Keyword Search on Relational Data Streams

Alexander Markowetz Yin Yang Dimitris Papadias
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{alexmar, yini, dimitris}@cs.ust.hk

ABSTRACT
Increasing monitoring of transactions, environmental parameters, homeland security, RFID chips and interactions of online users rapidly establishes new data sources and application scenarios. In this paper, we propose keyword search on relational data streams (S-KWS) as an effective way for querying in such intricate and dynamic environments. Compared to conventional query methods, S-KWS has several benefits. First, it allows search for combinations of interesting terms without a-priori knowledge of the data streams in which they appear. Second, it hides the schema from the user and allows it to change, without the need for query re-writing. Finally, keyword queries are easy to express.

Our contributions are summarized as follows. (i) We provide formal semantics for S-KWS, addressing the temporal validity and order of results. (ii) We propose an efficient algorithm for generating operator trees, applicable to arbitrary schemas. (iii) We integrate these trees into an operator mesh that shares common expressions. (iv) We develop techniques that utilize the operator mesh for efficient query processing. The techniques adapt dynamically to changes in the schema and input characteristics. Finally, (v) we present methods for purging expired tuples, minimizing either CPU, or memory requirements.

Categories and Subject Descriptors: H.2.3 Database Management-Languages, H.3.3 Information Search and Retrieval
General Terms: Algorithms
Keywords: Keyword Search, Data Streams

1. INTRODUCTION
With the rise of Web search engines, keyword search (KWS) has become a leading search paradigm. In conventional KWS, each document constitutes one *unit of information*, and is considered a result, if it contains all/some of the query's keywords. Recently, KWS has also been applied to relational DBMS, allowing data retrieval without SQL knowledge. In relational keyword search (R-KWS), the basic unit of information is a tuple/record. In contrast to KWS on documents, results in R-KWS cannot simply be found by inspecting units of information (records) individually. Instead, results have to be *constructed* by joining tuples.

Assume the movie database of Figure 1.1 that contains four tables: *director*, *movie*, *plays* and *actor*. Edges connecting the tables correspond to join conditions, e.g., a movie record can be joined with the tuples of its director (*movie.did* = *director.did*) and actors (*movie.mid* = *plays.mid*). For simplicity, consider that the database contains only the seven tuples of Figure 1.1. Given the R-KWS query q:= {Tarantino, Travolta}, the system returns two results: $t_1 \bowtie t_2 \bowtie t_5 \bowtie t_3$ and $t_3 \bowtie t_6 \bowtie t_7 \bowtie t_4$. The first one signifies that there is a movie (Pulp Fiction), which was directed by Tarantino and features Travolta. The second implies that there is movie (*mid*=5) that includes both Tarantino and Travolta as actors.

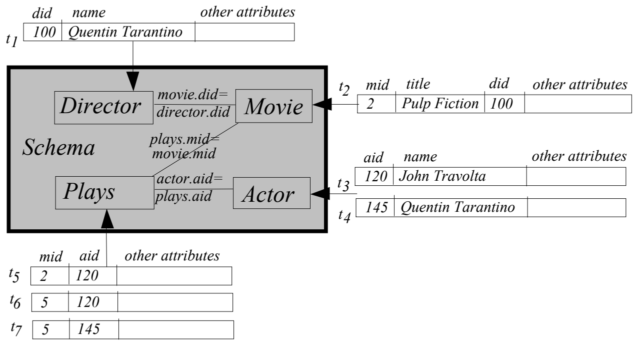


Figure 1.1 Example of relational KWS

R-KWS shows several major benefits over SQL queries. First, it liberates the user from having to study a (possibly messy) database schema. In the above example, a query can be issued without knowledge of tables, their attributes, or join conditions. Second, R-KWS allows querying for terms in unknown locations (tables/attributes). For instance, "Tarantino" appears both as an actor and a director. A user trying to identify projects in which Tarantino and Travolta cooperated does not care about their particular roles (e.g., actor or director). Finally, a single R-KWS query replaces numerous complex SQL statements. Finding the two interactions between Tarantino and Travolta requires the two SQL expressions in Figure 2.2. However, these are only the statements that actually output results. Many more SQL queries have to be issued, in order to cover every possible interaction, e.g. a movie starring Tarantino that was directed by Travolta. The overwhelming number of such SQL queries (often ranging in the thousands), prohibits the usage of hand-coded SQL on any database with a non-trivial schema.

```
SELECT *
FROM Director D, Movie M,
        Plays P, Actor A
WHERE D.name=Tarantino, A.name=Travolta
and D.did=M.did and P.mid=M.mid
and A.aid=m.aid
```

(a) $t_1 \bowtie t_2 \bowtie t_5 \bowtie t_3$

```
SELECT *
FROM Actor A1, Actor A2, Plays P1, Plays P2,
WHERE A1.name=Tarantino, A2.name=Travolta
and A1.aid=P1.aid and A2.aid=P2.aid and
P1.mid=P2.mid
```

