Towards Automatic Association of Relevant Unstructured Content with Structured Query Results

Prasan Roy Mukesh Mohania Bhuvan Bamba* Shree Raman†
IBM India Research Lab, New Delhi, India
{*prasanr, mkmukesh*}*@in.ibm.com, bhuvan@cc.gatech.edu, sraman@ecs.umass.edu*

ABSTRACT

Faced with growing knowledge management needs, enterprises are increasingly realizing the importance of seamlessly integrating critical business information distributed across both structured and unstructured data sources. In existing information integration solutions, the application needs to formulate the SQL logic to retrieve the needed structured data on one hand, and identify a set of keywords to retrieve the related unstructured data on the other. This paper proposes a novel approach wherein the application specifies its information needs using only a SQL query on the structured data, and this query is automatically "translated" into a set of keywords that can be used to retrieve relevant unstructured data. We describe the techniques used for obtaining these keywords from (i) the query result, and (ii) additional related information in the underlying database. We further show that these techniques achieve high accuracy with very reasonable overheads.

Categories and Subject Descriptors: H.2 [Database Management]: Systems-*Query Processing*

General Terms: Algorithms, Design, Experimentation

Keywords: Information Integration, Context, SQL, Keyword Search

1. INTRODUCTION

Digital information within an enterprise is composed of two kinds of content: *structured* and *unstructured*. The structured content includes mainly the operational business data such as sales, accounting, payroll, inventory, etc. while the unstructured content includes the reports, email, webpages, etc. The structured data is strictly typed and can be meaningfully decomposed into relations, and queried using a query language (SQL) with a well-defined semantics. The unstructured content, in contrast, is free-flow and untyped, and is typically queried using a set of keywords. This inherent difference in the way the structured and unstructured content are managed and queried creates an artificial separation between the two, which is unfortunate since they are complementary in terms of information content.

Effective knowledge management, however, requires seamless access to information in its totality, and enterprises are fast realizing the need to bridge this separation. This has led to a significant effort towards integration of structured and unstructured data [6, 17, 23, 20, 16, 11, 22, 14].

Existing solutions typically enable this integration by providing a single point of access to both structured and unstructured data sources. DB2 NSE [20], for instance, exposes the unstructured data as a virtual relational table (with one row per document) and introduces a `CONTAINS` predicate that can be used to filter the relevant documents from this table based on a set of keywords.[1] While this alleviates the issue of having to interface with each source individually, the application still needs to formulate the SQL logic to retrieve the needed structured data on one hand, *and* identify a set of keywords to retrieve the related unstructured data on the other. This is not ideal for the following reasons:

- The same information need is formulated using two disparate paradigms (SQL for structured data, and a set of keywords for the unstructured data), which is redundant effort.
- In many cases, it is hard (even impossible) for the application to identify appropriate keywords needed as above to retrieve related unstructured data.

In this paper, we introduce the *SCORE*[2] system that effectively addresses the limitations mentioned above by *automatically* associating related unstructured content with the SQL query result, thereby eliminating the need for the application to specify a set of keywords in addition to the SQL query. Specifically, SCORE works as follows:

1. The application specifies its information needs using only a SQL query on the structured data. SCORE executes the query on the RDBMS.
2. SCORE "translates" the given SQL query into a set of keywords that, in effect, reflect the same informa-

*Currently at Georgia Institute of Technology, Atlanta, GA
†Currently at University of Massachusetts, Amherst, MA

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

[1]See also SQL-Server [16], and Oracle [11].
[2]SCORE stands for Symbiotic Context-Oriented Information REtrieval – where *symbiotic* emphasizes the mutually beneficial role of the structured and unstructured data sources.

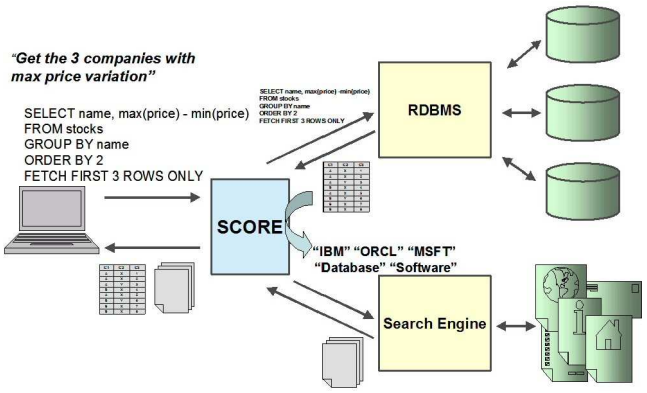

Figure 1: SCORE Overview

tion need as the input query. (This set of keywords is hereafter termed the *context* of the query).

3. SCORE uses these keywords to retrieve relevant unstructured content using a search engine. This unstructured content is then associated with the result of the given SQL query computed earlier, and returned to the application.

This is illustrated in the following example.

