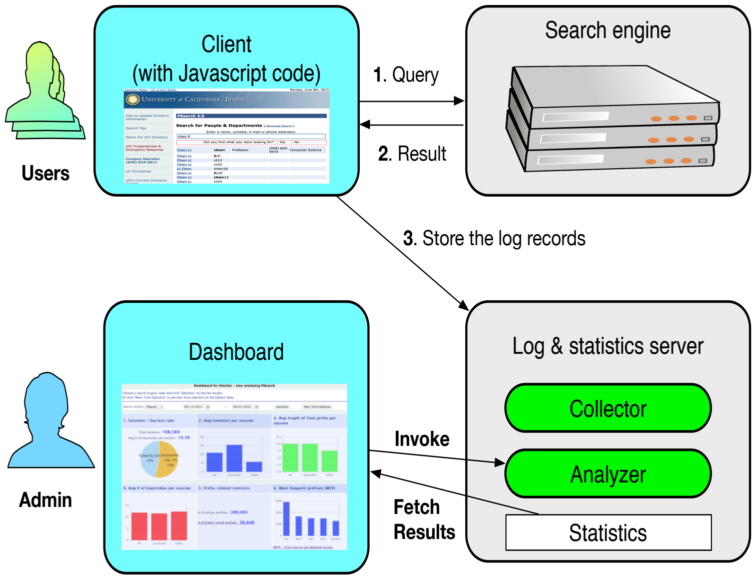


Fig. 1: Architecture of RILCA

Specifically, RILCA uses client-side techniques to collect information about user activities (Section 2), such as search results from the search engine, URLs clicked by the user, etc. Instead of using an ad hoc solution for storing and analyzing the collected log data, RILCA uses a relational DBMS to do the storage and analysis. We study how to use built-in capabilities of DBMS such as stored procedures to process and analyze log. In particular, we show how to use SQL to analyze log records to identify sessions, and the most important query in each session. This method can not only fully utilize existing indexing and query processing functionality of DBMS based on SQL, but also support real-time analysis on log records. We develop various optimization techniques to improve the efficiency. The analyzed results are displayed on a dashboard, which can be used by the system administrator to do real-time monitoring and analysis.

We have deployed RILCA to analyze two instant-search engines we developed in the past few years, namely PSearch [1] and iPubmed [2]. PSearch is a system that supports instant search on the directory at UC Irvine, with about 70,000 records. iPubmed is an instant-search system on more than 24 million medical publications licensed from MEDLINE. We report our experiences of using RILCA to do log collection and analysis on these two systems, and conduct experiments to evaluate the techniques.

1.1 Related Work

Collecting and analyzing log records in real time: Log records can be generated and collected through Web servers such as Apache [3]. These log records contain access information for the resource on a Web server. They also contain other information not relevant to the search, such as accessing Cascading Style Sheets (CSS), static HTML files, unrelated JavaScript files, or image files. The RILCA approach can collect additional information about user behaviors using Javascript. A similar, proprietary approach is taken by a company called Elicit Search [4], which collects user activities in real time by embedding a Javascript program on the Web server. This Javascript code sends information about user activities to a server that stores the data as log records, and analysis is done on the server. Our approach to collecting and analyzing log records in real time is similar. However, the internal structure and functionality of our method is different from that of Elicit Search, in that we collect other relevant event information, and we also use DBMS to do log analysis.
Instant Search Log Analysis: Log analysis of traditional search systems is a well-studied area [5, 6, 7, 8]. On the contrary, instant search log analysis is new. In [9], the authors showed how to analyze instant search log records by identifying sessions and query patterns. We extend the session-identification algorithm by collecting and using additional information that is relevant to user queries. Specifically, our focus is to develop a RDBMS-based method to identify sessions, the most important query in each session, and unique statistics about log records. We also focus on how to do optimization in this approach (Section 3).
Instant-fuzzy Search: Instant-fuzzy search is studied in [10, 11, 12]. The authors developed a solution by incorporating data structures, namely a trie, an inverted index, and a forward index. Traversing a trie with a fuzzy method is also described in [11, 12]. This method provides the combined effect of fetching certain prefixes with a fuzzy search and finding entries that start with the fetched prefixes.