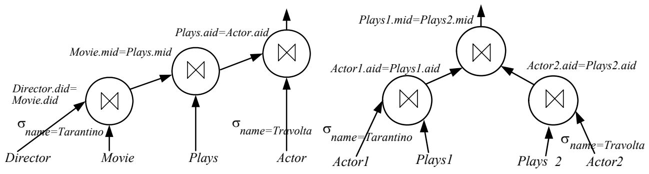
(b) $t_3 \bowtie t_6 \bowtie t_7 \bowtie t_4$

Figure 1.2 SQL statements for two results

The advantages of R-KWS have led to a variety of methods, surveyed in Section 2. However, the additional flexibility compared to conventional query languages, comes at the expense of high execution cost. Specifically, the search space is now considerably larger, since keywords may appear in arbitrary attributes of arbitrary tables, and all feasible combinations of keyword occurrences have to be explored. One common approach

SIGMOD'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006...$5.00.

to R-KWS processing generates an operator tree for each combination. Figure 1.3 shows two such trees, corresponding to the two results of Figure 1.2. Similar to the hand-coded SQL statements, a large number of additional trees have to be evaluated, although they do not produce output. This number can be reduced by restricting the domain of keywords, e.g., by specifying that Tarantino can only appear as a name in the *director* or *actor* tables. For generality, we assume no such restrictions since they require knowledge of the schema.



(a) $t_1 \bowtie t_2 \bowtie t_5 \bowtie t_3$ (b) $t_3 \bowtie t_6 \bowtie t_7 \bowtie t_4$

Figure 1.3 Operator trees for two results

In this paper we apply KWS to data streams with relational structure; i.e., all tuples from the same source have the same attributes and different streams can be joined according to certain conditions. As an example, assume that in Figure 1.1 records are not stored a-priori, but continuously arrive from distributed databases, box office sites etc. Increasing monitoring of transactions, environmental parameters, homeland security, RFID chips or interactions of online users rapidly establishes new data sources and application scenarios. Stream keyword search (S-KWS) provides an effective and intuitive way for dealing with the high complexity and dynamic nature of the data.

S-KWS amplifies the benefits, as well as the challenges, of R-KWS. In terms of benefits, unlike conventional databases, the stream schema is likely to change at runtime, as new sources are integrated and old sources cease to send input. Thus, it is difficult for a user to have complete and up-to-date knowledge of the schema. Even if this knowledge were available, any changes would require the user to instantly adapt his/her queries. On the other hand, S-KWS provides an intuitive way for posing queries, which is transparent to the schema and its potential changes.

Regarding challenges, S-KWS is more complex and expensive (in terms of both computation and memory overhead) than R-KWS. The complexity arises from the fact that the queries are long running, and their output must be continuously updated. R-KWS systems only deal with snapshot queries on a static schema without modules for handling result insertion (when new tuples arrive), deletion (when old tuples expire), and schema changes. The high computation and memory overhead is due to the fact that all the operator trees must be maintained during the entire lifespan of the query. In R-KWS, on the other hand, many trees can quickly be dropped. For instance, if there is no director with the name Tarantino, every tree containing the selection $\sigma_{\text{director.name=Tarantino}}$ (e.g., the one in Figure 1.3a) can immediately be discarded. In S-KWS these trees must remain active, because a tuple containing the keyword may arrive in the future. Furthermore, unlike R-KWS, where the system can take advantage of well-understood query optimization mechanisms from the underlying DBMS, S-KWS requires novel algorithms. The paper faces the above challenges through the following contributions:

- We provide formal semantics for S-KWS, addressing temporal validity and order of the results.
- We propose an algorithm for generating operator trees that outperforms existing methods (for R-KWS) both in terms of efficiency and applicability to a wider range of schemas.
- We integrate these trees into an operator mesh that reduces the CPU cost and memory consumption by sharing common expressions.
- We develop techniques that utilize the operator mesh for efficient query processing. The techniques adapt dynamically to changes in the schema and input characteristics.
- We present methods for purging expired tuples, minimizing the CPU or memory requirements.

The remainder of the paper is structured as follows. Section 2 outlines related work, and describes basic concepts in R-KWS. Section 3 defines the semantic model for S-KWS, presents the generation of operator trees and the construction of an equivalent operator mesh. Section 4 proposes query processing techniques and introduces mechanisms for purging old tuples and altering the stream schema. Section 5 contains the experimental evaluation, and Section 6 concludes the paper.

