Keyword Search on External Memory Data Graphs

Bhavana Bharat Dalvi[*] Meghana Kshirsagar[†] S. Sudarshan
Computer Science and Engg. Dept., I.I.T. Bombay
bhavana.dalvi@gmail.com, meghanak@yahoo-inc.com, sudarsha@cse.iitb.ac.in

ABSTRACT

Keyword search on graph structured data has attracted a lot of attention in recent years. Graphs are a natural "lowest common denominator" representation which can combine relational, XML and HTML data. Responses to keyword queries are usually modeled as trees that connect nodes matching the keywords.

In this paper we address the problem of keyword search on graphs that may be significantly larger than memory. We propose a graph representation technique that combines a condensed version of the graph (the "supernode graph") which is always memory resident, along with whatever parts of the detailed graph are in a cache, to form a multi-granular graph representation. We propose two alternative approaches which extend existing search algorithms to exploit multi-granular graphs; both approaches attempt to minimize IO by directing search towards areas of the graph that are likely to give good results. We compare our algorithms with a virtual memory approach on several real data sets. Our experimental results show significant benefits in terms of reduction in IO due to our algorithms.

1. INTRODUCTION

Keyword search on graph structured data has attracted a lot of attention in recent years. Graphs are a natural "lowest common denominator" representation which can combine relational, XML and HTML data. Graph representations are particularly natural when integrating data from multiple sources with different schemas, and when integrating information extracted from unstructured data. Personal information networks (e.g. [7]) which combine information from different sources available to a user, such as email, documents, organizational data and social networks, can also be naturally represented as graphs. Keyword querying is particularly important in such scenarios.

Responses to keyword queries are usually modeled as trees that connect nodes matching the keywords. Each answer tree has an associated score based on node and edge weights, and the top K answers have to be retrieved.

[*]Current affiliation: Google Inc., Bangalore

[†]Current affiliation: Yahoo! Labs, Bangalore

There has been a great deal of work on keyword querying of structured and semi-structured data in recent years. Several of these approaches are based on in-memory graph search. This category includes the backward expanding search [3], bidirectional search [16], dynamic programming technique DPBF [9], and BLINKS [13]. All these algorithms assume that the graph is in-memory.

As discussed in [3, 16], since the graph representation does not have to store actual textual data, it is fairly compact and graphs with millions of nodes, corresponding to hundreds of megabytes of data, can be stored in tens of megabytes of main memory. If we have a dedicated server for search, even larger graphs such as the English Wikipedia (which contained over 1.4 million nodes and 34 million links (edges) as of October 2006) can be handled. However, a graph of the size of the Web graph, with billions of nodes, will not fit in memory even on large servers.

More relevantly, integrated search of a users personal information, coupled with organizational and Web information is widely viewed as an important goal (e.g. [7]). For reasons of privacy, such search has to be performed on a users machine, where many applications contend for available memory; it may not be feasible to dedicate hundreds of megabytes of memory (which will be needed with multiple data sets) for occassionally used search.

A naive use of in-memory algorithms on very large graphs mapped to virtual memory would result in a very significant IO cost, as can be seen from the experimental results in Section 7. Related work is described in more detail in Section 2. None of the earlier work has addressed the case of keyword search on very large graphs.

In this paper we address the problem of keyword search on graphs that may be significantly larger than memory. Our contributions are as follows:

1. We propose (in Section 4) a multi-granular graph representation technique, which combines a condensed version of the graph (the "supernode graph") which is always memory resident, along with whatever parts of the detailed graph are in a cache. The supernode graph is formed by clustering nodes in the full graph into supernodes, with superedges created between supernodes that contain connected nodes. The multi-granular graph represents all information about the part of the full graph that is currently available in memory.

The idea of multi-level hierarchical graphs formed by clustering nodes or edges of a graph has been used in many contexts, as outlined in Section 2, and hierarchical multi-granular graph views have been used for visualization (e.g. [6]). However, as far as we are aware, earlier work on search (including keyword and shortest path search search) does not exploit the cache presence of parts of the detailed graph.

2. We propose two alternative approaches which extend existing search algorithms to exploit multi-granular graphs; both approaches attempt to minimize IO by directing search towards areas of the graph that are likely to give good results.

   The first approach, the iterative approach (Section 5), performs search on the multi-granular graph. The answers generated could contain supernodes. Such super nodes are expanded, and the search algorithm is executed again on the new graph state, till the top-K answers are found. Any technique for keyword search on graphs, such as backward expanding, bidirectional or DPBF, that does not depend on precomputed indices can be used within an iteration.

   The second approach, described in Section 6, is an incremental approach, which expands supernodes as before, but instead of restarting search from scratch, adjusts the in-memory data structures to reflect the changed state of the multi-granular graph. This saves significantly on the CPU cost, and as it turns out, reduces IO effort also. We present an incremental version of the backward expanding search algorithm of [3], although in principle it should be possible to create incremental versions of other search algorithms such as bidirectional search [16] or the search technique of BLINKS [13].

3. We compare our algorithms and heuristics with a virtual memory approach and the "sparse" approach of [14], on several real data sets. Our experimental results (Section 7) show significant benefits in terms of reduction in IO due to our algorithms, and our heuristics allow very efficient evaluation while giving very good recall.

2. RELATED WORK

Work on keyword querying can be broadly classified into two categories based on how they use schema information:

1. *Schema-based approaches:* In these approaches, a schema-graph[1] of the database is used, along with the query keywords and text-indices to first generate "candidate" or probable answer trees. SQL queries corresponding to each candidate tree are computed and each query is executed against the database to get results. Only some of the original candidate answer trees may produce results finally. DBXplorer [1], DISCOVER [15], [14], [20] and [21] present algorithms based on this model.

   Schema-based approaches are only applicable to querying on relational data. Moreover, none of the algorithms listed above provides an effective way of generating top-K results in the presence of ranking functions based on data-level node and edge weights.

2. *Schema-free approaches:* These approaches are applicable to arbitrary graph data, not just to relational data. Graph data can include node and edge weights, and answers are ranked on these weights. The goal of these algorithms is to generate top-k answers in rank order. Algorithms in this category include RIU [19], backward expanding search and bidirectional search [3, 16], the dynamic programming algorithm DPBF [9], and BLINKS [13]. All the above algorithms assume either implicitly or explicitly that the graph is in-memory, and would incur high random-IO overheads if the graph (as well as the associated bi-level index in the case of BLINKS) does not fit in memory.

[1]A graph with relations/tables as nodes, and edges between two nodes if there exists a foreign-key to primary-key relationship between the corresponding tables.

A graph search technique based on merging of posting lists coupled with graph connectivity information is outlined in the description of SphereSearch [10], but it is not clear if their technique works on graphs larger than memory, since they require connectivity information for arbitrary pairs of nodes, which would require extremely large amounts of space. Object Rank [2] also precomputes posting lists for each keyword, and merges them at run time, but suffers from an impractically high blow up of index space. EKSO [25] precomputes reachable tuples from each "root" tuple based on schema information, and builds virtual documents that are then indexed. This approach not only requires schema information, but also results in high index overheads.

There has been much work on keyword search on XML data, which is based on a tree model of data, e.g., [11]. The tree model of data allows efficient querying based on hierarchical posting lists, but these techniques do not generalize to the graph model.

Hierarchically clustered graph representations have been used to deal with graphs that are larger than main memory in a variety of applications, such as for relaxation algorithms on very large graphs (e.g. Leiserson et al. [18]), and visualization of external memory graphs (e.g., Buchsbaum and Westbrook [6]), and for computation on Web graphs (e.g. Raghavan and Garcia-Molina [23]).

Following [23], we can define a two-level graph representation, where graph nodes are clustered into supernodes, and superedges are created between supernodes. The supergraph, consisting of supernodes (without the corresponding set of nodes) and the superedges, is used in [23] for PageRank computation.

Nodine et al. [22] present schemes for storing graphs in disk blocks, which allow a node to be replicated in more than one disk block. Although efficient in terms of worst case IO, their schemes have impractically high space overheads. There has been a fair amount of work on external memory graph travel, e.g. Buchsbaum et al. [5], but this body of work concentrates on a complete traversal of a graph, not on searching for connections on a small part of the graph, which we address.

There has also been a good deal of work on shortest-path computation on disk-based graphs. Several techniques in this area, e.g. Shekhar et al. [24], and Chang and Zhang [8], are based on partitioning graphs, and maintaining information about boundary nodes of the partitions. Although these algorithms are efficient for planar graphs,[2] where clusters with a very small number of boundary nodes can be constructed relatively easily, they can be very inefficient on non-planar graphs, where the number of boundary nodes can be very high.

3. BACKGROUND

In this section, we first briefly describe the graph model for data, outline search algorithms proposed earlier for in-memory data, and outline a two-level graph representation and a two-phase search algorithm proposed earlier.

3.1 Graph Model

We use the directed graph model for data, and the rooted tree model for answers, following [3], which are summarized below. (Undirected graph models are a special case of directed graph models.)

- *Nodes*: Every node has an associated set of keywords. Each node in the graph has an associated node weight, or prestige, which influences the rank of answers containing the node.
- *Edges*: Edges are directed and weighted. Higher edge weights correspond to a weaker degree of connection. The directed model has been shown to help avoid short-cut answers through hub nodes which are unlikely to be meaningful semantically [3]. An approach of defining edge scores based on in/out degrees of nodes is presented in [3]. The search techniques we consider are not affected by how the edge scores are defined.
- *Keyword Query*: A keyword query consists of a set of terms k_i, $i = 1 \dots n$.
- *Answer Tree*: An answer is a minimal rooted directed tree, such that every keyword is contained in some node of the tree.

  The overall answer score is a function of the node score and the edge score of the answer tree. The search algorithms are not affected by the exact combination function, except that the combination function is assumed to be monotonic; an additive model or a multiplicative model can be used as described in [3]. The node score is determined by the sum of the leaf/root node weights. Several models of edge score have been proposed. The model used in [3] defined the edge score of an answer tree as the inverse of the sum of the weights of all edges in the tree.

  The answer model in [16] treats an answer tree as a set of paths, with one path per keyword, where each path is from the root to a node that contains the keyword; the edge score of an answer is defined as the sum of the path lengths. [13] also uses this model and points out that (a) this model allows queries to be answered in polynomial time, whereas the Steiner tree model of [3] is NP hard, and (b) this model also avoids the generation of a large number of similar answers with the same root. We use the set-of-paths model in this paper.

