6-2012

Dash: A Novel Search Engine for Database-Generated Dynamic Web Pages

Ken C. K. LEE
University of Massachusetts Dartmouth

Kanchan BANKAR
University of Massachusetts Dartmouth

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Chi-Yin CHOW
City University of Hong Kong

Honggang WANG
University of Massachusetts Dartmouth

DOI: https://doi.org/10.1109/ICDCS.2012.53

Dash: A Novel Search Engine for Database-Generated Dynamic Web Pages

Ken C. K. Lee[1] ken.ck.lee@umassd.edu
Kanchan Bankar[1] kbankar@umassd.edu
Baihua Zheng[2] bhzheng@smu.edu.sg
Chi-Yin Chow[3] chiychow@cityu.edu.hk
Honggang Wang[4] hwang1@umassd.edu

[1] Department of Computer & Information Science, University of Massachusetts Dartmouth, USA
[2] School of Information Systems, Singapore Management University, Singapore
[3] Department of Computer Science, City University of Hong Kong, Hong Kong
[4] Department of Electrical & Computer Engineering, University of Massachusetts Dartmouth, USA

***Abstract*—Database-generated dynamic web pages (db-pages, in short), whose contents are created on the fly by web applications and databases, are now prominent in the web. However, many of them cannot be searched by existing search engines. Accordingly, we develop a novel search engine named *Dash*, which stands for Db-pAge SearcH, to support db-page search. Dash determines db-pages possibly generated by a target web application and its database through exploring the application code and the related database content and supports keyword search on those db-pages. In this paper, we present its system design and focus on the efficiency issue.**

To minimize costs incurred for collecting, maintaining, indexing and searching a massive number of db-pages that possibly have overlapped contents, Dash derives and indexes *db-page fragments* in place of db-pages. Each db-page fragment carries a disjointed part of a db-page. To efficiently compute and index db-page fragments from huge datasets, Dash is equipped with MapReduce based algorithms for *database crawling* and *db-page fragment indexing*. Besides, Dash has a *top-k search algorithm* that can efficiently assemble db-page fragments into db-pages relevant to search keywords and returns the k most relevant ones. The performance of Dash is evaluated via extensive experimentation.

Keyword: Database-Generated Dynamic Web Pages, Search Engine, Database Crawling, Indexing, Top-k Search, MapReduce, Hadoop and Performance.

I. Introduction

In the ever-growing World Wide Web (or the web, for brevity, hereafter), search engines are indispensable facilities enabling individuals to find web pages of interest efficiently. However, as estimated in 2000, 550 billion web pages were not reachable by search engines [23]. Compared with 8.41 billion web pages collected and indexed by search engines as recorded in mid-March 2012 [24], only a very small fraction of web pages are really searchable! Even worse, those unsearchable web pages should have been blooming along with the growth of the entire web over the past decade.

Among various unsearchable web pages, *database-generated dynamic web pages* (abbreviated as *db-page*s, hereafter) are the majority [23]. Many present websites for e-commerce, e-learning, social networking services, etc., are all database driven that hosted web applications generate db-pages based on backend databases upon receiving input query strings such as HTML form submissions [22]. We use Example 1 (our running example) to exemplify query strings and db-pages.

Example 1. (Query string and db-page) Suppose that in www.example.com, a web application Search lists local restaurants according to search criteria in given query strings. For a query string '$\mathsf{c} = \mathsf{American}\&\mathsf{l} = 10\&\mathsf{u} = 15$', it generates db-page P1 about restaurants whose cuisines are American and average budget ranges between \$10 and \$15 per person as well as their customer comments. P1's content and URL, i.e., appending the query string to Search's URI (i.e., `www.example.com/Search`) are shown in Figure 1(a).[1]

`www.example.com/Search?c=American&l=10&u=15`

Restaurants	Budget	Rate	User comments
Burger Queen	\$10	4.3	Burger experts by David on 06/10
Wandy's	\$12	4.1	
Wandy's	\$12	4.2	Unique burgers by Bill on 05/10; Bad fries by Bill on 06/10

(a) P1's URL and content

`www.example.com/Search?c=American&l=10&u=20`

Restaurants	Budget	Rate	User comments
Burger Queen	\$10	4.3	Burger experts by David on 06/10
Wandy's	\$12	4.1	
Wandy's	\$12	4.2	Unique burgers by Bill on 05/10; Bad fries by Bill on 06/10
McRonald's	\$18	2.2	Regret taking it by David on 06/10

(b) P2's URL and content

Fig. 1. Example db-pages P1 and P2 and their URL

Other than P1, Search can generate many db-pages. For instance, it generates db-page P2 that lists American restaurants with a budget range between \$10 and \$20 per person and customer comments for another query string of '$\mathsf{c} = \mathsf{American}\&\mathsf{l} = 10\&\mathsf{u} = 20$', as in Figure 1(b). ■

In the following, let us first discuss the challenges faced in db-page *collection*, *indexing*, and *search*, which are the three most essential activities of search engines [9].

Conventionally, search engines discover static web pages by traversing their hyperlinks. Nevertheless, many db-pages are not linked by any others. As such, collecting db-pages is the first and important challenge to search engines. There are two approaches used by existing search engines to explore some db-pages. First, search engines may collect cached db-pages, which are generated for some query strings, from the caches of web proxies and web servers [8], [17]. Second, search engines may submit as many trial query strings as

[1] Some query strings are provided in HTTP requests through POST method. Here, we consider a query string as a part of an URL, i.e., GET method, but Dash can support both GET and POST methods.

possible to web applications to generate db-pages [19]. As in Example 1, query strings to Search may be formed by filling in c, l and u fields with some possible values. However, these two approaches cannot guarantee the completeness of collected db-pages. Besides, the second approach has to invoke web applications, which may generate many valueless db-pages, e.g., empty pages, error messages, and pages with identical contents. In addition, both websites hosting web applications and search engines will be easily exhausted by such overwhelming web application invocations.

Once web pages are retrieved via web page collection, their contents need to be indexed to facilitate search functions. Intuitively, each db-page, if collected, can be trivially treated as an independent web page. However, many db-pages, even generated by different query strings, would share similar (or even identical) contents. This is mainly because they are generated from some common records in a database. Handling numerous content-overlapped db-pages clearly increases index storage and access overheads. Thus, this intuitive approach is not efficient. What's worse, those content-overlapped db-pages are very likely to be relevant to queried keywords and returned as search results altogether, considerably deteriorating the quality of search results. As in Example 1, db-pages P1 and P2 have similar contents; and specifically, P1 is totally covered by P2. With respect to a queried keyword "burger", both P1 and P2 are qualified and returned. As no additional content provided by P2 (relative to P1) contains "burger", P2 would be interpreted as redundant in presence of P1 in the same search result.

Motivated by the lack of search engines to support the efficient retrieval and search of db-pages, we, in this paper, introduce a novel search engine named *Dash*, which stands for db-page search. Specialized for db-pages, Dash suggests URLs that refer to web applications and includes query strings for the applications to generate db-pages relevant to queried keywords. To address the aforementioned challenges, Dash is developed based on several innovative ideas, making its design and implementation very *unique*.