2. RELATED WORK

R-KWS semantics are commonly based on a graph representation of the database [DEGP98]. Each node in the *data graph G* represents a tuple, and edges connect tuples that can be joined. Figure 2.1 shows a schema with four tables, and the data graph for a small instance of the database. In our notation, s_i signifies a tuple of S, t_i one of T, etc. Two tuples (e.g., s_2, t_1) are connected in G, iff their relations (S, T) are connected in the relational schema and the tuples satisfy the corresponding join conditions. 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.

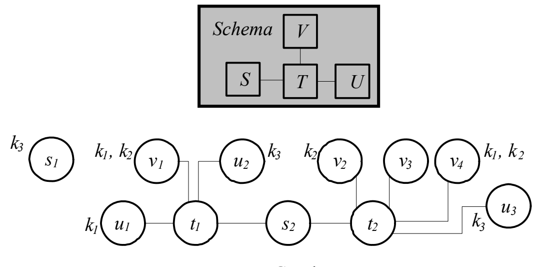


Figure 2.1 Relation schema and corresponding data graph

The result of an R-KWS query can be defined using the concept of *Minimal Total Join Network* (MTJNT) [HP02]. Given a query q:= $\{k_1, \ldots, k_m\}$, an MTJNT is a connected acyclic component of the data graph G that is (i) *total*, i.e., it contains all keywords k_1, …, k_m, and (ii) *minimal*, i.e., it is impossible to remove any node and still have a total network. In particular, minimalism is satisfied, iff every terminal node contains at least one *unique* keyword, i.e., one that is not contained in any other node of the MTJNT. The left side of Figure 2.2 shows several MTJNT for the data graph in Figure 2.1, assuming that q:= $\{k_1, k_2, k_3\}$. Intuitively, only MTJNT are valid R-KWS results. For instance, (v_1, t_1, u_2, s_2)[1] does not constitute an MTJNT since s_2 is redundant; the minimal result is (v_1, t_1, u_2). Similarly, (v_2, t_2, u_3) is not total as it does not include keyword k_1.

[1] For ease of presentation, we often denote an MTJNT as a sequence of nodes.

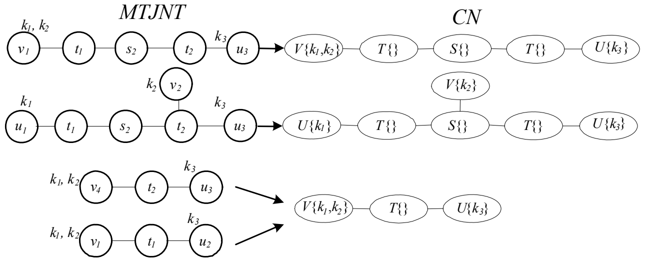


Figure 2.2 Examples of MTJNT and CN

There are various approaches for R-KWS processing. One family of systems, such as *Discover* [HP02, HGP03], *DBXplorer* [ACD02] and *Mragyati* [SJ01], translate an R-KWS query into a series of SQL statements, which are executed directly on secondary storage, using the underlying DBMS. In the sequel, we focus on *Discover*, an influential system that introduced several concepts related to our work. An *expanded schema*[2] is a graph whose nodes correspond to horizontal decompositions of relations, according to the set of keywords they contain. The nodes in the expanded schema are denoted as follows: $S\{K\}$ signifies all tuples of relation S that contain exactly the set K of query terms; $K \subseteq \{k_1, \ldots, k_m\}$. The set of elements in S that do not contain any keyword is denoted as $S\{\}$. Edges in the expanded schema connect two such sub-relations $S\{K'\}$ and $T\{K''\}$, iff their parent relations S and T are connected in the original database schema. *Candidate networks* (CN) are the projections of MTJNT on the expanded schema. A CN is a particular combination of keyword occurrences. For instance, the MTJNT (v_1, t_1, u_2) of Figure 2.1 maps to the CN ($V\{k_1, k_2\}$, $T\{\}$, $U\{k_3\}$) because $v_1 \in V\{k_1, k_2\}$, $t_1 \in T\{\}$ and $u_2 \in U\{k_3\}$. Figure 2.2 illustrates the mappings of several MTJNT (on the left) to CN (on the right). Note that multiple MTJNT, e.g., (v_1, t_1, u_2) and (v_4, t_2, u_3), can map to the same CN, and that a CN may contain the same sub-relation (e.g., $T\{\}$) multiple times.

Discover answers R-KWS queries by returning all MTJNT that do not exceed T_{max} nodes. T_{max} is a parameter used to avoid long chains of joins, which usually lead to uninteresting results. The system first generates the set of CN by traversing the expanded schema. Next, it creates an operator tree for each CN. Leaf nodes in the trees correspond to selections and inner nodes to joins. Figure 2.3 shows an operator tree for the CN ($V\{k_1, k_2\}$, $T\{\}$, $U\{k_3\}$). The selection $\sigma_{k_1 \wedge k_2 \wedge \neg k_3} V$ produces all tuples in the sub-relation $V\{k_1, k_2\}$, whereas $\sigma_{\neg k_1 \wedge \neg k_2 \wedge \neg k_3} T$ produces $T\{\}$. This tree generates the lower two MTJNT in Figure 2.2. Finally, operator trees are translated into SQL statements and executed by the underlying DBMS. Common sub-expressions (e.g., $V\{k_1, k_2\} \bowtie T\{\}$) are shared between trees for several CN (e.g., the first and third in Figure 2.2).