[2]The most common application for shortest-path computation is road networks, which are planar.

3.2 Keyword Search

Given a set of query keywords, generating the results consists of two main steps. The first step involves looking up an inverted keyword index to get the node-ids of nodes (corresponding to tuples containing one/more of the keywords). These are called the "*keyword nodes*".

In the second step, a *graph search* algorithm is run to find out trees connecting the keyword nodes found above. The algorithm finds rooted answer trees, which should be generated in ranked order. Once an answer tree is generated, it is displayed to the user.

In this section we describe the *Single-Iterator* Backward Expanding Search algorithm from [16], which is a variant of the (multi-iterator) Backward Expanding Search algorithm from [3]. In the rest of this paper we shall refer to this algorithm as *Backward Expanding Search* or *BES*.

BES takes as input, the set of keywords and the graph and outputs top-k answer trees containing those keywords. For each keyword term k_i, it first finds the set of nodes S_i that are relevant to (contain) keyword k_i, by using a disk resident keyword-index built on the indexable columns of the database. BES concurrently runs n copies of Dijkstra's single source shortest path algorithm; i.e., one instance per keyword. Each instance provides an iterator interface to incrementally retrieve the next nearest node. We call each instance of Dijkstra's algorithm as a *Shortest Path Iterator* or *SPI*. The source node for SPI i is (conceptually) keyword k_i, whose neighbors are the nodes containing k_i (referred to as keyword nodes).

The iterator traverses the graph edges in reverse direction from the keyword nodes. The idea is to find a common vertex from which a forward path exists to at least one node in each set S_i. Such paths will define a rooted directed tree with the common vertex as the root and the corresponding keyword nodes as the leaves. The tree thus formed is an answer tree. As each iterator generates more nodes, more answer trees are found. Answers are generated roughly in decreasing edge score order, although not exactly so.

None of the search algorithms in the literature *generate* answers in (decreasing) order of score, although they *output* answers in (decreasing) score order. Algorithms proposed earlier (including BES) as well as those proposed here generate answers roughly in (decreasing) edge score order, [3] and these answers must be temporarily stored in a result heap. However, these algorithms also provide, at any stage, a bound L_E such that no answer of higher edge score can be generated in future; and all answers of higher edge score have been generated already. A bound on the overall answer score L can be computed using the edge score bound and the maximum possible node score for the given set of keywords ([16, 13]). An answer tree can be output if its score is greater than the bound L. The bound L decreases as the algorithm progresses, allowing the top k answers to be output in decreasing order of their score.

Virtual Memory Search: We can run BES (or any other search algorithm) on an external memory graph representation which clusters nodes into disk pages, fetching nodes from disk as required. The set of nodes in memory would form a cache of the disk resident graph, akin to virtual memory. However, performance is significantly impacted, since

[3]DPBF [9] generates answers in decreasing edge score order, although not in decreasing order of overall score.

keyword search algorithms designed for in-memory search access a lot of nodes, and such node accesses lead to a lot of expensive random IO when data is disk resident. (This intuition is supported by our performance study in Section 7.)

3.3 2-Stage Graph Search

In earlier unpublished work Gupta and Bijay [12, 4] considered the problem of keyword search exploiting the 2-level graph structure, and presented a 2-phase algorithm for keyword search on 2-level graph representation.

3.3.1 2-Level Graph Construction

The 2-level graph representation is defined as follows [23]:

- **SuperNode**: The graph is partitioned into components by a clustering algorithm, and each cluster is represented by a node called the *super node* in the top-level graph. Each supernode thus contains a subset of the vertex-set V; the contained nodes are called *innernodes.*
- **SuperEdge**: The edges between the supernodes called *superedges* are constructed as follows: if there is at least one edge from an innernode of supernode $s1$ to an innernode of supernode $s2$, then there exists a superedge from $s1$ to $s2$.

During the supernode graph construction, the parameters for the clustering are chosen such that the supernode graph fits into the available amount of main memory. Each supernode has a fixed number of innernodes and is stored on disk.

In the context of [23] nodes and edges are unweighted. In the context of keyword search, edges are weighted, and the edge-weight of a *super edge* is defined as *min*{ *edge-weight* } computed over all the edges between the innernodes comprising the two supernodes. Using min helps in getting an upper bound on the score of any real answer that is a refinement of a given answer that includes a superedge, although Gupta and Bijay also consider using average in place of min.

In some contexts, the graph has a natural clustering, e.g. based on URL prefixes for web graphs (e.g. [23]) or network hierarchies (e.g. [6]). Several algorithms have been proposed for hierarchical graph clustering on arbitrary graphs, e.g. METIS [17].

Gupta and Bijay [12, 4] tried several techniques for clustering, and found a technique based on edge-weight prioritized breadth-first-search (*EBFS*) to work the best. In this technique, an unassigned node is chosen and a BFS is started from it. During BFS, nodes are explored in the order of their edge-weights from the parent node. The BFS is stopped once the number of explored nodes equals the predefined maximum supernode size. All these explored nodes are put in a new cluster and marked as assigned.

BLINKS [13] uses clustering on the data graph to create clusters, which it uses to reduce the size of an index structure called the forward index.[4] [13] explores two techniques for clustering the data graph, one based on breadth-first search (BFS) and one based on the METIS clustering technique [17], and their performance study shows that BFS works well.[5] We note however that clustering in BLINKS is done only to restrict the index size. BLINKS performs random accesses on both the graph and on the bi-level index, and thus requires the graph as well as the bi-level index to be memory resident; BLINKS is therefore inapplicable to the case where the graph is larger than memory.

The issue of which clustering technique works best is beyond the scope of this paper, but based on the above mentioned work, we chose to use the EBFS technique for clustering.

3.3.2 2-Phase Search Algorithm

Gupta and Bijay also describe a 2-stage search algorithm, which works as follows:

1. **[Phase-1]** Search only on the complete top-level supernode-graph and generate supernode results (results containing only supernodes). To generate top-k final results, a larger number N of supernode results are generated in phase-1.
2. **[Phase-2]** Expand all supernodes from each of the phase-1 results and obtain the *Expanded Graph* G_E. Invoke the search algorithm only on G_E, producing innernode results (results containing only innernodes).

Gupta and Bijay [4, 12] discuss several limitations of 2-stage search. Minimal answers at the expanded level may correspond to non-minimal answers at the supernode level. For example, nodes $n1$ and $n2$ containing keywords $k1$ and $k2$ may both be in one supernode, but an intermediate node connecting $n1$ and $n2$ may be in a separate supernode. Such answers will never be generated if only minimal answers are generated in phase-1. At the same time, it is not clear what non-minimal results to generate in phase-1. Further, the number N of supernode results to expand in phase-2 is decided arbitrarily, and there is no way to determine what N to use to generate the top-k answers. As a result there are no guarantees that the top-k results are generated in phase-2, and recall (fraction of relevant results found) in their implementation was quite poor.

4. MULTI-GRANULAR GRAPH REPRESENTATION

The 2-phase search algorithm requires the top level of the 2-level representation to be memory resident, and fetches parts of the lower level, corresponding to supernodes, into memory as required. Since the lower level in its entirety is larger than memory, parts of the lower level are cached in a fixed size buffer. Given the size of memories today, a substantial amount of memory may be available for caching, and if related queries have been executed earlier, relevant parts of the lower-level graph may already be in-memory when a query is executed. The 2-phase search algorithm is unable to exploit this fact.

We propose a *multi-granular graph* structure to exploit information present in lower-level nodes that are cache-resident at the time a query is executed. The multi-granular or *MG*

[4]The forward index is used to store precomputed node-to-keyword distances/paths, which can speed up search. The version of the forward index with clustering is called a "bi-level" index.

[5]BLINKS generates a node partition using BFS, and then modifies it to get an edge partition, where a node may occur in more than one cluster. Although the clustering is used for a different purpose than ours, and node partitioning is dual to edge partitioning, the goal of keeping related nodes in the same cluster is the same.

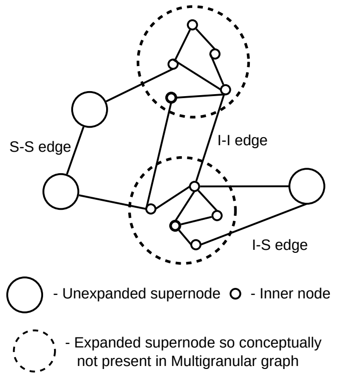
S-S edge
I-I edge
I-S edge
- Unexpanded supernode
- Inner node
- Expanded supernode so conceptually not present in Multigranular graph

Figure 1: Multi-granular Graph

graph is a hybrid graph that has both supernodes and innernodes at any instant. A supernode is present either in *expanded* form, i.e., all its innernodes along with their adjacency lists are present in the cache, or in *unexpanded* form, i.e., its innernodes are not in the cache. An example multi-granular graph is shown in Figure 1.

Since supernodes and innernodes coexist in the multi-granular graph, several types of edges can be present. Of these, the edges between supernodes and between innernodes need to be stored, the other edges can be inferred. Since graphs are weighted, edge weights have to be assigned to each type of edge, based on the edge weights in the underlying graphs.

- *supernode* $\rightarrow$ *supernode* ($S \rightarrow S$): edge-weight of $S1 \rightarrow S2 = min\{$ *edge-weight* $n1 \rightarrow n2 \mid n1 \in S1$ and $n2 \in S2\}$

  Using min provides an upper bound on the score of any pure answer which is a "refinement" of an answer that includes the superedge (the term "refinement" is defined formally shortly).

- *supernode* $\rightarrow$ *inner-node* ($S \rightarrow I$): Let supernode be S, inner-node be i, and supernode to which i belongs be I such that $S \neq I$.
  edge-weight $S \rightarrow i = min\{$ *edge-weight* $s \rightarrow i \mid s \in S\}$

  These edges need not necessarily be explicitly represented. During the graph traversal, if $S1$ is an unexpanded supernode, and we find a supernode $S2$ in the adjacency list of supernode $S1$, and $S2$ is expanded, we have found one or more $S \rightarrow I$ edges. We can enumerate such edges by locating all innernodes
  $\{i \in S2 \mid$ the adjacency list of i contains some innernode in $S1$ $\}$

- *inner-node* $\rightarrow$ *supernode* ($I \rightarrow S$):
  These edges arise when the search goes from an expanded supernode (to which the inner-node belongs) to an unexpanded supernode. The edge weight is defined in an analogous fashion to the previous case.

