Towards open-source shared implementations of keyword-based access systems to relational data

Alex Badan, Luca Benvegnù, Matteo Biasetton, Giovanni Bonato, Alessandro Brighente, Alberto Cenzato, Piergiorgio Ceron, Giovanni Cogato, Stefano Marchesin, Alberto Minetto, Leonardo Pellegrina, Alberto Purpura, Riccardo Simionato, Nicolò Soleti, Matteo Tessarotto, Andrea Tonon, Federico Vendramin, Nicola Ferro

University of Padua, Italy

{alex.badan, luca.benvegnu.2, matteo.biasetton, giovanni.bonato, alessandro.brighente, alberto.cenzato, piergiorgio.ceron, giovanni.cogato.1, stefano.marchesin, alberto.minetto, leonardo.pellegrina, alberto.purpura, riccardo.simionato.1, nicolo.soleti, matteo.tessarotto.1, andrea.tonon.3, federico.vendramin}@studenti.unipd.it

ferro@dei.unipd.it

ABSTRACT

Keyword-based access systems to relational data address a challenging and important issue, i.e. letting users to exploit natural language to access databases whose schema and instance are possibly unknown. Unfortunately, there are almost no shared implementations of such systems and this hampers the reproducibility of experimental results. We explore the difficulties in reproducing such systems and share implementations of several state-of-the-art algorithms.

1. INTRODUCTION

Keyword-based access to relational data is a key technology for lowering the barriers of access to the huge amounts of data managed by databases. It is an extremely difficult and open challenge [1] since it has to face the "conflict of impedance" between vague and imprecise user information needs and rigorously structured data, allowing users to express their queries in natural language against a potentially unknown database.

Improving a technology is greatly eased by the existence of a reference architecture paired with a proper evaluation framework in order to have a coherent vision of the components we have to leverage on and to precisely measure and track performances over time. Unfortunately, the situation is very fragmented in the case of keyword-based access to relational data, since both a fully-fledged reference architecture and an evaluation methodology are still missing [2].

In this paper, we start to investigate the problem of the *reproducibility* of the experimental results for keyword-based access systems. Reproducibility is becoming a primary concern in many areas of science and it is a key both for fully understanding state-of-the-art solutions and for being able to compare against and improve over them [6, 7]. However, there is a lack of commonly shared open source platforms implementing state-of-the-art algorithms for keyword-based access to relational data as, for example, Terrier[1] is in the information retrieval field; on the contrary, you mostly need to re-implement each system from scratch [4].

Therefore, as part of a student project during the course on databases of the master degree in computer science at the University of Padua, we considered several state-of-the-art algorithms and we implemented them from scratch, in order to understand the difficulties and pitfalls in implementing them and to go towards a shared implementation of them.

The paper is organized as follows: Section 2 presents the related work; Section 3 introduces the implementation of the different algorithm; Section 4 reports the lessons learned; finally Section 5 wraps up the discussion and outlooks some future work.

2. RELATED WORKS

In the following, we briefly summarize the algorithms implemented in this paper. For a broader perspective on keyword search over relational database, please refer to [13, 14].

2.1 Graph-based Approaches

This type of algorithms starts with the construction of a graph where nodes are tuples and edges are the forward (foreign) key constraints and their goal is to find a set of trees that connects a group of nodes containing all the keywords in an input query. Weights between nodes and graph type (direct or indirect) are dependent on each specific approach. Moreover, this type of algorithms differ for the use of Dijkstra or Steiner methods to visit the graph: the former allows you to find the shortest spanning tree using only the selected nodes, i.e. those that match with the query key-

[1] http://www.terrier.org/

words; the latter allows you to use other extra nodes which are not included in the previous selection but exist in the original graph.

Browsing ANd Keyword Searching I (BANKS-I) [3] The algorithm adds a backward weighted edge for each directed edge in the graph and assigns a relevance score to each node in it. Then, it proceeds by finding the shortest path from each keyword node to all the other nodes in the graph. Each path is computed by running Dijkstra's single source shortest path algorithm starting from each one of the keyword nodes and traversing all the edges in reverse direction. This process is called backward expanding search. When multiple paths intersect at a common node r in the graph, the resulting tree with root r and the nodes containing the keywords as leaves is examined. If the tree contains all the keywords in the input query, then its weight is computed and the tree is added to a heap data structure of fixed size, which contains all the result trees in decreasing order of relevance. When the heap is full or all the trees have been generated, the most relevant of them are returned until a predefined number of results has been returned.