EXAMPLE 1. *Consider an investment information system that helps users to analyze stock market data. The system maintains the stock ticker archived over the past week, company profile, information about the various institutional investors, mutual-fund portfolios, etc. (structured data). It also maintains a searchable repository of past week's news stories, advisories and recent analyst reports (unstructured data). Now, consider the scenario illustrated in Figure 1, wherein the application submits a query asking for the names of the three companies with maximum stock price variation in the past week. With this query as input, SCORE explores the query's result as well as neighboring tables*[3] *containing related information, and identifies the keywords that are most relevant to the profiles of these stocks. These keywords form the context of the input query. In this example, these keywords are "IBM", "ORCL", "MSFT" (obtained from the stocks table) and "Database", "Software" (obtained from the company profile tables). These keywords are then used to retrieve relevant news stories and related advisories and reports using the search engine, which are then returned to the application along with the SQL query result.*

As illustrated in the above example, SCORE computes the context of the input SQL query by analyzing not only the information present in the result of the query, but also related information present *elsewhere* in the database. This related information is found by exploiting the semantic information (foreign-key dependencies) embedded in the database. Moreover, since traversing all potentially related information present in the database does not scale, SCORE heuristically determines what subset of the available information to focus on.

SCORE's approach to integrating structured and unstructured information has several advantages over the existing approaches to the problem. Some of these advantages are listed below (ref. Section 2 for a detailed discussion.):

- SCORE dynamically associates documents with the SQL query result based on the *global* context of the result. For instance, SCORE associates "database" and "software" related docs when "IBM" is present with "ORCL" and "MSFT" in the SQL query result (ref. Example 1), and associates "microcomputers" and "hardware" related docs instead when "IBM" is present with "DELL" and "HP." Contrast this with static association wherein all assorted "IBM" related documents get associated whenever "IBM" appears in the SQL result.

- SCORE does not need the administrator to define a common schema that consolidates the schemas of the individual data sources. Also, as mentioned earlier, SCORE does not maintain static association of documents and relational data. Moreover, it does not require any special-purpose index on the data. This implies that administrator does not have the onus of maintaining SCORE in face of data and/or schema updates on the data sources.

- SCORE does not require any external semantic information (e.g. ontologies). In effect, it uses the semantics embedded in the structured data (foreign-key dependencies, statistical correlations) to integrate structured and unstructured data However, if the related ontologies are available, SCORE can be extended to make effective use of the same.

- SCORE does not require any non-standard interface to the data sources being integrated. It interfaces with the structured data source using standard SQL queries over JDBC/ODBC, and with the unstructured data source using keyword queries over HTTP. SCORE can thus work with any DBMS supporting SQL (DB2, Oracle, SQL-Server, MySQL, etc.) and any unstructured data source with a keyword-based search interface (web search engines such as Google, intranet search engines such as Google appliance and DB2 Omnifind, desktop email clients such as MS Outlook, etc.).

As a result of the above unique advantages of its approach, SCORE can work with an existing product/technology that interfaces with an RDBMS using SQL (e.g. SQL-based publish-subscribe systems), and gracefully extend it to retrieve relevant documents from an external source. In such an extended environment, SCORE would exist as a loosely coupled component with minimal intrusion into the core of the given product/technology.

Contribution. Clearly, the hardest task in SCORE is identifying the context of the input query—that is, deriving the set of keywords that succinctly represent the information need of the input SQL query. The naive solution that includes every term available in the query result is not appropriate, since the resulting set is very likely to be a potpourri of unrelated terms. The focus of this paper is thus on techniques to pick relevant and informative keywords from the entire set of available terms in the database. The main technical contributions of this paper are as follows:

- We develop an algorithm to automatically compute the context of a SQL query from the result of the query

[3]tables reachable from the stocks table through foreign keys; these include tables containing company profiles, etc.

as well as by exploring related data in the underlying database; the algorithm determines what relevant tables to explore without any help from the user. (Recall that the computed context is used to retrieve the relevant unstructured content to be associated with the SQL query result.)

- We analyze the implementation challenges in the above algorithm, and show how these challenges can be efficiently resolved by judicious use of data statistics (histograms) available in the RDBMS. We present alternative efficient implementation approaches for the above algorithm.
- We present an experimental study of the quality and performance of the proposed techniques. The study shows that our implementation achieves high quality with very reasonable performance overheads.

Organization. The rest of this paper is organized as follows. We begin with a discussion of related work in Section 2. In Section 3, we present the algorithm for computing the context of a SQL query. The related implementation approaches are discussed in Section 4. A preliminary performance study is presented in Section 5 and an anecdotal example appears in Section 6. Finally, in Section 7, we present the conclusion and interesting future work directions.

2. RELATED WORK

There has been significant work in both the DB and IR communities towards integrating unstructured and structured data into a single physical system [13, 10]. Several vendor database systems provide full text search embedded within SQL [20, 16, 11]. SCORE simplifies the use of this feature by automatically suggesting the appropriate set of keywords for querying the unstructured data. Moreover, unlike current information integration solutions [15, 18], SCORE does not need the user (or the DBA) to design a *common schema*, a task that needs considerable skill on the part of the user and can, at best, be semi-automated.

WHIRL [9] supports "similarity" joins between text-based fields in database tables. Chaudhuri et al. [7] address joins between the structured data and unstructured data (virtualized as tables) in a loosely coupled system. WSQ [14] uses virtual tables as an interface to a search engine; integration with the unstructured data is simulated by including joins with these virtual tables in the SQL query. To formulate the joins in each of these systems, the user needs to be aware of the specific columns to be used in the search. In contrast, the specific columns and rows from which the keywords are picked are determined automatically in SCORE, and can vary across different executions of the query as the database gets updated and the focus of the query result changes. Moreover, in these approaches, each row in the query result is associated with unstructured data *independently* of the other rows in the query, unlike SCORE.