- *inner-node* $\rightarrow$ *inner-node* ($I \rightarrow I$):
  The edge weight is the same as in the original graph.

When we execute search on the multi-granular graph, the answers generated may contain supernodes; we call such an answer a *supernode answer*. If an answer does not contain any supernodes, we call it a *pure answer*. Only pure answers are actually returned to the user. A pure answer a_p is said to be a *refinement* of a supernode answer a_s if the tree a_s can be derived from the tree a_p by a sequence of steps as follows: each step replaces a subtree of a_p by a supernode S_i, provided that all nodes in the subtree belong to supernode S_i. Edges into and out of the replaced subtree are replaced by the corresponding edges of type $I \rightarrow S$, $S \rightarrow I$, or $S \rightarrow S$ to or from S_i.

```
ITERATIVE SEARCH ALGORITHM()
 1   Input: A multigranular graph $G$
 2   Output: Top-$k$ pure results
 3  while stopping criteria not satisfied
 4      /* EXPLORE phase */
 5      Run any in-memory search algorithm on $G$ to generate
 6      the top-$n$ results.
 7         /* Fewer than $n$ results may be generated above, if
 8         there are fewer than $n$ results on G, or if in-memory
 9         stops because a node-budget is exceeded. */
10      /* EXPAND phase */
11     for each result $R$ in top-$n$ results
12        if $R$ is a supernode result then
13           $SNodes =$ all supernodes from $R$
14           $SNodeSet \leftarrow SNodeSet \cup SNodes$
15      Expand all supernodes in $SNodeSet$
16         and add them to $G$
17  end while
18   Output top-$k$ pure results
```

Figure 2: Iterative Expansion Algorithm

LEMMA 4.1. *Given a multi-granular graph where superedge weights are computed using the min function, and any set of supernodes expanded in cache, and given any pure answer a_p,*

1. *there is a unique tree a_s in the multi-granular graph such that a_p is a refinement of the a_s, and*
2. *the edge score of a_s is $\geq$ the edge score of a_p.* □

5. ITERATIVE EXPANSION SEARCH

The *Iterative Expansion* algorithm is a multi-stage algorithm which is applicable to multi-granular graphs. This algorithm runs in multiple stages, unlike the 2-stage algorithm outlined earlier, expanding some supernodes in each stage. The iterative expansion algorithm is shown in Figure 2.

Each iteration of Iterative Expansion can be broken up into two phases:
(a) Explore phase: Run an in-memory search algorithm on the current state of the multi-granular graph (the multi-granular graph is entirely in memory), and
(b) Expand phase: Expand the supernodes found in top-n results of the (a) and add them to input graph to produce an expanded multi-granular graph.

The graph produced at the end of Expand phase of iteration i acts as the graph for iteration $i + 1$. Any in-memory graph search algorithm can be used in step (a). The in-memory search algorithm is used unchanged on the multi-granular graph, and treats all nodes (whether supernode or innernode) in the same way. Iterative Search makes use of already expanded supernodes from previous iterations in generating better results in successive iterations. The problem of non-minimal answers faced by 2-phase search, outlined earlier, is avoided since the search is rerun with supernodes expanded. (An example of the execution of Iterative Search is provided in Appendix A.)

The algorithm stops at the iteration where all top-k results are pure. So the stopping criterion in the Iterative Algorithm (Figure 2) will just check whether top-k results formed in current iteration are all pure. Other termination heuristics can be used to reduce the time taken for query execution, at the potential cost of missed results.

As shown in Figure 2, if the top-k results are to be displayed as the final output, then in each iteration the algorithm generates the top-n results, for some $n \geq k$. Our current implementation simply sets $n = k$ for all iterations; experiments to study the effect of a value of n that is initially larger, but decreases in later iterations, are planned as part of future work.

We also observed that for some queries, generating n results (for $n > k$) took a long time since the lower ranked answers were very large trees, requiring a lot of exploration of the graph. To avoid this problem, we added a *node-budget* heuristic, which stops search in a particular iteration if either
(1) If number of nodes touched to generate the $(i+1)^{th}$ answer is greater than twice the number of nodes touched to generate i^{th} answer and $i > k$, or
(2) If the total number of nodes touched in an iteration exceeds a pre-defined maximum limit.
This heuristic trades off a potentially larger number of iterations for a reduced cost within an iteration.

An implicit assumption made in the above algorithm is that the part of graph relevant to the query fits in cache. But this assumption fails in some cases, for example if the algorithm explores a large number of nodes while generating results in an iteration, or when the query has many keywords each matching many nodes. In such a case, we have to evict some supernodes from the cache based on a cache replacement policy. Thus some parts of the multi-granular graph may shrink after an iteration.

Such shrinkage can unfortunately cause a problem of cycles in evaluation. For example, suppose a supernode S is found in a supernode result R which is among the top-n results of iteration i, then it gets expanded and a refined version of R say R' gets generated in iteration j where $j > i$. Now if in a later iteration, supernode S gets evicted from cache, then in some iteration $k > j$, there is possibility that result R gets generated again in the top-n results leading to S being expanded again and so forth. Evaluation may potentially not terminate unless we place some bound on the number of iterations and stop regardless of the answers generated.

To address this problem, we do not shrink the logical multi-granular graph, but instead provide a "virtual memory view" of an ever-expanding multi-granular graph. To do so, we maintain a list, *Top-n-SupernodeList*, of all supernodes found in the top-n results of all previous iterations. Any node present in *Top-n-SupernodeList* but not in cache is transparently read into cache whenever it is accessed.

THEOREM 5.1. *Iterative search (without the node-budget heuristic) correctly computes the top k results when it terminates.* □

Since at least one supernode is expanded in each iteration (except the last), the algorithm is guaranteed to terminate. The above results also hold with the node-budget heuristic, except in the case when the heuristic terminates an iteration because the number of nodes touched exceeds the maximum limit, without generating any answer. Although some answers may be missed as a result, such answers are likely to be large answers with low scores.

6. INCREMENTAL EXPANSION SEARCH

Although the Iterative Expansion algorithm tries to minimize disk-accesses by iteratively improving the results, it has the limitation that it restarts search every time and recomputes results, effectively throwing away the current state of the search algorithm each time. This can lead to significantly increased CPU time, which was borne out by our experiments. We describe below an alternative approach, which we call *incremental expansion*, which follows a different approach.

6.1 Incremental Expansion Algorithm

As for iterative search, the incremental expansion algorithm also performs keyword search on the multi-granular graph. When a supernode answer is generated, one or more supernodes in the answer are expanded. However, instead of restarting search when supernodes are expanded, incremental expansion updates the state of the search algorithm. The exact way in which the state is updated depends on the specific search algorithm. Once the state is updated search continues from where it left off earlier, on the modified graph. Thus search done earlier does not have to be redone if it did not pass through the expanded supernode.

Figure 3 shows the incremental expansion version of the Backward Expanding Search algorithm described earlier in Section 3.2. We refer to this algorithm as the Incremental Expansion Backward search, or just Incremental Expansion search for short.

The Incremental Expansion Backward search algorithm runs backward search on the initial multi-granular graph. There is one *shortest path iterator* (SPI) tree per keyword k_i, which contains all nodes "touched" by Dijkstra's algorithm, including explored nodes and fringe nodes, starting from k_i. More accurately, the SPI tree does not contain graph nodes, rather each tree-node of an SPI tree contains a pointer to a graph node. Each tree-node n also contains a pointer to the its parent tree-node p in the SPI tree; the graph node corresponding to p is the next node in the shortest path (currently known shortest path, in the case of fringe nodes) from the graph node of n to the keyword (or to be more precise, to any "origin" node containing the keyword). Although the multi-granular graph is shared, the SPI trees are independent and do not share tree-nodes.

The backward search algorithm [3, 16] expands each SPI tree using Dijkstra's algorithm. When a graph node has been reached in the SPI trees corresponding to every one of the keywords, an answer has been found. Since answers may be found out of order with respect to their scores, they are accumulated in a heap and an answer is output only when no better answer can be generated (see Section 6.2).

When an answer is output by the backward search algorithm, if it contains any supernode it is not a pure answer. In this case, one or more supernodes from the result are expanded. A supernode being expanded may appear in multiple shortest path iterator (SPI) trees (one tree exists per keyword).

To update the state of Dijkstra's algorithm to reflect the

```
INCREMENTAL-SEARCH()
1  $SPI-Tree[i]$ : Shortest Path Tree of iterator $i$ which
2      contains both explored and unexplored (fringe) nodes
3  $PQ[i]$ : Priority Queue of iterator $i$, that contains
4      nodes yet to be explored (i.e fringe nodes)
5  while number of pure results generated $< k$
6      $Result$ = BACKWARDSEARCH.GETRESULT()
7      /* getResult() returns one result */
8      if no result found then
9          exit
10     if $Result$ contains a supernode then
11         EXPANDANDUPDATEITERATORS($Result.root$)

EXPANDANDUPDATEITERATORS($root$)
1  for each shortest-path iterator $SPI[i]$
2      $toDelete$ = first supernode on the path in SPI[i]
3          from a keyword node to $root$
4      $toDeleteSNSet \leftarrow toDeleteSNSet \cup toDelete$
5
6  for each shortest-path iterator $SPI[i]$
7      $snodeHeap$ = min-heap of supernodes sorted on
8          their path-costs from the origin keyword-nodes
9          of iterator $i$
10     /* Sort supernodes in order of path-cost from origin
11     keyword node */
12     for each $snodeID \in toDeleteSNSet$
13         if $SPI$-$Tree[i]$ contains $snodeID$ then
14             $snode$ = $SPI$-$Tree[i]$.getNode ($snodeID$)
15             $snodeHeap$.add ($snode, snode.pathCost$)
16             /* Delete each supernode from current iterator */
17     while not $snodeHeap$.isEmpty()
18         $snode$ = $snodeHeap$.removeMin()
19         DELETESUPERNODEFROMITERATOR($snode, i$)
```

