Keyword Search over Relational Tables and Streams

ALEXANDER MARKOWETZ
University of Bonn, Germany.
YIN YANG
Hong Kong University of Science and Technology, Hong Kong.
AND
DIMITRIS PAPADIAS
Hong Kong University of Science and Technology, Hong Kong.

Abstract: *Relational keyword search* (R-KWS) provides an intuitive way to query relational data without requiring SQL, or knowledge of the underlying schema. In this paper we describe a comprehensive framework for R-KWS covering snapshot queries on conventional tables and continuous queries on relational streams. Our contributions are summarized as follows: (i) we provide formal semantics, addressing the temporal validity and order of results, spanning uniformly over tables and streams; (ii) we investigate two general methodologies for query processing, *graph based* and *operator based* that resolve several problems of previous approaches; and (iii) we develop a range of algorithms and optimizations covering both methodologies. We demonstrate the effectiveness of R-KWS, as well as the significant performance benefits of the proposed techniques, through extensive experiments with static and streaming datasets.

Categories and Subject Descriptors: H.2.3 Database Management Languages, H.3.3. Information Search and Retrieval

General Keywords: Algorithms

Keywords: Keyword Search, Relational Databases, Data Streams, Query Processing, Data Graph

1. INTRODUCTION

With the rise of the Web, the vast majority of users have adapted *keyword search* (KWS) as a primary tool to access information. In conventional KWS, each document/Web page constitutes one *unit of information*, and is considered a result, if it contains a subset of the query's keywords. Recently, KWS

Authors' addresses: Alexander Markowetz, Department of Computer Science, University of Bonn, Germany. E-mail: alex@iai.uni-bonn.de; Yin Yang, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong; E-mail: yini@cse.ust.hk; Dimitris Papadias, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong; E-mail: dimitris@cse.ust.hk

has also been applied to relational DBMS, allowing data retrieval without SQL. In *relational keyword search* (R-KWS), the basic unit of information is a record/tuple. In contrast to KWS on documents, R-KWS queries cannot be answered by inspecting records individually. Instead, results have to be *constructed* by joining tuples. R-KWS has several benefits over SQL queries. First, it liberates users from studying a (possibly messy) database schema. Queries can instead be issued without knowledge of tables, their attributes, or join conditions. Second, R-KWS allows querying for terms in unknown locations (tables/attributes). Finally, a keyword query replaces numerous complex SQL statements, whose number (frequently in the thousands), prohibits hand-coded SQL on any database with a non-trivial schema. For such broad queries, R-KWS poses the only practical solution.

The external simplicity of R-KWS hides a great internal complexity, involving a vast search space. Specifically, an R-KWS system must explore all possible keyword occurrences (in every table and attribute), as well as their interactions. There are two general methodologies for processing R-KWS queries: *graph based* (GB) and *operator based* (OB). The former maintains an in-memory *data graph*, where data tuples are represented by nodes, connected through edges iff they can be joined. Results (sub-graphs) are retrieved by means of graph traversal. In contrast, OB R-KWS enumerates and executes a set of operator trees, similar to hand-coded SQL. These trees are generated exhaustively, such that *any* combination of keyword occurrences is found and returned to the user.

In this paper we propose a general framework covering both snapshot R-KWS on static tables and continuous queries on relational streams. Our contributions are summarized as follows:

- We propose uniform semantics that accommodate general join conditions, and take into account temporal validity and order of results.
- We present GB and OB query processing methodologies that are applicable to both streams and static databases. The methodologies resolve problems of previous approaches related to duplicate elimination.
- We devise a streamlined graph traversal that significantly accelerates GB query processing. Keyword labeling schemes further improve performance for continuous queries, by indicating which keywords can be reached from a given node.
- We design a highly efficient algorithm for OB systems that integrates operator trees into a *mesh*, which can grow and shrink dynamically, adapting to data characteristics.
- We experimentally compare our methods using static and streaming data, and investigate their effectiveness under different settings.

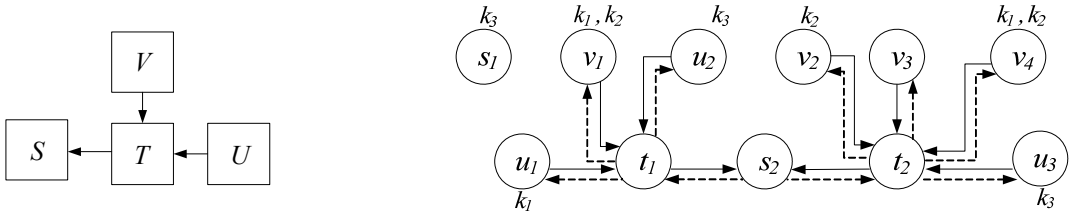
The remainder of the paper is structured as follows. Section 2 outlines related work on R-KWS and data streams. Sections 3 and 4 present generalized R-KWS semantics and processing methodologies for static databases and data streams, respectively. Section 5 enhances the efficiency of GB using keyword labels, and Section 6 proposes optimizations for OB. Section 7 evaluates the benefits of our algorithms experimentally. Finally, Section 8 concludes the paper.

2. RELATED WORK

Section 2.1 outlines GB, and Section 2.2 OB systems. For each approach we discuss one representative system, namely *Banks* and *Discover*, respectively. Section 2.3 presents existing work that falls outside these categories. Section 2.4 provides an overview on data streams.

2.1. Graph Based R-KWS

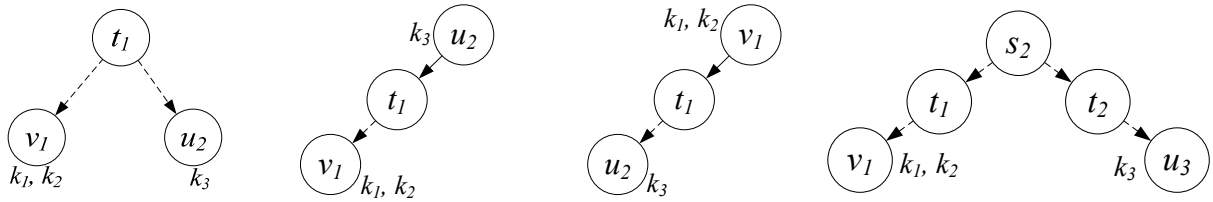
Graph based R-KWS query processing was introduced in *Banks* [Bhalotia et al. 2002], the first system for relational keyword search. *Banks* defines R-KWS semantics on a graph representation of the database. Each node in the *data graph G* corresponds to a tuple, and edges connect nodes/tuples that can be joined. Figure 1b illustrates the data graph for a database consisting of four tables (*S*, *T*, *U, V*), whose schema is shown in Figure 1a. In our notation, s_i signifies a tuple of *S*, t_i one of *T*, etc. Keywords $\{k_1, k_2, k_3\}$ are noted next to the tuples in which they occur, e.g., k_1 and k_2 exist in v_1. Two tuples (e.g., s_2, t_1) are connected in *G* by a (solid) edge, iff (i) their corresponding relations (*S*, *T*) are connected in the schema, and (ii) the tuples satisfy the corresponding join conditions. *Banks* limits join conditions to foreign to primary key relationships, and uses a pair of *directed* edges for each connection. Specifically, a *forward edge* (shown in solid lines) points from the tuple containing a foreign key to its primary key partner (e.g., v_1 has a foreign key referencing t_1), and a *backward edge* (in dotted lines) runs in the opposite direction.



(a) Schema (b) Data Graph

Fig. 1. Database and graph representation

The result of an R-KWS query $q = \{k_1, \ldots, k_m\}$ is the set of trees in *G* satisfying the following conditions: (i) edges (forward or backward) point from the root towards the leafs, (ii) each leaf node contains at least one keyword in *q*, and (iii) the nodes jointly cover all terms of *q*. Assuming the data graph of Figure 1 and the query $q = \{k_1, k_2, k_3\}$, Figure 2 depicts several result trees. Observe that the two trees in Figures 2a and 2b differ only in the direction of the edge between t_1 and u_2. From a user's perspective, these are de-facto duplicates, since both edges ($u_2 \rightarrow t_1$ and $t_1 \rightarrow u_2$) represent the same association between t_1 and v_1. Similarly, reversing the edge between t_1 and v_1 in Figure 2a yields another duplicate (Figure 2c). In general, for every result *r* with |*r*| nodes, *Banks* produces |*r*| copies, each of which has one of the |*r*| nodes as the root.



(a) Result with 3 nodes (b) Duplicate of (a) (c) Duplicate of (a) (d) Result with 5 nodes

Fig. 2. Example outputs of *Banks* for $q = \{k_1, k_2, k_3\}$

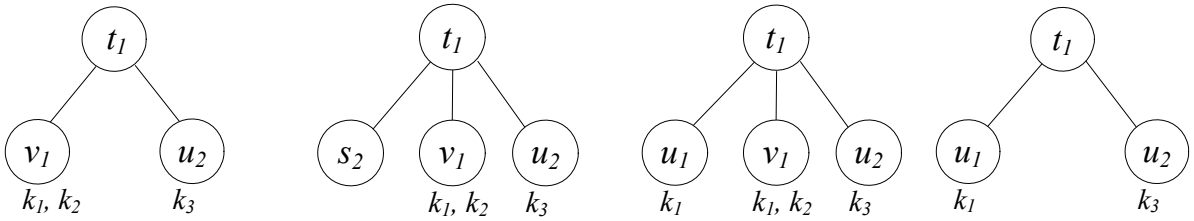
Banks maintains the data graph in main memory. In order to reduce the space consumption, the graph only consists of tuple identifiers; concrete attribute values are stored on disk. Bhalotia et al. [2002] report 100 MB of memory for the graph of the DBLP data set, containing roughly 120,000 nodes and 310,000 edges. For an incoming query, the system first identifies all tuples with keywords, using a disk-resident inverted index. Next, it traverses *G* for result trees. Specifically, it initiates a graph traversal at each node containing a keyword, following edges in their backward direction. Visited tuples record the source nodes

and their keywords. Once a node has been visited by all keywords, it generates a result by following the reverse paths to the sources. Assuming $q = \{k_1, k_2, k_3\}$ in Figure 1, traversals are initiated at v_1 and u_3 (among others), since these nodes contain keywords. When the traversals meet (e.g. at s_2) a new result is created, consisting of s_2 and the paths to v_1 and u_3 (Figure 2d). In contrast to this simplified description, *Banks* weighs nodes and edges to rank results, and attempts an early generation of high scoring results, using Dijkstra's algorithm.

Kacholia et al. [2005] accelerate GB query processing by *bidirectional-expansion* that traverses the data graph both *backwards* (from nodes that can reach all keywords, as in the original system) and *forwards* (from nodes containing at least one keyword) simultaneously. From a theoretical perspective, Kimelfeld and Sagiv [2005 and 2006] and Golenberg et al. [2008] investigate the complexity of relational keyword search, and propose algorithms that produce top-k results with the guarantee of *polynomial-delay*. *Blinks* [He et al. 2007] reduces the search space through a pre-computed two-level reachability index of the data graph. Reachability indexing is further explored by Markowetz et al. [2009].

2.2. Operator Based R-KWS

Discover [Hristidis and Papakonstantinou 2002] translates an R-KWS query into a series of SQL statements, executed directly on the underlying DBMS. The union of results answers the query. Although this methodology differs radically from GB, its semantics are also defined on a data graph G, which remains conceptual and is never materialized. Similarly to *Banks*, the system is restricted to foreign-to-primary key relationships. However, Hristidis and Papakonstantinou [2002] avoid de-facto duplicates, by considering a data graph without back-edges. Given the schema and data of Figure 1, the graph remains the same, but the dotted edges are removed. Consequently, the trees of Figures 2b and 2c do not exist in this graph, and hence do not form results. Tuples t_1, v_1 and u_2 (Figure 3a) are still connected and intuitively constitute a result that is equivalent to those of Figures 2b and 2c. However, the structure of Figure 3a does not have a distinct root node. Accordingly, *Discover* evades the concept of trees through *Join Networks of Tuples* (JNT), which are connected acyclic components of G.



(a) MTJNT (b) Non-minimal JNT (c) Non-minimal JNT (d) Non-total JNT

Fig. 3. Join Networks of Tuples

Figure 3 depicts several JNT for the data of Figure 1. JNT lack a root node: depending on the structure's orientation, every node can serve as root. Likewise, there are no distinct leaf nodes. We hence refer to nodes with degree ≤ 1 (in the JNT) as *terminal nodes*. Given a query, a JNT is called *total*, iff it contains all keywords. The JNT in Figures 3a through 3c are total for $q = \{k_1, k_3, k_3\}$; the one in Figure 3d lacks k_2. A JNT is called *Minimal Total JNT* (MTJNT), iff it is impossible to remove any node and find the remainder to be total. In particular, minimalism is satisfied, iff every terminal node contains at least one *unique* keyword (that is not contained in any other node of the JNT). The JNT in Figure 3b (resp. 3c) is not minimal because after node s_2 (resp. u_1) is removed, the remainder still constitutes a total JNT. *Discover* answers an R-KWS query q with the set of MTJNT. In contrast, *Banks* only requires terminal

nodes to contain *some* keyword. Consequently, its results form a superset of *Discover*'s. For instance, the JNT of Figure 3c is not a result in *Discover*, but the equivalent tree is a result of *Banks*. Finally, *Discover* imposes an upper limit of T_{max} nodes per MTJNT, in order to avoid long chains of joins, which usually lead to uninteresting results.

Discover's query processing relies on an *expanded schema* [1] and *candidate networks* (CN). Specifically, given a query $q = \{k_1, \ldots, k_m\}$, its corresponding expanded schema $EG(q)$ is a graph, constructed as follows. For each table S in the database and every set of keywords $K \subseteq q$, the expanded schema contains a node $S\{K\}$, where K may be empty. A pair of nodes $S\{K\}$ and $T\{K'\} \in EG(q)$ is connected by an edge iff their base relations S and T can be joined through a foreign-key to primary-key relationship. For example, assuming $q = \{k_1, k_2, k_3\}$, there are eight nodes in $EG(q)$ for each relation S, namely $S\{\}$, $S\{k_1\}$, $S\{k_2\}$, $S\{k_3\}$, $S\{k_1, k_2\}$, $S\{k_1, k_3\}$, $S\{k_2, k_3\}$ and $S\{k_1, k_2, k_3\}$. Every $S\{K\}$ is connected with each $T\{K'\}$ since S and T are related in the schema of Figure 1. *Candidate networks* (CN) are projections of MTJNT onto the expanded schema. In particular, a tuple s of relation S maps to node $S\{K\} \in EG(q)$, iff s contains all keywords in K, but does not contain any other term in $q \setminus K$. Figure 4 illustrates several MTJNT and their corresponding CN. For instance, the MTJNT (v_1, t_1, u_2) maps to the CN ($V\{k_1, k_2\}$, $T\{\}$, $U\{k_3\}$). Observe that an MTJNT projects to a unique CN, while it is possible for multiple MTJNT, e.g., (v_1, t_1, u_2) and (v_4, t_2, u_3), to map to the same CN. In contrast to MTJNT, a CN potentially contains multiple copies of the same node in $EG(q)$, e.g., $T\{\}$.

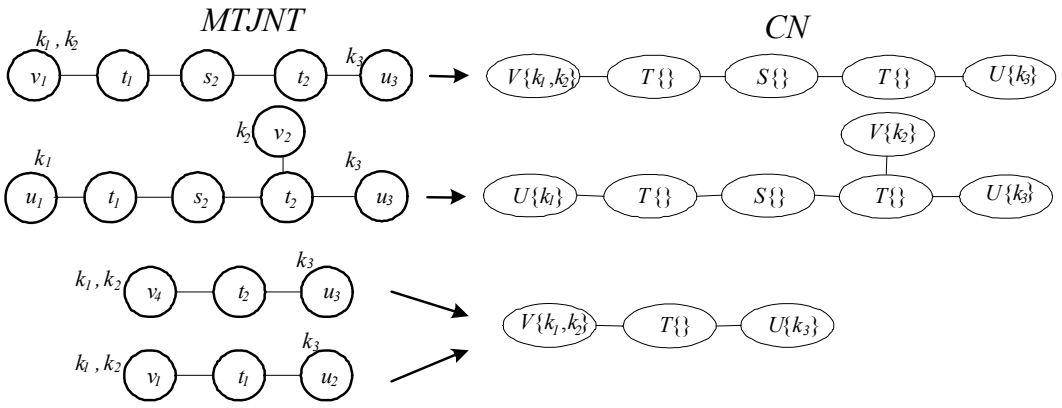
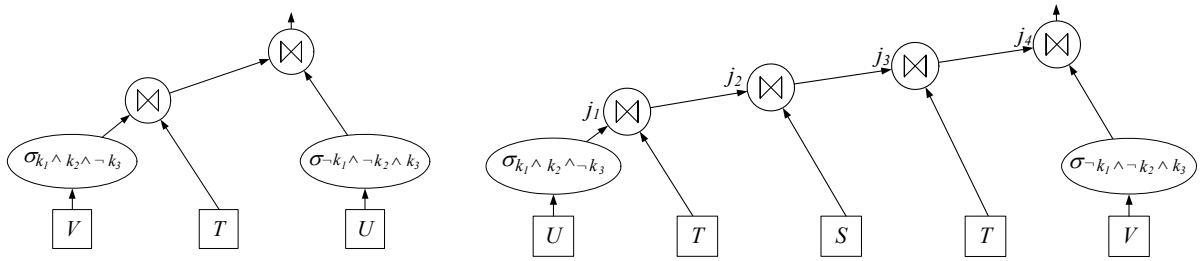


Fig. 4. Examples of MTJNT and CN

Discover answers a query by: (i) generating *all possible* CN, (ii) converting CN to operator trees, and (iii) executing operator trees to produce results. We first clarify the translation from CN to operator trees. Given a CN (e.g., at the bottom of Figure 4), *Discover* generates the corresponding operator tree, shown in Figure 5a, as follows. Each node with an non-empty keyword set ($V\{k_1, k_2\}$ and $U\{k_3\}$) translates into a selection over the base table ($\sigma_{k1 \wedge k2 \wedge \neg k3} V$ and $\sigma_{\neg k1 \wedge \neg k2 \wedge k3} U$), reporting tuples that contain precisely those keywords ($\{k_1, k_2\}$ and $\{k_3\}$). The remaining CN nodes with an empty keyword set (e.g., $T\{\}$) map to the base relation (e.g., T) *without selection*, referred to as *free tuple set* [Hristidis and Papakonstantinou 2002]. On top of the selections and free tuple sets, *Discover* builds a join tree, with conditions placed according to the edges of the CN. In our example, the edge between $V\{k_1, k_2\}$ and T translates into the foreign-key join between $\sigma_{k1 \wedge k2 \wedge \neg k3} V$ and T.

[1] The *expanded schema* is proposed, but not named, in [Hristidis and Papakonstantinou 2002]. We introduce the term for easier reference.



(a) Operator tree for ($V\{k_1, k_2\}$, $T\{\}$, $U\{k_3\}$) (b) Operator tree for ($U\{k_1, k_2\}$, $T\{\}$, $S\{\}$, $T\{\}$, $V\{k_2, k_3\}$)

Fig. 5. Example operator trees in *Discover*

Note that executing the tree in Figure 5a may yield joined tuples that are not *minimal*. A tuple $v \in V$ must have keywords k_1, k_2 to pass the corresponding selections. If there is a tuple $t \in T$ containing k_3 *Discover* may output *v-t-u*, because it does not impose any restriction on keywords in *T*. However, *v-t-u* is not a minimal result; node *u* is redundant, since *v-t* already includes all keywords. Moreover, the tree in Figure 5a may produce MTJNT that do not map to the corresponding CN (bottom of Figure 4) and thus lead to duplicates. Assume tuples *v′-t′-u′* joined by the operator tree. If t' contains keyword k_1, the resulting MTJNT *v′-t′-u′* maps to a different CN ($V\{k_1, k_2\}$, $T\{k_1\}$, $U\{k_3\}$), whose operator tree produces a duplicate copy. Finally, complex operator trees may generate results that are not even JNT, let alone MTJNT. Consider the tree shown in Figure 5b, translated from the topmost CN in Figure 4. Since node $T\{\}$ is involved twice, it is possible for a result to contain the same tuple $t \in T$ twice, e.g., *u-t-s-t-v*, violating the requirement that JNT be acyclic.

In addition, the CN generation of *Discover* suffers from duplicates. Its algorithm for creating CN maintains a queue *Q* of CN fragments. Initially, *Q* contains only one fragment, consisting of a single node $T\{k_1\}$, where *T* is an arbitrary relation, and k_1 is the first keyword in *q*. At each step, *Discover* extracts a CN fragment from *Q*, and adds a new node, forming a larger fragment. A complete CN is generated, whenever a fragment contains all keywords. Duplicate candidate networks result from the undirected addition of nodes. Assume that an initial fragment ($T\{k_1\}$) is expanded by adding $U\{k_2\}$ and $V\{k_3\}$ (among others). Consequently, two new CN-fragments are inserted into the heap. When ($T\{k_1\}$, $U\{k_2\}$) is later de-queued, it is expanded by adding $V\{k_3\}$, generating the CN ($T\{k_1\}$, $U\{k_2\}$, $V\{k_3\}$). Similarly, ($T\{k_1\}$, $V\{k_3\}$) is expanded by adding $U\{k_2\}$, generating the CN ($T\{k_1\}$, $V\{k_3\}$, $U\{k_2\}$). However, these two CN are identical. Duplicate CN translate into identical operator trees, and perform redundant computations, exacerbating the problem of duplicate MTJNT. Besides *Discover*, there are several other OB implementations, including *Mragyati* [Sarda and Jain 2001], *DBXplorer* [Agrawal et al. 2002], and *SPARK* [Luo et al. 2007]. Unfortunately, they do not describe CN generation, nor address the above issues. Markowetz et al. [2009] eliminate unproductive CN through reachability indexing, in order to accelerate query processing.