2 COLLECTING USER SEARCH ACTIVITIES

In this section we discuss how RILCA collects information about an instant-search engine and its user behaviors using client-side techniques. We first explain the limitation

[1] psearch.ics.uci.edu
[2] ipubmed.ics.uci.edu

of log records in a traditional search server. When a client submits a request to a search engine, the search server keeps track of information such as the client's IP address, date, time, return status, size of returned object, and accessed URL. While this information is very valuable, it does not include the "big picture" of the user behavior, when the front end is interacting with multiple servers, including servers maintained by other organizations. For instance, in the iPubmed system, after a user types in a few characters, the server returns a list of publication results. After clicking a result, the user is directed to a page with detailed information about the publication. This page is generated by an external server maintained by NIH, which is beyond the control of our team. In other words, the user behaviors generate log records at multiple servers managed by different organizations, making it hard to do global information collection and analysis.

To overcome this limitation, RILCA uses client-side Javascript code to gather more information about user behaviors. As shown in Fig. 1, we embed a Javascript program in each Web page returned by the search engine. This step can be done by adding an HTML "<javascript>" tag in the page that indicates the path to the Javascript program residing on the RILCA server, so the change made on the search engine server is minimal. When the initial result page from the search engine is returned, the embedded program sends to the RILCA server a log record with relevant information, such as the search query and number of results. After that, this program monitors interesting user activities in the browser, such as copying a text string on the page and clicking a URL. For each event, the program sends a log record to the RILCA server. All these interactions with the server are happening "behind the scene," since they can be implemented using AJAX to minimize the impact on the user search experience.

Table 1: Comparison of search log records produced by Apache and RILCA

Apache (time, query)	RILCA (time, query)
20:19:32 imp	20:19:32.324 imp
20:19:32 impro	20:19:32.569 impro
20:19:32 improv	20:19:32.802 improv
20:19:32 improve	20:19:33.034 improve
20:19:33 improve pa	20:19:33.231 improve pa
... ...	... ...
20:19:34 improve patient complian	20:19:35.157 improve patient complian
20:19:35 improve patient compliance	20:19:35.395 improve patient compliance
20:19:39 improve mepatient compliance	20:19:40.263 improve mepatient compliance
20:19:40 improve medicapatient compliance	20:19:40.705 improve medicapatient compliance
20:19:40 improve medication patient compliance	20:19:40.970 improve medication patient compliance
(no more log records)	20:20:34.282 (clicked an outbound link: http://www.ncbi.nlm.nih.gov/pubmed/19182563)

Table 1 shows sample log records of Apache and RILCA in the iPubMed search engine [2]. The Apache log records do not indicate the user actions after receiving the queries. In the RILCA log records, we can know that the user clicked an outbound link that guided them to another Web site. This log record provides more insights about whether the query results included what the user was looking for. The log records also include the number of results for each query (not shown in the table).

3 ANALYZING LOG RECORDS

In this section, we study how to utilize the collected query log data in RILCA. We first present a dashboard that shows the statistics of user queries, which can help the system administrator monitor and understand the workload on the search engine server, as well as gain insights of the performance of the engine. The next question is how to generate the data for the dashboard efficiently. We study how to achieve the goal by leveraging the existing query capabilities of relational DBMS systems.

3.1 Dashboard for Monitoring and Analyzing Search Queries

It is important for the system administrator of the search engine to monitor the status of the engine and understand the search behaviors of users. Fig. 2 shows a dashboard interface that can achieve the goal. It allows the administrator to specify a time range, then analyze the query records collected during this period and visualize the statistical results. If the administrator clicks the "Real-time statistics" button, the system will run the computation tasks on the latest query records, such as the last one hour, 12 hours, or one day, and the time duration is configurable. Compared to dashboards of traditional search engines, this dashboard is unique in two aspects due to the instant queries from users: (1) it needs new solutions to identify a session in a search, and (2) it also shows statistics related to query prefixes.