Figure 3: Incremental Expansion Search

```
DELETESUPERNODEFROMITERATOR($S, i$)
1  /* Delete $S$ from iterator $i$ */
2  $PQ[i]$.delete($S$)
3  $SPI$-$Tree[i]$.delete($S$)
4  deletedSet = $\emptyset$
5  DELETESUBTREE($S, i, deletedSet$)
6  Read $S$ from disk
7  /* Attach innernodes of $S$ to $SPI$-$Tree$ */
8  for each innernode $in \in S$
9      if $in$ is a keyword node then
10         $PQ[i]$.add($in$) with path-cost $= 0$
11         $SPI$-$Tree[i]$.add($in$)
12     else
13         FINDMINANDATTACH($in, i$)
14 for each node $\in$ deletedSet
15     FINDMINANDATTACH($node, i$)

DELETESUBTREE($Node, i, deletedSet$)
1  add Node to deletedSet
2  for each child-node $chld$ of $Node$
3      SPI-Tree[i].delete($chld$)
4      DELETESUBTREE($chld, i, deletedSet$)

FINDMINANDATTACH($node, i$)
1  /* Find best path connecting node $SPI$-$Tree[i]$ */
2  $Out = \{n | (node \rightarrow n)$ is an edge $\&(n.$ isExplored
3      $= true) \ \& \ (n \in$ SPI-Tree[i] $)\}$
4  $minCost = min\{$ edge-weight $(node, p) +$
5      path-cost $(p) \ | \ p \in Out\}$
6  $minNode = argmin_{p \in Out}\{$ path-cost $(p) +$
7      edge-weight $(node, p)\}$
8  if $minCost = \infty$ then
9      $PQ[i]$.delete ($node$)
10     $SPI$-$Tree[i]$.delete ($node$)
11 else
12     $minNode$. addChild ($node$)
13     $PQ[i]$.add ($node$) with path-cost $= minCost$
14     $SPI$-$Tree[i]$.add ($node$)
```

Figure 4: Functions called by Incremental Search

expansion of a supernode, tree-nodes representing the supernode are removed from all shortest path iterators where the supernode is present. If the supernode had been explored earlier in Dijkstra's algorithm, its removal may cause a change in the shortest path from some nodes to the corresponding keyword. Unlike in the normal Dijkstra's algorithm, the path cost of a fringe node may not just decrease, but may actually increase when the multi-granular graph changes. Similarly, the path cost of an explored node may also increase when the multi-granular graph is modified.

More specifically, when a supernode is expanded, it is deleted from the multi-granular graph and replaced by the corresponding inner nodes. If the supernode was explored earlier, it is present in one or more SPI trees. Deleting the node clearly changes the shortest path for all nodes in the SPI subtrees rooted in the supernode. Intuitively, if the path of any node in a shortest-path iterator contains a supernode, its path-cost (to the keyword) is a lower bound on its true path-cost (assuming superedge costs are defined by the min function). Expanding the supernode should improve (increase) this lower bound.

The following updates need to be done after a supernode S is expanded. For each SPI tree containing a tree-node s_i corresponding to S, the nodes in the subtree rooted at s_i are deleted from the SPI tree; these include all explored and fringe nodes whose best path goes through s_i. We then have to recompute their best path, which is done as explained shortly.

Further, for all inner nodes i of S, if there exists an outgoing edge to i, from any explored node in the SPI tree (after the subtree rooted at S is deleted), we attach (a tree-node representing) i to that node in tree. When there exist multiple such nodes with edges to i, we choose to attach it to that node p which gives the best path-cost (edge-weight($p \rightarrow i$) + path-cost(p)) to i. We further make i a fringe node, and place it in the priority queue (PQ) containing all fringe nodes.

Note that an inner node i of S may not get attached to any node above, but may be reached at a subsequent stage of the search. Conversely, an inner node may get attached to some tree-node as described above, but a better path to the keyword may be found subsequently, and it would get reattached appropriately to the SPI tree. (An example of the execution of Incremental Search is provided in Appendix B.)

Updating of costs of nodes in the deleted subtree rooted at S is done similar to the case of inner nodes: if they are connected to any explored node in the SPI tree (after deletion of the subtree), they are linked to the node that gives the best path-cost, and placed back in the priority queue PQ. Thus, a node that had been explored earlier and removed from PQ may go back to PQ after this update.

The above updates ensure that the following invariants of Dijkstra's algorithm hold in the new state of the graph: (a) the cost for every explored node is its best path cost, (b) for every fringe node (in PQ) the current cost is the best cost among paths that connect to one of the current explored nodes, and (c) every node adjacent to an explored node is either explored or in the fringe.

The above properties are an invariant for each step of Dijkstra's algorithm. Expansion of a supernode, which deletes a supernode, moves nodes in the corresponding deleted subtree back into the fringe with their cost correctly updated. Other nodes are not affected by the deletion since their (current) shortest path does not pass through the deleted node.

This leaves the question of which supernodes to expand at any point. In the Incremental Expanding search, supernodes are expanded only when a result is generated. We tried two schemes: (a) Expand only the closest supernode per keyword on the path from the keyword to the root of the result. (b) Expand all supernodes in the result. Expansion is performed separately on each SPI tree. When multiple supernodes in a given SPI tree have to be expanded, they are processed in order of their path-cost (from the origin keyword nodes).

In our experiments in Section 7, we found that option (a) (expand only closest supernode per keyword) is more efficient than expanding all supernodes in the result.

Similar to Iterative Expansion search, Incremental Expansion search also assumes that the supernodes expanded during evaluation of a query fit in the cache. If the numner of expanded supernodes exceeds cache size, nodes may be evicted, but we give a "virtual-memory" view of a multi-granular graph that does not shrink, by re-fetching previously expanded supernodes transparently, as required.

While this ensures correctness, it can lead to thrashing if evicted supernodes are frequently expanded again. We address heuristics to control thrashing later.

The Dijkstra invariants mentioned earlier can be used to show that at any state, the top k results at that state would be generated by Backward Expanding search. Now consider the state when the algorithm terminates, reporting the top k results (which are all pure when the algorithm terminates). Since the score of super-node result is an upper bound on the score of its refinement, there cannot be any other pure result with a higher score than the kth result. At any state of the multi-granular graph, at least one result is generated (assuming there is a result not generated earlier), and one or more supernodes are expanded. Since the total number of supernodes is finite, the algorithm will terminate. As a result, we have the following theorem.

THEOREM 6.1. *The incremental search algorithm correctly generates the top-K results.* $\Box$

At any state of the multi-granular graph, backward expanding search can take at most $m \times (n \log n + E)$, where m is the number of keywords, n the number of nodes and E the number of edges. When an answer is found, and a node expanded, at most $m \times (n \log n + E)$ time is required to delete nodes from the iterators (in practice much less time is required if the expanded node is not near the root of the shortest-path tree). Each node in the graph can be expanded at most once, leading to an overall polynomial time bound. The number of IO operations is at most equal to cache size in the absence of thrashing. Even with thrashing, we can show a polynomial bound on the number of IO operations.

However, these are worst case bounds, and in practice in most cases the number of nodes expanded and IO operations performed is much smaller than the graph size. The only exception is when thrashing occurs, which can be prevented heuristically.

6.2 Heuristics

We now introduce several heuristics that can improve the performance of the Incremental Expansion algorithm.

Thrashing can be avoided by the following heuristic, which we call *stop-expansion-on-full-cache*.

1. Nodes expanded during search cannot be evicted from the cache
2. Once the cache is full, and there are no nodes that can be evicted, node expansion is stopped. The top-k pure results from the current state of the graph are output (these can be computed by starting a fresh search which ignores supernodes).

With thrashing prevention, the number of IO operations is bounded by the cache size, at a possible loss of recall. However, our performance results show that recall is quite good with thrashing prevention.

We also observed that when results are generated in rank order, most of the early results tend to be supernode results. This happens due to two reasons: (a) the score of supernode results ignores the fact that when a supernode is expanded, in most cases it would get replaced by a path containing intra-supernode edges, and (b) the min edge technique for computing weights of $S \rightarrow S$, $I \rightarrow S$, and $S \rightarrow I$ often underestimates the cost of the edge between the corresponding innernodes.

To give lower priority to supernode results, we use a heuristic to increase the weights of edges which connect to a supernode. We define the intra-supernode weight of a supernode as the average of all *innernode* $\rightarrow$ *innernode* edges within that supernode. Supernode to supernode ($S \rightarrow S$) edges have the intra-supernode weights of both their endpoints added to the edge weight, while $S \rightarrow I$ and $I \rightarrow S$ edges have the intra-supernode weights of S added.

The above adjusted edge weights could be used only for altering the priority for search, in a manner similar to Bidirectional search [16]. However, doing so would not allow us to output answers early, and give limited benefits. Instead, in our heuristic the adjusted edge weights not only affect prioritization, but also heuristically affect the lower bound on future answers. This approach allows answers to be output earlier, even if there is a small chance of a better answer not being generated yet. We call this heuristic the *intra-supernode-weight* heuristic. Our performance study examines the effect of this heuristic.

The minimum fringe distance across all iterators gives a lower bound on the edge-cost of future answers [16]. This bound can be improved upon as shown in [13], but a cheaper alternative is to use the heuristic of [16] which computes the minimum fringe distance on each iterator, and adds these results to get a heuristic bound. We use only edge weights, not node weights, when computing these bounds. We call this heuristic the *sum-lower-bound* heuristic. As shown in [16] this heuristic gave good results in terms of reduced time while giving good recall.

7. EXPERIMENTAL EVALUATION

We implemented the algorithms described earlier on the BANKS codebase, and compared their performance with alternatives, under different parameter settings.

Database	Size of Graph	Nodes (tuples)	Edges	Indegree Avg	Indegree Max
DBLP	99 MB	1.77 M	8.5 M	2.34	784
IMDB	94 MB	1.74 M	7.94 M	2.28	693

Database	Supernode graph	Compr. Ratio	Super- -nodes	Super- -edges	Indegree Avg	Indegree Max
DBLP	16.9 MB	5.85	17714	1.38 M	38.9	767
IMDB	33 MB	2.84	17412	2.8 M	81.1	1686

Figure 5: Datasets

7.1 Search Algorithms Compared

The algorithms implemented were Iterative Expanding search, Incremental Expanding (Backward) Search with different heuristics, the in-memory Backward Expanding search run on a virtual memory view of data (described below), and the Sparse algorithm from [14].