Browsing ANd Keyword Searching II (BANKS-II) [10] Bidirectional Search improves on BANKS-I by allowing forward search from potential roots towards other keyword nodes. The algorithm uses two concurrent iterators, called *outgoing* and *ingoing* to explore the graph in both directions. Both the iterators use the Dijkstra's single source shortest path approach. BANKS-II also allows for preferential expansion of paths that have less branching using a spreading activation mechanism to prioritize the most promising paths and to avoid wasteful exploration of the graph.

Dynamic Programming Best-First (DPBF) [5]: it tries to build a minimum cost Steiner tree: for each neighbour v of the root in the graph, it creates a new tree with v as the new root; the algorithm tries to merge trees with the same root and a different set of keywords. If a tree contains all the searched keywords, it is a solution for the algorithm. Given a root and a specific set of keywords, DPBF keeps only the tree with lower weight, i.e. the best one for the specific root node and set of keywords, and it guarantees to be a total, minimal and controlled in size solution.

Steiner-Tree Approximation in Relationship graphs (STAR) [11]: it initially looks for tuples containing one or more of the query's keywords (let's call this set $V' \subset V$) and then tries to connect these tuples into a least-weight tree, i.e. computing the best Steiner tree between the given nodes which total weight is $\sum_{e \in E} w(e)$. In order to build a first interconnecting tree, STAR relies on a strategy similar to BANKS-I but, instead of running single source shortest path iterators from each node of V', STAR runs simple breadth-first-search from each terminal, called in a round-robin manner. As soon as the iterators meet, a result is constructed. In the second phase, STAR aims at improving the current tree iteratively by replacing certain paths in the tree by new paths with a lower weight from the underlying graph.

2.2 Schema-based Approaches

Both DISCOVER-I and II use generalized inverted-index, called Master Index and IR Engine respectively, to retrieve tuples containing the input keywords, given by the user, from the database. Both output a list of Candidate Networks, albeit defined slightly differently.

DISCOVER-I [9] Given a relational database and a set of keyword **K**, Discover exploits the database schema to generate a graph G_{TS}: from Basic Tuple Sets $\overline{R}_i^{k_j}$, set of tuples of the relation R_i containing the keyword $k_j \in K$, $\forall i, j$, it creates a node for each Tuple Set

$$R_i^{K'} = \bigcap_{k \in K'} \overline{R}_i^k - \bigcup_{k \notin K'} \overline{R}_i^k, \forall K' \subseteq \mathbf{K} \qquad (2.1)$$

sets of tuples of the relation R_i containing all and only the keywords of K', and one for each Free Tuple Sets, the database relations. Edges are representative of each PK to FK relationship. The algorithm then computes a complete non-redundant set of Candidate Networks up to a maximum number of joints T, pruning all the joining networks of tuples that are not minimal (i.e no tuples without keywords as leaves and do not contain the same tuple twice) and total (i.e. contain every keyword $\in$ **K**, AND semantics).To evaluate the Candidate Networks efficiently, Discover eventually computes and stores a list of intermediate results of common join subexpressions. Finding these intermediate results is an *NP-complete* problem, so it uses an iterative greedy algorithm based on frequencies: the most frequent joint expression is stored in a view at every iteration and then reused when possible. The results are finally displayed to the user ordered by size.

DISCOVER-II [8] The algorithm uses an IR engine module when a query Q is issued, in this way it extracts from each relation R the tuple set R^Q with $Score > 0$, where Score is a value for each tuple using a IR-style relevance scoring function. Candidate Networks are hence generated based on the tuple sets returned by the IR engine and the database schema, which are join expressions used to create joining trees of tuples which are potential answer to the issued query. The final step is to identify the top-k results, task which is performed by the execution engine module where it uses: Naive Algorithm which takes the top-k results from each CN and it combine in a sort-merge manner to identify the final top-k results of the given query; Sparse Algorithm computes a bound on the maximum possible score of the tuple tree derived from a CN.

3. IMPLEMENTATION

Students split up in groups of 3-4 people and each group was responsible for the implementation of one algorithm. All the algorithms have been implemented in Java while PostgreSQL has been used as relational engine.

The implementation of the different algorithms is available as open source under BSD license[2].