The dashboard shows the statistics about sessions, time and typing effort spent per session, and query prefixes. Formally, a session is a sequence of keystroke queries by a user issued without a major interruption to fulfill a single information need. For example, suppose a user is looking for a person named `John Doe` by typing in the name character by character. After receiving the first seven letters, the system can find the related record, and the user clicks the corresponding link for this person. In this case, the log records of these keystrokes form a session, indicating the process of finding this relevant answer. A session is called "successful" if the system can find the answer the user was looking for.

The dashboard shows the number of sessions, number of successful sessions, and the number of unsuccessful sessions. These numbers tell the administrator how successful the engine is to find right answers expected by the users. The dashboard includes the average number of keystrokes, the average length of query strings, and the average duration per session. These numbers indicate how much time and effort a user spent to find answers. The dashboard also shows the number of unique prefixes and empty-result prefixes. The prefixes with empty results are very useful to reveal abnormal search behaviors of the engine. As an example, by using this feature for the PSearch system, we were able to easily detect a bug in the engine. In particular, we noticed that many empty-result queries had two spaces in their keywords. By doing an investigation, we

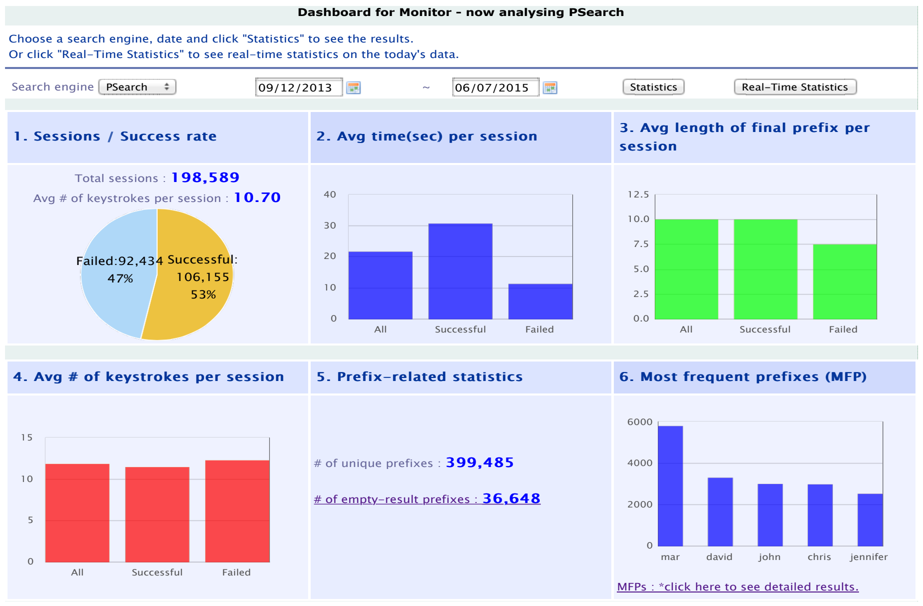

Dashboard for Monitor - now analysing PSearch
Choose a search engine, date and click "Statistics" to see the results.
Or click "Real-Time Statistics" to see real-time statistics on the today's data.
Search engine PSearch
09/12/2013 ~ 06/07/2015
Statistics
Real-Time Statistics
1. Sessions / Success rate
Total sessions : 198,589
Avg # of keystrokes per session : 10.70
Failed:92,434 47%
Successful: 106,155 53%
2. Avg time(sec) per session
3. Avg length of final prefix per session
All
Successful
Failed
4. Avg # of keystrokes per session
5. Prefix-related statistics
# of unique prefixes : 399,485
# of empty-result prefixes : 36,648
6. Most frequent prefixes (MFP)
mar
david
john
chris
jennifer
MFPs : *click here to see detailed results.

Fig. 2: Monitoring user queries using a dashboard

found that the client-side Javascript program had a bug related to how space was handled in keywords. This bug was fixed, and user search experience was improved. This example shows the value of this feature on the dashboard.

3.2 Generating Dashboard Statistics Using RDBMS

The next question is how to use the collected query log data to generate the statistics used in the dashboard. Instead of using ad-hoc solutions, RILCA uses relational database systems (RDBMS) to store and analyze the log data to achieve the goal. This approach can not only avoid the cost of duplicating the data at different places (e.g., the storage place and the analyzer module), but also leverage the powerful capabilities of RDBMS, including the SQL language, indexing, query processing and optimization.