A related approach involves populating pre-defined tables using information extracted from the unstructured data. These tables can then be queried and analyzed along with the existing structured data (e.g. [8]). In this approach, the user has to define *apriori* what information to extract from the text. SCORE, on the other hand, handles association of unstructured content with structured data based on full-text search, which is more general.

Fisher and Sheth [12] make a case for finding similarities in the metadata across structured and semi-structured data and exploiting these similarities for information integration. These solutions essentially rely on a high quality domain ontology for their operation [19]. SCORE, in contrast, does not need any external semantic information. However, if an ontology is available, then SCORE can be easily extended to exploit the same.

The context computation step in SCORE is also conceptually similar to question-answering and query-expansion in Information Retrieval [2]. Question-answering typically takes as input a natural language query (e.g. "Who invented the relational database model?") and generates a ranked list of possible answers, ideally containing the right answer ("E. F. Codd") with a high rank. Query-expansion, on the other hand, starts with an initial set of keywords (the search query) and, by explicit or implicit relevance feedback, adds in additional keywords relevant to the user's information needs. Context computation in SCORE is similar as it starts with an *empty* set of keywords, and adds keywords to this set. However, this feedback comes from a source (the structured database) that is different from the source that is to be queried, and is more focused since it is based on the semantic information embedded in the database.

Recent research [5, 1, 3] on enabling keyword-based query on (relational) databases can be used to integrate structured and unstructured data sources by enabling keyword-based query as the common medium of retrieving data from both the sources. While this has the advantage of simplicity, there is an accompanying loss in expressibility, as compared to SQL, in querying the relational database—this is because, as mentioned earlier, the keyword-based query paradigm assumes that the user knows the relevant terms that can encode her information need. SCORE, on the other hand, allows the user to express the information needs in terms of the more expressive SQL (cf. Example 1). Nevertheless, one way to look at the context computation in SCORE is as an *inverse* of the keyword search in databases—context computation determines a minimal set of terms in the query result that, when given to the keyword search in databases engine, would return roughly the same query result with a high rank.

3. SQL QUERY CONTEXT COMPUTATION

In this section, we discuss the techniques used in SCORE to compute the context of a given SQL query. First, in Section 3.1, we provide a working definition of the context of a query. Based on this definition, we develop an algorithm to compute the context from the result of the input query, without any augmentations. Next, in Section 3.2, we show how to determine the optimal augmentation of the query from the several alternatives, and how to extend the algorithm of Section 3.1 to work on the result of the augmented query. The consolidated algorithm is presented in Section 3.3.

3.1 Context from the Query Result

Consider a query Q on a single table R,[4] and let $Q(R)$ denote the result of the query Q. We model the context of Q as the set of terms that (a) are popular in $Q(R)$, and (b)

[4]While this discussion is limited to single-table queries for sake of clearer exposition, the extension of the ideas to the multi-table query case is not hard.

are rare in $R - Q(R)$. Intuitively, these terms differentiate the query result $Q(R)$ from $R - Q(R)$, the remaining rows in the table.

Let $N_Q(A,t)$ denote the number of rows in which the term t appears in the column $A \in cols(Q)$ (formally, $N_Q(A,t) = |\sigma_{A=t}(Q(R))|$). Further, let $N_R(A,t)$ denote the number of times the term t appears in the column $A \in cols(R)$ (formally, $N_R(A,t) = |\sigma_{A=t}(R)|$). Then, for a term $t \in A$[5] where $A \in cols(Q)$, SCORE measures the term t's relevance to the query Q as:

$$TW(A,t) = N_Q(A,t) \times \log\left(\frac{1 + |R| - |Q(R)|}{1 + N_R(A,t) - N_Q(A,t)}\right)$$

In the above, the first expression measures the popularity of t in $Q(R)$ while the second expression measures the rarity of t in $R - Q(R)$.[6]

A straightforward algorithm to compute the context of a query Q is to rank the terms in the query result on the basis of $TW(A,t)$ and pick the top N terms, where N is a user defined parameter. In the next section, we extend this algorithm to look for the context beyond the given query result.

3.2 Exploring Beyond the Query Result

The algorithm presented in the previous section computes the context of a query from its result. In this section we extend the algorithm to look for the context beyond the given query.

A simple way to include more terms of relevance to a particular row in the query result is to look at columns that are present in the underlying tables in the database, but do not appear in the query result because they are projected out for convenience. This is easily incorporated by analyzing the input SQL query and removing the projection constraints upfront. However, the algorithm so far is still limited to the boundaries of tables that, in effect, are decided by how *normalized* the database schema is.

Normalization results in distribution of related information across several tables, connected by foreign-key relationships. For instance, consider a bio-classification database with two tables: `Species` with one row per distinct species and `Genus` with one row per distinct genus. The information about the genre of a particular species in the `Species` table is encoded as a *foreign key* "pointing" to the corresponding row in the `Genus` table. In this section, we show how the algorithm presented in the previous section can be extended in order to exploit these relationships among the tables.

3.2.1 Alternative Relationship Directions: Forward vs. Backward

Before we get into the details of this extension, it is important to realize that there are two ways in which the foreign key relationships can be exploited. The first, as illustrated above, is by following the foreign-key "pointers" in the *forward* direction; this takes us from sub-concepts to encompassing super-concepts—for instance, from species to their genus in the bio-classification database. In relational terms, this involves joining with tables referenced by foreign key columns present in tables included in the query. The second way is by following the foreign keys in the *backward* direction; this takes us from super-concepts to *all* encompassed sub-concepts—for instance, from genus to all the species in that genus in the bio classification database. In relational terms, this involves joining with tables that contain foreign key columns that reference tables included in the query.