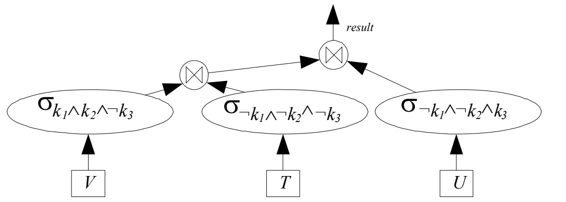

result
$\sigma_{k_1 \wedge k_2 \wedge \neg k_3}$
$\sigma_{\neg k_1 \wedge \neg k_2 \wedge \neg k_3}$
$\sigma_{\neg k_1 \wedge \neg k_2 \wedge k_3}$
V
T
U

Figure 2.3 Operator tree for CN ($V\{k_1, k_2\}$, $T\{\}$, $U\{k_3\}$)

[2] The concept was introduced [HP02], but had not been named. We use the term *expanded schema* for easier reference.

Another family of methods, including *Banks* [HBN+01] and *DBSurfer* [WLK04], maintains the actual data graph in main memory and generates results by graph traversal. Specifically, given a query, an inverted index identifies all tuples that contain at least one keyword. Each such tuple initiates a graph traversal, e.g., in Figure 2.1 traversals would start from nodes s_1, v_1, u_1, u_2, v_2, v_4 and u_3. Whenever a node is reached by all keywords, an MTJNT is constructed by following the reverse paths to the keyword occurrences. Duplicates are filtered in a second, post-processing step.

Requiring neither an in-memory data graph nor SQL, the *Ekso* system [SW05] computes R-KWS queries by means of extensive pre-processing. Given some pruning condition, *Ekso* determines for each node the set of all reachable nodes in the data graph. It then constructs a *virtual document* for each node, by writing the attribute values of all reachable nodes to a text file. These files are organized in an inverted index, on which queries are performed conventionally. Virtual documents are incompatible to most R-KWS semantics, such as MTJNT, since they cannot ensure minimalism.

Hristidis et al. [HPB03] have extended *Discover* to XML databases. In this case, the nodes of the data graph represent XML elements. Edges connect elements that are contained in each other, or reference each other. Cohen et al. [CKKS05] discuss semantics, and Gao et al., [GSBS03] ranked output for KWS over XML. Work by [HGP03, BHP04, CDHW04, LYMC06] focuses on ranking functions for R-KWS, mainly aimed at computing results in a top-*k* fashion. Since S-KWS results are sorted by time, our work does not require additional ranking.

There is an extensive literature on relational data streams. Under this paradigm, data elements (e.g., relational tuples), generated by 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 operator trees, where operators closely resemble their traditional counterparts such as selections or joins. Influential DSMS prototypes include: (i) *Aurora* [ACC+03], targeted mainly at processing sensor data, (ii) *TelegraphCQ* [CCD+03], focusing on the novel *Eddy* operator [AH00], (iii) *Stream* [ABW06], designed as a general purpose DSMS, and (iv) *Pipes* [KS04], a public infrastructure based on the XXL Java library for relational databases. Surveys of various DSMS can be found in [BBD+02, GO03].

Depending on the application characteristics, DSMS adopt different models regarding the validity of tuples. A popular model 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 and 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 time $s.t_{end}$. Two tuples can be joined only if their lifespans overlap. Join results must also be assigned a timestamp, since they may constitute input for a subsequent operator. Usually, the lifespan of a join result is defined as the intersection of all participating tuples' lifespans; e.g., if c is composed of tuples s and t, then $c.t_{start} = max(s.t_{start} + t.t_{start})$ and $c.t_{end} = min(s.t_{end} + t.t_{end})$.

KWS has also been applied to streaming documents (e.g., continuously arriving news articles). With few exceptions [YG99, FJL+01, IMS+06], most related work is proprietary. The main difference with respect to R-KWS and S-KWS is that documents do not have to be joined, but are evaluated individually (similarly to traditional KWS). In a recent poster [HVVY06], Hristidis et al. proposed KWS over multiple textual streams. Similar 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. Firstly, tuples in [HVVY06] have only one attribute, their text. Secondly, only tuples that contain keywords can contribute to a result. Thirdly, and most significantly, combinations (joins) of several tuples are not evaluated upon their (text) 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 the following, we present a comprehensive solution for keyword search on relational data streams, including formal semantics, efficient algorithms and optimizations for different problem settings.