2.3. Additional Work on Relational Keyword Search

Databases were first modeled as a graph by Dar et al. [1998]. Besides GB and OB, there is a *materialization based* (*MB*) approach to R-KWS, followed by *Ekso* [Su and Widom 2005] and *Ease* [Li et al. 2008]. Relying on extensive pre-computations, MB enumerates all sub-graphs in *G* of a certain diameter during pre-processing. For each such sub-graph G_S, it constructs a *virtual document* with the attribute values of all nodes in G_S. These documents are materialized on disk and organized in an inverted index. Upon query arrival, the system uses this index to determine the sub-graphs containing all

keywords, from which it removes unnecessary nodes to restore minimality, and eliminates duplicate results. Such an index grows quickly and soon exceeds the size of the database. In addition, MB is not applicable to dynamic scenarios involving data streams, since the data are not available in advance, rendering pre-computations impossible.

R-KWS has been extended in various directions. Hristidis et al. [Hristidis et al. 2003] relax the notion of minimality of *Discover* by requiring each terminal node in a result to contain a keyword, but not necessarily a unique one. Other systems, such as [Liu et al. 2006] support phrases (e.g., the title of an article), and lists of synonyms (e.g., "car OR automobile OR vehicle"). Several authors propose ranking schemes for top-*k* processing, based on IR ranking functions ([Hristidis et al. 2003], [Balmin et al. 2004], [Chaudhuri et al. 2004], [Golenberg et al. 2008], [Li et al. 2008], [Liu et al. 2006], [Luo et al. 2007]). Additionally, Golenberg et al. [2008] penalize results for overlap (redundancy). KWS has also been extended to XML databases, using various semantics ([Hristidis et al. 2003], [Guo et al. 2003], [Cohen et al. 2005], [Xu and Papakonstantinou 2005], [Liu and Chen 2007]). XML elements form nodes in a data graph; edges connect elements that are related through containment or by reference. Query processing depends on the database's physical layout. *Ease* [Li et al. 2008] supports uniform keyword search over structured, un-structured and semi-structured data. Furthermore, R-KWS has been applied to distributed systems containing multiple, possibly heterogeneous, data sources [Sayyadian et al. 2007]. The methods of [Yu et al. 2007] and [Vu et al. 2008] determine the most promising source for an R-KWS query among a set of distributed databases, using their summaries. Finally, Wu et al. [2007] propose keyword driven OLAP, and De Felipe et al. [2008] investigate R-KWS on spatial databases.

2.4. Data Streams

There is an extensive body of literature on relational data streams. Under this paradigm, data elements (relational tuples) from various sources are collected at a *data stream managing system* (DSMS), where users register continuous queries. When a new tuple arrives, all relevant queries are re-evaluated. Query processing is usually performed by routing tuples through trees of operators, resembling their traditional counterparts such as selections or joins. Influential DSMS systems include: (i) *Aurora* [Abadi et al. 2003], targeting mainly sensor data, (ii) *TelegraphCQ* [Chandrasekaran et al. 2003], focusing on the novel *Eddy* operator [Avnur and Hellerstein 2000], (iii) *Stream* [Arasu et al. 2006], designed as a general purpose DSMS, and (iv) *Pipes* [Krämer and Seeger 2004]. In-depth surveys on DSMS can be found in [Babock et al. 2002] and [Golab and Öszu 2003].

Depending on the application characteristics, DSMS follow different models regarding the validity of tuples. One popular approach assumes a *sliding window* of a given time frame w, i.e., a tuple s expires w time units after its arrival. In this case, all arrivals in the system correspond to insertions; deletions are implicit. Another common model assumes *positive-negative* tuples, i.e., the DSMS receives a negative tuple $-s$ that takes the same route through the operator tree as s, and erases all occurrences of its positive counterpart. In both cases, the *lifespan* of a tuple s is the interval $[s.t_{start}, s.t_{end})$ between its arrival $s.t_{start}$ and the (implicit or explicit) deletion $s.t_{end}$. Two tuples can be joined while their lifespans overlap. An abstract join operator j contains two input buffers *j.left-buffer* and *j.right-buffer*, storing alive tuples from the left and right input. When a new tuple s arrives from the left input, it is inserted in *j.left-buffer* and subsequently compared against all tuples t in *j.right-buffer*. For every pair that fulfills the join condition, a new composite tuple c is created and passed onto higher operators. The lifespan of c is commonly defined as the intersection of participating tuples' lifespans; e.g., $c.t_{start} = max(s.t_{start} + t.t_{start})$ and $c.t_{end} = min(s.t_{end}$

+ $t.t_{end}$). For sliding windows, *c's* lifespan is known at its creation time, and buffers are purged periodically of expired tuples. In the case of positive-negative tuples, the expiration time is not known. Instead, when a negative tuple *–s* arrives in the left input, *s* is removed from *j.left-buffer* and subsequently probed against *j.right-buffer*. For any join result *c*, a negative output *–c* is passed on to higher operators, informing them about *c*'s expiration. Tuples from the right input are handled symmetrically. Various implementations differ mainly in the organization of buffers, e.g., lists or hash tables.

KWS has also been applied to streaming documents (e.g., continuously arriving news articles). With few exceptions ([Yan and Garcia-Molina 1999], [Fabret et al. 2001], [Irmak et al. 2006]), most related work is proprietary. The main difference with respect to our work is that documents do not have to be joined, but are evaluated individually (as in traditional KWS). In a poster, Hristidis et al. [2006] propose KWS over multiple textual streams. Similarly to our work, results are constructed by combining units of information (emails, news articles) from several streams. The authors however do not follow a relational model, leading to several key differences with our problem setting. First, tuples in [Hristidis et al. 2006] have only one attribute, their text. Second, only tuples that contain keywords can contribute to a result. Third, and most significantly, combinations (joins) of several tuples are not evaluated upon their (textual) attribute, but tuples can always be joined, as long as the data streams from which they origin are sufficiently correlated. The correlation between streams is continuously updated, and stored in a stream schema. Unfortunately, the poster does not provide a formal definition of semantics, or details about algorithms and experiments.

In [Markowetz et al. 2007], we introduce keyword search over relational streams, focusing exclusively on OB processing. This paper extends our previous work on the following aspects: (i) we cover both static tables and streams, (ii) we present homogenized R-KWS semantics and query processing techniques for conventional databases, and remedy several common problems encountered by previous systems, (iii) we include GB processing, and (iii) we develop a novel optimization for OB processing.

3. RELATIONAL KEYWORD SEARCH ON TABLES

This section discusses methods for graph and operator based processing that avoid the shortcomings of prior systems and significantly improve performance of R-KWS in conventional databases. In our discussion, we assume an *undirected data graph G* that contains neither directional information nor back-edges. Two tuples are connected by an edge, iff they satisfy the join condition (symmetric or asymmetric), as specified by the application. In practice, many important join conditions, such as the similarity of textual attributes ([Chaudhuri et al. 1995], [Gravano et al. 2003]), are *symmetric*. Undirected edges also avoid the de-facto duplicates of *Banks*. If an application requires results to contain directed edges, it can fetch this information from the schema during post-processing at negligible cost.

Similar to *Discover*, we answer R-KWS queries with the set of MTJNT in *G* that contain up to T_{max} nodes. Recall from Section 2.2 that the concepts related to MTJNT do not involve edge directions, and thus directly extend to the undirected model. The parameter T_{max} allows adjusting run-time cost without restricting R-KWS semantically. Depending on the application, it can be set to an arbitrarily high number, including infinity. We follow the definitions of *Discover* because they are well-understood, popular, and extend easily to data streams. However, we do not champion any particular R-KWS semantics. Similarly, we do not emphasize the ranking of results, since output from streams naturally follows a temporal order. The next two sub-sections describe GB and OB query processing on static data. Both frameworks extend to alternative semantics and support arbitrary monotonous ranking functions.

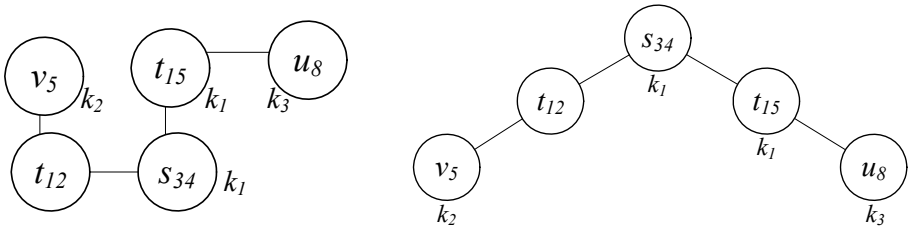
3.1. Graph Based Processing

Our method loosely follows the general paradigm of previous systems. Specifically, given an inverted index I (on disk), it traverses an undirected data graph G (in memory), searching for results. We reduce the problem of finding *all* results in the graph to retrieving those containing a particular node. By definition, every MTJNT includes a node sn containing an arbitrary (but fixed) query keyword, say k_1. It is hence sufficient to identify tuples/nodes containing this term (using I), and then traverse G locally, searching for MTJNT. Figure 6 depicts the pseudo-code for this query processing method, referred to as *GB*. Specifically, GB initiates a graph traversal around each node $sn \in I$.invertedList(k_1), using the *GSearch* procedure (Line 3). Note that it is possible for an MTJNT r to contain multiple nodes with k_1. In this case, r could wrongly be discovered by multiple calls to *GSearch*, each starting with a different seed node. To avoid such duplicates, GB assigns a distinct ID $n.nid$ to each node $n \in G$, and reports a result r only if the current seed node sn of *GSearch* has the smallest ID from all nodes in r that contain k_1 (Lines 5-7). The concrete values for node IDs do not influence correctness as long as no two nodes share the same ID. For ease of presentation, in the following we suppose that node IDs follow a lexicographic order on relation and tuple-ID; e.g., for $s_1, s_2 \in S$ and $t_1 \in T$, we have $s_1.nid < s_2.nid < t_1.nid$.

```
GB (DataGraph G, InvertedIndex I, Query q)          // Performs Graph Based R-KWS
1.  Initialize the result set RS to ∅
2.  For each sn ∈ I.invertedList(k1)                // for every node sn containing a selected keyword k1 of q
3.      List RS_sn = GSearch(G, q, sn)
4.      For each MTJNT r ∈ RS_sn
5.          If (r includes a node n containing k1) AND (n.nid < sn.nid)
6.              Discard r                           // this MTJNT has already been created by GSearch(G, q, n)
7.          Else Insert r into RS
8.  Return RS
```

Fig. 6. Algorithm *GB*

It remains to clarify the data graph traversal. In general, starting from the seed node sn, an MTJNT can be discovered in many ways. Any efficient algorithm, however, should enumerate each MTJNT precisely once. Additionally, its traversal should not stretch over the entire data graph, but must remain constrained to sn's vicinity. At the same time, it has to be extensive enough to ensure that *all* MTJNT containing sn are discovered. In order for *GSearch* to meet these criteria, we model each MTJNT as a unique tree, whose root consists of the seed node sn, while child nodes are ordered left-to-right by increasing nid. Given $q = \{k_1, k_2, k_3\}$ and assuming $sn=s_{34}$, Figure 7 depicts an MTJNT and its tree representation. Node s_{34} serves as the root and its children (t_{12} and t_{15}) are ordered left-to-right.



(a) MTJNT (b) Tree representation

Fig. 7. An MTJNT and its unique representation as a tree

The *GSearch* algorithm, illustrated in Figure 8, enumerates all possible trees in G rooted at sn, in the lexicographic order of their preorder traversal. Whenever the algorithm encounters a tree that corresponds

to an MTJNT, it reports a result. Specifically, *GSearch* maintains a queue Q of trees, each constituting a fraction of a potential MTJNT. At every step, one such tree t_{old} is de-queued (Line 5) and expanded by adding one new node n_{new}, resulting in a new tree t_{new}. To ensure that the preorder traversal of t_{new} is lexicographically greater than that of t_{old}, we enforce the following two requirements. First, n_{new} can only become child of a node nl on the rightmost root-to-leaf path of t_{old} (Line 6). Second, the *sid* of n_{new} must exceed that of its siblings' (Line 8). For example, a new node n_{new} can only be added to nodes s_{34}, t_{15} or u_8 in the tree of Figure 7b. If n_{new} is added to under s_{34}, the *nid* of n_{new} has to be larger than that of nodes t_{12} and t_{15}. The new tree t_{new} falls into one of three categories: (i) it forms an MTJNT, and is included in the result set (Lines 10-11); (ii) it has the potential to become an MTJNT, and is inserted in Q to be expanded later (Lines 13-15); (iii) none of the above, and the tree can be safely discarded (Line 16). In order to form an MTJNT at a later time (the second category), a tree must contain less than T_{max} nodes (Line 13), and all terminal nodes, except on the rightmost path, must have a *unique* keyword (Line 14), i.e., a keyword that does not exist in any other node in t_{new}. The algorithm terminates when Q becomes empty.

```
GSearch (DataGraph G, Query q, Node sn)          // Finds all MTJNT that contain a given node sn
1.  Initialize result set RS to empty
2.  Initialize queue Q to empty      // stores intermediate trees
3.  Insert a tree into Q consisting of a single node sn (the root)
4.  While (Q is not empty)
5.      De-queue the first tree t_old in Q
6.      For each node nl on the rightmost root-to-leaf path in t_old
7.          For each neighbor n_new of nl in G, where n_new ∉ t_old      // n_new must not already exist in t_old
8.              If (n_new.nid > nc.nid, for every child nc of nl )       // the nid must exceed the siblings'
9.                  Create a new tree t_new by adding n_new as the rightmost child of nl
10.                 If (t_new is an MTJNT)
11.                     Insert t_new into RS;
12.                 Else
13.                     If (t_new contains less than T_max nodes)
14.                     AND (each terminal node, except those the rightmost path, contains a unique keyword in q)
15.                         Append t_new to Q;          // t_new still has the potential of forming an MTJNT
16.                     Else, discard t_new
17. Return RS
```

Fig. 8. Algorithm *GSearch*

GSearch can accommodate a looser definition of minimality [Hristidis et al. 2003] by simply relaxing the condition of Line 14. Furthermore, it can capture phrases and synonyms [Liu et al. 2006] by treating them as single keywords. In particular, a node is considered to contain a synonym list l (resp. phrase), if and only if the corresponding tuple contains some (resp. all) keywords in l. Similar modifications for phrases and synonyms also apply to the rest of the proposed methods. The following two lemmas establish the correctness of *GSearch* and *GB*, respectively.

Lemma 3.1: *GSearch* computes the set of MTJNT containing node *sn* correctly, completely, and without duplicates.

Proof: Correctness is trivial since *GSearch* explicitly verifies that every output constitutes an MTJNT. Completeness is established because: (i) without pruning, *GSearch* enumerates every possible tree in the data graph containing *sn*, including those corresponding to MTJNT, and (ii) the pruning conditions eliminate exclusively trees that are guaranteed not to lead to MTJNT. Finally, *GSearch* does not generate duplicates, because it discovers trees at most once, in increasing order of their preorder traversal. □

Lemma 3.2: GB answers an R-KWS query q correctly, completely, and without duplicates.
Proof: The correctness of GB follows directly from the corresponding property of *GSearch*. Its results are complete, because each MTJNT must include at least one node containing k_1 and *GSearch* is called for every such node. Finally, since *GSearch* is free of duplicates, *GB* could only produce duplicates, if the same MTJNT was discovered by separate calls of *GSearch*. Such an MTJNT would have to include several nodes containing k_1 (potential root nodes). However, Lines 5 and 6 of *GB* ensure that MTJNT are recorded only when *GSearch* has been called for the node with the smallest *nid*. Thus, *GB* is free of duplicates. □

Finally, note that the pseudo-code of Figure 8 is *iterative* and maintains state (sub-graphs that have not yet been expanded) by storing trees in a queue. In addition, we perform the duplicate check (Lines 5 and 6 of *GB*) as part of *GSearch*; i.e., if n_{new} contains keyword k_1, its *nid* must be larger than that of *sn*, in order to be added to t_{new}. Although, this design avoids duplicates earlier, we present this check outside *GSearch*, in order to separate the functionality of *GB* and *GSearch*. GB can be adapted to produce results according to a monotone ranking function f, as follows. Instead of calling *GSearch* for each seed node individually, we initialize Q with the set of *all* seed nodes, and generate results through a single instance of the algorithm. Internally, Q is organized as a priority queue according to an estimate of the upper bound of f. At each step, *GSearch* retrieves the most promising partial result, and processes it. The highest ranking results are thus generated first.

3.2. Operator Based Processing

The concepts of operator based systems are oblivious to edge direction, and extend to our semantics more readily than their graph based counterparts. However, as discussed in Section 2.2, *Discover* incurs duplicate CN, duplicate MTJNT and invalid results. In the following, we remedy this behavior and simultaneously adapt operator based query processing to our semantics. In particular, we introduce three modifications: (i) *UniqueJoin*, (ii) a redefinition of $S\{\}$, and (iii) the *CNGen* algorithm for generating CN without duplicates. Compared to the former two, the latter is rather complicated, but inherits many characteristics from *GSearch*, simplifying the presentation.

Recall from Section 2.2 that a join operator consumes composite tuples from its left, and single records from its right input. Furthermore, the only valid output of each operator tree (i.e., MTJNT) is *acyclic*. This gives rise to the following invariant: all intermediate results, generated by join operators at lower levels, must also be free of cycles. To enforce this property, we replace all join operators with their *UniqueJoin* counterpart. *UniqueJoin* verifies that the compound tuple from its left input does not already contain the tuple from the right, in which case the tuples would form a cycle. Assume, for instance, a traditional operator j_{trad} that joins a compound tuple t_{left} and a basic tuple t_{right}, iff j_{trad}.condition(t_{left}, t_{right}) = true. The corresponding *UniqueJoin* evaluates that (i) j_{trad}.condition(t_{left}, t_{right}) = true, *and* (ii) $t_{right} \notin t_{left}$. For the remainder of the paper, we assume all join operators to use *UniqueJoin*.

Next, we re-define $S\{\}$ to denote the set of tuples in relation S that do not contain *any* keyword; i.e., $S\{\} = \{s \in S \mid \forall\, k \in q, s \text{ does } not \text{ contain } k\}$. The definition of candidate networks remains the same, but leaf operators are adapted to the redefined selections. For instance, given $q = \{k_1, k_2, k_3\}$, $S\{\}$ is translated into $\sigma_{\neg k1 \wedge \neg k2 \wedge \neg k3} S$. For the following discussion, we exclusively use the new definition of $S\{\}$. The combination of *UniqueJoin* and the redefinition of $S\{\}$ allows the following statements:

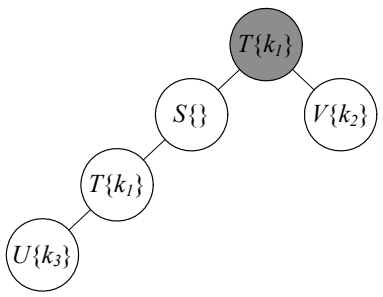
Lemma 3.3: An operator tree *op-tree* translated from a CN only generates correct output.

Proof: *Op-tree* outputs tuples that can be joined and hence form a sub-graph of G. The *UniqueJoin* operator ensures that this sub-graph is *acyclic*, i.e., a JNT. The selection operators ensure that (i) the JNT contains all keywords, i.e., is total, and (ii) has unique keywords in all external nodes, i.e., is minimal. Tuples contain *only specified* keywords, and no others, and thus cannot impair minimality. Hence, all output constitutes MTJNT. □

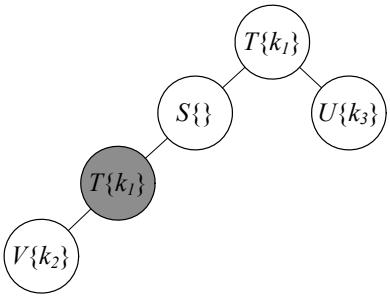
Lemma 3.4: If each CN is translated into one operator tree, then every MTJNT is retrieved exactly once.
Proof: Assume an MTJNT r generated by two different operator trees $op\text{-}tree_1$ and $op\text{-}tree_2$. Let cn_1 and cn_2 be their corresponding candidate networks. For every node $s \in S$ of r containing a set of keywords K, each candidate network contains a node $S\{K\}$. Furthermore, both CN share the same structure as r. With identical nodes arranged in an identical structure, we deduce that $cn_1 = cn_2$. Since every CN was translated into a *single* operator tree, we thus find $op\text{-}tree_1 = op\text{-}tree_2$, violating our initial assumption. Hence, we conclude that every MTJNT is retrieved precisely once. □

The above modifications avoid erroneous and duplicate results. OB can generate output according to a monotone ranking function f, by applying the method of [Hristidis et al. 2003]. In particular, OB stores all CN in a heap, with the most promising one ranking first. Then, it keeps executing the operator tree of the first CN, until it becomes less promising than the second-topmost, at which time it switches to the latter. Next we demonstrate how to compute the set of CN. Our *CNGen* algorithm constructs candidate networks by traversing the expanded schema, a process resembling *GSearch*. Both algorithms face similar challenges and lend themselves to comparable solutions. Akin to MTJNT, we model CN as unique trees, generated according to their unique preorder traversal. CN share the same structure as MTJNT, and must be total and minimal. In contrast to MTJNT, they can contain multiple identical nodes; e.g., the top CN in Figure 4 contains two instances of $T\{\}$. In the following, we focus on this distinction and its implications for duplicate avoidance.