Fig. 3 shows how we use an RDBMS to analyze query log records to generate tables with statistical information. A module called "Collector" receives log records from clients, and stores the data into a table called "Log records." The following shows the schema of this table. It has a primary key ("logno"), IP address, agent (browser), and cookie id of the client. It also stores the access time that includes the date time and millisecond time. The "operation_type" shows the type of user activity, such as search request, click on a URL, or copy on the result page. More details of the operation are stored in other attributes in the table.

```
CREATE TABLE LogRecord (
  logno int unsigned NOT NULL AUTO_INCREMENT,
  client_ip varchar(15) NOT NULL,
```

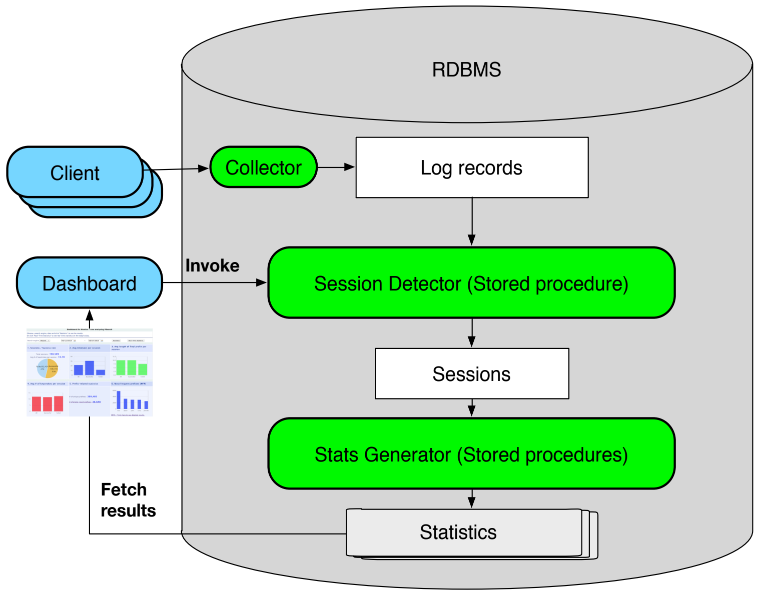


Fig. 3: Log analysis flow using a RDBMS

```
client_agent varchar(200) DEFAULT NULL,
client_cookieid varchar(50) DEFAULT NULL,
access_datetime datetime NOT NULL,
access_mstime smallint unsigned NOT NULL DEFAULT '0',
operation_type tinyint unsigned NOT NULL DEFAULT '0',
operation_detail varchar(200) DEFAULT NULL,
query_string varchar(200) NOT NULL,
PRIMARY KEY (logno) );
```

These records are processed by the “Session Detector” module to find sessions from the log records, and store the results in the “Sessions” table. These session records are further processed by the “Stats Generator” module to obtain statistical information to be used by the dashboard. In our implementation of RILCA on the two live systems, we used 10 stored procedures and 6 functions for the analysis process.

3.3 Identifying Sessions Using SQL

We will focus on how to use SQL stored procedures to identify sessions from the raw query log records. We also need to detect whether a session was successful. There have been many studies on this topic (e.g., [9, 13]). The main difference in this work is how to use SQL to analyze records with query prefixes, and understand whether those queries were successful or not. Specifically, we mainly consider newly available, rich information in the log records about user activities, such a click on a URL, copying a text from the result page, or closing/opening a new tab of a browser. In our analysis, we view a session as successful when the user clicked a link in the results, copied a text in the current search results, or closed the tab within a certain amount of time after the user entered the last query in the session.

Fig. 4 shows the sketch of this process as a SQL stored procedure. It first declares a cursor on a query that sorts the log records in an ascending order based on the client IP and access time, so that we can process the records in the order they were received per IP. For each log record, we compare it with the previous one. A new session is started when the current IP address is different from the previous IP address, the time difference between the current record and the previous one is more than five minutes, or the current record is opening a new tab in the browser (boxes 1 to 3). Otherwise, a new session is still considered to be started if the current record is a search request and the previous record was not (box 4), or the current and previous records are both search requests with dissimilar query strings based on edit distance (box 5). After detecting a new session has started, we assign a new session ID and insert a new session record into the "Sessions" table.