To effectively and efficiently collect db-pages and corresponding query strings, Dash directly explores the implementation of the web applications and the contents of underlying databases. Specifically, Dash performs reverse engineering on db-page generations. It first analyzes a given web application to determine how it accesses an underlying database to generate db-pages, with the help of some advanced software engineering techniques, such as data flow analysis [15] and symbolic execution [16]. As will be discussed later, the logic of a web application can be roughly formulated as a *parameterized query*, based on which query strings and db-page contents for the application can all be deduced. It is noteworthy that the assumptions about the availability of web application implementations and their database accesses to search engines are feasible, realistic and practical. First, many corporates nowadays rely on search engines to advertise their web pages in the Internet and they are even willing to permit those search engines to access their databases [12]. Second, only a portion of data in a database is needed to be accessed and it is expected to be publicized through db-pages generated by web applications. Third, many websites can explore their web applications and databases to determine all their db-pages as for their own keyword search. As opposed to some off-the-shelf embedded search engines, e.g., Google Custom Search [11], Apache Lucene [2], etc., that are designed for static web pages, Dash can be a tool for those websites to quickly offer keyword search on their own db-pages.

Besides, Dash exploits the concept of *db-page fragments*, i.e., indivisible parts of any db-page, to address the issue of potential content overlaps among different db-pages. Instead of generating all possible db-pages whose contents may overlap, Dash derives *disjoint* db-page fragments from an underlying database.[2] By deriving, storing and searching those db-page fragments, Dash effectively avoids processing and storing overlapped db-page contents. Accordingly, Dash is designed to manipulate db-page fragments for keyword search. To derive numerous db-page fragments from a huge dataset, Dash has MapReduce based algorithms to derive and index db-page fragments, respectively. In addition, a specialized index called *fragment index* is devised to index individual fragments according to their contents and their inter-relationship. Based on the index, a *top-k search algorithm* can synthesize k db-pages most relevant to queried keywords and suggest their URLs at the search time. The efficiency of those algorithms was evaluated via extensive experimentation. In our experiments, the top-k search took less than 0.3 millisecond to form and return relevant db-pages, which well indicates the feasibility of using db-page fragments to support db-page search.

The details about Dash will be presented in the rest of the paper, which is organized as follows. Section II reviews background and related work. Section III provides the preliminaries. Section IV provides the overview of Dash. Section V presents Map-Reduce based algorithms for database crawling and db-page fragment indexing. Section VI discusses top-k search algorithm. Section VII evaluates Dash performance. Section VIII concludes this paper, summarizes our contributions and states our future work.

II. Background and Related Works

This section reviews TF/IDF and inverted files, Google's MapReduce paradigm, and keyword search in relational databases, which are related to this research, and discusses the differences between those and Dash.

TF/IDF and Inverted Files. To determine web pages relevant to certain queried keywords, TF/IDF scheme [5] and inverted file [27] are commonly adopted by existing search engines.[3] With TF/IDF scheme, the relevance of a web page p to a set of queried keywords W is measured by its TF/IDF score, denoted by $\mathsf{TF\text{-}IDF}_W(p)$, as the sum of products of term frequency ($\mathsf{TF}_w(p)$) and inverse document frequency

[2]The concept of db-page fragments was also adopted in [7] for db-page content generation and management.

[3]To date, several TF/IDF variants and some other metrics, e.g., pagerank [21], are used to determine documents' weights.

rid	name	cuisine	budget	rate
001	Burger Queen	American	10	4.3
002	McRonald's	American	18	2.2
003	Wandy's	American	12	4.1
004	Wandy's	American	12	4.2
005	Thaifood	Thai	10	4.8
006	Bangkok	Thai	10	3.9
007	Bond's Cafe	American	9	4.3

restaurant(rid, name, cuisine, budget, rate)

cid	rid	uid	comment	date
201	001	109	Burger experts	06/10
202	004	132	Unique burger	05/10
203	004	132	Bad fries	06/10
204	002	109	Regret taking it	06/10
205	006	180	Thai burger	08/11
206	007	171	Nice coffee	01/11

comment(cid, rid, uid, comment, date)

uid	uname
109	David
120	Ben
132	Bill
171	James
180	Alan

customer(uid, uname)

Fig. 2. fooddb database

($\mathsf{IDF}_w(p)$) values with respect to individual keywords w in W, i.e., $\sum_{w \in W} \mathsf{TF}_w(p) \times \mathsf{IDF}_w$. A web page receives a relatively high TF/IDF score if it contains many queried keywords and/or those included keywords are not common to other web pages.

Inverted file, on the other hand, facilitates searches for web pages with high TF/IDF values by indexing web pages against their contained keywords. Its index structure is a collection of multiple inverted lists. Corresponding to a keyword w, an inverted list L_w maintains (the URLs of) those web pages that contain w and sorts them in descending order of their TF values with respect to w. As such, IDF_w can be quickly computed as an inverse of the number of web pages indexed in L_w; and web pages with higher TF values on w can be retrieved from an initial part of L_w.

TF/IDF scheme and inverted file are developed based on an assumption that all web pages are independent. This assumption, however, is invalid for db-pages as their contents may be derived from common records in backend databases. Thus, in Dash, a slightly modified relevance score function and a new index scheme to support keyword search based on db-page fragments are derived, as will be detailed later.

MapReduce Paradigm. Google's MapReduce paradigm [10] (or MR for simplicity) was introduced to provide scalable distributed data processing in clusters of commodity computers. Recently, various open-source MR implementations have been released, e.g., Hadoop [1] initially as a part of Nutch search engine [3]. Inspired by the *map* and *reduce* functions in functional programming, MR partitions and distributes data in computers (i.e., nodes) in a cluster, and performs a job in parallel among nodes in map phase and reduce phase. In the map phase, each participating node parses a segment of input data and generates intermediate results. Next in the reduce phase, each node gathers and processes intermediate results and delivers final results. In MR, all input data, intermediate results and output results are in form of (key,value) pairs.

As an example, let us consider how MR can create an inverted file over a collection of documents. Suppose all the documents presented as $\langle id, text \rangle$ pairs, where id and $text$ are the ID and content of a document, respectively. In the map phase, every node extracts keywords from $text$ from a $\langle id, text \rangle$ pair and generates intermediate results as $\langle w, \mathsf{TF}_w(id) \rangle$ pairs, where $\mathsf{TF}_w(id)$ is the term frequency of a keyword w in $text$ of the document referred by id. Next, in the reduce phase, each node gathers and sorts all $\mathsf{TF}_w(id)$ for the same word w; and it outputs each result inverted list L_w as $\langle w, \big(\mathsf{TF}_w(id_1), \mathsf{TF}_w(id_2) \cdots \mathsf{TF}_w(id_{|L_w|})\big) \rangle$.