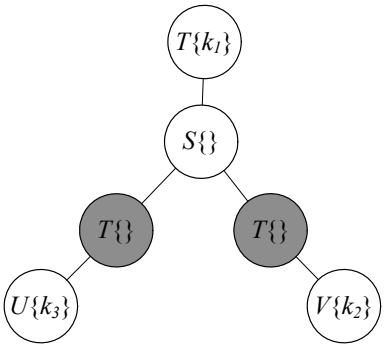
Similar to *GSearch*, *CNGen* requires a total ordering on nodes of the expanded schema. For our presentation, we assume that each relation (e.g., S) has an ID ($S.rid$). Nodes in the expanded schema have a keyword bitmap (*kbit*) according to the contained keywords, e.g., $S\{k_2\}.kbit = 010_b = 4$ and $S\{k_1, k_3\}.kbit = 101_b = 5$. The node order *nid* is a lexicographic combination of *rid* and *kbit*, e.g., $S\{k_1, k_3\}.nid < T\{k_2\}.nid$ and $S\{k_2\}.nid < S\{k_1, k_3\}.nid$. Having defined this order, CN can be represented as unique trees. The node containing keyword k_1 serves as root n_{root}. If several such nodes exist, the smallest *nid* breaks the tie. Child nodes are ordered left-to-right by *nid*. We observe two special cases, caused by the inclusion of multiple copies of the same node. In the first, a CN contains multiple instances of the root node. Here, the lexicographically smaller preorder traversal determines the CN's unique tree representation. Figures 9a and 9b depict the tree representation of a CN containing two instances of $T\{k_1\}$. Only the tree in Figure 9a represents the CN correctly because its preorder traversal ($T\{k_1\}$, $S\{\}$, $T\{k_1\}$, $U\{k_3\}$, $V\{k_2\}$) is lexicographically smaller than that of Figure 9b ($T\{k_1\}$, $S\{\}$, $T\{k_1\}$, $V\{k_2\}$, $U\{k_3\}$). In the second special case, a node has twin children. To establish a unique tree representation, we demand that the sub-tree rooted at the left twin is lexicographically smaller than the one rooted at its right sibling. Consider $S\{\}$ in Figure 9c, having two identical children of type $T\{\}$. The traversal of the sub-tree rooted at the left twin ($T\{\}$, $U\{k_3\}$) is smaller than that rooted at the right sibling ($T\{\}$, $V\{k_2\}$). This tree hence represents the CN correctly, whereas the tree of Figure 9d is incorrect.



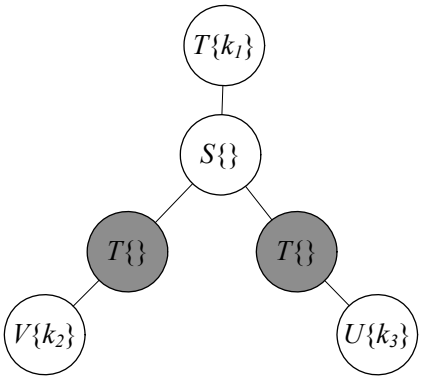
(a) Tree containing a copy of root node $T\{k_1\}$



(b) Duplicate tree



(c) Node $S\{\}$ has twin-children of type $T\{\}$



(d) Duplicate tree

Fig. 9. Candidate networks with multiple identical nodes

Figure 10 provides pseudo-code for *CNGen*, generating all CN whose tree representation is rooted at a node n_{root} of the expanded schema. Similar to *GSearch*, it starts with a tree t_{first} containing only n_{root}. At each step, the algorithm de-queues one tree t_{old} from Q, and spawns new trees by adding a node to the rightmost root-to-leaf path (Lines 6 to 9). In contrast to *GSearch*, it allows multiple instances of the same nodes to be added (Line 7). Again, we distinguish three cases: (i) if t_{new} constitutes a CN, it is included in the result set (Lines 10-11); (ii) if the tree is not a CN, but still has the potential of becoming one, it is inserted in Q (Line 24); and (iii) in any other case, the tree is discarded (Line 26). In order to have the potential for forming a CN in the future, the current tree must have less than T_{max} nodes, and every terminal node (except those on the rightmost path) must contain a unique keyword (Lines 14 and 15).

Similar to GB, a looser definition of minimality akin to [Hristidis et al. 2003] can be implemented by relaxing the conditions in Line 15. Note that these checks resemble Lines 13-14 of *GSearch*. However, *CNGen* imposes two additional conditions, in order to avoid the above duplicates. First, Line 17 evaluates whether t_{new} contains a second instance (n_{copy}) of its root node n_{root} (not necessarily n_{new}, but any older node in t_{new}). In this case, *CNGen* compares the preorder traversal of two trees: (i) t_{new} rooted at n_{root} and (ii) a version t_{copy} of t_{new} that has been rearranged so that n_{copy} serves as root. If the former is lexicographically larger, t_{new} constitutes a duplicate of t_{copy} and can be discarded. The latter tree is generated from a different seedling in Q and does not have to be explored at this point. Second, Line 21 checks all nodes along the rightmost root-to-leaf path in t_{new}. If any such node has an identical sibling, *CNGen* lexicographically compares the preorder traversal of the trees rooted at both nodes. If the left is larger, t_{new} can be discarded; the correct representation of the CN will be generated from another tree in Q.

The outer function *InitCNGen* calls *CNGen* for all nodes containing k_1, e.g., $S\{k_1\}$, $S\{k_1,k_2\}$, $T\{k_1\}$. These nodes then serve as roots of CN-trees. Since every CN must include at least one node containing k_1,

the set of expansions initiated by *InitCNGen* eventually produces *all* CN for the given query and schema. To ensure that individual calls of *CNGen* do not yield duplicates, CN are generated according to their *unique* tree representation. Furthermore, separate calls to *CNGen* must not generate identical CN. This could only happen for CN with multiple nodes containing k_1. To avoid such behavior, *InitCNGen* removes n_{root} from the expanded schema after *CNGen*(n_{root}) terminates (Line 4). Note that this does not cause any loss of results, since all CN containing n_{root} have been generated by *CNGen*(E, n_{root}).

```
InitCNGen (Expanded Schema E)
1.  Initialize result set RS to empty
2.  For each node n_root containing k_1
3.      Call CNGen(n_root) and add all its returned results to RS
4.      Remove n_root from E
5.  Return RS

CNGen (Expanded Schema E, Node n_root)          // Generates all CN rooted at a given node n_root
1.  Initialize result set RS to empty
2.  Initialize queue Q to empty      // stores intermediate trees
3.  Insert a tree into Q consisting of a single node n_root (the root)
4.  While (Q is not empty)
5.      De-queue the first tree t_old in Q
6.      For each node nl on the rightmost root-to-leaf path in t_old
7.          For each neighbor nn of nl in G  // Unlike GSearch, there is no check if t_old already contains n_new
8.              If (n_new.nid ≥ nc.nid, for every child nc of nl )   // the nid must be as large as the siblings'
9.                  Create a new tree t_new by adding nn as the rightmost child of nl
10.                 If (t_new is a CN)
11.                     Insert t_new into RS;
12.                 Else
13.                     Boolean has_potential = false
14.                     If (t_new contains less than T_max nodes)
15.                     AND (each terminal node, except those the rightmost path, contains a unique keyword in q)
16.                         has_potential = true
17.                     If (t_new contains a second node n_copy that is identical to n_root)          // anywhere in the tree
18.                         Create a copy t_copy of t_new and rotate it, so that n_copy serves as root
19.                         If (t_copy.lexiorder < t_new.lexiorder)
20.                             has_potential = false
21.                     If (∃ node n_right on the rightmost root-leaf path, with an identical sibling n_left)
22.                         If (tree-rooted-at-n_left.lexiorder > tree-rooted-at-n_right.lexiorder)
23.                             has_potential = false
24.                     If (has_potential), append t_new to Q;      // t_new still has the potential of forming an CN
25.                     Else, discard t_new
26. Return RS
```

Fig. 10. The iterative *CNGen* algorithm

Lemma 3.5: *InitCNGen* computes the set of CN (i) correctly, (ii) completely and (iii) without duplicates. The proof follows from the above discussion.

Finally, we address the translation of CN into operator trees. Similar to *Discover*, we map CN to left-deep trees, where leaf nodes are *source* operators that perform selections, and interior operators are *UniqueJoins*. Sources are ordered left-to-right in the order of their addition during *CNGen*: the leftmost source corresponds to the node (n_{root}) of the expanded schema from which the CN was discovered; the rightmost node was added last. Figure 11 shows the operator tree for the CN in Figure 9a. Join conditions correspond to parent-child relationships in the CN tree.

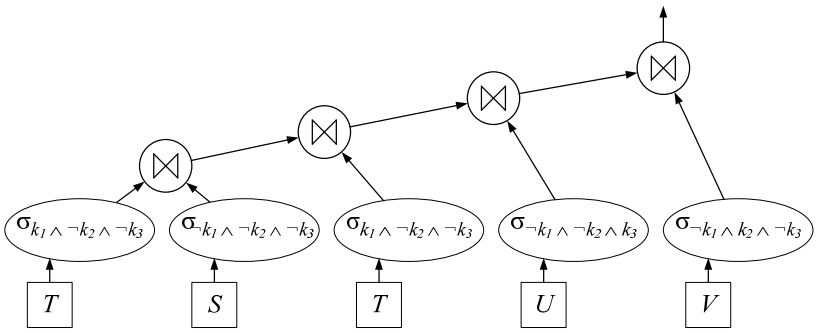


Fig. 11. Operator tree for the CN of Figure 9a

In our description of *CNGen*, we chose the lexicographic order of *rid* and *kbit* for simplicity. Since all operator trees are left-deep, it would be more efficient to arrange sources with the highest selectivity to the left. We should thus add such nodes as early as possible during *CNGen*. For instance, instead of k_1, we could choose the rarest keyword to determine n_{root}. Furthermore, without any specific knowledge about table size and distribution of keywords, it is reasonable to assume that the selectivity increases with the number of keywords; i.e., nodes with a large number of keywords should be visited first, and hence receive a small *nid*. In summary, the above algorithms adapt operator based R-KWS to our semantics of general join conditions, eliminating duplicate and invalid results.

4. RELATIONAL KEYWORD SEARCH ON STREAMS

This section focuses on continuous R-KWS queries on relational streams. Section 4.1 presents formal semantics. Sections 4.2 and 4.3 propose GB and OB query processing methodologies, further optimized later in the paper.

4.1. Semantics

We assume multiple relational data streams, whose tuples arrive in increasing t_{start} and may be deleted explicitly (through a negative tuple) or implicitly (according to the sliding window model). A *streaming relation* (SR) is the union of several streams with a common structure and meaning. For example, all cash registers in a large supermarket produce data streams in the format <product-id, price, time> that can be wrapped into a single SR. We suppose all streams to be bundled into SR, and hence use both terms interchangeably. A *streaming schema* denotes which streams can be joined and on what attributes. Nodes in this graph represent SR, and are connected by an edge, iff a common join attribute has been defined. In our examples, we assume four SR: *S*, *T*, *U* and *V*, arranged in a schema identical to Figure 1a; i.e., the only joins permitted are between tuples in *T* and those in *S*, *U*, or *V*. The schema would usually be provided by the system operator, but may also be altered by individual users (e.g., by excluding SR that are not relevant to a query). If a new data stream is instantiated, it can be integrated by either (i) merging it with an existing SR (if it adheres to the same format) or (ii) introducing a new SR.

We assume a continuous keyword query of the form $q := \{k_1, \ldots, k_m\}$, and define its semantics by identifying results on instantaneous views (snapshots) of the system. At every time instant τ, the *instantaneous data graph* $G(\tau)$ contains a node for each tuple *s* that is alive at τ. Tuples are connected by an edge, iff they can be joined. Figure 12 shows $G(\tau = 9)$ for the example schema, including lifespans of tuples. Note that, in case of positive-negative tuples, the end of a lifespan is not known in advance. In our example, we assume a query with three keywords k_1, k_2, k_3 whose appearance is denoted next to the

tuples. Results are defined using the concept of MTJNT. Similar to keyword search over static tables, we impose a limit T_{max} of tuples per MTJNT. Let $R(\tau)$ be the set of MTJNT in $G(\tau)$ that do not exceed T_{max} nodes. The result R of a continuous R-KWS query is the union of $R(\tau)$, for all τ. In Figure 12, (v_1, t_1, u_2), (v_1, t_1, s_2, t_2, u_3) and (u_1, t_1, s_2, t_2, v_2, u_3) are results in $R(\tau = 9)$. At time $\tau = 10$, v_1 expires and so do the former two MTJNT, while (u_1, t_1, s_2, t_2, v_2, u_3) continues as an element of $R(\tau = 10)$. We require results to be produced in ascending t_{start} order.

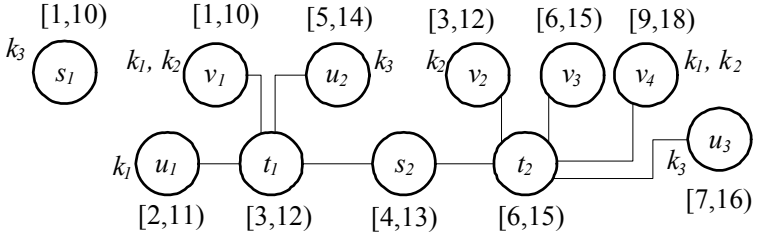


Fig. 12. Instantaneous data graph at $\tau = 9$

The following Lemma allows for an efficient generation and compact representation of R.

Lemma 4.1: Let $r \in R(\tau)$ be an MTJNT on $G(\tau)$. If every node n in r is alive at τ+1, then r is an MTJNT in $G(\tau$+1); i.e. $r \in R(\tau$+1).

Proof: Since attributes do not change values over time, any tuple that contains keyword k at time τ also contains k at time τ+1. Similarly, edges do not change: if r is connected at τ, it is also connected at τ+1. Therefore, if r is total and minimal at τ, it is also total and minimal at τ+1. □

According to this lemma, an MTJNT r is not affected by insertions or deletions of external nodes. Consequently, every MTJNT r needs to be constructed and reported only once (at $r.t_{start}$), rather than at every instant of its lifespan. The termination of a result r, depends on the stream model. Using a sliding window of duration w, we can compute the lifespan of r directly upon its creation, as: $r.t_{start} = max(n.t_{start})$ and $r.t_{end} = min(n.t_{start} + w)$, where n are the component tuples of r. For example, (v_1, t_1, u_2) in Figure 12 is first discovered at $\tau = 5$, in combination with its lifespan [5, 10). In the positive-negative model (where $n.t_{end}$ is not known in advance), r is terminated when any of its constituent tuples is deleted. At this point, the user must receive a negative result tuple $-r$. Finally, the proposed semantics also capture conventional relational tables (e.g., containing "static" information about product characteristics) by modeling them as streams of everlasting tuples (i.e., $t_{start} = 0$ and $t_{end} = \infty$).

4.2. Graph Based Processing

This approach extends the paradigm of graph based query processing to data streams. Continuous GB maintains the instantaneous data graph $G(\tau)$ in memory and retrieves MTJNT by means of graph traversal. Figure 13 depicts the general architecture, using the data of Figure 12. The two main components are: (i) a set of input buffers, one for each stream, and (ii) a data graph. The buffers store tuples that are currently alive; e.g., *T.buffer* collects tuples t_1 and t_2 from data stream T. Arrivals and expirations of tuples necessitate updates of the data graph, and trigger a search for (expired) MTJNT. A new tuple is first inserted in the appropriate buffer. A corresponding node is created in the data graph, where it must be connected to neighboring nodes. The latter are determined by joining the new tuple against the input buffers of neighboring relations.

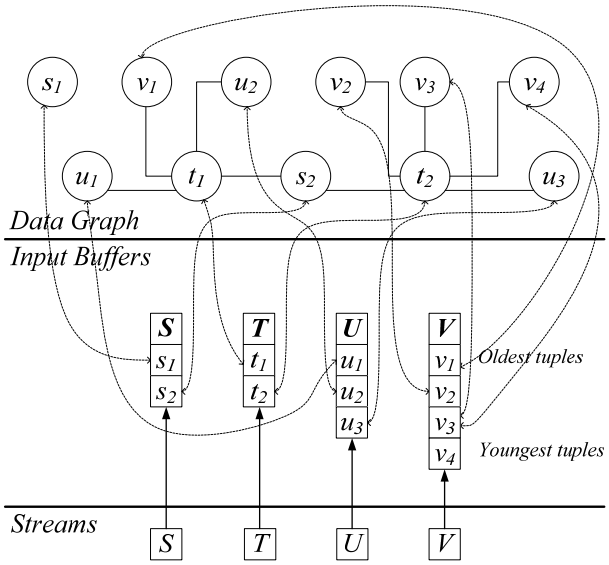


Fig. 13. A graph based architecture for continuous R-KWS

For example, t_2 is probed against *S.buffer*, *U.buffer* and *V.buffer*, thereby found to join with s_2 and v_2 (among others), and consequently connected to these nodes via edges. After each insertion, the resulting data graph is traversed for new MTJNT. Assume tuple $s \in S$, inserted at instant τ. A call to *GSearch*($G(\tau)$, q, s), retrieves the MTJNT in $G(\tau)$ that contain s. These results are returned to the user. Calling *GSearch* for every new tuple answers the continuous R-KWS query. Tuple expirations are handled according to the stream model. Given a negative tuple $-s$ at instant τ, the user must receive a negative MTJNT $-r$ for every MTJNT r that contains s. These MTJNT are identified by calling *GSearch*($G(\tau-1)$, q, s) on the current data graph. After this operation has completed, s is removed from the buffer and data graph, completing the update of $G(\tau)$. In the case of the sliding window model, there is no need to inform the user about expired MTJNT; instead it is sufficient to periodically scan the input buffers for expired tuples, and update the data graph accordingly.

Lemma 4.2: GB produces results (i) correctly, (ii) completely, (iii) without duplicates, and (iv) in the proper temporal order.

Proof: According to Lemma 3.1, results generated by *GSearch* are correct, complete and free of duplicates. The correctness GB follows directly from this property. The algorithm calls *GSearch* for *every* new tuple and hence finds *all* MTJNT. It avoids duplicates, because every MTJNT is found only once, from its youngest constituting tuple. Finally, results are generated directly upon the arrival of their youngest tuple, and hence in correct temporal order. □

The GB architecture easily adapts to changes in the schema. When a new data stream sends tuples for the first time, the system creates the corresponding buffer, and commences inserting tuples in the data graph. When a data stream is removed from the schema, the corresponding buffer is discarded. Relational tables are integrated as (static) input buffers, whose tuples straightforwardly participate in the data graph. If possible, these buffers should be stored in memory, to avoid disk access for every newly arriving tuple of neighboring streams. Section 5 optimizes GB further through keyword labeling.

4.3. Operator Based Processing

This approach uses relational stream operators to process continuous R-KWS queries. Figure 14 illustrates the system architecture. A query triggers the generation of candidate networks, subsequently translated into operator trees. These trees resemble their counterparts in conventional databases (e.g. Figure 11), receive streaming data from the bottom, and produce results (MTJNT) at the top. Join operators follow the description of Section 2.4, and additionally implement the *UniqueJoin* pattern (Section 3.2), ensuring that no basic tuple contributes more than once to a single result.

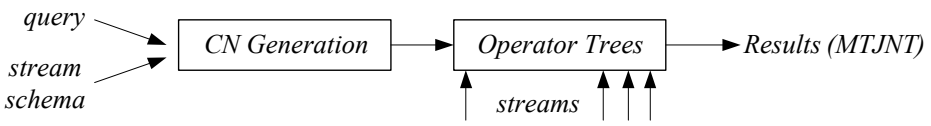


Fig. 14. Operator based query processing for continuous R-KWS

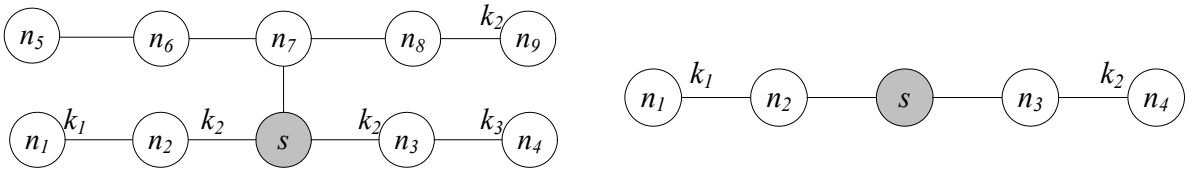
Lemma 4.3: OB produces results that are (i) correct, (ii) complete, (iii) without duplicates, and (iv) in the proper temporal order.