3. STREAM KEYWORD SEARCH

Section 3.1 describes general concepts and provides semantics for S-KWS. Section 3.2 presents an efficient, duplicate-free algorithm for candidate network generation. Section 3.3 discusses operator trees and meshes. Since R-KWS has a relatively long history and well-understood semantics, we adhere to R-KWS concepts as closely as possible.

3.1 Semantics

We assume a DSMS that monitors several relational streams and answers continuous keyword queries of the form q:= $\{k_1, \ldots, k_m\}$. Tuples arrive ordered by 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. For the remainder of the paper, we assume streams to be bundled into SR, and hence use the terms stream and SR interchangeably. Note that tuples do not necessarily have unique keys; e.g., there may exist two records with exactly the same values for all attributes originating from different cash registers.

A graph, called *streaming schema*, denotes which streams can be joined and on what attributes. Nodes in the streaming schema represent SR. Two SR are connected by an edge, iff they can be joined. 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). A newly arriving data stream 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. In our examples, we use a query with three keywords k_1, k_2, k_3 on four SR: S, T, U and V. The schema is the same as that in Figure 2.1, i.e., the only joins permitted are those between tuples in T and those in S, U, or V.

We define S-KWS 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. Two tuples are connected, iff they can be joined. Figure 3.1 shows $G(\tau = 9)$ for the example schema, including the lifespans of the tuples. Note that, in case of positive-negative tuples, these lifespans are not known in advance. The appearance of keywords is denoted next to the tuples. Similar to previous work on R-KWS, we impose a limit T_{max} of tuples per MTJNT, in order to avoid overly long chains of joins. Let $R(\tau)$ be the set of MTJNT in $G(\tau)$ that do not exceed T_{max} nodes. The result R of a continuous S-KWS query is the union of $R(\tau)$, for all τ. The MTJNT (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) in Figure 3.1 are elements of $R(\tau = 9)$. 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 $R(\tau = 10)$. Results are produced in ascending t_{start} order.

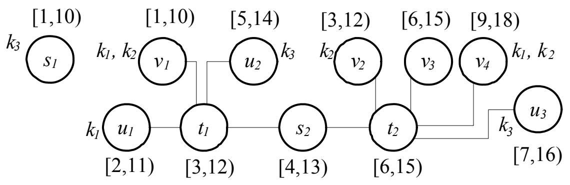


Figure 3.1 Instantaneous data graph at $\tau = 9$

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

Lemma 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 a MTJNT on $G(\tau+1)$; 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, since edges cannot change, if r is connected at τ, it is also connected at τ+1. Hence, if r is total and minimal at τ, it is also total and minimal at τ+1. □

According to Lemma 1, an MTJNT r is not affected by insertions or deletions of external nodes.[3] Consequently, every MTJNT r is constructed and reported only once (at $r.t_{start}$), rather than at every instant during its lifespan. The termination of a result r, on the other hand, depends on the stream model. For a sliding window of duration w, we can compute the lifespan of r, directly when it is created, 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 3.1 is output at τ = 5, in combination with the lifespan [5, 10). In the positive-negative model (where $n.t_{end}$ is not known in advance), r is terminated when the first of its constituent tuples is deleted through a negative tuple. When the result is terminated, the user receives a negative tuple $-r$.

Figure 3.2 illustrates our framework for keyword search on relational streams. Similar to R-KWS systems, we first generate all candidate networks that may produce results (i.e., MTJNT) for a given query and schema. Each CN is transformed to an operator tree. These trees are integrated in an operator mesh that exploits sharing opportunities. Tuples from the various stream sources are

[3] Other semantics, such as these used by *Banks* [HBN+01], do not show the property of Lemma 1, e.g., a result may be invalidated because of a new arrival, necessitating continuous monitoring of all results at each timestamp.

routed through the operator mesh and spawn output. In the remainder of this section, we describe an efficient algorithm for CN generation and present the structure of the operator mesh. Query processing depends on the particular problem settings, and is discussed separately in Section 4.

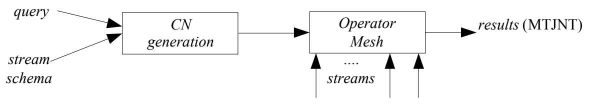
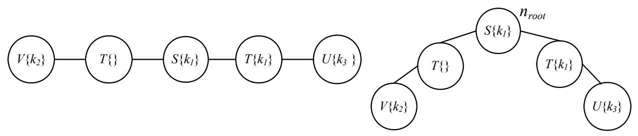


Figure 3.2 General framework

3.2 Candidate Networks for Data Streams

Recall from Section 2 that *Discover* [HP02] already contains a module for CN generation. Unfortunately, that module generates duplicate CN that have to be filtered out in a post-processing step. We propose *CNGen*, a novel algorithm that computes CN according to their unique pre-order traversal and, hence, it is free of duplicates. Futhermore, the algorithm contains elaborate pruning conditions that abort an expansion as soon as it becomes clear that it cannot lead to a new CN[4].