A naive approach to external memory search would be to run in-memory algorithms in virtual memory. To compare our algorithms with this approach, we have implemented this approach on the supernode graph infrastructure, treating each supernode as a page. We call this approach *VM-Search*. VM-Search runs Backward Expanding search on a virtual memory view of the innernode graph. If the search accesses a node that is not currently in cache, the node is transparently fetched (evicting other pages from the cache if required).

The Sparse algorithm of [14] is a schema-based approach for keyword search. Although it cannot be used on arbitrary graphs, we include it in our comparison to see how our algorithms compare with schema based algorithms when the schema is known. For this, we manually generate all relevant "candidate networks" (CNs) for a query by examining all the relevant answers, and compute the total SQL query execution time over relevant CNs (relevance was judged manually by examining results). Note that no result ranking is done in the case of Sparse. To match the cold-cache runs of our algorithms, and to get a fair comparison of the IO times, we restarted the (PostgreSQL) database-server and flushed the file-system buffers (as described in Section 7.2.3) before each SQL query. Further, indices were created on all columns used by the joins of the CNs.

7.2 Experimental Setup

We describe the data sets used, compression results and cache management, in this section.

7.2.1 Data Sets

We used two different datasets, the entire DBLP data as of 2003, and the Internet Movie database (IMDB) also as of 2003. The node and edge-weights for the graphs constructed from these datasets, were set as described in [16]. We clustered the datagraphs using the EBFS technique described in Section 3.3. Our default cluster (supernode) size was set to 100 innernodes, corresponding to an average of 7KB on DBLP and 6.8KB on IMDB. Supernode contents were stored sequentially in a single file, with an index for random access within the file to retrieve a specified supernode. We also performed some experiments with larger supernode sizes. Some statistics of these datasets and the properties of their supernode graphs are summarized in Table 5.

Although these datasets can fit into main memory on today's machines, as explained in Section 1, not all of main memory may be available for the search application. Web graphs, as well as data graphs obtained by information extraction from very large data sets may be much larger than memory. To keep our experiments manageable with known data sets, we used the above datasets, but allocate a correspondingly small amount of memory for search.

Nodes per S'node	S'node Graph	Compr. Ratio	Edges	Vertices	Indegree Avg	Indegree Max
100	16.9 MB	5.85	1378K	17.7K	40	767
200	13 MB	7.6	1106K	8.8K	62	775
400	11 MB	9	887K	4.4K	100	867
500	9.6 MB	10.3	820K	3.5K	116	780
800	8.3 MB	11.9	687K	2.2K	155	868

Figure 6: Compression ratios for DBLP for various Supernode sizes

7.2.2 Clustering Results

The data sets were clustered using the *EBFS* technique described in Section 3.3, which had been found to perform well with 2-stage search. Figure 5 shows that with 100 nodes per supernode, we get a compression ratio of 5.85 for DBLP and 2.84 for IMDB. A comparison of the performance of various clustering techniques is beyond the scope of this paper, but is an important area of future work.

The graph compression ratios obtained with the EBFS algorithm for different supernode sizes (or cluster sizes) on DBLP are shown in Figure 6 and they range from 6 to 12.

The compression ratios obtained are significant, and allow us to handle fairly large graphs with a reasonable amount of memory. Further compression may be required to handle extremely large graphs, using either better clustering techniques or by using a multi-level clustering technique; this is another area of future work.

7.2.3 Cache Management

The system used for experimentation had 3GB RAM, and a 2.4GHz Intel Core 2 processor, and ran Fedora Core 6 (linux kernel version 2.6.18). We used a total of 24MB by default for the supernode graph and cache (combined). For DBLP, the supernode graph with 100 innernodes per supernode occupied 17MB, allowing a cache size of 7MB or 1024 nodes (1/14th of the original data-graph size, or 1/17th in terms of supernodes) for our algorithms. For VM-Search, we used a cache of 3510 nodes (1/4th of the original data-graph size) by default, corresponding to 24MB, since VM-Search does not require the supernode graph. The cache uses an LRU-based page-replacement policy. The same policy is used across caches for all algorithms compared.

All results that we present for each of the algorithms were taken on a cold cache. To ensure that the expanded supernode pages read from disk for a query do not remain in the OS file buffer for subsequent queries, we force the linux kernel to drop the page cache, inode and dentry caches (after restarting the server) before executing each query. (On Linux kernels from version 2.6.16 upwards, this can be done by executing the command `echo 3 > /proc/sys/vm/drop_caches`, after executing the `sync` command to flush dirty pages back to disk.)

Blocks fetched and subsequently evicted from the cache during search may however still be resident in the file system buffers if they are refetched. This effect results in execution times being underestimated for VM-search, as well as in the case of thrashing. However, even with this bias, thrashing

Dataset	Query	Keywords	(# Keyword nodes) [total keyword supernodes]
DBLP	Q1	Christos Faloutsos Nick Roussopoulos	(81, 4, 161, 3) [158]
DBLP	Q2	continuous queries widom	(1182, 2005, 1) [2252]
DBLP	Q3	naughton dewitt query processing	(5, 8, 3236, 4986) [4830]
DBLP	Q4	vapnik support vector	(30, 4888, 1685) [4445]
DBLP	Q5	divesh jignesh jagadish timber querying XML	(1,4,4,7,595, 1450) [1258]
DBLP	Q6	sudarshan widom	(6, 1) [7]
DBLP	Q7	giora fernandez	(5,188) [172]
IMDB	Q8	steven spielberg	(1248, 19) [1136]
IMDB	Q9	brosnan bond	(8, 228) [126]
IMDB	Q10	bruce willis john	(1355, 325, 10805) [6546]
DBLP	Q11	krishnamurthy parametric query optimization	(51,585,3236, 3874) [4640]
IMDB	Q12	keanu matrix thomas	(4,430,3670) [2593]

Figure 7: List of Queries

results in bad performance, and VM-search performs badly, as results presented later in the section show, and thus the conclusion of our performance study are not significantly affected by this bias.

7.3 Experimental Results

We use the following metrics for comparison: number of *cache misses*, total time taken for the query to execute, which is split into CPU-time and IO-time, *recall* (obtained by comparison with the results produced by original in-memory BANKS algorithms and manual examination of a result), total number of nodes (including supernodes) touched and explored. The total time taken to execute a query is the time taken to output the top-k results.

We used a set of 12 queries, of which 8 were from DBLP and 4 from IMDB, which are shown in Figure 7. The figure also shows the number of nodes that match each keyword, and the total number of supernodes that contain these nodes.

7.3.1 Comparing Heuristics for Incremental Search

In this set of experiments, we compare different heuristic versions of Incremental search. We first implemented Incremental search without any of the heuristics, using the minimum of fringe distances across iterators as the bound for outputting answers. However this approach did not perform well, and gave poor results, taking unreasonably long times for many queries. We do not present results for this case.

We next studied two versions of Incremental expansion, one with and one without the intra-supernode-weight heuristic. Both versions used the sum-lower-bound heuristic, and neither version used thrashing control. Figure 8 compares the performance of these alternatives in terms of the cache misses and recall. It can be seen that the sum-lower-bound heuristic without the intra-supernode-weight heuristic, improved matters somewhat compared to using the min bound, but performance is still quite poor, with the last three queries thrashing and not finishing even after quite a long time.

Query	W/o Intra-Supernode Wts		W/ Intra-Supernode Wts	
	Cache Misses	Recall	Cache Misses	Recall
Q1	453	100	130	83
Q2	641943	100	391	100
Q3	317323	100	338	100
Q4	109	100	57	100
Q5	130	70	29	100
Q6	212	100	82	60
Q7	65	100	16	100
Q8	630741	100	534	100
Q9	385	100	88	100
Q10	-	-	1336	100
Q11	-	-	194955	100
Q12	-	-	680	100

Figure 8: Intra-Supernode Weight Heuristic

In contrast, with the intra-supernode-weight heuristic, the number of cache misses comes down drastically for most queries, except for Q11, which exhibits thrashing. In fact, remarkably, the number of cache misses is typically much lower than the number of supernodes that match the keywords in the queries. This indicates that performing keyword search on the supernode graph is very successful at avoiding fetching parts of the graph that do not contribute to answers. The recall (as a fraction of results that were manually judged as most relevant) is 100% for all but 2 queries, which had 83 % and 60 % recall. Thus the intra-supernode-weight heuristic reduces the number of cache misses drastically without significantly reducing answer quality.

The next set of experiments used the intra-supernode-heuristic, but studied the effect of using the min bound versus using the sum-lower-bound heuristic. We omit the details due to lack of space. The results showed that compared to the min-lowerbound heuristic, the sum-lowerbound heuristic typically reduces the time taken by about 10 to 30%, while recall (as we saw earlier) is very good. We use the sum-lowerbound heuristic for the remaining experiments.

The next set of experiments studied the effect of thrashing control for those queries that showed thrashing in earlier experiments (those that did not thrash would not be affected by thrashing control). Since, among queries Q1 to Q12, only Q11 showed thrashing we added four more queries, chosen to exhibit thrashing behavior, to this experiment:

DBLP	Q13	kevin statistical	(474,1696) [1541]
IMDB	Q14	zellweger jude nicole	(3,119,1085) [1074]
DBLP	Q15	yates analysis string	(31,15350,639) [8582]
IMDB	Q16	al pacino diane keaton	(1623, 12, 1015, 39) [2224]

For these experiments, we used the intra-supernode-weight heuristic along with the sum-lowerbound heuristic. Detailed results are omitted for lack of space, but it was seen that thrashing control sharply reduced the time taken for queries that exhibited thrashing, by up to an order of magnitude. The recall results were unchanged for queries that do not exhibit thrashing, but even for queries that show thrashing, recall did not go down significantly due to thrashing control. In fact, it went down for only one query, Q11, where recall went down from 100% to 75%, and 50% at different cache sizes. It was observed that queries where some of the keywords matched relatively few nodes had fewer cache misses, while those that had more keywords, each matching many nodes, had higher cache misses.