Proof: Correctness, completeness and the absence of duplicates follow Lemma 3.4. MTJNT are produced directly upon arrival of their youngest tuple and hence in correct temporal order. □

Operator based systems do not adapt as easily to changes in the schema as their graph based counterparts. When a new data stream connects to the system, an entire range of operator trees must be generated, before any data can be processed. Section 6.6 addresses the efficient creation of operator trees during this time-critical operation. In comparison, removing a data stream is easy. All operator trees that consume its tuples are guaranteed not to generate output, and can be removed whenever convenient. Again, we treat relational tables as streams of everlasting tuples, which are scanned once and "stream" into to operator trees at the beginning of query processing. Section 6 introduces several effective optimizations for OB.

5. OPTIMIZATIONS FOR CONTINUOUS GB

Graph based R-KWS over data streams can be greatly accelerated by restricting graph traversals. Assume $q = \{k_1, k_2, k_3\}$ and the graph of Figure 15a, where the current state (tree t_{new}) of *GSearch*(G, q, s) consists of nodes s and n_2. The algorithm visits nodes n_5 through n_9, among others, although none of these nodes contain (or lead to) the missing keywords k_1 or k_3. If *GSearch* had been informed accordingly, the algorithm could have avoided traversing this branch. Similarly, in Figure 15b, none of the tuples contains k_3. If this was known to the algorithm, graph traversal from s could be omitted altogether. Since keywords appear highly infrequently, both cases are common in practice. In this section, we introduce *keyword labeling*, a simple and effective method to summarize reachable keywords for a given node. It improves performance by avoiding unnecessary calls to *GSearch* and constraining graph traversals. For ease of presentation, we avoid stream specific notation (e.g., time-stamps), whenever possible.



(a) *GSearch* does not need to investigate n_5 - n_9 (b) *GSearch* does not need to be called at all

Fig. 15. Optimizing *GSearch*

A *keyword label* (KL) of format [k_i, h], stored at node n, indicates a path of h edges in the data graph, connecting n to an occurrence of keyword k_i. We use n:[k_i, h] to indicate that the label [k_i, h] is located at node n. There is no need to consider paths exceeding $T_{max} - 1$ edges, since MTJNT are limited to T_{max} nodes. Figure 16 depicts labels for the data of Figure 15a, assuming $T_{max} = 4$. Node s stores four KL: [k_1, 2], [k_2, 1], [k_2, 3] and [k_3, 2]. For example, s:[k_1, 2] corresponds to the path (s-n_2-n_1), connecting s to an occurrence of k_1, via two edges. We require keyword labels to be *min-complete*, i.e., each node must store a KL for every reachable keyword, indicating the *shortest* path. The labeling in Figure 16 is min-complete. Note that node s has two labels for k_2, namely [k_2, 1] and [k_2, 3], corresponding to paths of different lengths. The KL for the longer path (i.e., [k_2, 3]), is not required. On the other hand, removing the KL for the shortest path ([k_2, 1]) violates min-completeness. The benefits of a min-complete labeling during query processing are two-fold. First, *GSearch*(G, q, s) only needs to be called, if s node can reach *all* query terms, i.e., if the node stores a KL for every $k \in q$. In any other case, s is guaranteed not to participate in an MTJNT. Second, whenever a call to *GSearch* cannot be avoided, the labeling allows pruning the graph traversals. Recall that the original *GSearch* (Figure 8) uses two conditions (Lines 13 – 14) to verify whether a new tree t_{new} has the *potential* of becoming an MTJNT. Keyword labels enable stricter requirements that eliminate more trees. Since *GSearch* expands t_{new} from nodes in the rightmost root-to-leaf path, these must be able to reach the missing keywords - without exceeding a total of T_{max} nodes. Otherwise, t_{new} cannot become total, and is discarded.

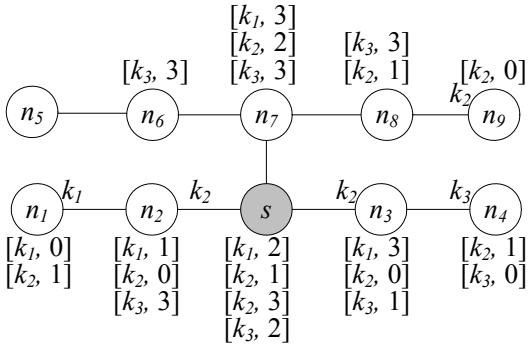
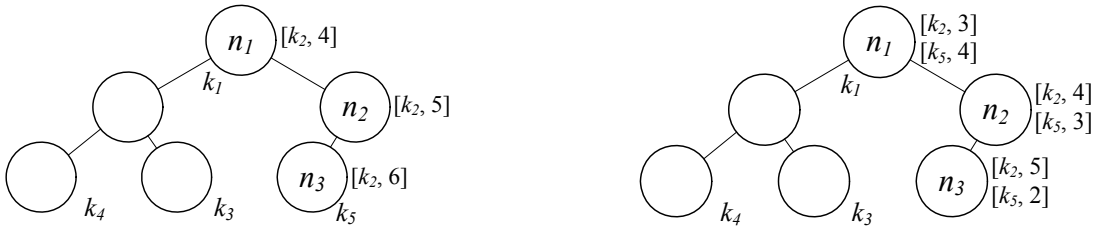


Fig. 16. A min-complete labeled graph for $T_{max} = 4$

Assume the query $q = \{k_1, \ldots, k_5\}$, and $T_{max} = 9$. The tree in Figure 17a lacks keyword k_2, and new nodes can only be added to n_1, n_2 or n_3 (for simplicity, we omit KL that are stored on the remaining nodes, or regard other keywords). Node n_1 can reach k_2 in four hops, the shortest path to any occurrence of k_2. However, the tree already contains six nodes, and would thus exceed T_{max}. It can hence be discarded directly, without further expansions. If more than one keyword is missing, we have to take into account that paths may overlap. Consider the tree in Figure 17b, lacking keywords k_2 and k_5. The KL indicate paths of three nodes to k_2 (via n_1), and two nodes to k_5 (via n_3), and the tree could become total by adding five nodes. However, this is not the minimal extension, since paths potentially overlap. The KL at n_2 indicate paths to k_2 and k_5, of four and three edges, respectively. If k_5 is situated along the path to k_2, four nodes suffice to reach both keywords. In our example, this marks the smallest number of nodes that could possibly be added. Since the tree already contains six nodes, it is bound to exceed $T_{max} = 9$ and can hence be discarded.



(a) k_2 is too distant (b) k_2 and k_5 are too distant

Fig. 17. Intermediate trees abandoned by *KL-aware GSearch* ($T_{max} = 9$)

In general, a *KL-aware GSearch* only inserts t_{new} into Q, iff there exists a set NL of labels, situated at nodes on the rightmost root-to-leaf path of t_{new}, that meets the following criteria:

(i) The KL in NL can reach all missing keywords; i.e., $NL.keywords \cup t_{new}.keywords = q$.

(ii) $|t_{new}| + \Sigma_n (max(h \mid n : [k, h] \in NL) \leq T_{max}$.

The second criterion takes the potential overlap between paths into account, by considering the longest path leaving a node n, rather than their sum. Assuming $T_{max} = 10$ in Figure 17b, the set $NL = \{n_1{:}[k_2, 3], n_1{:}[k_5{:}4]\}$ satisfies both conditions. Specifically, (i) the KL in NL contain all missing keywords (k_2 & k_5), and (ii) it is possible for the only concerned node n_1 to reach both keywords in four edges, while $|t_{new}| = 6$; hence, $|t_{new}| + 4 \leq T_{max}$. The following lemma states the correctness of the resulting *KL-aware GSearch*:

Lemma 5.1: *KL-aware GSearch* yields the same results as *GSearch*.

Proof: *KL-aware GSearch* does not produce results that are not generated by the original algorithm because the keyword labels only *filter* unpromising trees. On the other hand, *KL-aware GSearch* does not miss any correct result, because min-completeness ensures that discarded trees cannot reach the remaining keywords within T_{max} nodes. □

In order to remain min-complete, keyword labels must be updated whenever tuples arrive or expire. In the following, we present two implementations. The first is applicable to both stream models, sliding-windows as well as explicit deletions; the second is more efficient, but restricted to sliding-windows.

5.1. Predecessor-KL

A *predecessor-KL* is a triplet of the form [k, h, p]. Stored at node n, it indicates a path of length h, connecting n to an occurrence of keyword k. In this path, p is n's predecessor. We write $n{:}[k, 0, id]$, if n already contains term k. Figure 18 depicts a graph with occurrences of keywords k_1 and k_2. Node n_4 reaches k_2 via the path n_2-n_3-n_4, and stores [k_2, 2, n_3], indicating that its predecessor in this path is n_3. The same path connects k_1 to n_4, and the node also stores [k_1, 2, n_3]. The fact that n_4 can also reach k_1 in its neighbor n_1 is indicated by $n_4{:}[k_1, 1, n_1]$. To guarantee min-completeness, we maintain the following invariant: every node n must contain a predecessor-KL [k, h, p], for the *shortest path* leading from n through p to the occurrence of k. For example, in Figure 18, n_4 must keep both KL [k_1, 2, n_3] and [k_1, 1, n_1], since they represent the shortest path via predecessors n_3 and n_1 respectively. Node n_6 reaches k_1 contained in n_2 and n_1 through three and two steps, respectively. However, since both paths (to n_2 and n_1) share the same predecessor n_4, it suffices to keep only [k_1, 2, n_4]. Assuming $T_{max} = 4$, each path contains at most $T_{max} - 1 = 3$ edges, and there is no need to store $n_7{:}[k_2, 4, n_5]$. This invariant yields a min-complete labeling, and allows a correct *KL-aware GSearch*.

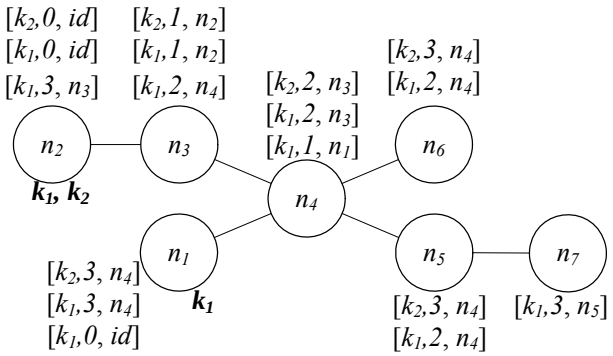


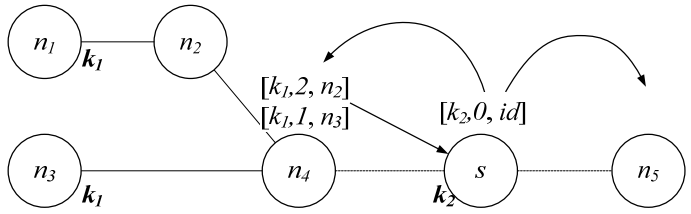
Fig. 18. A min-complete labeling with Predecessor-KL

An arriving tuple *s* can itself contain a keyword, or create new paths between keywords and nodes. Both cases require KL insertions and updates. Changes are not limited to *s* and its immediate neighbors, but are propagated in a radius of $T_{max} - 1$. Figure 19 describes the underlying algorithms *InitSendKL* and *SendKL*. The former initializes labels at node *s*; the latter propagates changes. *InitSendKL* works in two stages. First, it constructs KL at *s*, for all keywords contained in the node (Lines 2 and 3), and transmits this information to all neighboring nodes (Lines 4 and 5), using *SendKL*. Second, it propagates older KL from neighboring nodes to *s* (Lines 6 and 7), again by using *SendKL*. Consider inserting *s* in the graph of Figure 20a. First, *s* receives label [k_2, 0, *id*], since it contains k_2. This label is then passed to the neighboring nodes n_4 and n_5. Second, older labels [k_1, 1, n_3] and [k_1, 2, n_2] from n_4 are sent to *s*, and propagated further. As discussed below, *Send-KL* is recursive, and pushes labels deep into the graph. For example, the new KL [k_2, 0, *id*] is passed from *s* to n_4, and from there to n_3, n_2 and finally n_1.

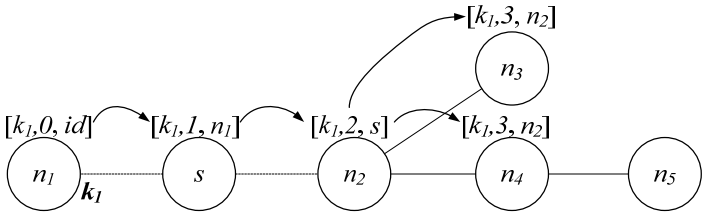
InitSendKL (Node *s*) // Adjusts Predecessor-KL to the arrival of a new node *s*
// *s*: the newly inserted node
1. Insert *s* in the data graph
2. For every keyword *k*, occurring in *s* // Create KL at new node
3. Add [*k*, 0, *s*] to *s.KLlist*
4. For every neighbor *n* of *s* in the data graph // Pass KL to neighbors – and further
5. *SendKL*(*s*, *n*, [*k*, 1, *s*])
6. For every neighbor *n* of *s* in the data graph // Pass KL from neighbors to new node – and further
7. For every KL [*k*, *h*, *pr*] ∈ *n.KLlist* where *pr* ≠ *s*
8. *SendKL*(*n*, *s*, [*k*, *h*+1, *s*])

SendKL (Node *pn*, Node *rn*, KL *newKL*) // Propagates a KL to a neighboring node
// *pn*: the node passing *newKL*
// *rn*: the neighboring node receiving *newKL*
// Assume that *newKL* = [*k*, *h*, *pn*]
1. If $h > T_{max} - 1$, Return
2. If (∃ [*k*, *h'*, *pn*] ∈ *rn.KLlist* | *h'* ≤ *h*)
3. Return // an existing KL indicates a shorter route from *rn* through *pn*
// Lines 4-6: store the new KL, and propagate it to neighboring nodes
4. Add *newKL* to *rn.KLlist*
5. For each neighbor *nn* of *rn* | *nn* ≠ *pn* // do not send the KL back to the sender
6. *SendKL*(*rn*, *nn*, [*k*, *h* + 1, *rn*])

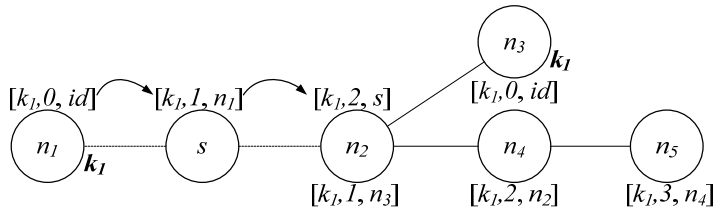
Fig. 19. *SendKL* and *InitSendKL*



(a) Insertion of node *s*



(b) KL are propagated as long as $h \leq T_{max} - 1 = 3$



(c) KL are propagated as long as they announce shorter routes

Fig. 20. Examples of *InitSendKL* and *SendKL* ($T_{max} = 4$)

SendKL propagates KL in a store-and-forward fashion. A node receiving a KL (i) stores this locally (Line 4), and (ii) forwards it to all neighbors, except for the transmitter (Lines 5 and 6). In Figure 20b, the new node *s* receives a KL from its left neighbor n_1 and forwards it to n_2. This node stores the KL (as [k_1, 2, *s*]) and propagates it to n_3 and n_4. There are two terminating conditions. First, a label is only propagated as long as its hop-count h does not exceed $T_{max} - 1$ (Line 1). For instance, assuming $T_{max} = 4$, the KL in Figure 20b is sent to n_4, but not to n_5. Second, KL are propagated only as long as they announce shorter paths (Lines 2 and 3). Consider Figure 20c. In contrast to Figure 20b, a second tuple (n_3) already contains k_1. The KL from n_1 is forwarded to *s* and further to n_2. Because n_2 can already reach k_1 via a shorter path (n_2-n_3), there is no need to send any new label to n_3 and n_4.

Similarly, expiring tuples can remove keyword occurrences, and destroy paths that connect nodes with keywords. Both cases lead to KL updates and deletions. Figure 21 describes the functionality of *InitRevokeKL* and *RevokeKL*. The former algorithm removes a deleted tuple *s*, and calls the latter to propagate this news (within a radius of T_{max}). Using *RevokeKL*, a node *pn* instructs its neighbor *rn* that its shortest path to keyword *k* ceased to exist, and that the new (longer) path from *rn* to *k* via *pn* takes *new-dist* hops. If *pn* cannot reach *k* anymore at all, it sets *new-dist* = ∞. Upon receiving such a message, node *rn* replaces its KL [*k*, *old-dist*, *pn*] with [*k*, *new-dist*, *pn*] (Lines 1, 4 & 5) and informs its neighbors (Lines 6 – 13). Consider the disappearance of *s* in Figure 22a. The node notifies its neighbor n_2, by calling *RevokeKL*(*s, n_2, k_1,* ∞). This node removes the old KL n_2:[k_1, *2*, *s*], and in turn informs its neighbors (Figure 22b), e.g. by calling *RevokeKL*(n_2, n_3, k_1, *4*). The receiver (e.g., n_3) then replaces its label [k_1, *3*, n_2] with [k_1, *4*, n_2], and continues propagating the message.

InitRevokeKL (Node *s*) // Adjusts KL-labeling to the expiration of node *s*

1. Remove *s* from the data graph
2. For all former neighbors *n* of *s*
3. For all keywords *k*
4. *RevokeKL*(*n, k, s,* ∞)

RevokeKL (Node *pn*, Node *rn*, Keyword *k,* int *new-dist*) // informs of increased path length
// *k*: keyword of concern
// *pn*: node whose distance to *k* has increased
// *new-dist*: the new distance between *pn* and *k*
// *rn*: a neighbor of *pn*, receiving the message

1. Remove [*k*, *old-dist*, *pn*] from *rn.KLlist*
2. KL $kl_1 = [k, h_1, n_1] \in rn.KLlist$, such that h_1 is minimal. // kl_1 indicates the shortest path to *k*
3. KL $kl_2 = [k, h_2, n_2] \in rn.KLlist \setminus kl_1$, such that h_2 is minimal. // kl_2 indicates the second shortest path to *k*
4. If ($new\text{-}dist \leq T_{max} - 1$)
5. Add [*k*, *new-dist*, *pn*] to *rn.KLlist*

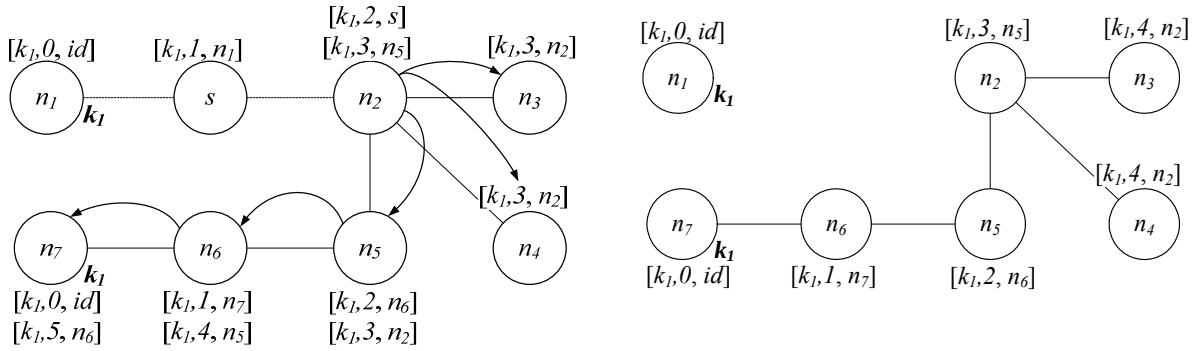
// Case 1: if the best path to *k* used to lead through *pn*; all neighbors are informed

6. If ($old\text{-}dist < h_1$)
7. For all neighbors *nn* of *rn*| $nn \neq pn$
8. $bdist = \min (h_{best} \mid [k, h_{best}, n_{best}] \in rn.KLlist \mid n_{best} \neq nn)$ or ∞ if no such KL exists
9. *RevokeKL*(*nn, rn, k, bdist* + 1) // assuming ∞ + 1 = ∞
10. Else