Currently SCORE exploits the foreign-key relationships in the forward direction only. This is justified since exploration in the forward direction works well in most target applications. For instance, the Star/Snowflake schema, commonplace in the OLAP/BI environments, consists of a "fact" table with foreign-keys to multiple "dimension" tables, and queries on this schema typically involve the fact table. Exploiting backward relationships is an interesting future work.

3.2.2 Exploring Forward Relationships

Moving back to the details of the algorithm, recall that for each row in the query result, we want to follow the foreign-key pointers and gather more terms, beyond those present in the query result. In relational terms, this amounts to augmenting the query by adding a *foreign-key (left-outer) join* with the referenced table. Note that, with this perspective, following foreign keys in the forward direction also has the desirable effect that the extra information is just an appendage to the original query result, which remains untouched. In other words, the original query result can be extracted from the augmented query result by simply reintroducing the projection constraints; formally, if Q is the original query and AQ is the augmented query then $Q = \pi_{cols(Q)}(AQ)$.

In essence, thus, the algorithm aims to extend the input query (with its projection constraints already removed) by augmenting with other tables in the database reachable through foreign-key relationships. To achieve this, two issues need to be addressed. First, joining with *all* possible tables reachable through foreign-key relationships might be too expensive, and unnecessary, and we need to select a subset. In the rest of this section, we show how SCORE selects the optimal subset from the possible alternatives.

Since we need to limit the computation involved, we define a parameter M as the maximum number of tables that can be augmented to the query. Ideally, we would like to select a subset of M tables on which the query has maximum focus; but before we can do that, we need to quantify the focus of the query on a table.

Let $F \in cols(Q)$ be a foreign-key column in the query, and let R be the table referenced by F[7] Since each term t in the column F references a single row in R, we can interpret $TW(F,t)$, our measure of the query's focus on the term t (Section 3.1), as a measure of the query's focus on the corresponding row in R as well. Now, given two foreign-key columns $F_A, F_B \in cols(Q)$ referencing tables R_A and R_B respectively, we say that the query is more focused on R_A than on R_B if there exists a term $t \in F_A$ such that the

[5]We abuse notation in this paper, with $t \in A$ meaning $|\sigma_{A=t}(Q(R))| > 0$.

[6]This definition of TW is in the spirit of the "tf-idf" weightings commonly used in Information Retrieval literature [2]; it is intuitive and works well on our benchmark. However, we emphasize that SCORE is extensible enough to support reasonable alternatives.

[7]We assume that each foreign key references a unique table. In case foreign keys F_1 and F_2 both reference the same table R, then we alias (clone) R into R_1 and R_2 and interpret the latter as two different tables (same semantics as R AS R_1, R AS R_2 in SQL's FROM clause).

query's focus on t is more than its focus on any term in F_B. The focus of a query Q on the table referenced by a foreign-key column $F \in cols(Q)$ is accordingly measured using a column weight function CW defined as:

$$CW(F) = \max_{t \in F} TW(F, t)$$

A simple-minded algorithm to select the optimal subset of tables could thus be to find, by traversing the schema graph, all the tables reachable from the tables already present in the query by foreign-key relationships and pick the M tables with the maximum focus. However, this algorithm is not practical—not only because it is exponential-time in the number of tables, but also because the query's focus on a table is known only if the corresponding foreign key $F \in cols(Q)$ (in other words, we only know the query's focus on the tables directly referenced by the query, but not on the tables referenced in turn by the foreign-keys in these tables).

In SCORE, thus, we follow a greedy strategy wherein we iteratively build up the set of tables to augment with. The algorithm maintains the set S of candidate foreign-key columns in the query as augmented so far (call this the augmented query AQ); since $S \subseteq cols(AQ)$, we know $CW(F)$ for each foreign key column $F \in S$. In each iteration, the algorithm picks $F \in S$ with the maximum $CW(F)$ and augments AQ with the table referenced by F; as it does so, it computes the weights of the foreign-key columns added as a result and replaces F in S by these foreign keys. The algorithm is presented formally in Section 3.3.

3.2.3 Scaling the weights of added terms

Since the terms in the columns added as a result of the query expansions as discussed above are not part of the original query result, it can be argued that the weights of these terms should be scaled down. Since this is subjective, depending on the way the data is structured and on the specific queries that form part of the workload, we leave the decision to the user (or the DBA) and define a tunable scaling parameter $\beta \in [0, 1]$ that scales down the weights of these terms from what they would have been if these columns formed part of the original query. For all the cases we came across, however, it seemed reasonable to take $\beta = 1$ (no scaling).

3.3 The Algorithm

The discussion so far on how to compute the context of a given query leads to the algorithm in Figure 2. In this section, we describe this algorithm in detail.

The procedure QueryContext takes as input the query Q. The first step (line 1) is to remove the projection constraints in Q (ref. Section 3.2). Next, the set S of the initial candidate foreign-key columns is constructed (line 2) and CW is computed for these columns (lines 3-4). The procedure next enters a loop, wherein in each iteration (lines 6-14) it picks the most "promising" candidate foreign-key, joins with the referenced table and computes CW for the foreign-key columns added to AQ as a result, closely following the discussion in Section 3.2.2. The loop exits when no more candidate foreign keys exist, or when M augmentations have already been performed. The terms in AQ's result are then, as discussed in Section 3.1, ranked according to their term weights (TW), and the top N terms, along with the corresponding column names and their overall weights, are returned as the context of the query Q (lines 15-16).