CNGen requires a total ordering (*nid*) on nodes of the expanded schema. The concrete ordering does not influence correctness, but as discussed in Section 3.3, it affects performance. For the remainder of the presentation, we assume that each streaming relation (e.g., *S*) has an ID (*S.sid*). Nodes in the expanded schema have a keyword bitmap (*kbit*) according to the keywords contained in the node, e.g., $S\{k_2\}.kbit$ = 010 = 4 and $S\{k_1, k_3\}.kbit$ = 101 = 5. The node order *nid* is a lexicographic combination of *sid* and *kbit*, e.g., $S\{k_1, k_3\}.nid < T\{k_2\}.nid$ and $S\{k_2\}.nid < S\{k_1, k_3\}.nid$. For simplicity, we first consider that there are no multiple appearances of the same node in a CN. In this case, given the *nid* order and having designated a root node n_{root}, every CN can be represented as a unique tree, where the children of a node are arranged from left to right by *nid*. Figure 3.3 shows a CN for q:= $\{k_1, k_2, k_3\}$ and the corresponding tree for $n_{root} = S\{k_1\}$. Note that n_{root} may actually be a terminal node of the CN, in which case it only has one child.



(a) Candidate network (b) Tree for n_{root}=$S\{k_1\}$

Figure 3.3 A CN and its interpretation as a tree

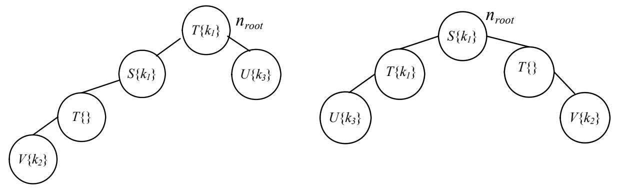
Figure 3.4 shows a simplified version of *CNGen*, assuming that a node contributes to a CN at most once. *InitCNGen* calls *CNGen* for all nodes containing k_1.[5] All these nodes, e.g., $S\{k_1\}$, $S\{k_1,k_2\}$, $T\{k_1\}$, will become roots. In case that several nodes have k_1, the minimal *nid* breaks the tie. Each application of *CNGen* starts with a tree t_{first} containing only n_{root}. For every node n_{new} of the expanded schema that can be added to t_{first}, *CNGen* creates a new tree t_{new} consisting of t_{first} plus n_{new} (Line 7). These trees are then processed similarly (Lines 10-11). Trees exceeding T_{max} do not spawn further trees. This expansion generates all trees rooted at n_{root}. Since every CN must contain k_1, the set of expansions initiated by *InitCNGen* will eventually produce *all* CN for the given query and schema.

```
InitCNGen(Expanded Schema E)
1. For all nodes $n_{root}$ containing $k_1$, ordered by increasing nid
2.    CNGen($n_{root}$)
3.    Remove $n_{root}$ from E

CNGen(expanded schema node $n_{root}$)
1. Initialize queue q     // stores intermediate trees
2. Construct a tree $t_{first}$ consisting of a single node $n_{root}$
3. Insert $t_{first}$ into q
4. While ($q \neq \emptyset$)
5.    Tree $t_{old}$ = q.first
6.    $\forall$ node $n_{new}$ in E that can legally be added to a node $n_{old}$ of $t_{old}$
7.       Create a new tree $t_{new}$ by adding $n_{new}$ as a child of $n_{old}$
8.       If ($t_{new}$ is a CN)
9.          Output $t_{new}$; Break
10.      If ($t_{new}$ has the potential of becoming a CN)
11.         Insert $t_{new}$ in q
```