We briefly summarize the main features of each implementation:

- BANKS-I The required graph is created prior to the execution of the algorithm executing multiple queries on the selected database without any prior knowledge of its schema. In our implementation, for each node in the graph, only the ID of the tuple is saved and an external file is used to store the information. The graph is represented using adjacency lists. In order to speed up the creation of the graph, we used the fastutil library[3],

[2] https://bitbucket.org/ks-bd-2015-2016/ks-unipd

[3] http://fastutil.di.unimi.it/

in addition to the Java Standard Library. Finally, to optimize the performance while executing multiple instances of Dijkstra's algorithm (one for each keyword node), we chose to compute the next node to be returned by the algorithm only when it was required. In fact, most of the times, only a handful of nodes were required to build a valid result tree. With this approach we avoided executing Dijkstra's algorithm on the whole graph, optimizing the execution time and the memory usage. For the execution of each query the memory limit was set to 5 Gb while the maximum running time was set to 1 hour.

- BANKS-II We divided our implementation in two phases: pre-processing and the BANKS-II execution. We implemented the graph in a OO manner. The vertexes are stored using a two-layered HashMap, to improve performances and to reduce as much as possible the probability of collisions. The first layer refers to the relation name, while the second layer uses as index the primary key of the tuple. The paper suggested a set of different weight assignment strategies (i.e. Pagerank); we decided to use equally weighted edges for simplicity. Once the user provides the keywords, the pre-processing ends with the creation of the matching sets. These sets are also organized in a two-layered HashMap data structure: vertexes are inserted in the respective matching sets using SQL queries to find tuples with attributes having matching values. During the BANKS-II execution, *outgoing* and *ingoing* iterators start to explore the graph, using two priority queues to extract the highest activated node. We created these queues using the standard Java PriorityQueue class, defining a custom compare method. When a new solution (a rooted tree connecting all the matching sets) is found, we store it in a PriorityQueue; these results are ranked with a score given by the overall sum of the edge weights. Our implementation creates the graph from the DB once, and then accepts keywords from the user to perform multiple searches.

- DISCOVER-I We first implemented the Master Index to perform the full text search through the function *to_tsvector* provided by PostgreSQL and then we created the initial graph G_{TS} using an adjacency matrix. We observed that a lot of queries were repeated while computing Tuple Sets because the same Basic Tuple Sets were created many times by the DBMS "on the fly". Thus we decided to store a new table for each Basic Tuple Set to be reused while performing unions and interceptions as in (2.1).To further improve performances of full text search we preprocessed the main search fields to remove english stopwords and to convert all tokens to lowercase characters. We then stored the lexemes in a dedicated Generalized Inverted Index. The Candidate Network algorithm then prune, expand and accept the Candidate Networks using static methods. The accepted ones (treated as graphs) were rewritten, by the Plan Generator, replacing repeated joins inside them (i.e. the same couples of nodes and the connecting edge) with a reference (i.e. a new node) to a new temporary table. Finally queries were evaluated from the resulted graphs with a BFS algorithm in order to guarantee the correct order of joins.

- DISCOVER-II The graph was built using the information schema and the algorithm was implemented in java using native collections. The IR Engine module identifies all database tuples that have a non-zero score for a given query, then each occurrence of a keyword is recorded as a tuple-attribute pair. Moreover, the IR Engine relies an inverted index that associates each keyword that appears in the database with a list of occurrences of the keyword, which we implemented using the GIN index of PostreSQL. To obtain better performances, we store as a view the tuple sets identified for a query Q. The Candidate Network Generator involves a combination of free and not-free tuple set, the former represent the tuple set with no occurrences of the query keyboards, that help via foreign-key join the connection of non-free tuple set, the latter contains at least one keyword. The Execution Engine module receives as input the set of CNs along with the non-free tuple sets, it contacts the RDBMS's query execution engine repeatedly to identify the top-k query results. We implemented two algorithm: the Naive algorithm, considering a Boolean-OR semantics, issues a SQL query for each CN to answer a top-k query, then combines the results in a sort-merge manner to identify the final top-k results of the query; the Sparse algorithm discards at any point in time any (unprocessed) CN that is guaranteed not to produce a top-k match for the query, it computes a bound MPS_i on the maximum possible score of a tuple tree derived from a CN C_i. If MPS_i does not exceed the actual score of k already produced tuple trees, then CN C_i can be safely removed from further consideration. MPS_i is the score of a hypothetical joining tree of tuples T that contains the top tuples from every non-free tuple set in C_i. The CNs for a query are evaluated in ascending size order, so that smallest CNs, i.e. the least expensive to process and the most likely to produce high-score tuple trees, are evaluated first.