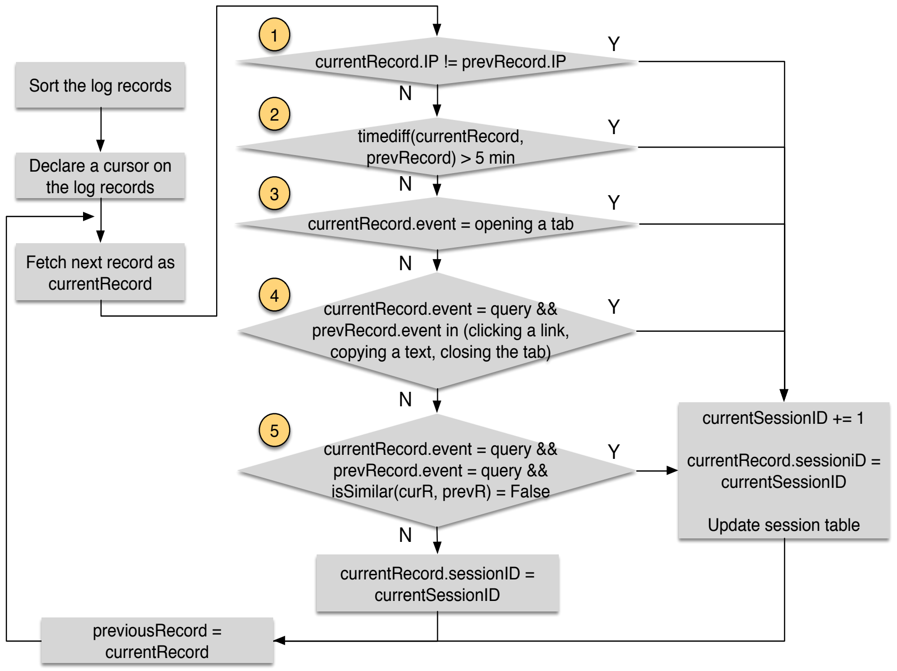

Sort the log records
Declare a cursor on the log records
Fetch next record as currentRecord
1
currentRecord.IP != prevRecord.IP
2
timediff(currentRecord, prevRecord) > 5 min
3
currentRecord.event = opening a tab
4
currentRecord.event = query && prevRecord.event in (clicking a link, copying a text, closing the tab)
5
currentRecord.event = query && prevRecord.event = query && isSimilar(curR, prevR) = False
Y
N
currentSessionID += 1
currentRecord.sessioniD = currentSessionID
Update session table
currentRecord.sessionID = currentSessionID
previousRecord = currentRecord

Fig. 4: Detecting sessions from log records using a stored procedure

The following is part of the stored procedure that includes the cursor declaration and **IF** statements that are used to identify new sessions.

```
DECLARE cursor1 CURSOR FOR
```

```
SELECT logno, ... FROM LogRecord WHERE
TIMESTAMPDIFF(SECOND, clientaccesstime, currenttime)
<= 500 ... ORDER BY client_ip, clientaccesstime logno;

FETCH cursor1 into c_logno, c_accesstime, ...

// #1. If the IP address of the current log record and
// the previous log record do not match: a new
// session has started
IF (c_client_ip != pre_client_ip) THEN
SET startnewsession = 1; ...

// #2. If the time difference between the current log
// record and the previous log record is greater than
// 5 minutes: a new session has started
ELSEIF ( timestampdiff(SECOND, pre_accesstime,
c_accesstime) > 300 ) THEN
SET startnewsession = 1; ...

// #3. If the current operation is opening a new tab:
// a new session has started
ELSEIF ( c_operationtype = 5 ) THEN
SET startnewsession = 1; ...

// #4. If the previous operation is in (clicking a
// link, copying a text, closing the tab) and the
// current operation is querying: a new session has
// started
ELSEIF ( (pre_operationtype = 2 OR pre_operationtype = 3
OR pre_operationtype=4) AND c_operationtype=1 ) THEN
SET startnewsession = 1; ...
```

3.4 Optimizing SQL Queries

Our experiences on the two real systems suggested that a simple implementation of the stored procedures can be slow, especially for large amounts of log records. To solve the problem, we develop several optimization techniques to improve the performance.