In MR, communication and I/O costs are important performance factors and they need to be minimized for high processing efficiency. Thus, the design of MR applications as a workflow of MR jobs is critical to performance. As will be discussed later, we devise MR based algorithms for database crawling and fragment indexing; and smart rearrangement of operations can boost the processing efficiency.

Keyword Search in Relational Databases. In addition to web pages which are categorized as unstructured data, keyword search on semi-structured data (e.g., XML documents [13]) and structured data (e.g., relational databases) has started to receive attention from research community and industry. We review works on keyword search in relational databases below.

In relational databases, information is stored and accessed as the attribute values of records in multiple relations, as a result of database normalization [6]. Despite many database systems support string comparison functions, e.g., LIKE and CONTAINS, in their queries, it is not so straightforward to locate (joined) records that contain queried keywords in any of their attributes; and some recent research studies [14], [18], [20], [26] have been conducted to tackle this issue. Their common idea is to (i) locate records whose attributes contain any queried keywords, and then (ii) join those records as long as they are linked through referential constraints.

To illustrate the idea, let us consider fooddb database as depicted in Figure 2. Here, rid, cid, and uid are the primary keys in three relations restaurant, comment, and customer, respectively; and comment has two foreign keys rid and uid referencing restaurant and customer, respectively. Given an example queried keyword "burger", all records that contain "burger" are identified, i.e., record 001 in relation restaurant and records 201, 202, and 205 in relation comment. Then, those linked through foreign keys are joined, and we obtain following three result records.

1) "205, 006, 180, Thai burger, 08/11" (from comment),
2) "202, 004, 120, Unique burger, 05/10" (from comment), and
3) "001, Burger Queen, American, 10, 4.3, 201, 001, 109, Burger experts, 06/10" (from restaurant $\bowtie$ comment). [4]

These existing approaches could return some useful results but they have several obvious defects, which limit their practicality. First, they cannot provide a complete view of searched information. Refer to the example. The first two result records do not come with any restaurant information, since their corresponding restaurant records *do not* contain "burger". Also, some uninterested values, e.g., rid, uid, etc., are included. As a result, these output records are uneasy for

[4] We omit the result record schemas to save space.

general users to interpret. Likewise, the three result records do not mention customers who make the retrieved comments. Second, different queries need to issue to retrieve candidate records from one or multiple relations individually. Such query evaluation can take a long processing time, especially when many records contains queried keywords.

Google Search Appliance [12] performs keyword search in a single relation, which may be derived from other relations. Then, all the attribute values of each record in the relation collectively resemble a document. It guarantees all attribute values are included and saves join cost at the search time. Continue the example. A derived relation is formed as restaurant outer-joined with comment and customer. Then, for the same queried keyword "burger", both restaurant information and customer names are included in a search result.

However, all these works are just designed to retrieve separate (joined) records with queried keywords but not deriving information from groups of records in the same relation. For example, in our fooddb database, "Wandy's" has two comments "Unique burger" and "Bad fries". However, as two records, they are retrieved independently; and either one can only be shown in some cases. Differently, Dash is designed to support keyword search on db-pages that are derived based on a collection of database records according to web applications. Those db-pages should be in more user understandable form than raw database records.

III. Preliminaries

This section discusses a generalized execution model of web applications and the idea of reverse engineering, based on which we can derive all db-pages generated by a web application and corresponding query strings from a database.

Here, without loss of generality, web application $\mathcal{A}$ is regarded as a *wrapper* of some access to an underlying database $\mathcal{D}$. While the implementation of $\mathcal{A}$ can significantly vary, depending on development methodologies used, developers' programming style, etc., the execution of $\mathcal{A}$ can logically be divided into three major steps: (a) *query string parsing*, (b) *application query evaluation*, and (c) *query result presentation*.

When it is invoked, $\mathcal{A}$ receives a query string from a web server $\mathcal{W}$. Then, in (a), a query string q is parsed into some values used in the remaining execution. In (b), application queries are formulated according to some query parameters (from q, other query results or both) and evaluated in $\mathcal{D}$. Finally in (c), the query results are formatted in a HTML page, which is then returned to a web browser via $\mathcal{W}$.

With respect to application queries, query parameters should be a part of operand relations in $\mathcal{D}$. Thus they are accessible/deducible from $\mathcal{D}$ and they can be used to deduce all query strings Q. They can also be used to determine corresponding query results, which in turn constitute the contents of all db-pages P. In other words, we only need to reverse engine the query string parsing step. As such, the corresponding *reverse query parsing* can derive query strings according to some possible query parameter values. Example 2 below shows the three execution steps of Search, our example web application introduced in Example 1, and how query strings and db-page contents can be deduced by means of reverse engineering.

Example 2. (Search's Implementation and Reverse Engineering) The implementation of Search, as Java servlet is outlined in Figure 3. Here, Search performs query string parsing (in line 1-3), extracting **cuisine**, **max** and **min** values from l, c, and u of a request (i.e., a query string) q as parameters to an application query Q (at line 6) evaluated in fooddb database. The query result is presented by a function output at (line 7), to a response (i.e., a db-page) p.[5]

public class **Search** extends HttpServlet{ public void doGet(HttpServletRequest q, HttpServletResponse p) ...{	
Query String Parsing	1. String **cuisine** = q.getParameter('c'); 2. String **min** = q.getParameter('l'); 3. String **max** = q.getParameter('u');
	4. Connection cn =DriverManager.getConnection(fooddb);
Application Query Evaluation	5. Q = 'SELECT name, budget, rate, comment, uname,' + ' date FROM (restaurant LEFT JOIN comment) ' + ' JOIN customer WHERE (cuisine = "' + **cuisine** + '") AND (budget BETWEEN '+ **min** + ' AND ' + **max** + ')'; 6. ResultSet r = cn.createStatement().executeQuery(Q);
Result Presentation	7. output(p, r); }}

Fig. 3. Search's implementation

According to Q, proper values for cuisine and budget should present in operand relations in fooddb database. Refer to the database contents as already shown in Figure 2, available cuisine and budget values are {American, Thai} and {9,10,12,18}, respectively. One possible combination of parameter values for cuisine, min and max can be American, 10 and 12, respectively. As they are extracted from l, c and u of q, q should be 'c = American&l = 10&u = 12'. Meanwhile, the query result (which produces a db-page with the same content as P1 in Example 1) can be obtained. ■

Other than the presented idea, issues regarding system design and optimization strategies are valuable to be investigated, so as to turn this idea into a practical solution. The coming three sections present the design of Dash and its algorithms.

IV. Overview of Dash

Intuitively, all possible query strings and query results (i.e., db-page contents) need to be derived in advance for keyword search. Nevertheless, such an intuitive approach is infeasible, because of very high processing, storage and search cost incurred for a large quantity of db-pages. Besides, the presence of overlapped db-page contents can deteriorate the quality of search results. Rather, Dash exploits the concept of *db-page fragments*. As will be discussed later, each db-page fragment carries a disjoint part of a db-page content and a db-page can be reconstructed based on some db-page fragments online. Also, the costs for collecting, storing, and searching db-page fragments are reasonably smaller than those for all db-pages. Further, db-pages sharing db-page fragments for sure have overlapped contents, and they can be easily identified to be excluded from search results.