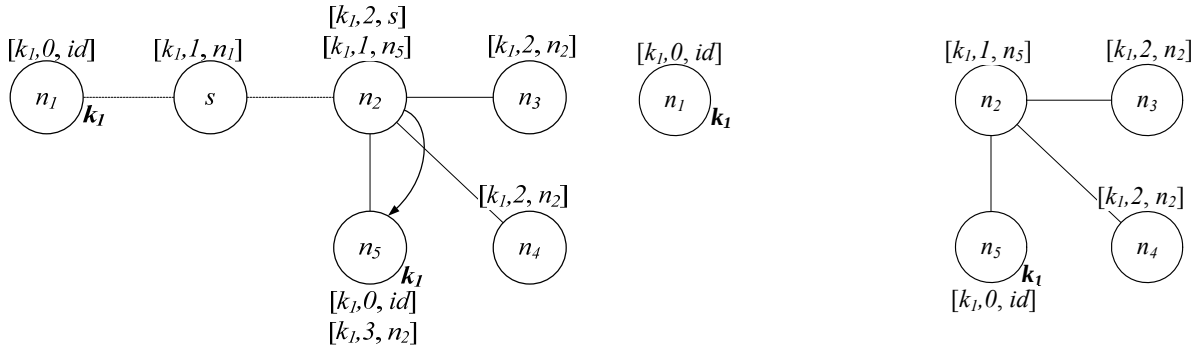
// Case 2: If the second best path to *k* used to lead through *pn*, only node n_1 supplying the shortest route is informed

11. If ($old\text{-}dist < h_2$)
12. $bdist = \min (h_{best} \mid [k, h_{best}, n_{nest}] \in rn.KLlist \mid n_{best} \neq n_1)$ or ∞ if no such KL exists
13. *RevokeKL*(n_1, *rn, k, bdist* + 1)

Fig. 21. *RevokeKL* and *InitRevokeKL*



(a) The shortest path from n_2 to k_1 expires (b) Node n_2 informs all neighbors

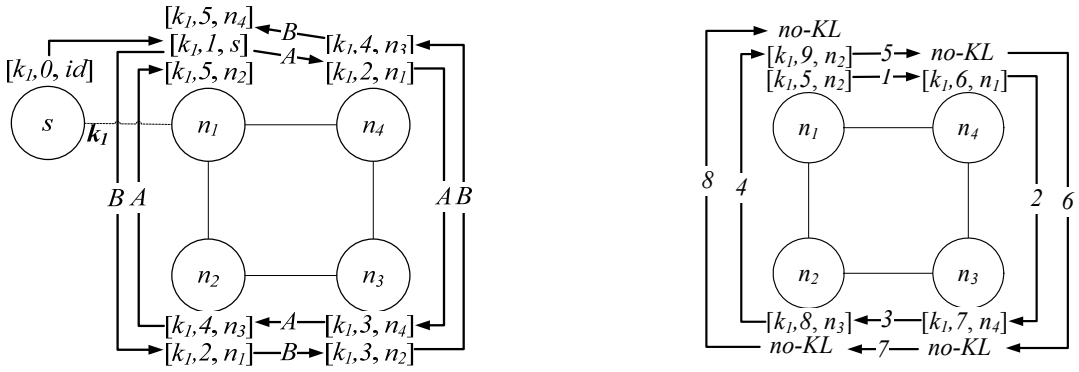


(c) The second shortest path from n_2 to k_1 expires (d) Node n_2 informs only n_5

Fig. 22. *RemoveKL* and *InitRemoveKL*

When a node *rn* receives a call of *RevokeKL*, there are three cases. In the first, the expiring path (via *pn*) has been *the shortest* path between *rn* and keyword *k* (Lines 6-9). When this happens, *all* neighbors of *rn* must be notified. Consider node n_2 in Figure 22a, whose shortest route to k_1 leads through node *s*. If n_2 receives *RevokeKL*(s, n_2, k_1, ∞), it informs all neighbors n_3, n_4, and n_5. These nodes update their KL accordingly (Figure 22b) and subsequently notify their own neighbors. In the second case, the expiring path (via *sn*) has been *the second shortest* path between *rn* and *k* (Lines 7-11). Most neighbors of *rn* do not need to update their KL, because these indicate already the *shortest* path via *rn*. The only node that requires an update is the neighbor of *rn* that supplies the shortest path (Lines 11-13). Assume the expiration of *s* in Figure 22c. When n_2 receives *RevokeKL*(s, k_1, n_2, ∞), it updates its KL but does not inform n_3 and n_4, because its shortest path to k_1 (via n_5) did not change. The only label that has to be updated (by calling *RevokeKL*(n_2, k_1, n_5, ∞)) resides at the node n_5, which provides the shortest path between n_2 and k_1. Figure 22d depicts the resulting keyword labels. In the last case, where the expiring path (via *pn*) is neither the shortest nor the second shortest path between *rn* and *k, RevokeKL* terminates directly after updating the KL at *rn*, and does not inform neighboring nodes.

In applications involving graphs, updates along cycles commonly pose a challenge. Even though *RevokeKL* does not address cycles explicitly, it handles them effectively. In Figure 23a, tuple *s* contains keyword k_1, and nodes n_1 through n_4 form a circle. Accordingly, KL have been propagated in two directions: clockwise (*A*) and counterclockwise (*B*). Now, assume that *s* expires. This is the only occurrence of k_1, and all KL must be removed. Figure 23b investigates the updates on the clockwise-oriented KL. Upon expiration, *s* calls *RevokeKL* on n_1. This node removes [k_1, 1, *s*], and then informs its neighbors. Due to n_1:[k_1, 5, n_2], it assumes that it is able to reach k_1 via n_2 and hence updates n_4 by calling *RevokeKL*(n_4, n_1, k_1, *6*). This node replaces [k_1, 2, n_1] with [k_1, 6, n_1] and informs n_5. Update operations thus complete a circle, return to node n_1, and set out on a second tour. So far, only the hop-count *h* has increased. Yet, this process stops when *h* reaches T_{max}–1. Assume T_{max} = 10. The fourth call to *RevokeKL* causes n_1:[k_1, 5, n_2] to be replaced with n_1:[k_1, 9, n_2], and triggers updates on n_4, by calling *RevokeKL*(n_4, n_1, k_1, *10*). This node removes n_4:[k_1, 6, n_1], does *not* create a new KL because $h = T_{max}$, and calls *RevokeKL*(n_3, k_1, n_4, *11*). The KL at n_3 is removed accordingly and the update request is propagated further. After the circle has been completed, all KL have been removed correctly. In practice, T_{max} is small, and *RevokeKL* converges rapidly.



(a) KL on a data graph containing a cycle (b) KL are removed when $h = T_{max}$

Fig. 23. Revoking KL on cycles (T_{max} = 10)

5.2. Time-KL

For sliding windows, we propose a more efficient labeling that does not require explicit removal. Specifically, a *Time-KL* is a triplet [k, h, t_{end}] indicating a path of length h to an occurrence of keyword k, which exists until t_{end} (when its earliest tuple expires). Figure 24 depicts a data graph with Time-KL. Node n_4 is connected to k_1 in n_7 via two hops. Among the path's nodes, n_7 expires at 74, n_6 at 42, and n_4 at 38. The latter marks the minimum, and n_4 stores [k_1, 2, 38]. Similarly, there is a path of length one from n_4 to k_1 (in n_5). This node expires at time 15, and n_4 stores the KL [k_1, 1, 15]. A third path connects n_4 to k_1 (at n_3). The path has a length of three, and the earliest node (n_2) expires at 21. Recall that the min-completeness property requires the labels to indicate the shortest path between a node and a keyword *at any time*. Both KL [k_1, 2, 38] and [k_1, 1, 15] must be stored, since each indicates the shortest path for some period of time. On the other hand, the third path does not need to be recorded, because it is longer and expires sooner than the other two. We say that KL [k, h_1, t_{end-1}] *dominates* another [k, h_2, t_{end-2}], iff $h_1 \leq h_2$ and $t_{end-1} \geq t_{end-2}$. The min-complete property is met, iff the graph contains all KL that are not dominated by others (i.e., those in the *skyline*). The labeling in Figure 24 satisfies this invariant.

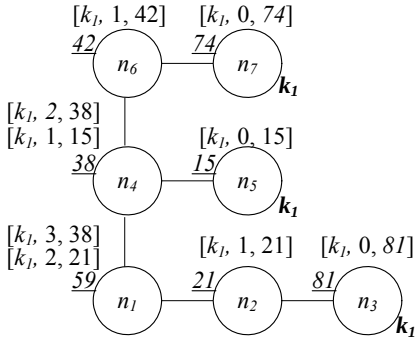


Fig. 24. KL with timestamps

A new tuple s invokes new time-KL, thereafter propagated in a store-and-forward fashion. Figure 25 shows the pseudo-code for *InitSentTimeKL* and *SendTimeKL*. The former algorithm (i) creates KL for keywords in the new node s, thereafter broadcasted through the data graph (Lines 2-5), and (ii) propagates KL from neighboring nodes to s and onward (Lines 6-8). The latter algorithm transfers a label *newKL* from one node *pn* to a neighbor *rn*. There are two terminating conditions. First, if the hop-count h exceeds T_{max}, *newKL* is disregarded. Second, if *newKL* is dominated by an older label at *rn*, it is also ignored (Lines 2-3). In any other case, *newKL* indicates the shortest path from *rn* to k, at least for some period, and is stored (Line 8) and propagated further to neighboring nodes (Lines 9-10). Older labels that are now dominated by *newKL*, become obsolete, and are removed (Lines 5-7). Outdated Time-KL are ignored by *GSearch* and removed whenever convenient, e.g., when their node expires.

6. OPTIMIZATIONS FOR CONTINUOUS OB

As illustrated in Sections 2.2 and 3.2, an R-KWS query on static tables commonly involves an immense number of operator trees. For continuous queries over data streams, this obstacle becomes even more pronounced. If a selection on a table (e.g. $T\{k_1\}$) returns no tuples, all operator trees using this input (e.g. Figure 11) can be discarded immediately. For data streams, this is not permissible. Even though the selection $T\{k_1\}$ does not currently produce tuples, it may do so in the future, and all operator trees must thus be maintained. This section proposes optimizations that enable efficient OB R-KWS over data

```
InitSendTimeKL (Node s)                                  // Updates TimeKL for the arrival of a new node s
1.  Insert s in the data graph
2.  For every keyword k, occurring in s                  // Create KL at new node
3.      Add [k, 0, s.t_end] to s.KLlist
4.      For every neighbor n of s in the data graph      // Pass KL to neighbors – and further
5.          SendTimeKL(s, n [k, 1, s.t_end]
6.  For every neighbor n of s in the data graph          // Pass KL from neighbors to new node – and further
7.      For every KL [k, h, t_end] ∈ n.KLlist
8.          SendTimeKL(n, s, [k, h + 1, t_end])

SendTimeKL (Node pn, Node rn, KL newKL)                  // Propagates a TimeKL to a neighboring node
// pn: the node passing the KL
// rn: the neighboring node receiving a KL
// newKL = [k, h, t_end] the keyword label
1.  If (h ≥ T_max − 1), Return
2.  If (∃ [k, h_old, t_end-old] ∈ rn.KLlist | h_old ≤ h & t_end-old ≥ t_end)   // if there exists a dominating entry, abort
3.      Return
4.  For all [k, h_old, t_end-old] ∈ rn.KLlist             // remove dominated KL
5.      If (h_old ≥ h) AND (t_end-old ≤ t_end)
6.          Remove [k, h_old, t_end-old] from rn.KLlist
7.  Add [k, h, min(newKL.t_end, rn.t_end)] to rn.KLlist  // Add the new KL
8.  For each neighbor nn of rn                           // Propagate to neighbors
9.      SendTimeKL(rn, nn, [k, h + 1, min(newKLt_end, rn.t_end)]
```

Fig. 25. *SendTimeKL* and *InitSendTimeKL*

streams. In particular, Section 6.1 integrates individual operator trees into a single mesh, sharing common sub-expressions. A first approach creates a *Full-Mesh* (FM) during pre-processing, allowing to dedicate runtime resources exclusively to tuple processing. Sections 6.2 and 6.3 describe two optimizations for FM, namely *demand-driven operator execution* and *temporary operator disconnection*. Section 6.4 presents a *Partial-Mesh* (PM) that does not require pre-processing, but grows and shrinks dynamically. Section 6.5 addresses the purging of dead tuples. Finally, Section 6.6 addresses changes in the schema.

6.1. Operator Mesh

We integrate all operator trees into an *operator mesh,* reducing CPU cost (for evaluating joins) as well as memory overhead (for intermediate results). The mesh has $|SR|\cdot 2^{|K-1|}$ clusters, where $|SR|$ is the number of streaming relations and $|K|$ the number of query keywords. Each cluster contains the operator trees for all CN discovered from a certain n_{root}. The trees in a cluster overlap on their left, because they include at least the same n_{root}, but usually they share larger parts. The entire operator mesh has $|SR|\cdot 2^{|K|}$ leafs/sources, one for each node of the extended schema. The maximum depth of the mesh is $T_{max} + 1$. The number of edges depends on the schema complexity. Output is produced at all levels, since operator trees vary in height. Different clusters are interconnected only through their source operators; joins from different clusters do not connect directly. In addition, we introduce a central *output operator* that collects results from all topmost operators (those producing MTJNT). Figure 26 shows the shared execution of four operator trees. Their corresponding CN all have been created by *CNGen* for $n_{root} = S\{k_1\}$. The join j_1 ($S\{k_1\} \bowtie T\{\}$) is shared by ($S\{k_1\}$, $T\{\}$, $V\{k_2\}$, $T\{k_1\}$, $U\{\ k_3\}$), ($S\{k_1\}$, $T\{\}$, $U\{k_2, k_3\}$) and ($S\{k_1\}$, $T\{\}$, $V\{k_2,k_3\}$). Note that the figure depicts only a small subset of the particular cluster.

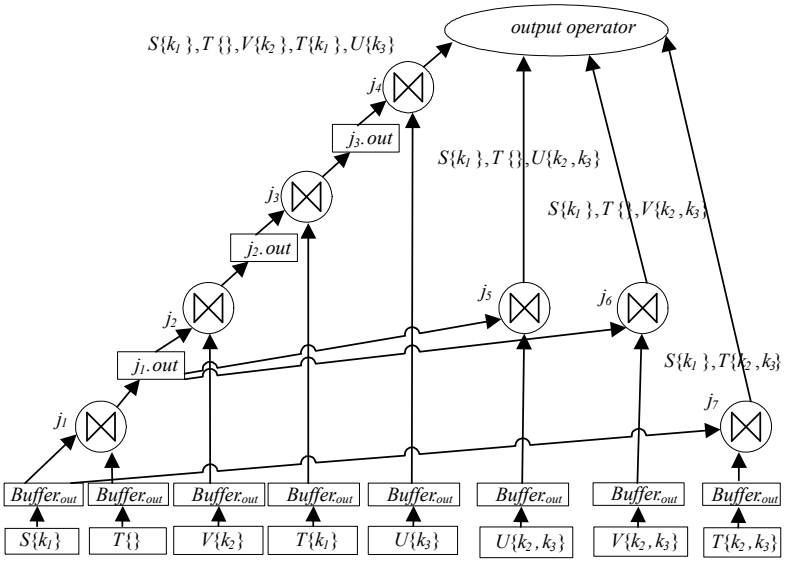


Fig. 26. Meshed trees for four CN in the same cluster

Mesh creation is performed in parallel to CN generation. Specifically, the first node in a cluster corresponds to the root node n_{root}, from which *CNGen* starts. Whenever the algorithm generates a new tree t_{new} from t_{old} (by adding a new child n_{new} to a parent n_{old}), a join $t_{new}.op$ is added to the mesh. The left child of $t_{new}.op$ is $t_{old}.op$ (the operator that was inserted when t_{old} was created), and the right child is the source of n_{new}. For each tree t in *CNGen*, we maintain a pointer to the corresponding operator $t.op$, in order to decide *where* to place subsequent joins, when t is expanded. The algorithm is initialized with $t_{first}.op$ pointing to the source of n_{root}. For instance, the mesh of Figure 26 at first contains only $S\{k_1\}$. When *CNGen* visits $T\{\}$, we add j_1 and connect it to $S\{k_1\}$ on its left and $T\{\}$ on its right. Subsequent insertions of $V\{k_2\}$, $T\{k_1\}$, $U\{\ k_3\}$ in the CN cause the addition of j_2, j_3 and j_4. Similarly, when at a later point *CNGen* inserts $U\{k_2, k_3\}$ to the tree containing $S\{k_1\}$ and $T\{\}$, j_5 is added to the mesh and connected to j_1 (representing $S\{k_1\} \bowtie T\{\}$) and source $U\{k_2, k_3\}$.

We further compact the mesh by sharing buffers. In traditional DSMS, a join operator j has two individual input buffers, $j.left\text{-}buffer$ and $j.right\text{-}buffer$. In our system, these buffers are replaced by the output buffers of the child operators, e.g., in Figure 26, $j_1.out$ replaces $j_5.left\text{-}buffer$ and $j_6.left\text{-}buffer$. Because a single operator may have thousands of parents, this concept of *state sharing* greatly reduces memory consumption. Note that tuples in the buffers are naturally ordered by t_{start} (the instant at which they were produced). More complex indexing schemes are not required, since buffers in the mesh commonly contain few tuples, if any. R-KWS meshes are larger and more densely connected than any other operator graph for relational data. However, they also show beneficial characteristics. In particular: (i) they have a distinct structure, i.e., clustered left-deep trees, and (ii) their join and selection operators are rather selective. In the following, we exploit these characteristics for further optimization.

6.2. Demand-Driven Operator Execution

Our first solution generates a *full mesh* (FM) of operators, prior to the actual query processing. This mesh is maintained in main memory throughout the lifespan of the query. It allows *demand-driven operator execution*, an inter-operator messaging system that eliminates ineffective join operations. We observe that join operators commonly execute without any prospect of contributing to an actual result, because joins at

higher levels lack input from their right child. In Figure 26, assume tuples from $S\{k_1\}$ and $T\{\}$, while $V\{k_2\}$, $U\{k_2, k_3\}$ and $V\{k_2, k_3\}$ are empty. None of the joins j_2, j_5, or j_6 requires the output of j_1 because they do not receive right input. In the worst case, j_1's results expire before the arrival of any tuples from $V\{k_2\}$, $U\{k_2, k_3\}$ or $V\{k_2, k_3\}$. The join has wasted CPU and memory, without any contribution to the query. Even if $V\{k_2\}$ had tuples available and j_2 consumed input from j_1, the execution of both operators could still be wasted, e.g., if j_4 happens to lack right input.

Under demand-driven operator execution, a join is considered to be either *running* or *sleeping*. Running operators process input; sleeping ones ignore it. A join operator is sent to sleep, if (i) it has no input from the right child (a source), or (ii) *all* its parents are sleeping. Sending operators to sleep does not affect the result's correctness or completeness because either (i) the operator cannot produce output, or (ii) its output would not be consumed. Figure 27 shows the state diagram for a join operator. States are characterized by two binary flags: *d* indicating that at least one parent operator is running, and *r* specifying that the operator's right input is not empty. An operator only runs in the topmost state, (*d* / *r*). When it leaves this state (Transition 2 or 3) it *goes to sleep* (or *halts*), to *wake up* (or *restart*) later (Transitions 9 and 10). Operators exchange messages regarding their state, in order to ensure that all *d* and *r* flags are up-to-date. Particularly, a join operator communicates changes (running/sleeping) to its left child that adjusts its *d* flag accordingly. Likewise, sources inform their parents (i.e., joins for which they constitute the *right child*), whenever their buffer runs empty, or when a new tuple arrives to a previously empty buffer, so that these joins maintain correct *r* flags.

Assume the operators in Figure 26, where all sources produce tuples, and consequently all join operators are running. When $U\{k_2, k_3\}$ dries up, it informs its parent j_5, which turns off its *r* flag, goes to sleep (Transition 2), and informs its left child (j_1), by calling j_1.*Pstop*. Upon receiving this notification, j_1 decreases its counter of running parents (Transition 1), but takes no further action, because it still has other running parents (j_2 and j_6). When $V\{k_2, k_3\}$ stops producing output, j_6 halts, and j_1 is left with a single running parent (j_2). If now $T\{k_1\}$ also dries up, j_3 adjusts its *r* flag, goes to sleep and informs j_2. This operator decreases its counter (*rParents* = 0), halts (Transition 3), and calls j_1.*Pstop*. This join also finds all its parents sleeping, and likewise halts.

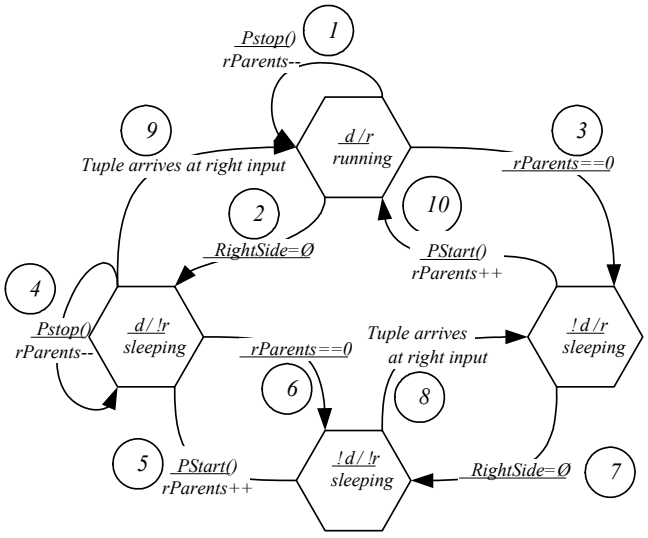


Fig. 27. States and transitions for join operators

Before going to sleep, an operator sets a local timestamp *stopTime* = *now*. When it later wakes up, it processes all tuples from its left and right input that are (i) alive and (ii) arrived after *stopTime*. To ensure the correct temporal order of results and to avoid duplicates, tuples are processed according to increasing order of t_{start}, and joined against those of the opposite input that have a smaller t_{start}. Before processing tuples, the newly awaking join has to ensure that its left input buffer is up-to-date. After all, the left child may also be sleeping, causing its output buffer to be incomplete. Thus, the operator calls *leftChild*.*Pstart*, asking its left input to wake up and update its output buffer.