– *Splitting tables*: For those log records that are already processed by the session detector, we move them into a separate table. In this way, we can reduce the size of the table that is used to store newly arrived records and detect sessions. Similarly, the sessions of those processed log records are also moved to a separate table. This approach is similar to the idea used in Apache that stores query log records into different files.
– *Using the* `MEMORY` *storage for the "Sessions" table*: In this way, this table can be stored in memory for fast access and low latency.

– *Database tuning*: When detecting sessions, initially we need to declare a cursor by sorting a large number of log records based on their IP address and access time. The default DBMS setting may not be efficient for such queries. We can solve the problem by tuning system parameters such as cache size, buffer-pool size, log file size, and sort-buffer size, which can greatly affect the performance of the sorting process.

After identifying each session, we extract the final query as the most important query in the session, since a user stopped typing in more keywords after that. We also add this information to the session-related statistics. According to [9], 61% of PSearch [14] log records show a so called "L-shaped pattern" such as "`c`", "`ca`", "`can`", "`canc`", "`cance`", and "`cancer`". In this L-shaped pattern, a user gradually creates a new query by adding one more character to the previous query. Therefore, we can choose the most important query in the session by selecting the final query. There can be some sessions where the length of the intermediate query is greater than that of the final query. We choose the final query as the most important query in the session, since the user did not stop querying after checking the results of the intermediate query.

4 Experiments

In this section, we present experimental results of RILCA on the two live instant search systems. Table 2 shows the data sets. The MEDLINE data set had information about 24 million medical publications. We extracted about 1.9 million records and used them for the experiments. We also collected log records from the live iPubMed [2] Web server. We first deployed the Javascript code on iPubMed in September 2013, and since then we collected additional log data from the RILCA server as well as Apache log records. The PSearch data set contained information about the UCI directory including their name, e-mail address, department, and office phone number, all of which were open to the public. The backend search engine supports instant, error-tolerant search. We collected log records by embedding a Javascript program on its returned pages. In addition, we also had log records from its Apache server.

Table 2: Two data sets

Data Set	**MEDLINE**	**PSearch**
Number of query log records	1,004,886	1,026,435
Number of sessions	65,260	102,743
Average character number per query	19.9	9.6
Average word number per query	2.8	1.7

For both systems, we installed a MySQL database on the server to store and analyze log records. All experiments were done on a server with 94GB of RAM and four Intel Xeon CPUs. Each CPU had six cores with a clock speed of 2.93GHz.

4.1 Generating Query Workloads

To measure the performance of the log collector and session detector, we simulated an environment with multiple concurrent users. We set the number of concurrent users to be 5, 10, 15, and 20, respectively. We used Jmeter to simulate a user who sent about 14,000 queries sequentially. We used the real PSearch log records collected by RILCA. Table 3 shows the setting. Note that even though the number of concurrent users was between 5 and 20, the environment inserted 200 to 500 queries per second with a high insertion rate. For the live PSearch query session, on average each session had 11.64 keystroke queries with a duration of 20.56 seconds. Each user on average sent 0.6 queries per second, which corresponds to the case where 20 users inserted 12 queries per second, not 496 queries per second in the simulation setting. Thus, we can say that the simulation environment was approximately equivalent to a situation where there were 800 concurrent users in the live system.

Table 3: Query Workload for PSearch

User Number	Query Number	Duration	Inserted Record Number per Second
5	70,556	352 sec	200
10	141,893	400 sec	354
15	213,598	535 sec	399
20	283,941	572 sec	496

4.2 Log-Collection Overhead on Search Time

We evaluated the overhead of RILCA by measuring the execution time of a query when the number of concurrent users increased from 15 to 20. The purpose of this experiment was to check if RILCA can introduce a significant amount of overhead to a search process. As we can see in Fig. 5, the performance degradation of the search process was small. For instance, when there were 15 concurrent users, RILCA increased the average search time from 215ms to 225ms. This low additional overhead is due to the fact that RILCA uses AJAX to do asynchronous communication with its server. The search time was a bit high because of the simulated environment with many concurrent users. We observed a similar minor time increase in the setting where the number of users is smaller and the search time was lower.