Procedure QueryContext(Q)
Input: Query Q
Output: Context of the query Q
Begin
1. Let $AQ = Q$ without the project constraints
2. Let $S = \{A \mid A \text{ is a FK} \in cols(AQ)\}$
3. For each $A \in S$
4. Let $CW(A) = \max_{t \in A} TW(A, t)$
5. Let $n = 0$
6. While $S \neq \phi$ and $n < M$
7. Let $n = n + 1$
8. Let $F = \text{argmax}_{A \in S} CW(A)$
9. Let $S = S - \{F\}$
10. Let R be the table referenced by F
11. Let $AQ = AQ \rtimes_F R$
12. For each FK $A \in cols(R)$
13. Let $CW(A) = \max_{t \in A} TW(A, t)$
14. Let $S = S \cup \{A\}$
15. Let $K = \{[t, A, w] \mid t \in A, A \in cols(AQ), w = TW(A, t)\}$
16. Return top N elements in K based on w
End
M is the maximum number of augments allowed
N is the maximum number of terms allowed in the context

Figure 2: Context Computation Algorithm

4. IMPLEMENTATION

In the previous section, we formalized the context computation process in terms of an algorithm. The challenge in efficiently implementing this algorithm (cf. Figure 2) arises primarily due to the interplay between the term weight computations and the query augmentations in the algorithm. Recall that in every iteration, in order to decide which foreign-key column to follow, the algorithm needs the column weights $CW(F)$ of the foreign-key columns $F \in S$ at that point; however, to compute the same, the algorithm needs to know all the terms $t \in F$ and their distribution in the query result, so that it can compute $TW(F, t)$. We considered three alternate implementation approaches.

The **Brute-Force Approach (BFA)** solves this problem by actually executing the augmented query so far before *each* iteration. Furthermore, after all the augmentations are done, the final query is executed as well and the result used to compute the context. ***Advantages:*** Simplicity. Follows the algorithm exactly. ***Disadvantage:*** Does not scale.

The **Simple Histogram-based Approach (SHBA)** refines the Brute-Force approach by exploiting histogram-based estimation techniques [21] to avoid expensive query executions. Only the initial query is actually executed; the term-weights and column-weights on the augmented columns in each step are *estimated* using histograms. Since the quality of the context relies on value correlations across columns, the one-dimensional histograms are not appropriate for these estimations—we need two-dimensional histograms. Unfortunately, since DB2 does not support two-dimensional histograms, these histograms are computed in a preprocessing step and stored locally. However, note that the augmentations are essentially foreign-key joins. Thus, in order to estimate the distribution of a column added as a result, we only need the two-dimensional histograms for the column against the underlying table's primary key; the extra space needed to store these histograms is thus not significant. Further note that since primary keys are involved, the histograms necessarily needs to bucket the primary key values. ***Advantages:*** Efficient, scales well. ***Disadvantages:*** Term and

column weight estimations using histogram might result in different choices in augmentation as compared to the exact brute-force approach. Thus, the returned context may not include some keywords present in the brute-force implementation's context ("false-negatives"). Furthermore, since the bucketing of the primary keys in the histograms makes relevant values in a bucket indistinguishable from the irrelevant values, the context might include several irrelevant keywords ("false-positives").

The **Modified Histogram-based Approach (MHBA)** further refines the Simple Histogram-based approach to remove the false-positives problem. The idea is to augment the queries based on estimated term and column weights, just like the Simple Histogram-based approach; but once the final augmented query has been formed, execute it on the database. The context is then computed based on the exact result of this query, eliminating the possibility of false-positives. ***Advantages:*** No false-positives. Efficient, scales well. Note that since histograms are only used for augmentation, for a given table, only the two-dimensional histograms for the foreign-keys in the table against the table's primary key are needed. ***Disadvantages:*** False-negatives—the returned context may miss some relevant keywords; however, as we demonstrate in Section 5, the difference is marginal on our benchmark.

5. EXPERIMENTAL STUDY

In this section, we present a preliminary experimental study to evaluate the techniques and design decisions proposed above.

System Parameters and Configuration. For the purpose of this experimental study, the parameters M, N and β (ref. Section 3) were assigned values 3, 10 and 1.0 respectively. Each histogram consisted of only the top 1000 buckets; the remaining buckets were assumed to be of equal size. The implementations were done in Java with J2SE v1.4.2 (approx. 3000 lines of code), and executed on a Pentium-M 1.5 GHz IBM Thinkpad T40 with 768 MB RAM running Windows XP SP1. The relational database used was IBM DB2 v8.1.5 co-located on the same machine. The page size was 4KB and the associated total bufferpool size was 250 pages (1000KB). The communication between SCORE and DB2 was through JDBC.

Dataset. The evaluation was performed on a subset of the Open Music Directory data (http://www.musicmoz.org) containing information about 10^6 music tracks, 10^5 records, 10^5 members and $3 * 10^4$ bands, each stored in a separate table and linked using foreign keys. To make the schema more complex, we introduced a dummy table and added foreign keys to this table from the other tables. The total size of the dataset was 116 MB.