Based on the above results, in the rest of the paper by default we use the sum-lower-bound heuristic, coupled with

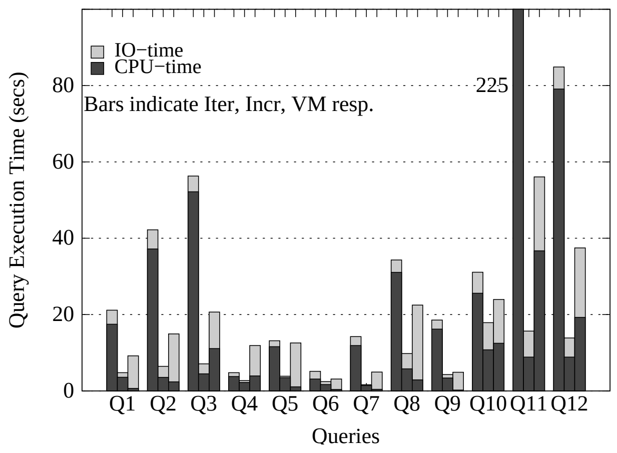

Figure 9: Execution Times the 10th Result

intra-supernode weights and thrashing control, as the defaults for Incremental expansion.

7.3.2 Comparison With Alternatives

Figure 9 shows the execution time for for answering queries using different search algorithms. For each query the figure shows three bars, for Iterative, Incremental, and VM-search respectively. For incremental, we use the sum-lower-bound heuristic, coupled with intra-supernode weights and thrashing control. In all cases, we measure the time to generate 10 answers.

For iterative, as for incremental, we used the intra-supernode-weight heuristic, along with the sum-lower-bound heuristic. We still found that performance was very poor if the number of iterations became very large, so we use the heuristic of stopping after 30 iterations. Even with this heuristic, which can lead to reduced recall, iterative performs quite poorly in terms of time taken. Recall was 100% for most queries, although Q9 had 90% recall, Q7 had 80% recall, and Q11 had 63% recall.

It can be seen that Incremental significantly outperforms Iterative, and VM-search. Note that the timing numbers for VM-search are an underestimate, since a node evicted during evaluation may be in the file system buffers, and may not require a real IO operation. VM-search has a significantly higher IO time than Incremental for most queries, but generally has a lower CPU time. The comparison of Incremental with VM-search in terms of cache misses, shown in Figure 10 shows an even greater improvement for Incremental over VM-search, by a factor of 10 to 100 in most cases.

Figure 10 compares the cache misses for Incremental without thrashing control versus VM-search with 3 different amounts of memory allocated to the cache. The number of supernodes in the cache is shown for each cache size; for Incremental, 17 MB of space was used by the supernode graph, which was instead used for the cache in VM-search, leading to a larger cache size for the same total amount of memory. Note that the graph uses logscale on the y axis. It can be seen that Incremental had far fewer misses than VM-search even if we compare Incremental with 24 MB cache (corresponding to 1024 supernodes in the cache) with VM-search with a 31 MB cache. The cases with high cache misses indicate thrashing. Although we do not show cache miss results for Iterative, we found them to be comparable to the results

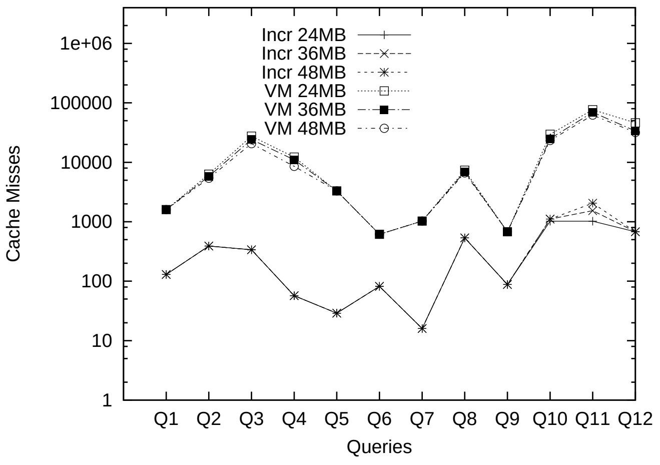

Figure 10: Cache Misses With Varying Cache Size

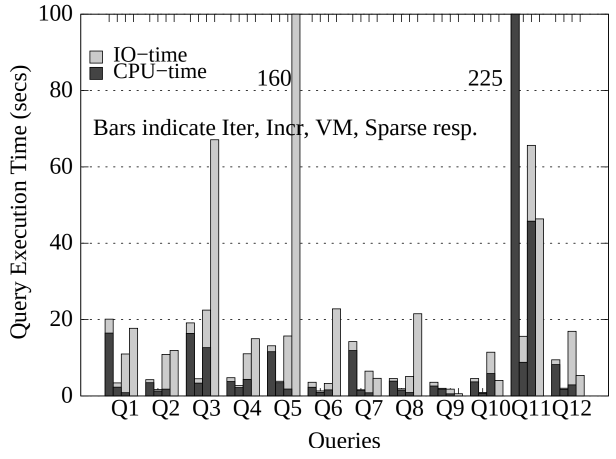

Figure 11: Execution Times for the Last Relevant Result

for Incremental. We also tried a variant of VM-search which stops node expansion once the cache is full, but continues search; this variant results in very poor recall, down to 0% for many queries (see Appendix C.1).

To verify that the above results hold for more queries, we ran 8 other queries, with 3 or 4 keywords each, on DBLP. We found that the performance (runtime and recall) was similar to that for the earlier queries. Incremental always performed better than VM, and provided run time benefits of 50 to 70% on most of the queries, and cache misses were typically less by a factor of 30 to 100. We also ran VM-Search on queries Q13 to Q16, which exhibited thrashing behaviour on Incremental, and found it performed poorly, and failed to report results even after a very long time.

In general, with Incremental search, queries where one or more keywords matched relatively few nodes had the lowest execution times, even if other keywords matched many nodes; those that had more keywords, each matching many nodes, had higher execution times and cache misses. In contrast, time and cache misses for VM were more directly related to the total number of nodes matching keywords.

The next set of experiments compare our technique with the Sparse technique [14]. For Sparse, we do not know apriori how many queries are required to get 10 results, and Sparse also does not support generating of results in the desired score order. In order to perform as fair a comparison as possible, for Sparse we executed all queries that could generate answers smaller (in terms of number of edges) than the answer which we manually judged was the last relevant answer.

Figure 11 shows execution times for generating the last relevant result if the number of relevant results is less than 10, or the 10th result otherwise. As was the case for the 10th result, Incremental significantly outperforms Iterative, and also outperforms VM-search. Sparse performs much worse than Incremental, with the sole exception being Q9. Comparing the numbers in this graph with that in Figure 9, we can see that it takes significantly less time to generate the last relevant result than to generate the 10th result.

We additionally compared two versions of Incremental Expansion, approach: (a) one which expands only the closest supernode to each keyword, and (b) one which expands all supernodes in the result. Across all the queries we ran, the total time taken was roughly comparable for both approaches, but approach (b) had a significantly (between 5 to 50 percent) higher cache miss rate in most cases, although there were a few cases where it had a lower cache miss rate. Approach (b) also explored/touched more nodes. Overall, the approach (a) is preferable, and all our experimental results used this approach. We omit detailed results for lack of space.

We carried out an additional experiment of measuring performance with varying supernode sizes. We omit the results for lack of space, but we observed that performance improved marginally when we increased supernode size from 100 to 200, but decreased sharply when supernode size was increased further to 400. Since our cache size was fixed at 24 MB, the number of supernodes that fit in cache reduced from 1024 to 256, which lead to thrashing.

8. CONCLUSIONS AND FUTURE WORK

In this paper we considered the issue of keyword search on graphs that may be larger than memory. We showed how to create and exploit a multi-granular representation of data. We developed the iterative and incremental approaches to extending existing search algorithms to work on multi-granular graphs, and showed incremental expansion search significantly outperforms alternative techniques.

We are currently developing a version of the Bidirectional search algorithm that works on the multi-granular graph representation. Developing clustering techniques that are more effective than EBFS is another important area of our onging research. Preliminary results indicate that clustering techniques based on finding communities in graphs can provide very good compression, allowing our techniques to be used effectively on data sets much larger than memory. We are also currently creating very large datasets, using Wikipedia and Web crawl data.

An alternative to storing a large graph in external memory is to distribute it across the main-memory of multiple nodes in a parallel environment. Keyword search in such an environment, exploiting the multi-granular graph representation, is an ongoing area of our research.

9. REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhari, and Gautam Das. DBXplorer: A system for keyword-based search search over relational databases. *ICDE*, 2002.

[2] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. ObjectRank: authority-based keyword search in databases. In *VLDB*, 2004.

[3] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. *ICDE*, 2002.

[4] Kumar Gaurav Bijay. Towards external memory algorithms for keyword-search in relational databases. *Bachelors Thesis, under the guidance of S. Sudarshan*, 2006.

[5] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000.

[6] A. L. Buchsbaum and J. Westbrook. Maintaining hierarchical graph views. In *SODA*, pages 566–575, 2000.

[7] Soumen Chakrabarti, Jeetendra Mirchandani, and Arnab Nandi. Spin: searching personal information networks. In *SIGIR*, page 674, 2005.

[8] Edward P. F. Chan and Ning Zhang. Finding shortest paths in large network systems. *ACM-GIS*, pages 160–166, 2001.

[9] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. *ICDE*, 2007.

[10] J. Graupmann, R. Schenkel, and G. Weikum. The SphereSearch engine for unified ranked retrieval of heterogeneous XML and web documents. In *VLDB*, 2005.

[11] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[12] Nitin Gupta. EMBANKS: Towards disk based algorithms for keyword-search in structured databases. *Bachelors Thesis, under the guidance of S. Sudarshan*, 2006.

[13] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. BLINKS:ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style keyword search in relational databases. *VLDB*, 2002.

[15] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, 2002.

[16] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. *VLDB 2005*, pages 505–516, 2005.

[17] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing*, 1995.

[18] Charles E. Leiserson, Satish Rao, and Sivan Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers (extended abstract). In *FOCS*, pages 704–713, 1993.

[19] W. S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Query relaxation by structure and semantics for retrieval of logical web documents. *IEEE Trans. Knowl. Data Eng.*, page 14(4), 2002.

[20] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, 2006.

[21] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: Top-k keyword query in relational databases. In *SIGMOD*, 2007.

[22] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16:181–214, 1996.

[23] Sriram Raghavan and Hector Garcia-Molina. Representing Web graphs. *ICDE*, pages 405–416, 2003.

[24] Shashi Shekhar, Andrew Fetterer, and Bjajesh Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In *SSD*, pages 94–111, 1997.