[5]The details of these functions are not important to the context and omitted to save space.

As depicted in Figure 4, Dash is developed and equipped with a suite of algorithms, namely, *database crawling algorithm*, *fragment indexing algorithm*, and *search algorithm*, together with a specialized *fragment index*, all dealing with db-page fragments.

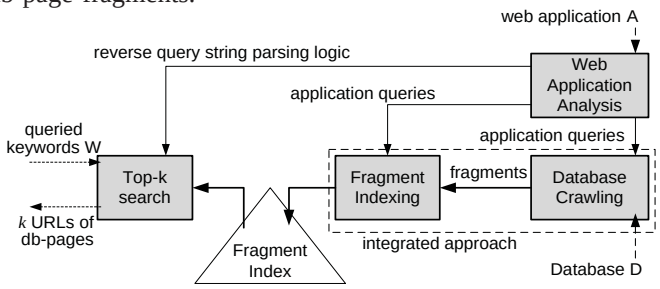

Fig. 4. The organization of Dash search engine

First of all, web application analysis identifies the logic of the three execution steps of a given web application $\mathcal{A}$ and application queries issued by $\mathcal{A}$. Then, based on identified application queries, database crawling looks up a database $\mathcal{D}$ for db-page fragments, and fragment indexing constructs a fragment index. The fragment index indexes the content of db-page fragments with respect to query parameters and captures the relationship among db-page fragments to facilitate db-page reconstruction at the search time. Thereafter, top-k search assembles db-page fragments into db-pages according to queried keywords, with the help of the fragment index. It formulates query strings and URLs (based on reverse query string parsing) of reconstructed db-pages. Finally, it returns the URLs of the k most relevant ones.

Since web application analyzer is subject to programming languages used in web application implementation and many studies have been done in static program analysis [4], in the next two sections, we focus only on database crawling, fragment indexing, and top-k search and assume only one web application in our discussion.

V. Database Crawling and Fragment Indexing

This section presents MapReduce (MR) based algorithms for database crawling and fragment indexing. The algorithms easily scale up for larger datasets by expanding an underlying computer cluster.

To facilitate our following discussion, we make three assumptions. First, we consider that a web application $\mathcal{A}$ issues only one application query. Second, every application query is formulated as a parameterized project-select-join (PSJ) query, which is formally expressed in Definition 1. Third, the content of a db-page equals to the result of an application query.

Definition 1. Parameterized Project-Select-Join (PSJ) Query. *A parameterized PSJ query Q is expressed in relational algebra:*

$$\pi_{a_1,a_2,\cdots a_l} \quad \sigma_{c_1\otimes_1 v_1 \wedge c_2 \otimes_2 v_2 \wedge \cdots c_m \otimes_m v_m} \quad (R_1 \bowtie R_2 \bowtie \cdots R_n)$$

where $R_1, R_2 \cdots R_n$ are n operand relations joined through inner- or outer-joins; each selection attribute c_i is compared with one query parameter value v_i, according to a comparison operator $\otimes_i$; and $a_1, a_2, \cdots a_m$, are m projected attributes. For simplicity, we consider comparison operator $\otimes_i$ as $=$, $\geq$, or $\leq$. The conditions are in conjunctive form. □

With respect to a given parameterized PSJ query, db-page fragments are formed and each of them is formulated in Definition 2.

Definition 2. Db-Page Fragment. *Given a parameterized PSJ query (as in Definition 1), a db-page fragment is formulated as*

$$\pi_{a_1,a_2,\cdots a_l} \quad \sigma_{c_1=v_1 \wedge c_2=v_2 \wedge \cdots c_m=v_m} \quad (R_1 \bowtie R_2 \bowtie \cdots R_n)$$

In the db-page fragment, all records share the same selection attribute values, i.e., $v_1, v_2, \cdots v_m$, and we denote $\langle v_1, v_2, \cdots v_m \rangle$ as the identifier *of this db-page fragment.* □

In order to quickly find db-page fragments relevant to queried keywords, an *inverted fragment index* is built to index db-page fragment identifiers (i.e., $v_1, v_2 \cdots v_m$) against the keywords extracted from all projected attributes (i.e., $a_1, a_2, \cdots a_m$). The structure of the inverted fragment index is identical to the conventional inverted file (as reviewed in Section II). In addition, a *fragment graph* is maintained to capture the relationship between db-page fragments. Both inverted fragment index and fragment graph constitute a *fragment index*. We leave the detailed discussion of the fragment graph in the next section.

Before presenting two different algorithms, namely, *stepwise algorithm* and *integrated algorithm* for database crawling and fragment indexing in two following subsections, we use Example 3 to illustrate the notion of fragments and the structure of an inverted fragment index.

Example 3. (**Db-page fragments and inverted fragment index**) Continue Example 2. According to the application query Q, db-page fragments are derived from operand relations: restaurant, comment and customer based on two selection attributes: cuisine and budget, as depicted in Figure 5. Next, an inverted fragment index is created to index the identifiers and the numbers of keyword occurrences of those fragments against keywords, e.g., "burger", "coffee" and "fries", as shown in Figure 6. ■

A. Stepwise Algorithm

The stepwise approach considers database crawling and fragment indexing as two separate steps. In the database crawling step, db-page fragments are derived from operand relations. In more details, all records from individual operand relations are first exported from a database to a MR cluster. Next, as defined below, a crawling query, which projects all the selection attributes in addition to the projection attributes from the same operand relations, is used to collect records in all db-page fragments.

$$\text{crawling query: } \pi_{a_1,a_2,\cdots a_l,c_1,c_2,\cdots c_m} \ (R_1 \bowtie R_2 \bowtie \cdots R_n)$$

Here, joins over three or more relations are performed through several MR jobs. Thereafter, retrieved records are grouped according to the db-page fragment identifiers (the values of their selection attributes (i.e., $c_1, c_2, \cdots c_m$)). This grouping can be conducted in another single MR job.

Identifier (cuisine,budget)	Projection attributes					
	name	budget	rate	comment	uname	date
(American,9)	Bond's Cafe	9	4.3	Nice coffee	James	01/11
(American,10)	Burger Queen	10	4.3	Burger experts	David	06/10
(American,12)	Wandy's	12	4.1			
	Wandy's	12	4.2	Unique burger	Bill	05/10
	Wandy's	12	4.2	Bad fries	Bill	06/10
(American,18)	McRonald's	18	2.2	Regret taking it	David	06/10
(Thai,10)	Thaifood	10	4.8			
	Bangkok	10	3.9	Thai burger	Alan	08/11

Fig. 5. Db-page fragments

keyword	(identifier):occurrence
burger	(American,10):2, (American,12):1, (Thai:10):1
coffee	(American,9):1
fries	(American,12):1

Fig. 6. Inverted fragment index (sample)

Next, in the fragment indexing step, individual db-page fragments are indexed against their contained keywords. With every db-page fragment treated as a single document, fragment indexing step is performed exactly as building an inverted file on documents, as already discussed in Section II. This step is also performed by a single MR job.