Continuing the example of Figure 26, consider that the only sources with output are $S\{k_1\}$, $T\{\}$, $V\{k_2\}$, $U\{k_3\}$ and $T\{k_2, k_3\}$, and the only running join operators are j_4 and j_7. (The output operator is always running.) Join j_4 does not generate results, due to lack of left input (j_3 is sleeping). When $T\{k_1\}$ begins producing output, it causes j_3 to adjust its *r* flag, wake up (Transition 9), and call j_2.*Pstart*. This operator (j_2) restarts and informs j_1. Consequently, all joins except j_5 and j_6, are running again. This concludes our discussion of *demand-driven operator execution.* Note that this method is not restricted to keyword search; it can equally benefit other data stream applications.

6.3. Temporary Operator Disconnection

While the previous optimization targets individual join operations, a significant portion of computational expenses is inherent to the structural properties and complexity of the operator mesh. Consider the selection $T\{\}$ in Figure 26 that supplies j_1 with right input. For every tuple $t \in T\{\}$, the system (i) checks if j_1 has demand, and (ii) verifies whether j_1.*left-buffer* contains any tuples. If both conditions are met, it (iii) compares *t* with all tuples in j_1.*left-buffer* to identify join partners. Compared to the first two steps, the latter only takes place rarely, since (i) demand driven operator execution is highly effective, and the majority of operators are dormant, and (ii) due to high join selectivity and keyword scarceness most operators on higher levels have empty left input buffers. The actual comparison of join attributes (iii) is thus performed too infrequently to incur significant cost. In contrast, steps (i) and (ii) have a substantial impact. At first sight, both operations only involve a simple lookup. Yet, their tremendous cost is due to the immense frequency at which they are executed. First, for a single tuple *t*, they are performed on several thousands of join operators that have $T\{\}$ as their right input (the number of joins that have $T\{\}$ as left input is negligible). Second, this large number of lookups has to be performed *for every arriving tuple*.

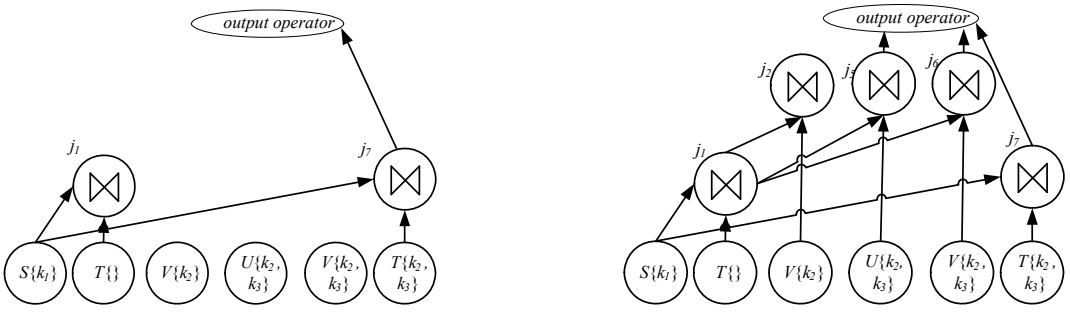
We minimize this effect through *temporary operator disconnection*. Whenever a join operator lacks demand or left input, it temporarily disconnects from its right child. In Figure 26, the operator j_1 disconnects from $T\{\}$, whenever $S\{k_1\}$ dries up, or when all parents (j_2, j_3, j_4) cease to requests output. None of the tuples arriving later at $T\{\}$ causes access to j_1 or j_1.*left-buffer*, and the CPU cycles for steps (i) and (ii) are saved. If a disconnected join operator (e.g., j_1) encounters both demand and left input, it reconnects to its right child (e.g., $T\{\}$). The temporary disconnection does not compromise the completeness of results, since during this period, new tuples from the right child are guaranteed not to find join partners. Yet, the simple measure significantly reduces CPU consumption.

As shown in the experimental evaluation, FM combined with demand driven operator execution and the temporary operator disconnection is highly efficient. However, data processing has to be delayed until the mesh is complete. For certain applications, this is not acceptable. Furthermore, the size of the mesh can exceed the available main memory, especially if multiple queries are active in parallel. Our next approach avoids initialization and reduces memory consumption by adapting the mesh dynamically.

6.4. Partial-Mesh

A *Partial-Mesh* (PM) is built at runtime and breaks the distinction between operator initialization and tuple processing. The method maintains relatively few *active* operators in memory, i.e., only those with input. Specifically, it is each operator's responsibility to create its parents before it can produce output. Conversely, it destroys its parents (and other operators up the tree), if it cannot supply them with input. Especially in large meshes, operators are commonly idle; some never execute throughout the query lifespan. Their absence does not affect result's completeness, but dramatically reduces memory consumption. In the following we describe how to grow and shrink the operator mesh.

Initially, the partial mesh contains only the $|SR|\cdot 2^{|K|}$ sources. Join operators are created later, as tuples travel upwards. For our left-deep operator trees, we demand that a join must be part of the mesh, iff it has left input. Recall that the operator mesh is composed of $|SR|\cdot 2^{|K-1|}$ clusters, one for each source containing k_1. Figure 28 illustrates the generation of a part of the cluster in Figure 26. When the leftmost source $S\{k_1\}$ first produces output, it creates its direct parents j_1 and j_7, along with others that are not depicted (Figure 28a). Joins that produce MTJNT (e.g., j_7) connect to the (permanent) output operator. In contrast, when j_1 generates results, it creates its own parents, e.g., j_2, j_5 and j_6 (Figure 28b). These directly process their input; e.g. when j_1 outputs a first tuple t, and instantiates j_2, this operator immediately probes t against $T\{\}$. Again, first results from j_2 trigger the addition of new join operators.



(a) $S\{k_1\}$ creates j_1 and j_7 (b) j_1's output creates j_2, j_5 and j_6

Fig. 28. Growing a cluster of operators from $S\{k_1\}$

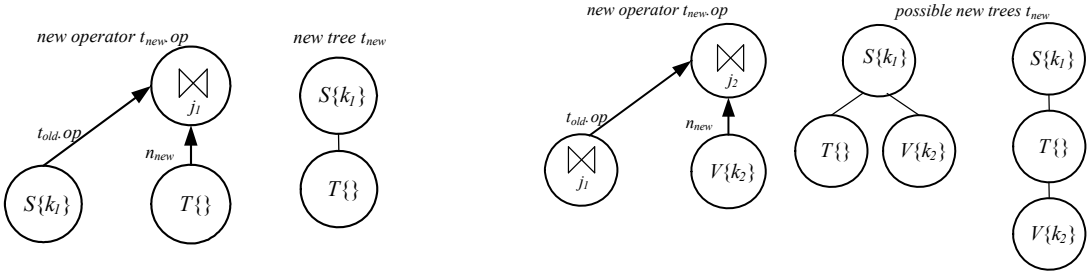
It remains to show how an operator at an arbitrary level in the mesh determines its direct parents. Recall from Section 3.2 and 6.1 that whenever *CNGen* creates a new tree t_{new} (by adding a node n_{new} to a previous tree t_{old}), a join $t_{new}.op$ is inserted into the operator mesh. The left input of $t_{new}.op$ is $t_{old}.op$ and the right one is the source of n_{new}. In PM the problem is reversed: we have an operator $t_{new}.op$, but we need the corresponding tree t_{new} in order to decide *which* parents to create. Figure 29 illustrates *TreeGen*, an algorithm for reconstructing a tree t_{new}, given its last added operator $t_{new}.op$.

TreeGen (*Operator* $t_{new}.op$)

1. if $t_{new}.op$ is a selection
2. Tree t_{new} = a tree with a single node n_{root}
3. else // $t_{new}.op$ is a join
4. Tree t_{old} = *TreeGen*(*left child of* $t_{new}.op$)
5. Let n_{new} be the node corresponding to the right child of $t_{new}.op$
6. Let n_{old} be the node joined with n_{new} in $t_{new}.op$
7. Tree t_{new} = add n_{new} as the rightmost child of n_{old} in t_{old}
8. return t_{new}

Fig. 29. Algorithm *TreeGen*

In essence, the algorithm checks the join condition of $t_{new}.op$: if n_{old} is the source joined with n_{new}, then t_{new} is generated by adding n_{new} as the rightmost child of n_{old} in t_{old}. Tree t_{old} is reconstructed recursively in the same manner. Figure 30 explains *TreeGen* by retracing the steps of Figure 28. When $S\{k_1\}$ produces its first output, *TreeGen*($S\{k_1\}$) returns a tree t_0 that contains a single node $S\{k_1\}$. The parents of $S\{k_1\}$ in the mesh are computed by simulating one loop of *CNGen*($S\{k_1\}$), i.e., adding nodes to t_0 according to the rules of Section 3.2. Each parent (e.g., j_1, j_7) is inserted in the mesh and connected to its left and right inputs. Similarly, when $j_1 = S\{k_1\} \bowtie T\{\}$ starts generating results, it has to create the layer of its parents. The call *TreeGen*(j_1) returns the tree t_1 of Figure 30a, derived by adding a child $T\{\}$ to the only node $S\{k_1\}$ of t_0. The expansion of t_1 reveals the parents of j_1 (e.g., j_2, j_5, j_6). Continuing the example, when j_2 starts producing results, it has to create its own parents. *TreeGen*(j_2) checks which component of j_1 joins with $V\{k_2\}$ in j_2. If $V\{k_2\}$ is joined with $S\{k_1\}$, t_2 is derived by adding $V\{k_2\}$ as the rightmost child of $S\{k_1\}$ in t_1 (left tree in Figure 30b). Otherwise ($V\{k_2\}$ is joined with $T\{\}$), t_2 is derived by adding $V\{k_2\}$ to $T\{\}$ (right tree in Figure 30b). Note that during the computation of t_2, we must also reconstruct t_1, since previous trees have been discarded. Keeping intermediate trees would require a large amount of memory, defeating the purpose of PM.



(a) *TreeGen*(j_1) (b) *TreeGen*(j_2)

Fig. 30. Examples of *TreeGen*

Conversely to generating parents, any operator without output destroys its parents, thereby freeing memory. In Figure 28b, if j_1 stops producing output, its buffer eventually runs dry. Consequently, the parents j_2, j_5 and j_6 are removed, leading back to the partial mesh of Figure 28a. Join operators that have been removed from main memory are regenerated whenever necessary, e.g., fresh output by j_1 at a later time leads to the anew creation of j_2, j_5 and j_6. The destruction of parent operators recursively travels up the operator mesh, e.g., if $S\{k_1\}$ dries up, the entire cluster in Figure 28b is reduced to its sources.

6.5. Purging Expired Tuples

When a source tuple s is deleted, all intermediate results that include s must be removed from the system. Under the positive-negative stream model, purging is part of query processing; i.e., a negative tuple $-s$ travels up the mesh, expunging all composite tuples containing s. The sliding window model allows different variants for removing tuples. Under this model, source buffers are ordered by $s.t_{end}$ (since $s.t_{end} = s.t_{start} + w$), and can be purged by simply inspecting the topmost tuples. In contrast, the output buffers of joins are not sorted on t_{end} (since join results do not expire according to their creation order), and deletions involve complete buffer scans. Thus, in the sequel we assume that source buffers are immediately purged, and propose two algorithms, *eager* and *lazy*, for removing tuples from the output buffers of join operators.

The *eager* approach, illustrated in Figure 31, mimics the bottom-up method used under the positive-negative model. Specifically, whenever a source tuple expires, the corresponding leaf operator removes

the tuple from its output buffer and informs its parents. Any join operator receiving such a note checks its own output buffer, and (should it find expired tuples) informs its parents. The approach is memory-optimal, since deleted tuples are removed immediately from all affected operators. However, it is CPU-intensive, due to the recursive call for all parents (potentially thousands) in lines 7-8.

```
Eager (Operator op)
1.  boolean tell_parents = false
2.  For all tuples s in op.out
3.      If s expires
4.          tell_parents = true
5.          Remove s from op.out
6.  If (tell_parents)
7.      For all parent operators p of op
8.          Eager(p)
```

Fig. 31. *Eager* purging

In contrast, the *lazy* approach reduces CPU consumption by removing expired tuples only when these are encountered during join execution. Assume, for instance, that in Figure 26 $S\{k_1\}$ and $T\{\}$ have tuples from which j_1 produces output. Whenever a tuple in $V\{k_2\}$, $U\{k_2, k_3\}$ or $V\{k_2, k_3\}$ arrives, it is probed against $j_1.out$. The probe loops over the buffer and inspects each tuple for matching join attributes. During the loop, all dead tuples in $j_1.out$ are removed, essentially for free. *Lazy* thereby incurs minimal CPU overhead, but provides no guarantee regarding when a dead tuple is removed. If $V\{k_2\}$, $U\{k_2, k_3\}$ or $V\{k_2, k_3\}$ dry up, $j_1.out$ will not be purged and continue to waste memory.

For FM, we combine *lazy* with demand driven operator execution, in order to limit the time that expired tuples remain in the system. Recall that the troublesome case involves an operator (j_1) with output, whose parents (j_2, j_5, j_6) have no right input. Using demand driven operator execution, j_1 must be sleeping, since all its parents are also dormant. The problem of deleting expired tuples is hence reduced to purging the output buffers of sleeping operators. When an operator *op* halts, its output buffer may still contain live tuples that cannot be expunged since *op* may wake-up soon. However, after *op* sleeps for w seconds, its entire output has expired, and its buffer can be discarded. On the other hand, if *op* restarts before w, the expired tuples will be removed by join processing. Even if a tuple in the output buffer expires before *op* halted, it cannot remain in the system for more than $2w$ after its expiration.

In order to monitor outdated buffers, *lazy* maintains a doubly linked list Q of sleeping operators. If an operator *op* halts, an entry $e = <op, stopTime>$ is appended to Q. Additionally, *op* keeps a pointer to e. A continuous process watches Q's head. When the topmost operator op_{top} (the first to halt in Q) has been sleeping for w ticks ($op_{top}.stopTime + w = now$), it is de-queued and its buffer is cleared of all content. Should an operator wake up before it is de-queued, it removes its entry from Q by following the corresponding pointer. *S*ince the removal of outdated buffers relies on demand driven operator execution, this optimization is only applicable to FM. In contrast, *lazy* purging for PM does not provide any guarantees regarding when an expired tuple is deleted.

6.6. Changes in the Schema

Schema changes may be caused by the appearance or disappearance of either a source (SR), or an edge indicating which SR can be joined. In the following, we focus on changes due to SR; those incurred by edges are handled similarly. First, we address appearances. A new SR S_{new} at time t_{new}, introduces $2^{|K|}$ new nodes in the expanded schema and produces an equal number of source operators. Let M_{old} (M_{new}) be the

operator mesh before (after) t_{new}. Directly switching from M_{old} to an empty M_{new} is not permissible, since older tuples (and intermediate results) that are still alive at t_{new} would be lost. Instead, M_{new} is generated on top of M_{old}, so that all operators of M_{old} (and their intermediate results) become part of M_{new}. Specifically, we apply *CNGen* using the same *nid* as M_{old} for old nodes, and assign to each new node a *nid* that is larger than that of all older sources. Consequently, every operator cluster in M_{old} becomes part of a cluster in M_{new}. Additionally, M_{new} contains $2^{|K-1|}$ additional clusters, rooted at sources of S_{new}. In order not to suspend query processing, the migration from M_{old} to M_{new} occurs successively. During the transition, tuples are routed up the mesh as usual. Each new join operator that receives tuples from both children processes them directly, ensuring that tuples which arrived after t_{new} are properly joined with older ones, and no results are lost during mesh migration.

The disappearance of an SR causes the removal of $2^{|K-1|}$ sources from the mesh. All direct parents of these sources are also purged. The removal of parents travels recursively up the mesh. This process may cause some other operators to remain without parents. Such operators must also be deleted from the mesh. To achieve this, for every direct parent p of a deleted source, we insert the left child into a list l_{rem}, and delete p. After this stage terminates, each operator in l_{rem} that has no parents is removed and its left child is appended to l_{rem}. The process terminates when l_{rem} is empty. SR disappearances require no immediate attention and can be performed whenever the system has resources to spare. The above discussion applies to FM as well as PM. The only difference is that in PM new operators are only created as high as there is data. This concludes the algorithmic part of the paper; next, we evaluate the proposed methods experimentally.

7. EXPERIMENTAL EVALUATION

Section 7.1 compares OB and GB on snapshot queries using the TPC-H benchmark. Section 7.2 evaluates the two methodologies and the impact of various optimizations on continuous queries over relational streams. Section 7.3 concludes with guidelines regarding the most suitable method depending on the problem characteristics. All algorithms are implemented in C++ and experiments are performed on a 3.2GHz Dual-Pentium IV with 2 GB of RAM.

7.1. Snapshot R-KWS Queries over Tables

We compare GB and OB implemented as described in Section 3 (without the optimizations of later sections). We loaded the TPC-H dataset into a relational database powered by MySQL 5.1, using the MyISAM storage engine and its internal full-text index. For our experiments we focus on the six tables of Figure 32: *Part* (0.2M entries), *Supplier* (10K), *PartSupp* (0.8M), *Customer* (150K), *Orders* (1.5M), and *LineItem* (6M). Two tables can join if and only if there is a foreign-key to primary-key between them, shown as arrows in Figure 32. We restrict the length of join sequences to T_{max}, which ranges between 4 and 6.

We designed seven sets of R-KWS queries QS_1-QS_7, listed in Table I. We use the following types of keywords in the queries: (i) people's or companies' names (denoted as *PeopleName*), which appear in the columns *Customer.Name*, *Supplier.Name* and *Orders.Clerk*; (ii) terms from the name of a part, e.g., "ivory", from the *Part.Name* attribute; (iii) years, which are present in *LineItem.ShipDate*, *LineItem.CommitDate*, *LineItem.ReceiptDate*, *Orders.OrderDate*; and (iv) terms from *Part.Brand*, *Part.Mfgr*, *Part.Size* and *Part.Container*. We manually selected all queries, such that they produce at least

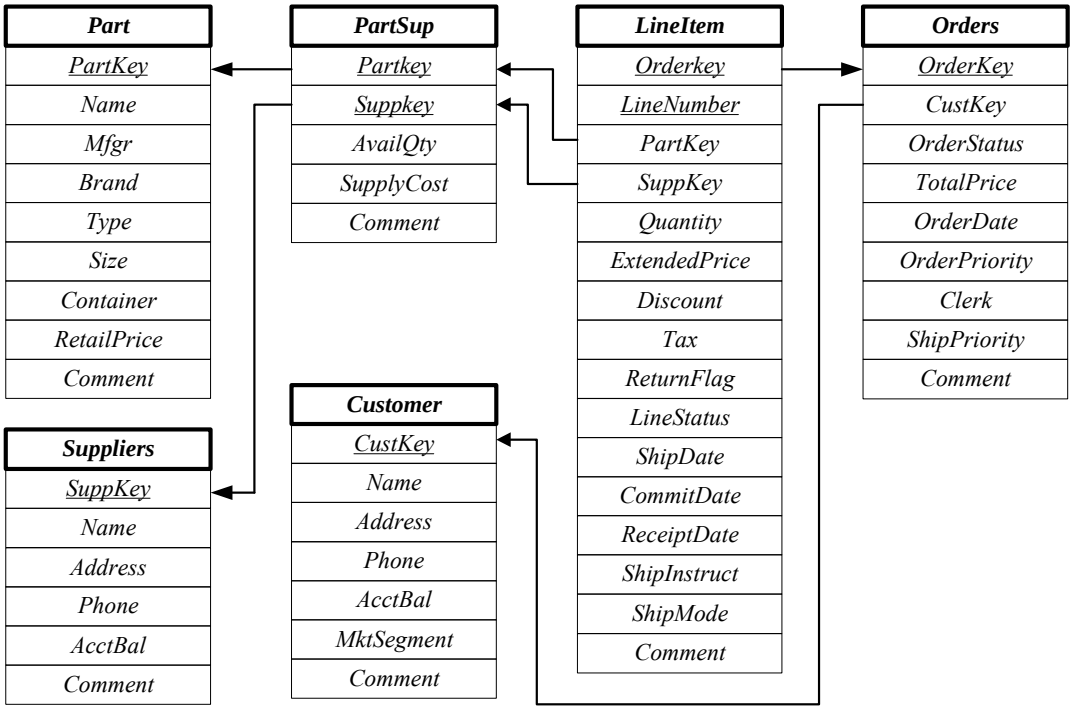


Fig. 32. TPC-H tables used in the experiments

one result at the minimum T_{max} value (i.e., 4). This simulates the fact that users usually have background knowledge on the data. Furthermore, every class of queries mimics a practical search task. Specifically, queries in: (i) QS_1 and QS_2 retrieve connections between multiple people, (ii) QS_3/QS_4 find co-occurrences of different parts, (iii) QS_5 (resp. QS_6) are similar to QS_1 (resp., QS_3), with an additional keyword specifying the year that these relationships occur; (iv) QS_7 consist of attributes of one particular part.