Figure 3.4 The basic *CNGen* algorithm

CNGen must avoid three types of duplicates. The first are isomorphic duplicates, e.g. Figure 3.5a shows how the CN of Figure 3.3 could be discovered a second time, by calling *CNGen* for $n_{root} = T\{k_1\}$. In order to eliminate this redundancy, *InitCNGen* removes n_{root} from the expanded schema after *CNGen*(n_{root}) terminates. Note that this does not cause any result loss, since all CN containing n_{root} are generated by *CNGen*(n_{root}). The second type of duplicates refers to trees that originate from the same root, but follow different insertion order for the remaining nodes. For instance, the tree in Figure 3.5b is a duplicate of that in Figure 3.3b created by starting with S$\{k_1\}$ and adding $T\{k_1\}$ before $T\{\}$. *CNGen* avoids this problem by creating trees according to their unique pre-order *nid* traversal. The term *legally* in Line 6 means that a node can be inserted only to the rightmost root-to-leaf path, and its *nid* must be larger than any of its siblings'. For instance, assume that the left branch of the tree in Figure 3.5b has already been created. $T\{\}$ could not be added to $S\{k_1\}$ because its *nid* is smaller than that of its sibling $T\{k_1\}$. Thus, this tree is not generated by *CNGen*. The third type of duplicates occurs due to multiple appearances of the same node in a CN. For instance, the replacement of $T\{k_1\}$ by $T\{\}$ in Figure 3.3 would lead to another valid CN with two nodes for $T\{\}$. This CN could be generated twice, by adding either the first, or the second occurrence of $T\{\}$ as the left child of $S\{k_1\}$. Such duplicates can be avoided by observing the lexicographic order of sub-trees rooted at the problematic nodes, but, for simplicity, we omit the details in pseudo-code of Figure 3.4.



(a) Starting with different n_{root} (b) Starting with the same n_{root}

Figure 3.5 Types of duplicates

[4] *Discover* uses an alternative definition of $S\{\}$, where essentially $S\{\} = S$, leading to a different set of CN and, hence, pruning conditions. In this paper, we use $S\{\}$ to signify tuples of *S* that contain no keyword.

[5] The choice of k_1 is arbitrary. Section 3.3 discusses selection of efficient root nodes.

Furthermore, note that we do not have to generate all trees, but only those corresponding to CN. One solution would be to create all trees of size up to T_{max}, and filter out non-CN. *CNGen* behaves more efficiently, by abandoning expansions, as soon as it becomes clear that the current tree cannot lead to a CN (Line 10). A tree has *the potential to become a CN*, if it (i) has fewer than T_{max} nodes, and (ii) every leaf that is *not* on the rightmost path contains a *unique* keyword. Such leaves will not be expanded in the future, and must therefore contain unique keywords; otherwise, they would violate CN minimalism. Since CN generation depends only on the query and the schema graph, but not on the type of underlying data (i.e., it is not specific to stream applications), *CNGen* can also be used to speed up *Discover* or similar R-KWS systems. Furthermore, as we discuss next, the order by which plans are generated by *CNGen* allows optimization through extensive operator sharing.

3.3 Operator Trees and Mesh

Each CN can be mapped to a left-deep operator tree, where leaf nodes are *source* operators that perform selections, and interior operators are *joins*. 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 is the last one added. Figure 3.6 shows the operator tree for the CN in Figure 3.3. Join conditions correspond to parent-child relationships in the CN tree.

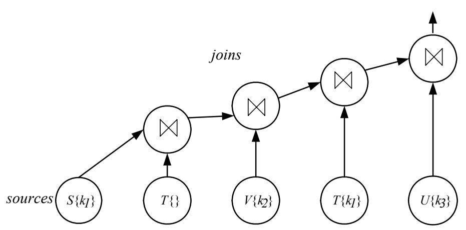


Figure 3.6 Operator tree for the CN of Figure 3.3

Operator trees that contain multiple occurrences of the same node need special treatment, because they could generate invalid results. Assume the CN in Figure 3.3, where T{k_1} has been replaced by *T*{}, resulting in another CN. Its operator tree is similar to Figure 3.6, except that the fourth source from the left is a selection on *T*{}. The tree could produce both outputs of Figure 3.7. The first one (Figure 3.7a) does not constitute a valid result because the same tuple t_1 appears twice (it is not MTJNT since it contains a circle t_1, s_1, t_1). However, we cannot drop the operator tree, because it is needed to produce valid MTJNT such as in Figure 3.7b. To avoid invalid output, join operators in S-KWS applications ensure that when joining a left t_{left} and a right t_{right} tuple, the latter is not already part of t_{left}.

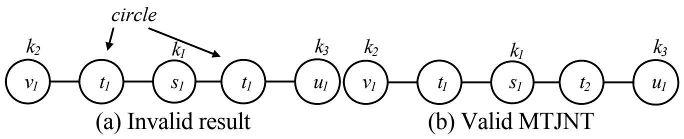


Figure 3.7 An invalid result, and an MTJNT by the same CN

The order in which *CNGen* adds nodes to CN trees is arbitrary, but fixed in advance. For simplicity, in Section 3.2 we used a lexicographic order, but others may enhance performance. Specifically, since all operator trees are left-deep, it is desirable to arrange the sources with the highest selectivity to the left. Hence, we want to 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 arrival rates and distribution of values, 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 by *CNGen*, and hence receive a small *nid*.