[25] Qi Su and Jennifer Widom. Indexing relational database content offline for efficient keyword-based search. In *IDEAS*, pages 297–306, 2005.

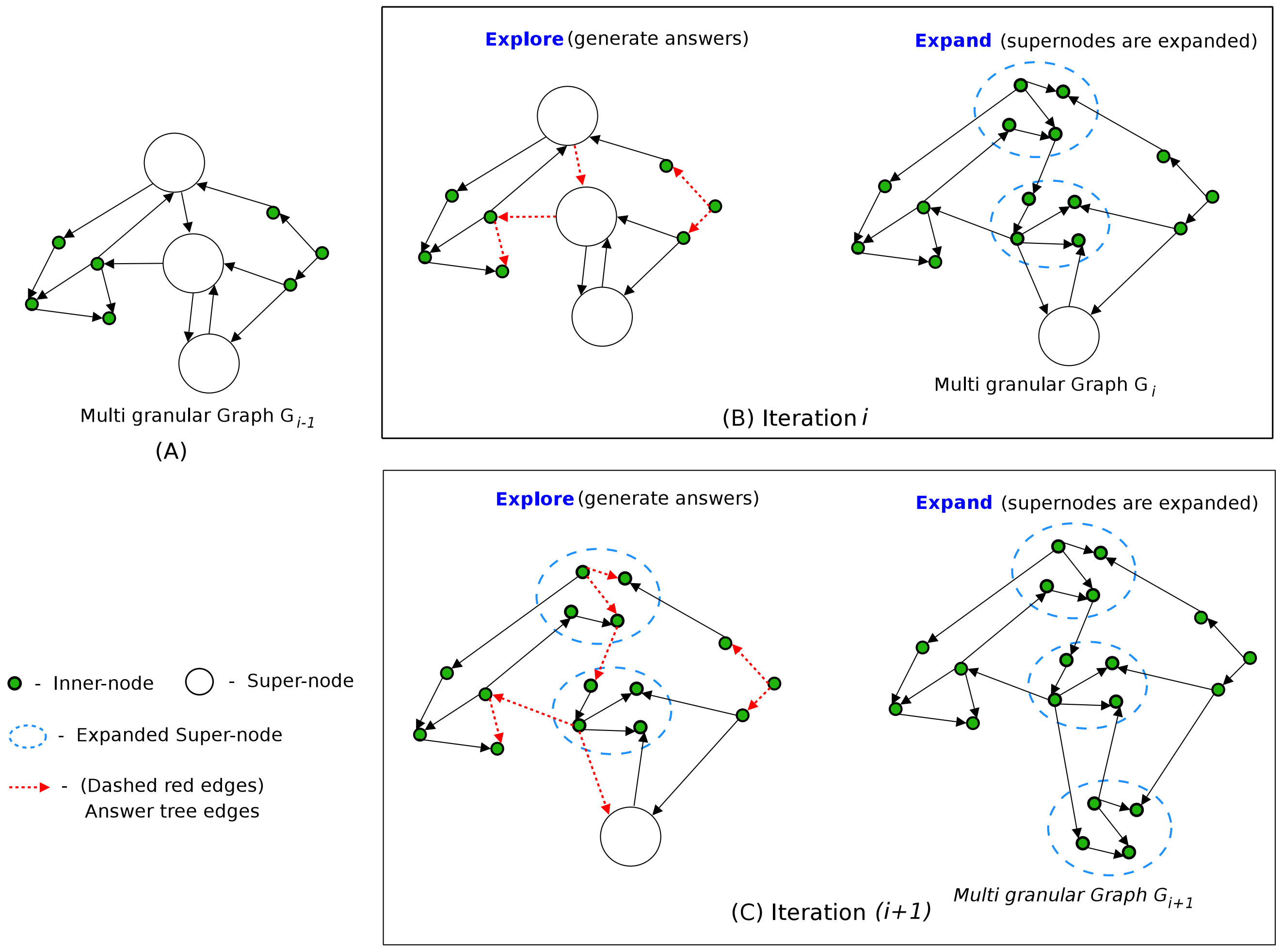
Explore (generate answers)
Expand (supernodes are expanded)
Multi granular Graph G_{i-1}
(A)
Multi granular Graph G_i
(B) Iteration i
- Inner-node
- Super-node
- Expanded Super-node
- (Dashed red edges) Answer tree edges
Explore (generate answers)
Expand (supernodes are expanded)
(C) Iteration $(i+1)$
Multi granular Graph G_{i+1}

Figure 12: Iterative Search: Example

APPENDIX

A. ITERATIVE SEARCH EXAMPLE

Figure 12 shows an example of Iterative Search. In the figure, part (A) shows G_{i-1}, the state of a multi-granular graph at the end of an iteration $i-1$. Part (B) shows how the Explore phase of iteration i takes G_{i-1} as input, and runs the in-memory search algorithm on it. The result trees generated are shown by dashed red edges. Part (B) also shows the state after the supernodes in these results are expanded and replaced by corresponding innernodes in the Expand phase, creating a new graph G_i. G_i acts as input to iteration $i+1$. In each iteration, the Explore phase restarts the search on the graph generated as the output of the Expand phase of the previous iteration.

B. INCREMENTAL SEARCH EXAMPLE

Figure 13 shows the key step in Incremental Search, where the multi-granular graph is updated after a supernode $S1$ is expanded (i.e., replaced by its inner nodes). This step is carried out by the procedure *DeleteSupernodeFromIterator()* in Figure 4).

Part (A) shows a supernode result in which $S1$ is the first supernode on path from keyword $K1$ to the root of the result tree. Part (B) shows the part of SPI tree of $K1$ which contains $S1$. Since nodes are not physically shared between the SPI trees of different keywords, $S1$ is separately replaced by inner nodes in each of the SPI trees where it is present. The diagram shows the deletion from SPI tree of $K1$ only. Deletion from other SPI trees is identical.

Part (C) shows the deletion of $S1$ from the SPI tree. Not only is $S1$ deleted from the SPI tree, but also all affected paths, i.e. the shortest paths going through $S1$, are deleted (by the call to *DeleteSubTree()* in line 5 of Figure 4). This implies that the entire subtree (D_s) of $S1$ will be deleted and will get temporarily disconnected from the unaffected part of SPI tree.

Next we replace $S1$ by its inner nodes. Part (D) shows this step. Each inner node n_j is attached to the unaffected and already explored node '$minNode_j$' of the multi-granular graph that results in the minimum path cost '$minCost_j$' from $K1$ (by the call to *FindMinAndAttach()*, in line 13 of *DeleteSupernodeFromIterator*, in Figure 4). If no such $minNode_j$ is found, n_j does not get attached to the SPI tree. All inner nodes which get attached are put in the priority queue with the newly computed path-cost ($minCost_j$), and get explored later.

Part (E) shows the last step where, each node n_j in D_s

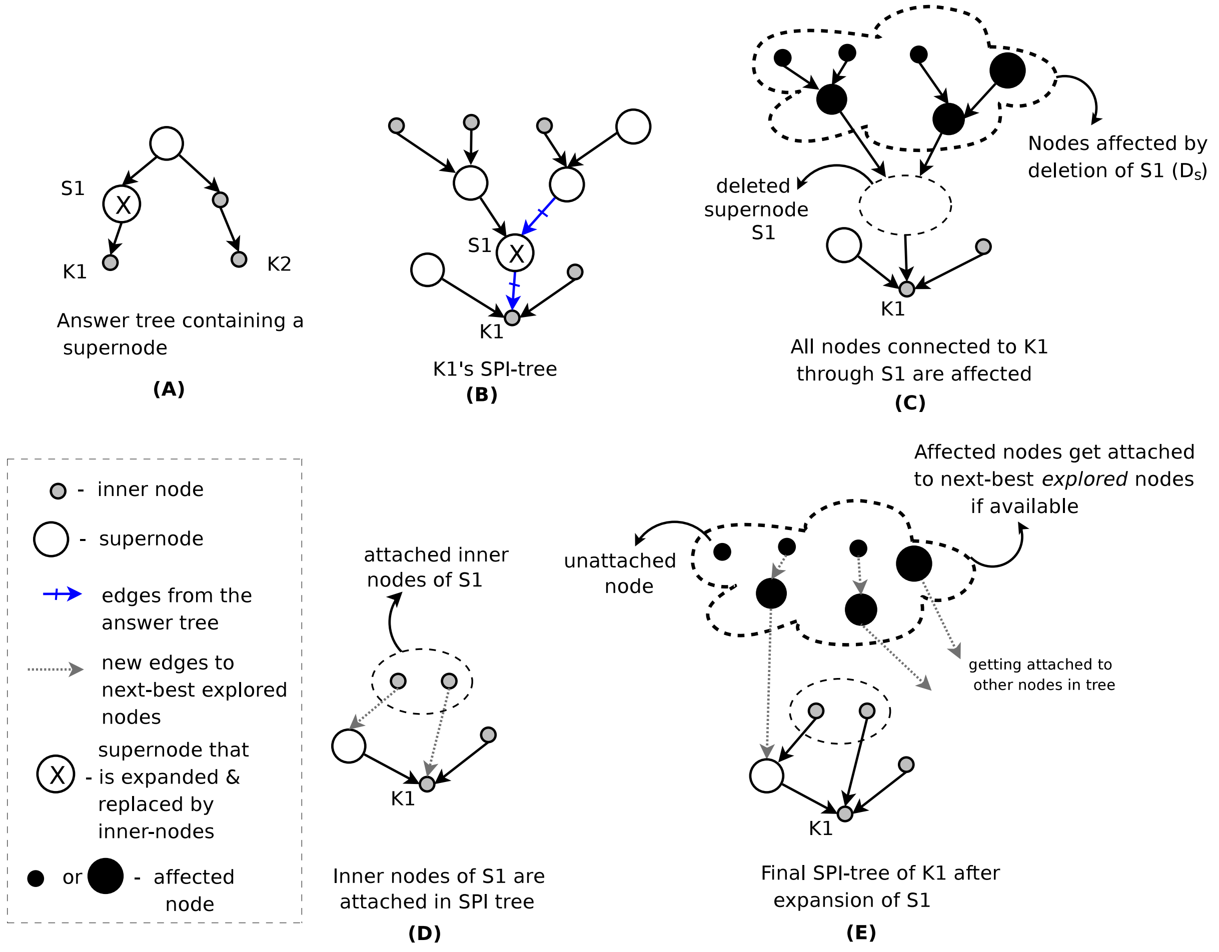
S1
K1
K2
Answer tree containing a supernode
(A)
S1
K1
K1's SPI-tree
(B)
Nodes affected by deletion of S1 (D_s)
deleted supernode S1
K1
All nodes connected to K1 through S1 are affected
(C)
inner node
supernode
edges from the answer tree
new edges to next-best explored nodes
supernode that is expanded & replaced by inner-nodes
or - affected node
attached inner nodes of S1
K1
Inner nodes of S1 are attached in SPI tree
(D)
Affected nodes get attached to next-best *explored* nodes if available
unattached node
getting attached to other nodes in tree
K1
Final SPI-tree of K1 after expansion of S1
(E)

Figure 13: Incremental Search: Updates to the multi-granular graph upon expansion of $S1$

is attached to an explored and unaffected node $minNode_j$, with the best path-cost from $K1$ (by the call to *FindMinAndAttach()*, in line 15 of *DeleteSupernodeFromIterator*). If no such node $minNode$ is found, n is simply deleted from the SPI tree.