- DPBF The project has been divided into two phases: creation and serialization of the database-graph and *dpbf* implementation. To get all vertexes containing query's keywords quickly, we used hashmap data structure to store the graph and sets of vertexes associated to their words. Completed the graph, our algorithm gets the vertexes whence start to build the resulting trees. (i) For each of these we move up one level, removing similar trees with higher costs and then (ii) all trees with same root node and different sets of keywords are merged. Points (i) and (ii) are repeated until at least one result is found. We had to implement some Java data structure to do that correctly and efficiently.

- STAR has been implemented in a schema-agnostic way so the first step of the algorithm loads all DB metadata needed to build the tree. Given an i-keywords query we look for the starting tuples in a multithreaded manner. All those tuples are stored in an ad-hoc graph data-structure based on Java TreeSet class. Considering these tuples as expanding trees we try to connect them following both ingoing and outgoing foreign keys, dynamically loading referenced tuples from DB, until we get a tree containing all the query's keywords. Given that our test DBs were not weighted we decided to set

$w(e) = 1 \; \forall e \in E$. At this point we worked on this tree to optimize its weight reshaping its structure: we iteratively remove the highest weight path replacing it with a lower weight one.

4. LESSONS LEARNED

For each examined system, we briefly summarize the difficulties and lessons learned in reproducing it.

- BANKS-I During the implementation of the algorithm, we encountered some difficulties due to missing information in the descriptions of crucial parts of the algorithm. The chosen technique to run efficiently multiple instances of Dijkstra's single source shortest path algorithm was not described in the original paper, even though the choice of one implementation over another influenced considerably the memory usage and the execution time. We also encountered some difficulties due to the lack of precision in the explanation of how to evaluate the result trees. In fact, the way to handle some particular cases such as nodes containing all the keywords in the input query or trees composed by only two nodes, were not described. It became noticeable that the major weakness of the algorithm was the memory occupation that became more evident when testing it on greater datasets.

- BANKS-II We found the paper sufficiently clear in defining the Bidirectional Search algorithm and the search approach; however, the implementation of some procedures has not been trivial. As an example, the paper does not state a proper algorithm to update the reached-ancestors' distances and activations when adding a new node to new the better path (ATTACH and ACTIVATE function). Therefore, we opted for a recursive solution for these procedures. To increase the performances of the searches, we opted for two heuristic solutions: we set a lower maximum distance of the answer trees than the one suggested by the paper; we set a minimum threshold value for the activation of a node, to avoid wasteful spreading towards the graph.

- DISCOVER-I The paper provided was heavily focused on a theoretical perspective therefore including a lot of formal definitions of problems and structures, but lacking of practical directives. Certain parts of the algorithm, like the Master Index, were almost treated like a blackbox. Specifically Oracle8i interMedia Text were used but we had to implement our own Master Index using full text search provided by PostgreSQL. We made our own assumptions to implement it, for example by searching and indexing text fields only and removing English stop words. This probably lead to different execution times, making time performances comparison with the ones in [4] less significant. This is reasonable because of the larger impact that the indexes have on the queries computation compared to the advantages brought by the introduction of the Plan Generator and its execution. Moreover the Plan Generator was challenging to implement, because Candidate Networks may contain joins of tuples from the same relation through an entry of another one. But this operation is refused from the DBMS because the resulting table would have repeated column names, forcing us to change them to be unique. Eventually there were no suggestion about how to evaluate the final queries: there is the need of an order in considering the tables, with join operations on the correct columns. For this reason we decided to evaluate every Candidate Network with a BFS algorithm, that guaranties correctness.

- DISCOVER-II We observed that in a dense dataset as IMDB the execution time using OR semantic grows exponentially then queries don't give results in a useful time with a number of keywords greater than 5. If we used an AND semantic, since it keeps more CNs, a post-processing on the stream of results needed and so worsens the performance. Furthermore, not clear if the authors keep the duplicate CNs queries or if they remove before (not specified if Set or Bag semantic) and we found biased queries with the inclusion of relation names in them, e.g. char_name. We chose to not implement the AND semantic because we consider such method highly inefficient. For better performance we decided to use hashMap in the score function that was filled when the lists were created.

- DPBF During the algorithm implementation we have faced three different problems: making the graph creation and the algorithm implementation efficient and using Java and its data-structures. We tried to make the graph creation as fast as possible, assuming that it is created only once. So we serialized the graph into a file. The main cause of problems was the lack of details in the paper. In particular only some general cases were described, ignoring specific ones. The last class of problems deals with Java. DPBF main data structure is a priority queue, based on trees' weights. Unfortunately Java PriorityQueue hasn't a useful interface compared to our needs; furthermore it gave us a lot of problems in sorting weights.