Finally, Example 4 below illustrates this stepwise approach.

Example 4. (**Stepwise algorithm**) Figure 7 demonstrates the stepwise algorithm that derives an inverted fragment index according to the application query Q of Search (in Figure 3).

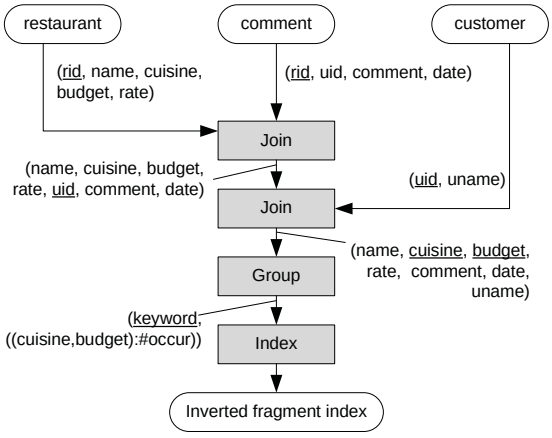

Fig. 7. The Stepwise approach

First, records from the three operand relations: restaurant, comment and customer in separate files are joined through two MR jobs. In the figure, the underlined attributes are used as keys in map outputs. Further, joined records are grouped according to the values of the selection attributes, i.e., cuisine and budget to form db-page fragments (as shown in Figure 5). Finally, cuisine and budget values are indexed against individual keywords (as exemplified in Figure 6). ■

B. Integrated Algorithm

The stepwise algorithm is straightforward. Yet, it is inefficient due to its high data transmission and I/O overheads. As already illustrated in Example 4, it involves multiple MR jobs to join the operand relations. Projection attributes are copied in intermediate results along the join processing, but they are only used in the fragment indexing step, i.e., the last task of the approach. This approach also has to store bulky intermediate joined results in files and transmit them between cluster nodes in map and reduce phase, incurring high I/O and transmission costs. Further, projection attributes of a record can be replicated whenever the record joins with multiple records, further amplifying the mentioned overheads.

To effectively reduce the overheads, we develop an integrated algorithm. Different from the stepwise algorithm, the integrated approach first determines the identifiers of individual db-page fragments and then estimates the numbers of keyword occurrences (from individual operand relations) in identified db-page fragments.

In details, the integrated approach includes three steps: (1) derivation of query parameter values, (2) extraction of keywords from operand relations and estimation of their occurrences in db-page fragments, and (3) consolidation of all the keywords for db-page fragments. We detail the steps followed by an illustrative example below.

(1) Query parameters derivation. For each operand relation R_i, we extract a selection attribute (c_i) and a join attribute (j_i), which is used to join other operand relations, together with counts (θ_i) of records sharing the same c_i and j_i values. Its logic is equivalent to the following aggregate query:

$$\text{aggregate query: } c_i,\ j_i\ G_{\text{count}(*)\ \text{as}\ \theta_i}\ (R_i)$$

Here, for notational convenience, we assume R_i has only one selection attribute c_i and one project attribute j_i. The use of θ_i serves two purposes. First, it represents the number of duplicate records and saves duplicates from being transferred. Second, it is used to determine keyword occurrence in the next step. Notice that in MR, the evaluation of θ_i with respect to c_i and j_i can be performed during the join, as j_i is used as both a join key and one of group-by keys for θ_i.

Finally, selection attribute values, join attribute values and the record counts from all operand relations denoted by $\mathcal{R}$ are derived. $\mathcal{R}$ is expressed as follows:

$$\mathcal{R} = \pi_{\ c_1,c_2,\cdots c_n,\ j_1,j_2,\cdots j_n,\ \theta_1,\theta_2,\cdots\ \theta_n}\ (R_1 \bowtie R_2 \bowtie \cdots R_n)$$

(2) Keyword extraction and occurrence determination. Based on $\mathcal{R}$, we extract keywords from projection attributes of records in individual operand relations and determine their numbers of occurrences in db-page fragments. Systematically, we determine (i) db-page fragments that projection attribute values are put into and (ii) the number of times the projection attributes are replicated in join. Both of them can be performed logically together as the following query:

$$\text{project query: } \pi_{a_i,c_1,\cdots c_n,\frac{\prod_{x\in[1,n]}\theta_x}{\theta_i}\ \text{as}\ \Theta_i}(\mathcal{R} \bowtie_{c_i,j_i} R_i)$$

Notice that $\prod_{x\in[1,n]}\theta_x$ presents the total number of records having the same values for $c_1, c_2, \cdots c_n,\ j_1, j_2, \cdots j_n$. It includes θ_i records from R_i and hence a record with the same

c_i and j_i is expected to be joined with $\frac{\prod_{x\in[1,n]}\theta_x}{\theta_i}$ (i.e., Θ_i) records in other relations.

Next, we extract keywords from the projection attribute (a_i) and assign them to a db-page fragment with identifier equal to selection attribute values $(c_1, \cdots c_n)$. Further, the number of occurrences for any keyword equals its occurrence in a record multiplied by Θ_i. Finally, keywords are outputted with their assigned db-page fragments and numbers of occurrences. Likewise, we adopt the same idea to determine the total number of keywords contributed by each record.

(3) Keyword occurrence consolidation and sorting. In (2), the same keywords for a db-page fragment may be extracted from different records in the same or different operand relations. In this step, for each keyword, we sum up the numbers of occurrences for the same db-page fragments and sort all db-page fragments according to their total numbers of occurrences. Finally, each sorted list becomes an entry in an inverted fragment index.

Example 5. (Integrated Algorithm) As depicted in Figure 8, the integrated algorithm first performs join on the selection attributes and join attributes of operand relations Restaurant, Comment and Customer,

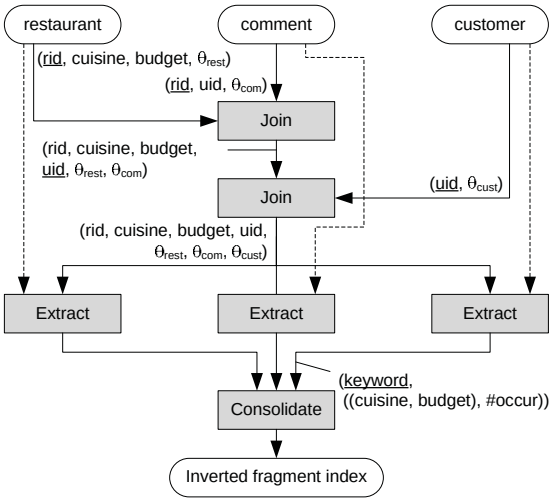

Fig. 8. The integrated approach

After all, query parameters (cuisine, budget), some join attributes (rid, cid, uid), and some counts are derived. One example result record is

cuisine	budget	rid	uid	$\theta_{rest.}$	$\theta_{comm.}$	$\theta_{cust.}$
American	12	004	132	1	2	1

This record means Restaurant record (rid=004) joins with two Comment records, which in turn join with a single Customer record (uid=132).