Figure 13 shows the update to the SPI tree of $K1$ when $S1$ is expanded. All SPI trees containing $S1$ are updated similarly. The same procedure is followed for each expanded supernode, updating each SPI tree in which it occurs.

C. OTHER EXPERIMENTS

In addition to the experiments reported in Section 7 we performed several other experiments, which are described below.

C.1 Thrashing Control Experiments

As mentioned earlier in Section 7.3.1, the thrashing-control heuristics from Section 6.2 resulted in improved performance, with good recall. Figure 14 shows further results on the effectiveness of the thrashing-control heuristics. Q10 to Q15 are queries which exhibit thrashing behaviour. The graph in Figure 14 compares the total query execution time (plotted in log scale) for these queries, with thrashing-control (TC) and without thrashing-control (NoTC), on three different cache-sizes: 1024, 1536 and 2048 nodes. It can be seen that with thrashing control, the query execution times are significantly better, with differences of nearly two orders of magnitude in some cases. It is clear that thrashing control is essential for good performance. It can also be seen that

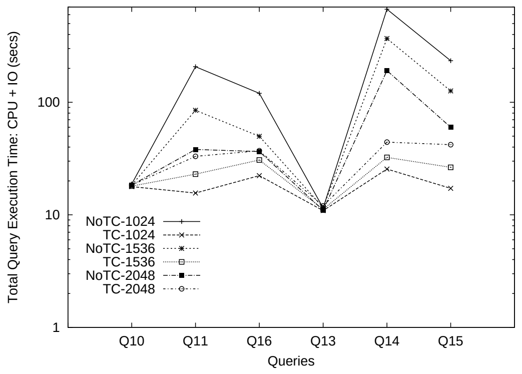

Figure 14: Effect of Thrashing Control on Incremental Expansion

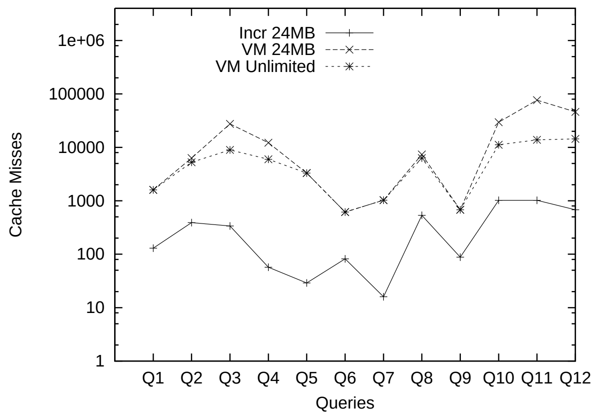

Figure 15: Cache Misses for VM-Search with Unlimited Cache Size

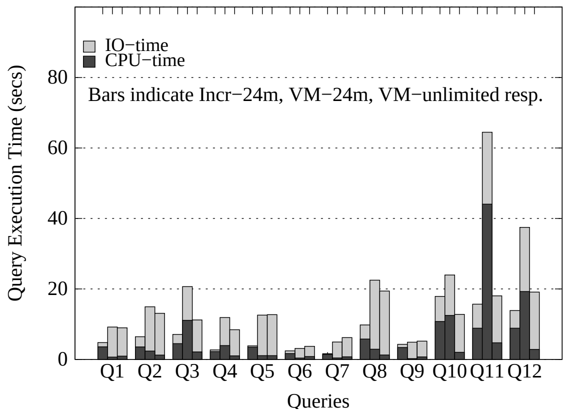

Figure 16: CPU and IO Time for VM-Search with Unlimited Cache Size

as the cache size increases, time taken decreases for the case of no thrashing control since the number of repeat IOs is reduced. In contrast, it increases for the case of thrashing control since more nodes are expanded. As explained in Section 7.3.1, recall went down for only one query, Q11.

As mentioned in Section 7.3.2, we also tried a variant of VM-Search which prevents thrashing by stopping expansion of nodes once the cache is full; search continues till completion, on whatever part of the graph is in the cache at that point. This variant showed very poor recall for queries that exhibited thrashing. With a cache size of 24 MB (3510 nodes), we studied the performance on Q2, Q3, Q4, Q11 and Q15, which exhibited thrashing. With thrashing control as above, only Q2 had 100% recall. Q3 had 33% recall, Q4 had 60% recall, while Q11 and Q15 had 0 % recall.

C.2 VM-Search with Unlimited Cache Size

Compared to Incremental Search, VM-Search performs extra IO operations because it is forced to expand all supernodes encountered during search, whereas Incremental expands only those that result in supernode answers. For the case where there is thrashing, there are also extra IOs due to repeated fetches into cache. To separate out the first component, which is intrinsic, from the second, which depends on the cache size, we ran another set of experiments, comparing Incremental with VM-Search using unlimited cache size (in

Query	With Thrashing Control			
	CPU Time	IO Time	Cache Misses	Recall
Q1	4.094	1.377	132	83
Q2	4.035	3.667	391	100
Q3	5.378	2.754	338	100
Q4	1.446	0.578	57	100
Q5	3.644	0.417	29	100
Q6	1.796	0.863	83	60
Q7	0.825	0.55	16	100
Q8	4.581	3.981	512	100
Q9	3.289	1.286	92	100
Q10	7.101	3.727	510	83
Q11	5.76	4.93	510	75
Q12	7.397	3.925	512	100

Figure 17: Results for Incremental with 512 Supernode Cache Size

the experiments, we used a cache size of 25000 supernodes, which is greater than the database size of around 17000 supernodes). The results are shown in Figure 15, which shows the cache misses, and Figure 16, which shows the CPU and IO time taken.

The results show that the cache misses and CPU/IO time are significantly better for VM-Search with unlimited cache, compared to VM-Search with a 24 MB cache, for queries where VM-Search with a 24 MB cache showed thrashing. However, Incremental with 24 MB cache (equivalent to 1024 supernodes, after accounting for the supernode graph size) still outperforms VM-Search with unlimited cache, by a factor of 10 or more on cache misses. For total time taken, Incremental with 24 MB cache beats VM-Search with unlimited cache on all but one query, Q10, where VM-Search is somewhat faster. Further, for most of the queries Incremental continues to be up to 2 faster than VM-Search with unlimited cache.

C.3 Incremental wih Smaller Cache Size

Our earlier experiments had a minimum cache size of 1024 nodes, corresponding to about 1/17th of the total number of supernodes. To measure performance at even smaller cache sizes, we performed experiments on Incremental Search, setting the cache-size to 512 nodes. Figure 17 shows the results, including CPU and IO time, cache misses, and recall. We observe that for most queries in our query set, 512 nodes are sufficient to hold the portion of the graph relevant to the query. For a few queries, the cache misses were larger in number, and thrashing-control kicks in. Even for such queries, it can be seen that the recall is quite good.

C.4 Varying Number of Results Retrieved

The performance results in Section 7 were for fetching the top 10 results. In the next set of experiments, we compare the performance of Incremental as well as VM-Search when fetching the top 20 results, instead of the top 10 results. We set the cache size to 24M (equivalent to 1024 supernodes for Incremental), and measured the number of cache misses and time taken.

The results are shown in Figure 18 and 19. It can be seen that the number of cache misses do increase when going from 10 to 20 results, but typically by a factor of only around 2. However the relative performance of VM-Search and Incremental remains essentially the same when retrieving the top 20 results as when retrieving the top 10 results. In terms of time taken, the increase is generally less than a factor of 2, but the relative performance of Incremental and VM-Search again remains essentially unchanged.

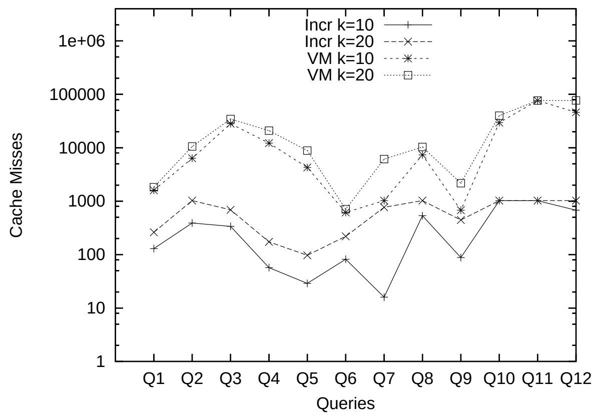

Figure 18: Cache Misses with Varying Number of Results Retrieved

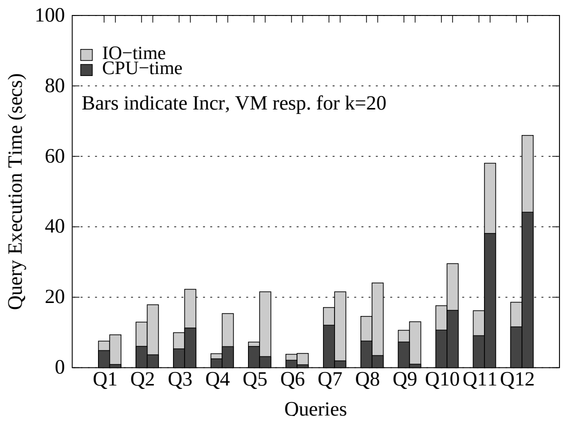

Figure 19: CPU and IO Time for Retrieving 20 Results