4.3 Scalability of Log Analyzer

Fig. 6 shows the time of the query analyzer (including the session detector and statistics generator) on different numbers of log records. (Note that we applied the optimizations in the experiments, and the effect of each optimization will be covered in the following subsection.) As the figure shows, when the number of query records increased, the total analyzer time also increased linearly, with a constant initial cost. For instance, when there were 82,500 log records with 10 concurrent users, it took about 6.7 seconds to analyze these records.

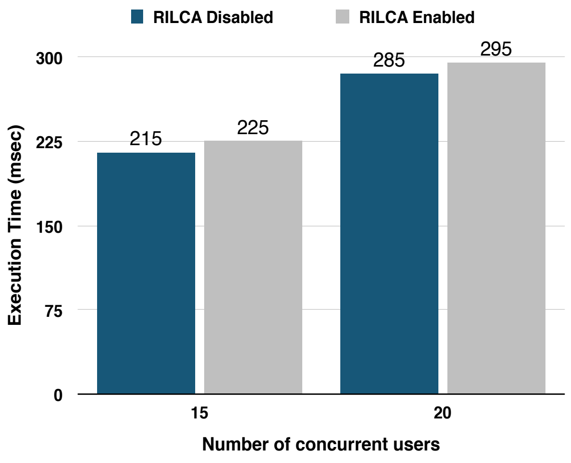
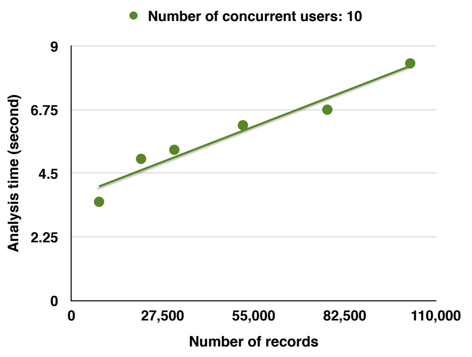
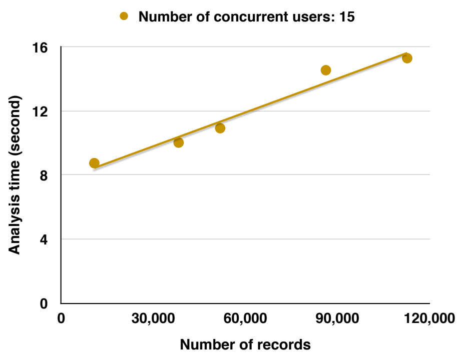


Fig. 5: RILCA overhead on search performance



(a) Number of concurrent users: 10



(b) Number of concurrent users: 15

Fig. 6: Average analysis time and throughput from the stress test

4.4 Effect of Optimization Techniques

Performance Improvement with RDBMS Tuning: We did an experiment to show the effect of DBMS tuning. We first used the default MySQL setting. In this case, when the number of concurrent users was 5, it took 376 seconds to analyze 17,000 log records. After adjusting a few database parameters (cache size, buffer-pool size, log-file size, and sort-buffer size), the analysis time decreased to 27.8 seconds to analyze 20,000 log records. Table 4 shows the results of this optimization.

Performance Improvement with In-Memory Table: We evaluated the effect on the analysis performance by declaring the "Sessions" table in memory. As shown in Fig. 7, this approach can reduce the analysis time significantly. For example, for 100K log records,

Table 4: Performance improvement by DBMS parameter tuning

Content	Before optimization	After optimization	Improvement
Average analysis time (msec)	22	1.3	20.7
Throughput (record/sec)	45	720	675

it took 134,000ms to do the log analysis, while the time decreased to 33,570ms after we declared the table to be in memory, with a reduction about 75 percent.