Workloads. The generated workloads consist of selection queries resulting in a specified number $NTRACKS$ of tracks; for each query, the specific $NTRACKS$ tracks are chosen such that they have a common band (called *band* queries), or a common record (called *record* queries). These workloads are generated randomly from the data given $NTRACKS$ as a parameter and have an equal mix of band and record queries.

Each generated query is also associated with a *target context term*—that is, the term of maximum relevance to the query among all the terms in the database. Specifically, each band/record query has the band/record's name (as mentioned in the corresponding row in the bands/record table) as its target context term.

Evaluation Metrics. This study uses the following metrics for measuring context quality and computational overheads. ***Context Quality Metric:*** Recall that the target context term t_Q is, by design, the term in the entire database most relevant to the query Q. Thus, we expect t_Q to be the the highest ranked keyword in the context returned by SCORE for each query. We choose a quality metric that quantifies this expectation.[8]

Given the query Q, let t_Q be its target context term and let $C(Q) = [t_1, t_2, \ldots, t_N]$ be the list of N keywords retrieved by SCORE as its context, ranked in decreasing order of their term-weights. We define the *Reciprocal Rank (RR)* of SCORE for the query Q as:

$$RR(Q) = \begin{cases} 1/n & \text{if } t_Q \in C(Q) \text{ and } t_Q = t_n \\ 0 & \text{if } t_Q \notin C(Q) \end{cases}$$

The *Mean Reciprocal Rank (MRR)* of SCORE on a query workload S is then defined as $\text{MRR(S)} = \frac{1}{|S|} \sum_{Q \in S} RR(Q)$, i.e. the mean RR of SCORE over the queries $Q \in S$.

We use this MRR as our quality metric in the experiments in this section. MRR is not a novel metric; in fact, it is the standard metric for the quality of Question-Answering systems in IR [24], a closely related problem (ref. Section 2). ***Computational Overhead Metric:*** We measure the computational overheads in terms of the additional time spent on a query when using SCORE as compared to when *not* using SCORE.

EXPERIMENT 1

Purpose: To evaluate efficacy of the query augmentation algorithm and study how augmenting the queries impacts the context quality and computational overheads for BFA, SHBA and MHBA respectively.

Methodology: We generated a workload with fixed result size $NTRACKS = 32$, and for increasing number of augmentations $M = 0$ (no augmentation), $1, 2, 4$ and 8 invoked the BFA, SHBA and MHBA implementations for the workload and computed the respective MRR and the overheads.

Result: The quality (MRR) and overhead results are summarized in the following table.

#aug (M)	0	1	2	4	8
BFA quality (MRR)	0.00	0.47	0.98	1.00	1.00
BFA overhead (ms)	5	28	67	171	502
SHBA quality (MRR)	0.00	0.10	0.15	0.16	0.16
SHBA overhead (ms)	7	12	30	87	167
MHBA quality (MRR)	0.00	0.47	0.61	1.00	1.00
MHBA overhead (ms)	5	20	31	54	92

Discussion: Recall that the target context term for a record query is reached after augmentation with the records table, while that for a band query is reached after augmentation

[8]Note that we measure the quality of the context instead of the quality of the search results. This is because (a) the search engine is an *external* component in SCORE, and factoring its quality in this evaluation is not fair, (b) the techniques proposed in this paper only involve obtaining the context, and (c) assuming that the search engine is well-behaved in the sense that better quality context will lead to better quality search results, measuring the quality of the context is equivalent to measuring the quality of the corresponding search result.

with both the records and the bands table.[9] We see that the MRR for BFA improves from 0.00 for no augmentation (expected, since the target context term is not present in the TRACKS table) to 0.47 for $M = 1$ (most record queries find the target context term) and 0.98 for $M = 2$ (most record and band queries find their target context term). This implies that for almost all queries in the workload, the augmentations occur in the optimal order, clearly validating the choice of TW and CW, and attesting to the efficacy of the context computation algorithm.

However, BFA's accuracy comes at the price of a high overhead. Comparing with the results for SHBA, we see that using in-memory histogram-based estimations instead of actually querying the data results in drastic reduction of the overheads, but the MRR falls drastically as well. In contrast, the MRR for MHBA is almost the same as that for BFA and, surprisingly, is achieved with even less overhead than SHBA for $M > 2$. This happens because, as the number of augmentations increase, so do the number of augmented columns; eventually, for $M > 2$, the overhead for estimating the distributions for all the augmented columns becomes more than the cost of actually evaluating the final augmented query.

Overall, for all the approaches, the overheads increase rapidly with increasing number of augmentations; the increase is highest for BFA and the lowest for MHBA. For MHBA, however, the overhead of 92 ms for even $M = 8$ is not very significant. We thus conclude that MHBA is clearly the implementation of choice, achieving a quality comparable to BFA with a very reasonable overhead even in the presence of a large number of augmentations.

EXPERIMENT 2

Purpose: To study how increasing query size impacts the context quality and computational overheads for BFA, SHBA and MHBA respectively.

Methodology: We generated workloads with result size $NTRACKS = 12, 20, 28, 36, 44$ and 52 respectively. For each workload, we invoked each SCORE implementation and computed the respective MRRs and the overheads. (The number of augmentations were fixed at $M = 4$.)

Result: Summarized in the following table.