Next, in the second step, a Restaurant record joins with the above result record. From it, keywords such as "Wandy's" are put into db-page fragments with keys of (American,12). Also, their numbers of occurrences are multiplied by 2. In the final step, an inverted fragment index is generated. ■

VI. Top-k Db-Page Search

In Dash, top-k db-page search algorithm combines some db-page fragments into db-pages and returns k db-pages, mostly relevant to queried keywords. The relevance of a db-page to queried keywords is estimated through a slightly modified TF/IDF scheme. Since no db-pages are physically derived, the exact IDF is not available. Rather, Dash approximates the IDF of a keyword w as the inverse of the number of db-page fragments containing w. Intuitively, w, if it is common to many db-page fragments, is expected to be included in many db-pages possibly sharing those db-page fragments.

On the other hand, to facilitate the determination of a db-page's size and a query string, a fragment graph is derived and maintained. Both fragment graph and inverted fragment index constitute a fragment index. In what follows, we present the fragment graph and top-k search algorithm.

A. Fragment Graph

A fragment graph captures the relationship between db-page fragments. In a fragment graph, every node presents a single db-page fragment and indicates the total number of keywords included in the fragment. An edge connects two fragments f and f', if f and f' can be combined to form a db-page, according to an application query, and the formed db-page contains no other db-page fragments. Example 6 shows a fragment graph with respect to our example db-page fragments.

Example 6. (**Fragment Graph**) Figure 9 shows a fragment graph of our example db-page fragments, as discussed in Example 3.

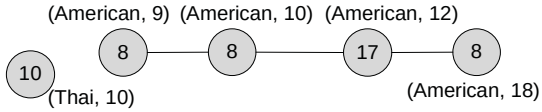

Fig. 9. Fragment graph

Here, all db-page fragments about American cuisine are connected. Take the node representing a fragment (American, 9) as an example. The value 8 means eight keywords contained in the fragment, i.e., Bond′s, Cafe, 9, 4.3, Nice, Coffee, James, and 01/11, from six projection attributes specified in the application query Q from Search. Another node about Thai cuisine is disconnected from others, because db-page fragments with different cuisine values are definitely not included in any db-page provided by Search. ■

A fragment graph can be created incrementally. In each turn, a db-page fragment as a node is added to graph until all db-page fragments are added. When a db-page fragment f is added into a fragment graph, f is checked against individual nodes. If f and any node f' can be combined into a db-page that contains no other nodes, an edge connecting f and f' are formed. On the other hand, if f is covered by a db-page formed by two connected nodes f_0 and f_1, the corresponding edge between f_0 and f_1 is removed and new edges are formed between f and f_0 and between f and f_1. Strategically, a lot of comparisons can be saved if db-fragments are pre-sorted based on their query parameter values before they are added to a fragment graph.

B. Top-k Search Algorithm

Our top-k search algorithm accepts three parameters, namely, (i) *a set of queried keywords* (W), (ii) *a requested number of URLs of db-pages mostly relevant to* W (k), and (iii) *a db-page size threshold* (s). The size threshold s (expressed as the number of words) is used to control the minimum size of suggested db-pages. Logically, a db-page can be simply formed by a few db-page fragments and thus this small db-page should have a large number of queried keywords. However, such db-page with no much contents other than the queried keywords might not be valuable to users. Therefore, a larger value of s can force the search algorithm to combine more db-page fragments if possible into a coarser db-page. Meanwhile, huge db-pages, which are likely to contain a lot less relevant information, are avoided to be derived.

The logic of the top-k search algorithm is outlined in Algorithm 1. First, it determines a set of db-page fragments $\mathcal{F}$ relevant to queried keywords W from an inverted fragment index (line 1). Next, $\mathcal{F}$ are inserted into a priority queue $\mathcal{Q}$ sorted by their TF/IDF scores in descending order (line 2). If multiple queued entries have the same TF/IDF score, the tie is resolved arbitrarily. A result set $\mathcal{O}$ is vacated initially (line 3).

Algorithm 1 Top-k Db-Page Search

input: queried keywords W; a req. no. of answer URLs k; db-page threshold size s; inverted fragment index I; fragment graph G;
output: at most k URLs;
1. determine db-page fragments $\mathcal{F}$ relevant to W from I;
2. initialize a priority queue $\mathcal{Q}$ with $\mathcal{F}$;
3. initialize an output set $\mathcal{O}$ to $\{\}$;
4. **while** ($\mathcal{Q}$ is not empty and $|\mathcal{O}| < k$) **do** {
5. dequeue $\mathcal{Q}$ to get the head entry ϵ;
6. **if** ϵ is not expandable with respect to s **then**
7. add ϵ to $\mathcal{O}$; goto 4
8. expand ϵ based on G to ϵ'
9. add ϵ' to $\mathcal{Q}$; }
10. output the URLs of db-pages in $\mathcal{O}$;

Thereafter, $\mathcal{Q}$ is accessed iteratively (line 4-9). The iteration ends when $\mathcal{Q}$ becomes empty or k result db-pages are collected. Then, the URLs of collected result db-pages are formulated and returned (line 10).

At each turn, a head entry, ϵ, i.e., a pending db-page with the greatest TF/IDF score among all queued entries, is accessed from $\mathcal{Q}$ (line 5). If ϵ cannot be expanded, it is added into $\mathcal{O}$ (line 6). ϵ is not expandable because (i) the size of ϵ is not smaller than s, or (ii) no db-page fragments are available to be combined with. Otherwise, ϵ is expanded to ϵ' (line 7) and then ϵ' is inserted to $\mathcal{Q}$ for later examination (line 8). Whenever possible, relevant db-page fragments are favored to combine to maintain high TF/IDF score. If a relevant db-page is used in expansion, it is removed from $\mathcal{Q}$. Due to additional text, ϵ' should have its TF/IDF score no greater than ϵ. Due to this monontonicity property, the first k collected db-pages should have the greatest relevance scores.

After the iteration, query strings of k db-pages in $\mathcal{Q}$ with the greatest TF/IDF scores are deduced and corresponding URLs are returned as the search result (line 11). Our final example provided below demonstrates this search algorithm.

Example 7. (Top-k search algorithm) With respect to a queried keyword "burger", k and s, which are set to 2 and 20, respectively, the search first identifies three db-page fragments, namely, $(\mathsf{American}, 10)$, $(\mathsf{Thai}, 10)$ and $(\mathsf{American}, 12)$.

First, $(\mathsf{American}, 10)$ (whose TF is $\frac{2}{8}$) is dequeued and it is expanded by merging it with $(\mathsf{American}, 12)$ as it is also relevant to "burger". The expanded db-page is formed with TF equal to $\frac{3}{25}$ and the query parameters are $(\mathsf{American}, (10, 12))$. Then, $(\mathsf{American}, 12)$ is no longer queued.