Table I. Query sets

Query Set	Template
QS_1	$PeopleName_1$, $PeopleName_2$
QS_2	$PeopleName_1$, $PeopleName_2$, $PeopleName_3$
QS_3	$PartName_1$, $PartName_2$
QS_4	$PartName_1$, $PartName_2$, $PartName_3$
QS_5	$PeopleName_1$, $PeopleName_2$, $Year_1$
QS_6	$PartName_1$, $PartName_2$, $Year_1$
QS_7	$PartBrand_1$, $PartName_1$, $PartMfgr_1$, $PartType_1$, $PartContainer_1$

Figure 33 depicts the total run time (y-axis) of GB and OB, as well as the result set cardinality $|R|$ (below the x-axis) for the seven query sets. For each query set, we generated 5 queries, and report the median values after setting T_{max} to 4, 5, and 6. Queries in QS_3 and QS_4 yield numerous results because terms from part names have low selectivity (the TPC-H generator uses a small dictionary to generate part names). In contrast, people's names are much more selective, leading to fewer results in the corresponding queries. Naturally, $|R|$ and the query cost increases with T_{max}. For fixed T_{max}, all queries in the same set have similar cost because they search similar parts of the data graph (in GB), and create identical operator trees (in OB).

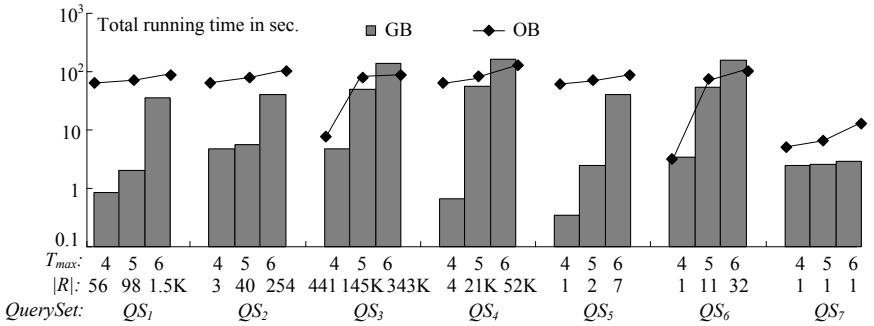


Fig. 33. Query processing time for various T_{max}

GB usually outperforms OB, sometimes by orders of magnitude because it utilizes the data graph pre-computed at an initialization phase; consequently, during query processing it only accesses the database to read from the inverted index. The pre-computation cost (around 220 seconds) is not included in the diagram because the same data graph is used by all queries. However, the graph consumes 0.737GB of RAM, which is a constant overhead of GB given that it must reside in main memory. The parameter T_{max} generally has a significant impact on the performance of GB because a higher T_{max} increases the portion of the data graph traversed for each query. This effect depends on the density of the sub-graph where the search is performed. For QS_7, queries often comprise of keywords extracted from various attributes of a single record. Consequently, nodes far from the seeds may not reach new keywords, leading to early pruning of the partial result by *GSearch*, independently of the T_{max} value. The performance of OB is dominated by join operations. The cost is relatively low for when the joins involve small tables (QS_3 at T_{max}=4, QS_6 at T_{max}=4, and QS_7). In all other cases, the overhead of OB is high due to the size of the tables (e.g., *LineItem* has over 6M records). T_{max} generally does not have a significant impact on OB (except for QS_3 and QS_6, where a small T_{max} avoids expensive joins) since a long join sequence often terminates after not obtaining results from first few tables due to high selectivity of the keywords.

7.2. Continuous R-KWS Queries over Streams

Our stream implementations follow the *Pipes* data stream framework [Krämer and Seeger 2004]. Due to lack of real datasets, we resort to synthetic streams. In particular, we construct a schema containing |*SR*| streaming relations, connected in the shape of a ternary tree: each SR can be joined with up to four other SR (one parent and three children). An SR has one attribute for each edge, used to evaluate equi-joins with the corresponding neighbor. Note that while the schema forms a tree, the data graph can take arbitrary shapes, and it may contain cycles. Additionally, the schema is larger and more densely connected than that of TPC-H (Figure 32). Each SR generates one tuple per second. Attribute values are randomly and independently chosen in the range [1, *sel*]. Two tuples of neighboring SR can thus be joined with probability 1/*sel*, the join selectivity. A tuple contains several different keywords, each with an independent probability *KWF*. We assume a sliding window of *w* minutes, and answer a continuous R-KWS query with |*K*| keywords over five hours. Table II illustrates the ranges and the default values (in boldface) of the experimental parameters.

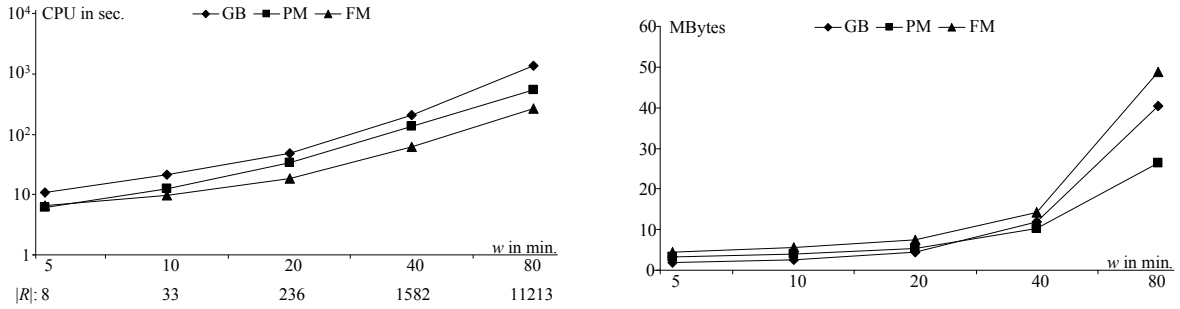
Initially, we investigate the three core methodologies for continuous R-KWS queries: graph based (GB), a full mesh of operators (FM), and a partial mesh of operators (PM). Subsequently, we study the effect of individual optimizations. The GB implementation uses keyword labels including temporal information (Time-KL). FM includes lazy purging, demand driven operator execution and temporary

operator disconnection. Recall that these optimizations are not applicable to PM. We investigate peak memory and total CPU as a function of w, KWF, $|SR|$, sel, $|K|$, and T_{max}. In each experiment, we vary one parameter and set the remaining ones to their default. Additionally, we report the number of generated results $|R|$ (shown under the x-axis of the CPU chart). If affected by the parameter under investigation, we also state the duration of FM mesh-initialization (below the x-axis of the memory chart), which indicates the size and complexity of the operator mesh.

Table II. Parameters under investigation

Parameter	**Range & Default**
W	5, 10, **20**, 40, 80 minutes
KWF	0.003, 0.007, **0.01**, 0.013, 0.016
$\|SR\|$	5, 10, **15**, 20, 25
$1/sel$	1/500, 1/750, **1/1000**, 1/1250, 1/1500
$\|K\|$	2, **3**, 4, 5
T_{max}	2, 3, **4**, 5, 6, 7, 8
$\|Q\|$	**1**, 2, 4, 8, 16, 32, 64

Figures 34a and 34b illustrate the impact of the window size w on the total CPU time (in seconds) and the peak memory (in Mbytes). While the number of live tuples grows linearly with w, the number of combinations in which they can be joined (i.e., edges in the data graph) grows quadratically, as reflected in the number of results $|R|$. Consequently, GB consumes more space and CPU, to store the data graph and construct edges, respectively. Both OB approaches behave similarly, as tuples are more likely to travel up the mesh, requiring CPU for construction and space for storage of intermediate results. As expected, FM incurs the least computational overhead, while PM excels in terms of space.



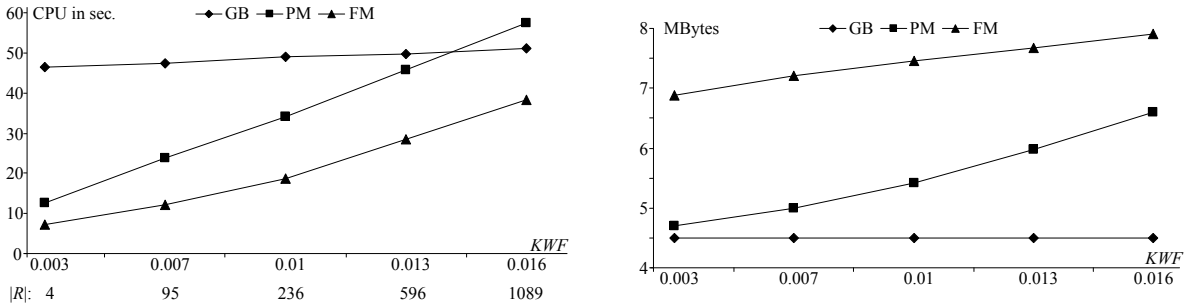
(a) Total CPU cost (b) Peak memory

Fig. 34. Window size w

Figure 35 investigates the impact of the keyword frequency KWF. The relative performance of FM and PM remains similar to Figure 34. Resource consumption of both OB systems grows with KFW, as tuples are more likely to pass selection operators and climb the operator mesh. The increasing number of intermediate results is reflected in the cardinality of the result set $|R|$. In comparison, GB requires less memory, but uses significant amounts of CPU. This approach remains almost insensitive to varying values of KWF, because the dominant factor, the maintenance of the materialized data graph, is independent of keyword appearances. The slight increase in the CPU and memory consumption of GB are attributed to (i) a growing number of graph traversals, and (ii) storage and maintenance of additional keyword labels. Later in this section, we investigate the impact of KL individually.

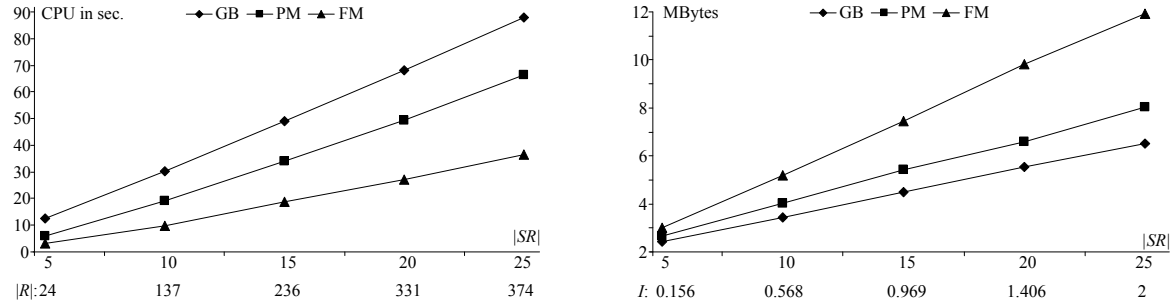
Figure 36 evaluates the effect of the number of streaming relations $|SR|$. Because each SR is limited to four neighbors, an increasing $|SR|$ causes a linearly more complex streaming schema and data graph. The

resource consumption of all methods grows proportionally. In the case of GB, the enlarged data graph requires more memory (for storage) and CPU (for construction). For OB systems, the greater streaming schema causes the operator mesh to grow, as can be seen from the prolonged initialization phase (*I*) of FM (see Figure 36b). The same is true for the number of tuples reaching intermediate operators rises, as reflected in the number of results |*R*|. The relative performance of all three systems is similar to the diagrams of Figure 35.



(a) Total CPU cost (b) Peak memory

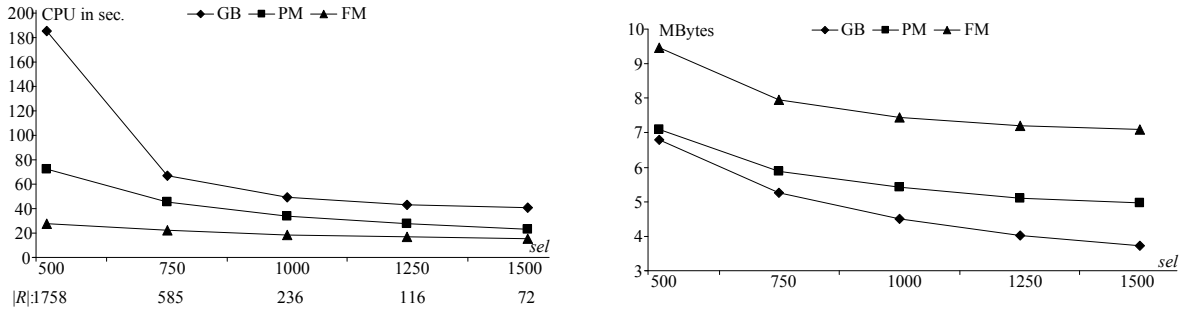
Fig. 35. Keyword frequency *KWF*



(a) Total CPU cost (b) Peak memory

Fig. 36. Number of stream relations |*SR*|

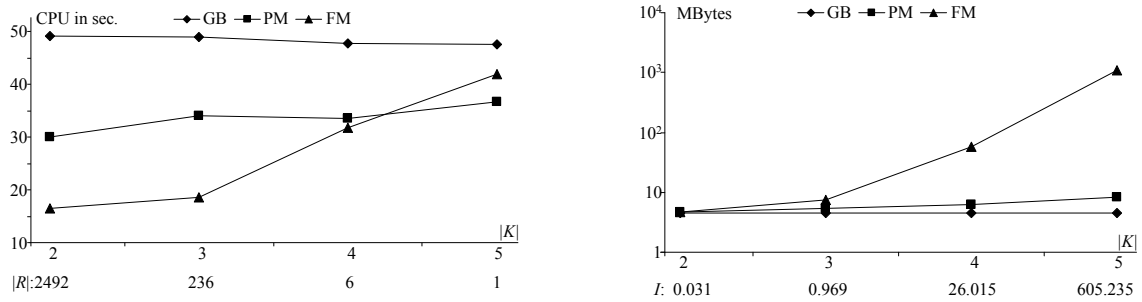
Figure 37 investigates the impact of the join selectivity. An increase in *sel* (drop in 1/*sel*) causes a quadratic decrease in the number of edges in the data graph, and the number of results |*R*| shrinks accordingly. Consequently, GB incurs a quadratically smaller overhead for creating and storing the data graph. Likewise, KL have to be propagated over a reduced number of links. FM and PM display an analogous behavior, as tuples are less likely to reach higher levels of the operator mesh. PM benefits most from a high *sel*, as fewer operators have to be instantiated and executed. Again, FM is most efficient in terms of CPU, and GB in terms of space.



(a) Total CPU cost (b) Peak memory

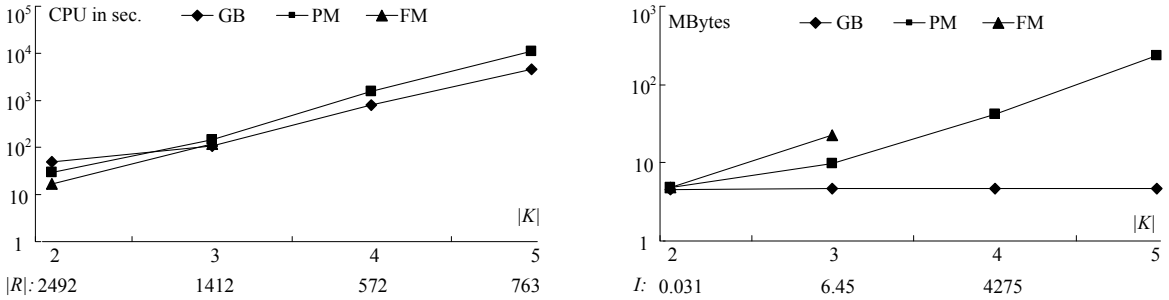
Fig. 37. Join selectivity 1/*sel*

Figure 38 depicts the effect of the query keywords $|K|$. Since this parameter has no impact on the number of tuples, or the way they can be joined, GB remains almost unaffected. For OB, however, more query terms cause an exponential growth in the size and complexity of the operator mesh, as observed from the initialization time of FM (parameter I in Figure 38b). Three keywords require only 2.5 seconds of initialization, whereas five keywords take almost half an hour. Since most operators in this mesh are commonly idle, they are never created by PM; hence, the increasing advantage of this approach in terms of memory. Whereas in Figure 38 T_{max} is fixed to 4 (default value), Figure 39 repeats the experiment after setting T_{max}:= $|K|$+2. The comparison with Figure 38 reveals that in this case the number of results increases fast with $|K|$ and T_{max}, which leads to an analogous growth of processing costs. Notably, for $|K|$ > 3, the memory requirement of FM exceeds the amount of main memory in our system (i.e., 2GBytes); at $|K|$=5, the CN generation module alone takes longer than 5 hours. GB is most scalable with respect to $|K|$ and T_{max}, as its memory consumption remains almost constant in all settings.



(a) Total CPU cost (b) Peak memory

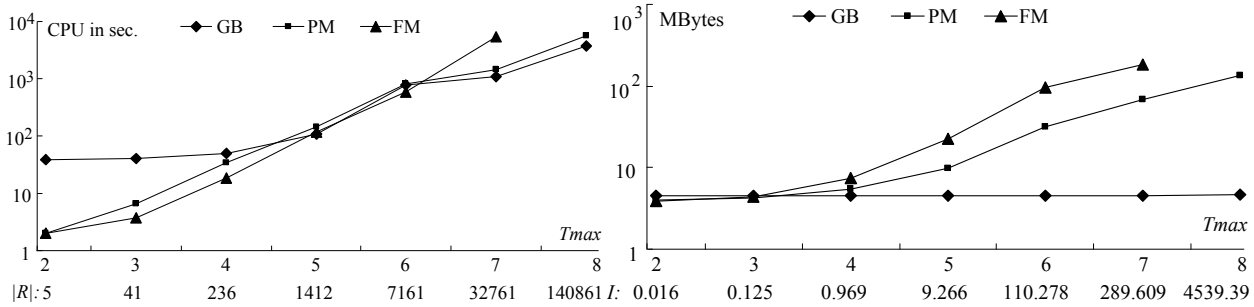
Fig. 38. Number of keywords $|K|$



(a) Total CPU cost (b) Peak memory

Fig. 39. Number of keywords $|K|$, with T_{max}:= $|K|$+2

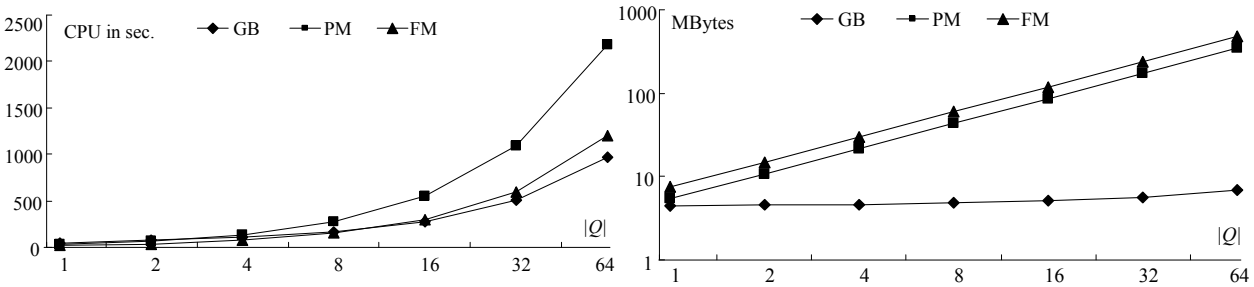
Figure 40 investigates the effect of T_{max}, which has a similar impact to $|K|$ because it does not influence the number of tuples, or the way that they can be joined. GB's memory consumption thus remains constant. However, for high values of T_{max}, MTJNT can become larger and more complex. Thus, keyword labels must be propagated in a greater radius, and every call to *GSearch* explores a larger fraction of the data graph. Consequently, GB's CPU overhead grows with T_{max}. For OB systems, an increase in T_{max} leads to an exponential growth of the operator mesh. In case of FM, operator initialization for T_{max} = 6 exceeds two minutes, compared to less than one second for T_{max} = 3. Akin to the mesh size, CPU and memory grow fast, since (i) the mesh requires more space, (ii) the number of intermediate results increases, and (iii) their generation incurs additional CPU. The number of intermediate tuples is reflected in the cardinality of the result set $|R|$. For T_{max} = 8, FM was too slow to process tuples at the given arrival rate and PM is the only viable OB solution.



(a) Total CPU cost (b) Peak memory

Fig. 40. Limit T_{max} of nodes per MTJNT

Whereas the previous experiments assume a single query, Figure 41 depicts the resources consumed by $|Q|$ parallel queries. For queries consisting of random keywords drawn from a large dictionary, the chance that two queries share any common term is negligible; hence, we assume all $|Q|$ queries to be disjoint. Consequently, both PM and FM execute $|Q|$ independent queries, and their costs increase linearly with $|Q|$. In contrast, a large portion of GB's overhead is due to the storage and maintenance of the data graph, which is shared among all queries. Therefore, GB scales better than OB approaches with the number of parallel queries.