NTRACKS	12	20	28	36	44	52
BFA qual (MRR)	1.00	1.00	1.00	1.00	1.00	1.00
BFA ovrhd (ms)	52	99	158	171	183	196
SHBA qual (MRR)	0.22	0.21	0.18	0.20	0.21	0.30
SHBA ovrhd (ms)	86	77	82	82	99	97
MHBA qual (MRR)	1.00	1.00	1.00	1.00	1.00	1.00
MHBA ovrhd (ms)	27	40	53	63	58	68

Discussion: The results further corroborate our observations in Experiment 1 about the relative context quality determined by the respective approaches and their relative overheads. In these results, we further observe that the overhead for BFA increases rapidly with increasing query size. The overhead for MHBA, in sharp contrast, shows a very slow and roughly linear increase. It is again evident that MHBA is consistently more efficient as compared to SHBA; the overhead for MHBA even for a query size of 52 rows is 68 ms, which is again very reasonable, almost one-third of the corresponding overhead for BFA. These observations clearly validate the design choices made in formulating the MHBA approach.

[9]Incidentally, this motivates the need to explore beyond the query result that spans only the tracks table.

EXPERIMENT 3

Purpose: To study how increasing the number of buckets per histogram (equivalently, memory overhead) impacts the context quality and computational overheads for BFA, SHBA and MHBA respectively.

Methodology: We generated a workload with result size $NTRACKS = 32$. Then, fixing the number of buckets per histogram (NBUCKETS) as 125, 250, 500 and 1000 respectively, we invoked each SCORE implementation and computed the respective MRRs and the overheads. (The number of augmentations were fixed at $M = 4$.)

Result: Summarized in the following table.

NBUCKETS	125	250	500	1000
BFA quality (MRR)	1.00	1.00	1.00	1.00
BFA overhead (ms)	170	173	170	172
SHBA quality (MRR)	0.03	0.02	0.10	0.22
SHBA overhead (ms)	4	10	35	77
MHBA quality (MRR)	0.60	0.71	1.00	1.00
MHBA overhead (ms)	50	53	52	57

Discussion: From the results, we note that SHBA is most sensitive to NBUCKETS; this is expected since it performs most computations on the histograms. Each histogram multiplication performed is roughly linear in NBUCKETS; however, given the large number of histograms involved (one per column in the augmented query), the overall increase is very rapid. In contrast, MHBA performs histogram computations only for the foreign key columns in the augmented query, which are a small fraction of the total number of columns. Accordingly, MHBA's overhead increases rather slowly with increasing NBUCKETS. The context quality of SHBA and MHBA improves with increasing NBUCKETS. For NBUCKETS = 125, the fraction of the database represented in the histograms is small, leading to a large number of false-negatives in SHBA and MHBA. However, even at such a small value of NBUCKETS, MHBA achieves MRR of 0.60, which is encouraging. As NBUCKETS increases, the fraction of the database represented in the histograms increases, leading to fewer false-negatives and therefore a better quality context. (BFA context computation does not involve histogram operations; BFA results appear in the results above as a baseline.)

EXPERIMENT 4

Purpose: To study how increasing the database size impacts the computational overheads for MHBA.

Methodology: We generated a workload with result size $NTRACKS = 32$. Starting with the initial database of size 116MB, we generated artificial datasets of size 232MB, 464MB and 928MB by repeated duplication. For each generated database, we invoked the MHBA SCORE implementation and computed the respective overheads on the generated workload. (The number of augmentations were fixed at $M = 4$).

Result: Summarized in the following table.

DBSIZE (MB)	116	232	464	928
MHBA overhead (ms)	57	67	171	571

Discussion: MHBA scales well with increasing DBSIZE. Even for DBSIZE of 928MB, the overhead is only 571ms. This is because the augmentations introduced in the input query are merely foreign-key joins that can efficiently exploit the primary key indexes for fast execution.

6. ANECDOTAL EXAMPLE

The following query, that selects ten titles by The Beatles, was executed on the dataset mentioned in Section 5:

`SELECT * FROM TRACKS WHERE TRACKS.title IN` *('Octopus"s Garden', 'Things We Said Today', 'A Taste Of Honey', 'Thank You Girl', 'Yesterday', 'All You Need Is Love', 'Twist And Shout', 'I"ll Get You', 'She Loves You', 'I"m Gonna Sit Right Down and Cry (Over You)')*

In response to the above SQL query, SCORE (MHBA implementation) returns the following context:

Keyword	Source Column	TW
George Harrison	`MEMBER.NAME`	642.55
guitarist, vocalist, sitar	`MEMBER.ROLE`	642.55
The Beatles	`BAND.NAME`	642.55
1957	`BAND.YEAR`	333.38
The Quarrymen	`BAND.FOUNDEDAS`	242.62
Pop	`RECS.STYLE`	218.49
England	`BAND.COUNTRY`	192.14
Yesterday	`TRACKS.TITLE`	101.99
She Loves You	`TRACKS.TITLE`	101.99
Octopus's Garden	`TRACKS.TITLE`	81.59

As we can see, the input SQL query was augmented with the `RECS`, `MEMBER` and `BAND` tables. The system identified "George Harrison" and "guitarist, vocalist, sitar" from the `MEMBER` table, and "The Beatles" from the `BAND` table as the most relevant terms, which is encouraging. It also identified the band's foundation year, its foundation name and its country from the `BAND` table, and the style from the `RECS` table. Apart from these keywords from the neighborhood, the context includes, with lower weight, the tracks titles themselves.