Next, $(\mathsf{Thai}, 10)$ with TF of $\frac{1}{10}$ is accessed and it cannot be expanded. Then, it is directly placed into the result set. Further, $(\mathsf{American}, (10, 12))$ is retrieved. It is greater than $s = 20$ and so it is put into the result set. Finally, two URLs: `www.example.com/Search?c=Thai&l=10&u=10` and `www.example.com/Search?c=American&l=10&u=12` corresponding to the two resulted db-pages are returned. ■

VII. Performance Evaluation

This section evaluates the performance of (i) the two approaches for database crawling and fragment indexing (Section V), (ii) fragment graph construction (Section VI-A), and (iii) top-k search algorithm (Section VI-B). The efficiency of those directly indicates the practicality of Dash.

To save space, we present a representative subset of experiments and their results, based on experiment settings as summarized in Table I.

Parameters	Values
datasets	small, medium and large
application queries	Q1, Q2, and Q3
no. of returned db-pages (k)	1, 5, 10, 20
db-page threhold size (s)	100, 200, 500, 1000
keywords	cold (bottom 10%), warm (middle 10%) and hot (top 10%)

TABLE I
Experiment parameters

In this evaluation, we used TPC-H benchmark database [25], which has been also used in the performance evaluation of research works on keyword search in relational databases [14], [18], [20], [26]. We generated three TPC-H datasets (i.e., small, medium, and large) that provide operand relations of different sizes, as listed in Table II.

	R	N	C	O	L	P
small	389B	2KB	23MB	163MB	725MB	23MB
medium	389B	2KB	116MB	830MB	3,684MB	116MB
large	389B	2KB	234MB	1,668MB	7,416MB	232MB

TABLE II
Three experimented data sets

Besides, we defined three application queries Q1, Q2, and Q3 for the experiments. They are detailed in Table III. [6] Q1 includes small operand relations (i.e., R N, and C); Q2 and Q3 query three common large operand relations (i.e., C, O, and L), while Q3 includes one additional operand relation (i.e., P). These queries take $r, $min, and $max as query parameters and project all attributes whose contents are collected as keywords.

[6] In these queries, R, N, C, O, L and P represent relations: region, nation, customer, orders, lineitem, part, respectively, in TPC-H schema.

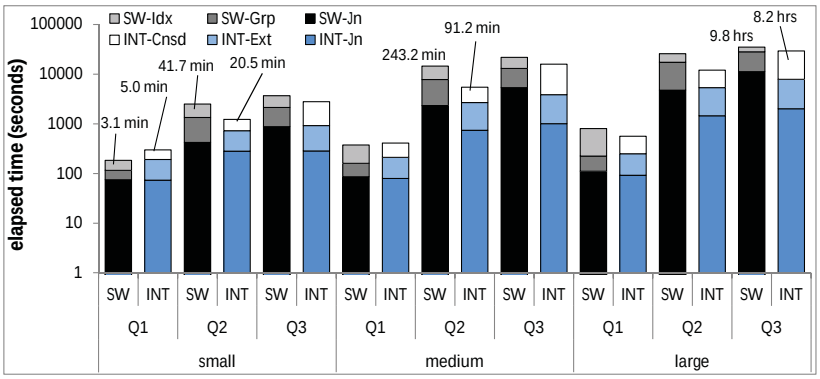

Fig. 10. Database crawling and fragment indexing performance

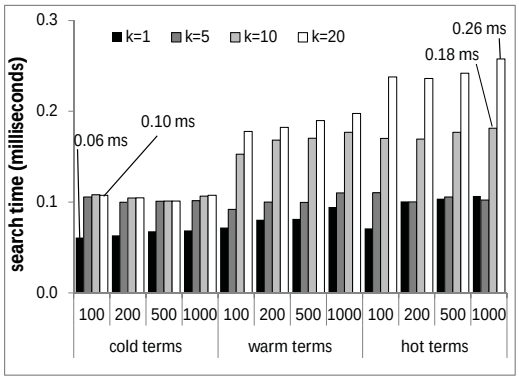

Fig. 11. Top-k search performance (Q2, medium)

Q1:	select * from (R ⋈ N) ⋈ C where R.RID = $r and C.ACCBAL between $min and $max
Q2:	select * from (C ⋈ O) ⋈ L where C.CID = $r and L.QTY between $min and $max
Q3:	select * from (C ⋈ O) ⋈ (L ⋈ P) where C.CID = $r and L.QTY between $min and $max

TABLE III
THREE EXPERIMENTED APPLICATION QUERIES

We also use different requested numbers of result db-pages (k) (i.e., 1, 2, 5, 10), various minimum size thresholds (s) (i.e., 100, 200, 500, 1000) and different groups of keywords to evaluate top-k search performance. All the experiments were conducted in a cluster of four Intel Xeon 2.8Ghz 4GB RAM computers running Debian Linux, Hadoop 0.20.3 and Sun JRE 1.7.0 and connected through a gigabit ethernet. In the following, we first report the efficiency of database crawling and fragment indexing, and then present that of fragment graph building and top-k search algorithm.

A. Evaluation of Database Crawling and Fragment Indexing

The first set of experiments evaluate the elapsed time of the stepwise algorithm (SW) and the integrated algorithm (INT) for database crawling and fragment indexing, based on three queries Q1, Q2, and Q3 in small, medium and large datasets.

Figure 10 shows their resulted elapsed time in unit of seconds (in log scale). As shown in the figure, their elapsed time dramatically increases with the dataset size. For small operands like R and N accessed by Q1, database crawling and fragment indexing can finish within a few minutes. In contrast, for large operands, the same algorithms can spend 8-9 hours to complete. On the other hand, INT runs generally faster than SW. Compared with SW, INT saves 21.4% elapsed time, on average, and 64% elapsed time for the best case situation (i.e., Q2 against medium). SW runs faster than INT only when very small operand, e.g., R and N, are accessed. From this, we can see the outperformance of INT.

We also examined the impact of the number of nodes for reduce tasks on the performance while fixing the cluster size. However, the difference is not significant (3-8%). This can be explained that most of the MR jobs for the approaches are I/O bound, so map tasks are the main performance bottleneck and adding more nodes for reduce tasks cannot improve performance much. Besides, Hadoop assigns nodes for map tasks according to the number of file blocks. For the same datasets, the same numbers of cluster nodes for map tasks are resulted. We omit the experiment results to save space.