(a) Total CPU cost (b) Peak memory

Fig. 41. Number of concurrent queries $|Q|$

Next, we evaluate specific optimizations. Figure 42 evaluates the effect of *keyword labeling* (KL) on GB, using keyword frequency (*KWF*) as an exemplary parameter. While peak memory for both approaches remains identical and almost constant, KL reduce the overall CPU consumption. The un-optimized approach explores the data graph in a diameter of T_{max} around every new tuple. Consequently, the workload stays unaffected by *KWF*. In contrast, the optimized system only traverses tuples near a keyword occurrence. As *KWF* increases, this case becomes progressively more frequent, rendering KL less beneficial. Eventually, for a very high *KWF*, the workload of both approaches converges. On the other end of the spectrum, a smaller KWF promises even greater savings from KL.

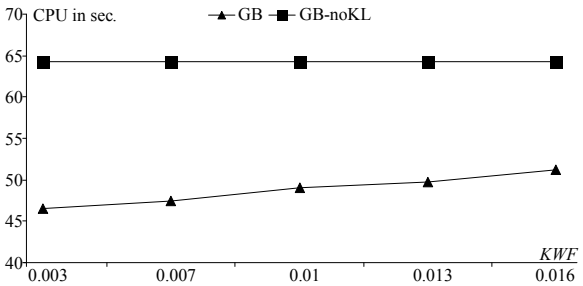


Fig. 42. Effect of keyword labels on GB vs. keyword frequency *KWF*

Figure 43 illustrates the importance of duplicate elimination in *GSearch*, by comparing the number of results returned by our duplicate-free *GSearch* with a straightforward breath-first graph traversal. For w = 80, *GSearch* finds 11,213 unique results, whereas the naïve approach outputs 18,844, including duplicates. In addition to the overhead for retrieving these duplicates, the naïve method entails the cost of eliminating them during a post-processing step.

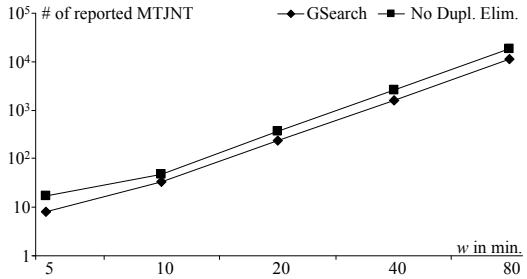
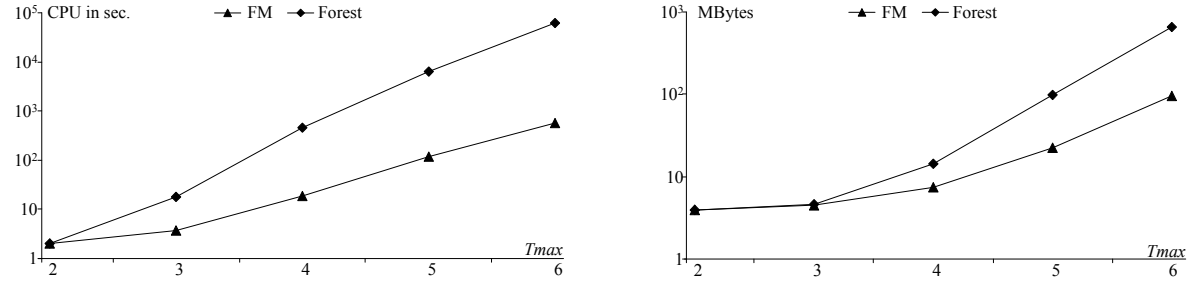


Fig. 43. Effect of duplicate elimination in *GSearch* vs. window size w

The next experiment evaluates the benefits of the operator mesh for OB systems. As an exemplary parameter, we chose the query cardinality $|K|$. Figure 44 depicts the resource utilization of two systems: (i) FM, using an operator mesh, and (ii) a forest of independent operator trees. The operator mesh reduces both CPU and memory requirements by orders of magnitudes. Indeed, the forest approach crashes for any complex scenario, e.g. an increased T_{max}.



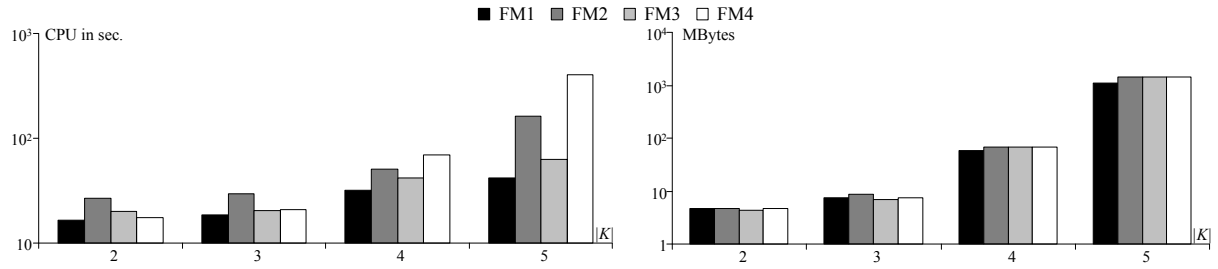
(a) Total CPU cost (b) Peak memory

Fig. 44. Effect of mesh for OB vs. the number of keywords $|K|$

Next, we evaluate four different variations of FM. The first (FM1) is fully optimized, and uses demand driven operator execution, lazy purging, and temporary operator disconnection (this is the default implementation used in the previous experiments). The second variation (FM2) does *not* use demand-driven operator execution; instead it pushes intermediate tuples up the operator mesh, even if they are not needed. In contrast, FM3 is fully optimized, but applies eager purging, as described in Section 6.5. Finally, FM4 uses demand-driven operator execution and lazy purging, but omits temporary operator disconnection, i.e. its join operators remain connected to their right child at all times[2]. Since the benefits of the various optimizations do not become apparent under all settings, we evaluate the four implementations on two exemplary parameters, the query cardinality $|K|$, and the join selectivity $1/sel$. Figure 45a illustrates the impact of $|K|$ on CPU consumption. As suggested by the high cost of FM2, demand-driven operator execution reduces computational expenses at all settings. Similarly, the poor performance of FM4 illustrates the benefits of temporary operator disconnection on large operator meshes

[2] FM4 served as the default FM implementation in [Markowetz et al. 2007].

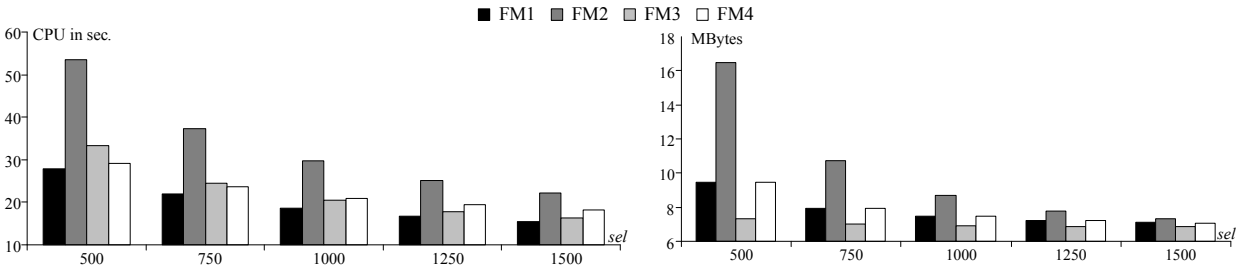
($|K|$=4 and $|K|$=5). Finally, *lazy* (FM1) consistently outperforms *eager* pruning (FM3). For this particular parameter, all four implementations consume roughly the same amount of memory (Figure 45b).



(a) Total CPU cost (b) Peak memory

Fig. 45. Effect of FM optimizations vs. the number of keywords $|K|$

Figure 46 studies the effect of the join selectivity. Small values of *sel* lead to more intermediate results, reflected in an increasing consumption of CPU as well as memory. A comparison of FM1 and FM2 reveals that demand-driven operator execution achieves the most significant gains, especially for low *sel*. The relatively good performance of FM4 indicates that temporary operator disconnection is not as beneficial under this setting as in Figure 45. As predicted in Section 6.5, *lazy* purging (FM1) outperforms *eager* (FM3) in terms of CPU, but consumes more peak memory.



(a) Total CPU cost (b) Peak memory

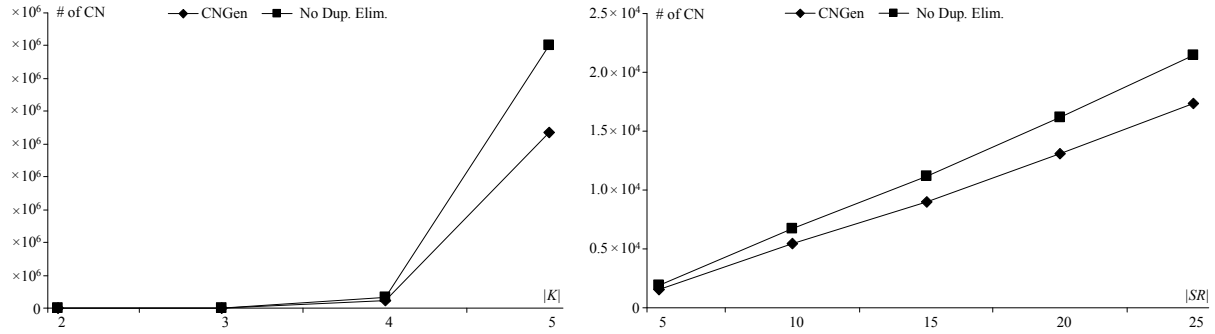
Fig. 46. Effect of FM optimizations vs. the join selectivity 1/*sel*

Finally, we evaluate the importance of duplicate elimination during CN generation. Figure 47 illustrates the number of CN generated by two algorithms. The first is *CNGen*, which avoids duplicates by observing a unique preorder traversal. The second removes all duplicate-specific checks from *CNGen* (i.e., Line 8 and Lines 17-23 in Figure 10), resulting in a CN generation module similar to that proposed in *Discover*[3]. For the default parameters ($|K|$ = 3, $|SR|$ = 15 and T_{max} = 4), the naïve algorithm generates roughly 3,000 duplicates, in addition to about 9,000 unique CN. For more complex settings, this number increases rapidly. Duplicate elimination is a key feature of *CNGen* since duplicate CN are expensive to create, detect and remove.

7.3. Summary of Experimental Evaluation

For conventional tables, GB is more efficient than OB, often by a wide margin. Moreover, since the construction of the data graph is relatively efficient, and the graph can be dynamically maintained, GB is preferable for datasets with frequent updates. However, GB consumes a considerable amount of main

[3] For static databases, *Discover* executes a partial CN as soon as it is generated, and prunes it if the join yields an empty result set. Depending on the data, it may produce fewer CNs than in Figure 47. However in a stream setting, all CNs must be created before their execution, and this optimization is no longer applicable.



(a) Number of keywords $|K|$ (b) Number of streams $|SR|$

Fig. 47. Duplicates avoided by *CNGen*

memory to store the data graph. In contrast, OB utilizes the functionality provided by a DBMS, and, thus, can answer R-KWS queries using much less memory than GB. Therefore, the choice between the two methods depends on the application scenario. On servers dedicated for R-KWS queries, GB is the best choice due to its high performance. On servers running multiple applications and only answer R-KWS queries infrequently, OB might be preferable due to its low memory footprint. Finally, OB is the only feasible solution, if the data graph is too large to fit into main memory.

For continuous R-KWS queries over data streams, FM is usually the most CPU-efficient method for a single query, except for queries involving numerous keywords and/or a large value of T_{max}. On the other hand, GB and PM are more economical in terms of memory consumption. In particular, GB is the best choice for long queries and large values of T_{max}. Moreover, GB scales better with the number of registered queries, and it is the clear choice for systems involving numerous simultaneous queries. Compared to previous GB methods, *GSearch* avoids duplicate results, which reduces the total cost (to retrieve and then eliminate duplicates). Furthermore, the proposed keyword labeling schemes limit the scope of graph traversals, further enhancing performance. The optimizations of OB also achieve significant benefits with respect to the basic methods.

8. CONCLUSION

R-KWS has several advantages over conventional query languages; most notably, it handles broad query tasks whose complexity does not permit hand-coded structured queries. At the same time, it presents considerable algorithmic challenges because query processing has to explore a vast search space. We face these challenges through a series of contributions. First, we provide R-KWS semantics that are well-defined and easily extensible to streaming environments. Then, we develop GB and OB processing techniques that match these semantics and remedy problems encountered in previous systems. Subsequently, we adapt our framework to relational streams, and propose a wide range of optimizations. Finally, we support our claims through an extensive set of experiments. In the future, we plan to further improve R-KWS performance by means of indexing. In parallel, we intend to integrate ranking into continuous R-KWS query processing. For example, if there are a sudden burst of results, it may be desirable to report only the top-k answers for the affected period.

REFERENCES

ABADI, D.J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2003. Aurora: a new Model and Architecture for Data Stream Management. In *The VLDB Journal* (*VLDBJ*), 12(2): 120–139.

AGRAWAL, S., CHAUDHURI, S., AND DAS, G. 2002. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*), February 26-March 1, 2002, San Jose, CA: 5-16.

ARASU, A., BABU, S., AND WIDOM, J. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In *The VLDB Journal* (*VDLBJ*), 15(2): 121–142.

AVNUR, R., AND HELLERSTEIN, J.M. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), May 16-18, 2000, Dallas, Texas: 261-272.

BALMIN, A., HRISTIDIS, V., AND PAPAKONSTANTINOU, Y. 2004. ObjectRank: Authority-based keyword search in databases. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB*), August 31 - September 3, 2004, Toronto, Canada: 564-575.

BHALOTIA, G., HULGERI, A., NAKHE, C., CHAKRBARTI, S., AND SUDARSHAN, S. 2002. Keyword Searching and Browsing in Databases using BANKS. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*), February 26 - March 1, 2002, San Jose: 431-440.

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M.J., HELLERSTEIN, J.M., HONG, W., KRISHNAMURTHY, S., MADDEN, S.R., RAMAN, V., REISS, F., AND SHAH, M.A. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the Biennial Conference on Innovative Data Systems Research* (*CIDR*), January 5-8, 2003, Asilomar, CA.

CHAUDHURI, S., DAYAL, U. AND YAN T.W. 1995. Join Queries with External Text Sources: Execution and Optimization Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), May 22-25, 1995, San Jose, California: 410-422.

COHEN, S., KANZA, Y., KIMELFELD, B. AND SAGIV, Y. 2005 Interconnection Semantics for Keyword Search in XML. In *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management* (*CIKM*), October 31 - November 5, 2005, Bremen, Germany: 389-396.

DAR, S., ENTIN, G., GEVA, S., AND PALMON, E. 1998. DTL's DataSpot: Database Exploration Using Plain Language. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB*), August 24-27, 1998, New York City: 645-649.

DE FELIPE, I., HRISTIDIS, V., AND RISHE, N. 2008. Keyword Search on Spatial Databases. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*), April 7-12, 2008, Cancún, México: 656-665.

FABRET, F., JACOBSEN, H. A., LLIRBAT, F., PEREIRA, J., ROSS, K.A., AND SHASHA, D. 2001. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD Record*, 30(2): 115–126.

GOLAB, L., AND ÖSZU, T.M. 2003. Issues in Data Stream Management. In *SIGMOD Record*, 32(2): 5–14.

GOLENBERG, K., KIMELFELD, B., AND SAGIV, Y. 2008. Keyword Proximity Search in Complex Data Graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 10-12, 2008, Vancouver, Canada: 927-940.

GRAVANO, L., IPEIROTIS, P., KOUDAS, N., AND SRIVASTAVA, D. 2003. Text Joins in an RDBMS for Web Data Integration. In *Proceedings of the International World Wide Web Conference* (*WWW*), 20-24 May, 2003, Budapest, Hungary: 90-101.

GUO, L., SHAO, F., BOTEV C., AND SHANMUGASUNDARAM J. 2003. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 9-12, 2003, San Diego: 16-27.

HE, H., WANG, H., YANG, J., AND YU, P. 2007. BLINKS: Ranked Keyword Searches on Graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 12-14, 2007, Beijing, China: 305-316.

HRISTIDIS, V., GRAVANO, L., AND PAPAKONSTANTINOU, Y. 2003. Efficient IR-Style Keyword Search over Relational Databases. In *Proceedings of 29th International Conference on Very Large Data Bases* (*VLDB*), September 9-12, 2003, Berlin: 850-861.

HRISTIDIS, V., AND PAPAKONSTANTINOU, Y. 2002. DISCOVER: Keyword Search in Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB*), August 20-23, 2002, Hong Kong, China: 670-681.

HRISTIDIS, V., PAPAKONSTANTINOU, Y., AND BALMIN, A. 2003. Keyword Proximity Search on XML Graphs. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*), March 5-8, 2003, Bangalore, India: 367-378.

HRISTIDIS, V., VALDIVIA, O., VLACHOS, M., AND YU, P. 2006. Continuous Keyword Search on Multiple Text Streams. In *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management* (*CIKM*), November 6-11, 2006, Arlington, Virginia: 802-803.

IRMAK, U., MIHAYLOV, S., SUEL, T., GANGULY, S., AND IZMAILOV, R. 2006. Efficient Query Subscription Processing for Prospective Search Engines. In *Proceedings of the USENIX Annual Technical Conferenc*e, May 30 - June 3, 2006, Boston, Massachusetts: 375 – 380.

KACHOLIA, V., PANDIT, S., CHAKRABARTI, S., SUDARSHAN, S., AND DESAI, R., KARAMBELKAR, H. 2005. Bidirectional Expansion for Keyword Search on Graph Databases. In *Proceedings of the International Conference on Very Large Data* Bases (*VLDB*), August 30 - September 2, 2005, Trondheim, Norway: 505-516.

KIMELFELD, B., AND SAGIV, Y. 2005. Efficiently Enumerating Results of Keyword Search. In *Proceedings of the International Symposium on Database Programming Languages* (*DBPL*), August 28-29, 2005, Trondheim, Norway, Lecture Notes in Computer Science 3774: 58-73.

KIMELFELD, B., AND SAGIV, Y. 2006. Finding and Approximating Top-*k* Answers in Keyword Proximity Search. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (*PODS*), June 26-28, 2006, Chicago, Illinois: 173-182.

KRÄMER, J., AND SEEGER, B. 2004. PIPES: A Public Infrastructure for Processing and Exploring Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 13-18, 2004, Paris, France: 925-926.

LI, G., OOI, B.C., FENG, J., WANG, J., AND ZHOU, L. 2008. EASE: Efficient and Adaptive Keyword Search on Unstructured, Semi-structured and Structured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 10-12, 2008, Vancouver, Canada: 903-914.

LIU, F., YU, C., MENG, W., AND CHOWDHURY, A. 2006. Effective Keyword Search in Relational Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 27-29, 2006, Chicago, Illinois: 563-574.

LIU, Z., AND CHEN, Y. 2007. Identifying Meaningful Return Information for XML Keyword Search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 12-14, 2007, Beijing, China: 329-340.

LUO, Y., LIN, X., WANG, W., AND ZHOU, X. 2007. SPARK: Top-*k* Keyword Query in Relational Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 12-14, 2007, Beijing, China: 115-126.

MARKOWETZ, A., YANG, Y., PAPADIAS, D. 2007. Keyword Search on Relational Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). June 11-14, 2007, Beijing, China: 605-616.

MARKOWETZ, A., YANG, Y., PAPADIAS, D. 2009. Reachability Indexes for Relational Keyword Search. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*). March 29-April 4, 2009, Shanghai, China: 1163-1166.

SARDA, N.L., AND JAIN, A. 2001. Mragyati: A System for Keyword-Based Searching in Databases. *Technical Report CoRR*, cs.DB/0110052.

SAYYADIAN, M., LEKHAC, H., DOAN, A., AND GRAVANO, L. 2007. Efficient Keyword Search across Heterogeneous Relational Databases. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*), April 15-20, 2007, Istanbul, Turkey: 346-355.

SU, Q., AND WIDOM, J. 2005. Indexing Relational Database Content Offline for Efficient Keyword-Based Search. In *Proceedings of the International Database Engineering and Applications Symposium* (*IDEAS*), 25-27 July, 2005, Montreal, Canada: 297-306.

VU, Q.H., OOI, B.C, PAPADIAS, D., AND TUNG, A. 2008. A Graph Method for Keyword-based Selection of the Top-*k* Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 10-12, 2008, Vancouver, Canada: 915-926.

WU, P., SISMANIS, Y., AND REINWALD, B. 2007. Towards Keyword-Driven Analytical Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 12-14, 2007, Beijing, China: 617-628.

XU, Y., AND PAPAKONSTANTINOU, Y. 2005. Efficient Keyword Search for Smallest LCAs in XML Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 14-16, 2005, Baltimore, Maryland: 537-538.

YU, B., LI., G., SOLLINS, K., AND TUNG, A. 2007. Effective Keyword-based Selection of Relational Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*), June 12-14, 2007, Beijing, China: 139-150.

YAN, T.W., AND GARCIA-MOLINA, H. 1999. The SIFT Information Dissemination System. In *ACM Transactions on Database Systems* (*TODS*), 24(4): 529–565.