Notice that SCORE is able to filter out terms such as "CD" in `RECS.format` that relate little to the input query even though they are present in most rows of the query result. This validates our choice of TW (ref. Section 3.1), that not only considers the popularity of a term in the query result, but also its rarity in the rest of the underlying database.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced a *context-oriented* approach to associate unstructured content with structured database query results and described the design and implementation of SCORE, a system based on this approach. We developed an efficient algorithm to compute the context of a SQL query by analyzing the query result as well as related information elsewhere in the database. The computed context is used to retrieve relevant unstructured content and associated with the result of the given SQL query. We also described the implementation challenges involved, and presented a solution that achieves high context quality with reasonable overheads by judicious exploitation of available histogram information. Finally, we presented an experimental study that showed that SCORE is able to compute the high-quality context of SQL queries with reasonable performance overheads.

SCORE, as discussed in this paper, associates unstructured content as a whole with the entire query result; we are currently working in an extension wherein the query result is first clustered based on the context [4], and then relevant unstructured content is associated with each individual cluster. An interesting future direction for this work is to incorporate the ideas here in an XML/XQuery environment. Also, instead of considering the context of a single, isolated query, it would be interesting to investigate the context of an entire query session instead. As mentioned earlier, we also plan to extend SCORE by enabling it to exploit the foreign-key relationships in the backward direction as well.

Acknowledgments

We would like to thank Laura Haas, Pat Selinger and Bill Cody for their ongoing help and support in the execution of this project, and Dan Wolfson, Ullas Nambiar, Laurent Mignet, Krishna Kummamuru and Himanshu Gupta for helpful discussions.

8. REFERENCES

[1] Agrawal, S., Chaudhuri, S., and Das, G. DBXplorer: A System for Keyword-Based Search over Relational databases. In *ICDE* (2002).

[2] Baeza-Yates, R., and Ribeiro-Neto, B. *Modern Information Retrieval*. Addison Wesley/ACM, 1999.

[3] Balmin, A., Hristidis, V., and Papakonstantinou, Y. ObjectRank: Authority-Based Keyword Search in Databases. In *VLDB* (2004).

[4] Bamba, B., Roy, P., and Mohania, M. OSQR: Overlapping Clustering of Query Results. In *CIKM* (2005).

[5] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., and Sudarshan, S. Keyword Searching and Browsing in Databases using BANKS. In *ICDE* (2002).

[6] Bruce, H., Halevy, A., Jones, W., Pratt, W., Shapiro, L., and Suciu, D. Information retrieval and databases: Synergies and syntheses. http://www2.cs.washington.edu/nsf2003, 2003.

[7] Chaudhuri, S., Dayal, U., and Yan, T. W. Join queries with external text sources: Execution and optimization techniques. In *SIGMOD* (1995).

[8] Cody, W. F., Kreulen, J. T., Krishna, V., and Spangler, W. S. The integration of business intelligence and knowledge management. *IBM Sys. J. 41*, 4 (2002).

[9] Cohen, W. W. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. *ACM Trans. Inf. Syst. 18*, 3 (2000).

[10] DeFazio, S., Daoud, A., Smith, L. A., Srinivasan, J., Croft, B., and Callan, J. Integrating IR and RDBMS Using Cooperative Indexing. In *SIGIR* (1995).

[11] Dixon, P. Basics of Oracle Text Retrieval. *IEEE Data Engg Bull. 24*, 4 (2001).

[12] Fisher, M., and Sheth, A. Semantic enterprise content management. In *Practical Handbook of Internet Computing* (2004), Chapman & Hall/CRC.

[13] Fuhr, N. A Probabilistic Relational Model for the Integration of IR and Databases. In *SIGIR* (1993).

[14] Goldman, R., and Widom, J. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *SIGMOD* (2000).

[15] Haas, L. M., Lin, E. T., and Roth, M. A. T. Data integration through database federation. *IBM Sys. J. 41*, 4 (2002).

[16] Hamilton, J., and Nayak, T. Microsoft SQL Server Full-Text Search. *IEEE Data Engg Bull. 24*, 4 (2001).

[17] Jhingran, A., Mattos, N., and Pirahesh, H. Information integration: A research agenda. *IBM Sys. J. 41*, 4 (2002).

[18] Josifovski, V., Schwarz, P. M., Haas, L. M., and Lin, E. Garlic: a new flavor of federated query processing for DB2. In *SIGMOD* (2002).

[19] Kashyap, V., and Sheth, A. Semantic heterogeneity in global information systems. In *Cooperative Information Systems* (1998), Academic Press.

[20] Maier, A., and Simmen, D. DB2 Optimization in Support of Full Text Search. *IEEE Data Engg Bull. 24*, 4 (2001).

[21] Poosala, V. *Histogram-based estimation techniques in database systems*. PhD thesis, University of Wisconsin, Madison, WI, USA, 1997.

[22] Raghavan, P. Structured and unstructured search in enterprises. *IEEE Data Engg Bull. 24*, 4 (2001).

[23] Somani, A., Choy, D., and Kleewein, J. C. Bringing together content and data management: Challenges and opportunities. *IBM Sys. J. 41*, 4 (2002).

[24] Voorhees, E., and Tice, D. The TREC-8 question answering track evaluation. In *Proc. Eighth Text Retrieval Conference (TREC-8)* (1999).