- STAR In our work we had to deal with some difficulties. The lack of implementation details in [11] gave us a wide range of possibilities especially in the tree building phase. The most interesting and challenging issue concerned the choice of the keyword-containing tuples: if some of the DB's columns contain repeated strings searching a keyword results with high probability in many matching tuples, which causes longer execution times and lower accuracy for some queries. To avoid these issues we tried to add some heuristics sorting the expanding trees by number of query's keywords contained, i.e. trees with more keywords are expanded first. Furthermore, to increase DB throughput, we used multiple threads and connections to the DB during the queries execution. Java standard library does not have a graph data structure we first tried JGraphT library, but, after some benchmarks, we decided to write our own graph class based on Java TreeSet.

5. CONCLUSIONS AND FUTURE WORK

The paper briefly summarized an ongoing project for releasing an open source Java implementation of the most common algorithms for keyword-based search over relational data. The project has been initiated as a Master students project where teams of 3-4 students implemented each algorithm and noted the difficulties and issues in trying to reproduce them. All the developed implementations are available as open source at: https://bitbucket.org/ks-bd-2015-2016/ks-unipd.

We are currently working on testing all these different implementations on the same hardware in order to compare their efficiency. We are also processing the datasets prepared by [4] in order to make them closer to the standard practices adopted in *Information Retrieval (IR)* evaluation [12] and then we will systematically investigate the effectiveness of the implemented algorithms.

Finally, we will work on harmonizing the different implementations, factoring out commonly used data structures and shared functionalities, in order to go towards a unified open source framework for keyword-based search over relational data.

6. REFERENCES

[1] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Rè, D. Suciu, M. Stonebraker, T. Walter, and J. Widom. The Beckman Report on Database Research. *ACM SIGMOD Record*, 43(3):61–70, September 2014.

[2] S. Bergamaschi, N. Ferro, F. Guerra, and G. Silvello. Keyword-based Search over Databases: A Roadmap for a Reference Architecture Paired with an Evaluation Framework. *LNCS Transactions on Computational Collective Intelligence (TCCI)*, 9630:1–20, 2016.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In R. Agrawal, K. Dittrich, and A. H. H. Ngu, editors, *Proc. 18th International Conference on Data Engineering (ICDE 2002)*, pages 431–440. IEEE Computer Society, Los Alamitos, CA, USA, 2002.

[4] J. Coffman and A. C. Weaver. An Empirical Performance Evaluation of Relational Keyword Search Techniques. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1(26):30–42, 2014.

[5] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding Top-k Min-Cost Connected Trees in Databases. In R. Chirkova, A. Dogac, M. T. Özsu, and T. Sellis, editors, *Proc. 23rd International Conference on Data Engineering (ICDE 2007)*, pages 836–845. IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[6] N. Ferro. Reproducibility Challenges in Information Retrieval Evaluation. *ACM Journal of Data and Information Quality (JDIQ)*, 8(2):8:1–8:4, January 2017.

[7] N. Ferro, N. Fuhr, K. Järvelin, N. Kando, M. Lippold, and J. Zobel. Increasing Reproducibility in IR: Findings from the Dagstuhl Seminar on "Reproducibility of Data-Oriented Experiments in e-Science". *SIGIR Forum*, 50(1):68–82, June 2016.

[8] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proc. 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 850–861. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, 2003.

[9] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In P. A. Bernstein, Y. E. Ioannidis, R. Ramakrishnan, and D. Papadias, editors, *Proc. 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 670–681. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, 2002.

[10] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-A. Larson, and B. Chin Ooi, editors, *Proc. 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 505–516. ACM Press, New York, USA, 2004.

[11] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner-Tree Approximation in Relationship Graphs. In Z. Li, P. S. Yu, Y. E. Ioannidis, D. Lee, and R. Ng, editors, *Proc. 25th International Conference on Data Engineering (ICDE 2009)*, pages 868–879. IEEE Computer Society, Los Alamitos, CA, USA, 2009.

[12] M. Sanderson. Test Collection Based Evaluation of Information Retrieval Systems. *Foundations and Trends in Information Retrieval (FnTIR)*, 4(4):247–375, 2010.

[13] H. Wang and C. C. Aggarwal. A Survey of Algorithms for Keyword Search on Graph Data. In C. C. Aggarwal and H. Wang, editors, *Managing and Mining Graph Data*, pages 249–273. Springer-Verlag, New York, USA, 2010.

[14] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Morgan & Claypool Publishers, USA, 2010.