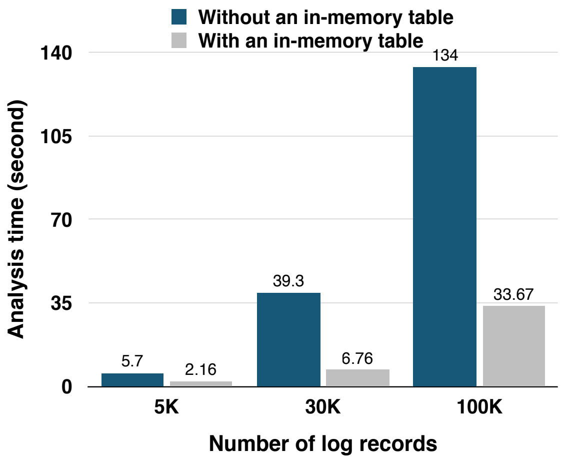


Fig. 7: The benefits of using an in-memory “Session” table

4.5 Comparison of Two Systems

We use RILCA to analyze the results collected from the two live systems in the past two years, and the results are shown in Table 5. As we can see, the average duration of a session on iPubMed was larger than that in PSearch. The difference is due to the fact that the average length of a query for iPubMed was nearly twice the size of that for PSearch.

5 CONCLUSIONS

In this paper, we studied how to collect information about user queries in instant search. In our proposed approach called RILCA, we embed a Javascript program in the result page of the search engine to collect user activities on the client side. Then we studied how to use a relational database system to store this information and analyze the log records. In particular, we showed how to leverage the querying capabilities of RDBMS

Table 5: Statistics about two search engines

Content	**iPubMed**	**PSearch**
Duration	9/'13 - 5/'15	9/'13 - 5/'15
# of sessions	4,347	197,814
# of successful sessions	649	105,726
# of unsuccessful sessions	3,698	92,088
Avg. number of queries per session	12.26	10.70
Avg. duration of a session (seconds)	56.65	21.70
Avg. time per each query (seconds)	3.58	0.63
Avg. # of keystrokes per successful session	22.98	11.45
Avg. length of the final query in a session	21.55	7.50
Avg. number of keystrokes per session	26.29	11.83
# of queries that generated no search results	5,172	36,536
# of unique queries	25,461	398,417
Most popular keywords whose length is equal to or greater than 3	cancer, practical clinical competence, brain	david, john, chris, jennifer

to detect sessions and produce insightful statistics that can be displayed on a dashboard, which can be used by a system administrator to do real-time monitoring of the server. We discussed several techniques to improve the performance of this analysis task. Our experiments on the real data sets collected from two live instant-search systems demonstrated the effectiveness and efficiency of these techniques.

References

1. Search: now faster than the speed of type, http://googleblog.blogspot.com/2010/09/search-now-faster-than-speed-of-type.html
2. iPubMed Search, http://ipubmed.ics.uci.edu
3. Apache, http://httpd.apache.org/
4. Elicit Search, http://elicitsearch.com/
5. Jansen, B.: Search log analysis: What it is, what's been done, how to do it. In: Library & information science research 28.3, pp. 407–432. (2006)
6. JANSEN, B.: The methodology of search log analysis. In: Handbook of research on Web log analysis, pp.99–121. (2008)
7. Anderson, J.: Analyzing Clickstreams Using Subsessions. In: DOLAP , pp.25–32. (2000)
8. Dogan, R., Murray, G., Neveol, A. Lu, Z.: Understanding PubMed user search behavior through log analysis, In: Database 2009 (2009)
9. Cetindil, I., Esmaelnezhad, J., Chen, L., Newman, D.: Analysis of instant search query logs. In: WebDB, pp. 7-12 (2012)
10. Cetindil, I., Esmaelnezhad, J., Kim, T., Li, C.: Efficient instant-fuzzy search with proximity ranking. In: ICDE, pp. 328-339. (2014)
11. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: WWW, pp.371-380. (2009)

12. Li, G., Wang, J., Li, C., Feng, J.: Supporting efficient top-k queries in type-ahead search. In: SIGIR, pp.355-364. (2012)
13. Silverstein, C., Marais, H., Henzinger, M., Moricz, M.: Analysis of a very large web search engine query log. SIGIR Forum 33(1), pp.6-12. (September 1999)
14. Psearch, http://psearch.ics.uci.edu
15. MEDLINE Data, http://www.nlm.nih.gov/bsd/licensee/medpmmenu.html
16. IMDB Data, http://www.imdb.com/interfaces