B. Evaluation of Fragment Graph Building and Top-k Search

In the second set of experiments, we measure the elapsed time incurred for fragment graph building and top-k search. Due to limited space, we focus only on medium dataset in the rest of the discussion. First, we report (i) fragment graph building time, (ii) the quantity of db-page fragments, (iii) the average number of keywords contained in a db-page fragment, with respect to different queries in Table IV. From the result, we can see that the fragment graphs can be built by a single computer within few minutes.

	building time	#db-page fragments	average # keywords
Q1	27.5 sec	700,851	12.6
Q2	515.0 sec	7,481,097	22.5
Q3	518.8 sec	7,481,097	86.2

TABLE IV
DB-PAGE FRAGMENT GRAPH BUILDING PERFORMANCE

Next, we evaluate the top-k search performance. Here, we report the results based on db-page fragments derived for Q2 in medium datasets. All top-k searches are performed based on three different sets of keywords. The selection of keywords is based on their DFs. In details, we order all keywords according to their DFs. Among all those, 30 hot keywords, 30 warm keywords and 30 cold keywords are extracted from top 10%, middle 10% and bottom 10% of the keywords. As anticipated, hot (cold) keywords are included in many (few) db-page fragments.

With respect to different k values and size threshold s, the average elapsed time is plot in Figure 11. As shown in the figure, searches on cold keywords result in more or less the same elapsed time 0.08 millisecond for k=1 and 0.10 millisecond for other k settings. It is because only a few db-page fragments are retrieved and expanded. In the experiment, there are about 6 db-pages formed for a cold keyword. Thus, very similar performance is obtained for k=5, 10 and 20. On the contrary, when warm keywords and hot keywords are experimented, longer elapsed time is resulted, because many candidate db-page fragments are collected and examined. Also, for those warm and hot keywords, s makes a greater impact

on the efficiency than cold keywords while the average size of fragments are listed in Table IV. From the experiments, all search time is very small and the longest search time is still less than 0.27 millisecond.

In summary, the presented evaluation results indicate (i) reasonable performance of database crawling and fragment indexing and the outperformance of the integrated algorithm over the stepwise algorithm, and most importantly (ii) the efficiency of using db-page fragments to derive db-pages according to runtime settings of k, s and queried keywords at the search time. These experiment results demonstrate the efficiency and thus the practicality of our Dash.

VIII. Conclusion

We developed Dash, a novel search engine that supports keyword search on db-pages, whose contents are generated on the fly by web applications and databases. Based on the implementation of a web application and the content of its database, Dash can deduce db-pages and their query strings/URLs to answer keyword search. In this paper, we presented the details of Dash and made the following contributions:

1) We exploited the concept of db-page fragments. A db-page fragment contains a disjointed db-page content. By using db-page fragments in place of db-pages, a massive number of db-pages, which may have overlapped contents, can be avoided from being collected, indexed and searched in Dash.
2) We developed MapReduce based algorithms, namely, *stepwise algorithm* and *integrated algorithm* for database crawling and fragment indexing. The integrated algorithm smartly arranges tasks to minimize the overall processing time.
3) We designed the fragment index that consists of (i) *an inverted fragment index*, which is to facilitate the lookup of relevant db-page fragments, and (ii) *a fragment graph* that indicates whether db-page fragments can be combined to form a db-page.
4) We devised the top-k search algorithm that constructs k db-pages relevant to queried keywords and that controls the minimum size of formed db-pages.
5) We implemented all the proposed algorithms and fragment index, and evaluated them via extensive experiments. The experiment results consistently demonstrate the efficiency of our proposed approach.

As the next steps of this presented work, we consider several directions. First, in presence of updates in an underlying database, a fragment index would become outdated and need to be updated. It should be very costly to rebuild the entire fragment index. Some efficient update mechanisms that can efficiently update (affected portions of) a fragment index are desirable and they are valuable to be investigated. Second, the presented Dash is assumed to support keyword search on db-pages generated by a single web application. For quite common situations, multiple web applications would derive db-pages based on some common contents from a database. As such, the contents of those db-pages could still be overlapped. A new approach is demanded to eliminate duplicate contents of db-pages from different web applications. Last but not least, our discussion simply considered that all db-page fragments are needed to be derived. There exists a tradeoff between (i) the amount of db-page fragments to be collected and (ii) crawling and index efficiency.

Acknowledgments

In the research, Ken C. K. Lee was in part supported by UMass Dartmouth's Healey Grant 2010-2011. Baihua Zheng was partially funded through a research grant 12-C220-SMU-002 from the Office of Research, Singapore Management University. Chi-Yin Chow was partially supported by grants from the City University of Hong Kong (Project No. 7200216 and 7002686). Honggang Wang was in part supported by UMass President's 2010 Science & Technology (S&T) Initiatives.

References

[1] Apache Hadoop, 2011. http://hadoop.apache.org/.
[2] Apache Lucene, 2011. http://lucene.apache.org.
[3] Apache Nutch, 2011. http://nutch.apache.org/.
[4] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008.
[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Pearson, 1999.
[6] C. Beeri, P. A. Bernstein, and N. Goodman. A Sophisticate's Introduction to Database Normalization Theory. In *VLDB*, pages 113–124, 1978.
[7] J. Challenger, P. Dantzig, A. Iyengar, and K. Witting. A Fragment-Based Approach for Efficiently Creating Dynamic Web Content. *ACM Trans. Internet Technology*, 5(2):359–389, 2005.
[8] Y.-K. Chang, I.-W. Ting, and Y.-R. Lin. Caching Personalised and Database-Related Dynamic Web Pages. *IJHPCN*, 6(3/4):240–247, 2010.
[9] B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2009.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
[11] Google Custom Search Engine, 2011. http://www.google.com/cse/.
[12] Google Database Crawling and Serving, 2011. http://code.google.com/apis/searchappliance/documentation/52/database_crawl_serve.html.
[13] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *ACM SIGMOD Conf.*, pages 16–27, 2003.
[14] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, pages 670–681, 2002.
[15] U. P. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.
[16] J. C. King. Symbolic Execution and Program Testing. *J. ACM*, 19(7):385–394, 1976.
[17] W.-S. Li, W.-P. Hsiung, and K. S. Candan. Multitiered Cache Management and Acceleration for Database-Driven Websites. *Concurrent Engineering: R&A*, 12(3):221–235, 2004.
[18] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective Keyword Search in Relational Databases. In *ACM SIGMOD Conf.*, pages 563–574, 2006.
[19] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's DeepWeb Crawl. In *VLDB*, pages 1241–1252, 2008.
[20] A. Markowetz, Y. Yang, and D. Papadias. Keyword Search over Relational Tables and Streams. *ACM TODS*, 34(3), 2009.
[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, InfoLab, Stanford Unversity, 1999.
[22] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *VLDB*, pages 129–138, 2001.
[23] The Deep Web: Surfacing Hidden Value, 2001. http://quod.lib.umich.edu/cgi/t/text/text-idx?c=jep;view=text;rgn=main;idno=3336451.0007.104.
[24] The size of the World Wide Web, 2011. http://www.worldwidewebsize.com.
[25] TPC-H Database, 2011. http://www.tpc.org/tpch/.
[26] B. Yu, G. Li, K. R. Sollins, and A. K. H. Tung. Effective Keyword-based Selection of Relational Databases. In *ACM SIGMOD Conf.*, pages 139–150, 2007.
[27] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Survey*, 38(2), 2006.