The forest of operator trees for all CN answers an S-KWS completely because there exists an operator tree for every possible MTJNT, by which the MTJNT can be produced. Results are output as soon as their youngest tuple arrives; the correct output order is hence preserved. However, executing the operator trees independently would incur very high cost due to their potentially huge number. Recall that this problem also exists in R-KWS, but in a much milder form; if a selection on a relation, (say $S\{k_1\}$) returns no tuples, one can immediately discard all trees (e.g., the one in Figure 3.6) containing the operator. In S-KWS, this is not permissible, because even though the selection $S\{k_1\}$ does not currently produce tuples, it may do so in the future. It is hence crucial to optimize S-KWS operator execution.

The *operator mesh* integrates all operator trees in order to reduce the CPU cost (for evaluating joins) and memory overhead (for intermediate results). The mesh has $|SR|\cdot 2^{|K-1|}$ clusters, where $|SR|$ is the number of stream 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 since they include at least the common n_{root}, but often share larger common parts. Figure 3.8 shows the shared execution of the tree in Figure 3.6 together with three more CN that were created 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 this is only a small subset of all CN in the cluster.

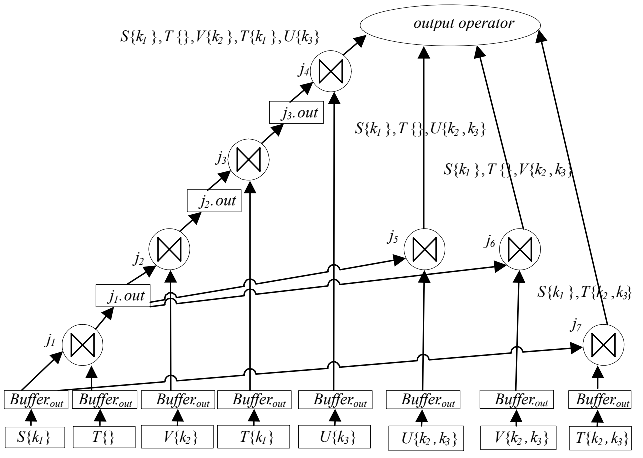


Figure 3.8 Meshed trees for four CN in the same cluster

Mesh creation is performed in parallel with CN generation. Specifically, the first node in a cluster is the root node n_{root}, from which *CNGen* originated. When *CNGen* generates a new tree t_{new} from t_{old} (by inserting 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*, we require a pointer to the corresponding operator *t.op*, in order to decide *where* to place subsequent joins due to expansion of t. The algorithm is initialized with $t_{first}.op$ pointing to the source of n_{root}. For instance, the mesh of Figure 3.8 at first contains only $S\{k_1\}$. When $T\{\}$ is visited by *CNGen*, j_1 is added to the mesh and connected to $S\{k_1\}$ on the left and $T\{\}$ on the 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 $U\{k_2, k_3\}$.

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 $T_{max} + 1$ and 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).

We further integrate operators by sharing buffers. In traditional DSMS, a join operator j has two individual input buffers, *j.left-buffer* and *j.right-buffer*. In our system, these buffers are replaced by the output buffers of the child operators, e.g., in Figure 3.8, j_1*.out* replaces j_5*.left-buffer* and j_6*.left-buffer*. This concept of *state sharing* reduces memory consumption dramatically, since a single operator in the mesh may have thousands of parents. Note that tuples in the buffers are naturally ordered by their t_{start}, i.e., the time instant at which they were produced. More complex indexing schemes are not required, since buffers in S-KWS meshes commonly contain very few tuples, if any.

In summary, this section provides a complete solution for generating candidate networks, which are optimized through the operator mesh. The extension of the proposed techniques to R-KWS systems that are based on similar concepts (e.g., *Discover*) is straightforward and expected to yield significant gains in terms of performance (absence of duplicates, early pruning). Our focus however lies in on stream systems. Compared to R-KWS, S-KWS has three important differences: (i) long running queries that require continuous update of results, as new tuples arrive and old ones expire; (ii) a huge mesh that cannot be trimmed and (iii) potentially changing schema due to incorporation or removal of stream sources. These differences necessitate novel query processing techniques, as discussed in the following section.

4. QUERY PROCESSING

S-KWS meshes are larger and more densely connected than operator graphs in any other data stream application, necessitating massive optimization. However, they also provide beneficial characteristics, in particular: (i) S-KWS meshes have a distinct structure, i.e., clustered left-deep trees, and (ii) their join and selection operators are rather selective. In the following we propose two query processing methods for exploiting these properties. Section 4.1 describes *Full-Mesh* (FM) *S-KWS*, which creates the entire operator mesh at a preprocessing step, so that, at runtime, the system resources are exclusively dedicated to tuple processing. Section 4.2 presents *Partial-Mesh* (PM) *S-KWS* that does not require pre-processing, but dynamically grows and shrinks the operator mesh at runtime. Section 4.3 proposes algorithms for purging dead tuples. Finally, both Full and Partial Mesh can handle changes in the schema as discussed in Section 4.4.

4.1 Full-Mesh

FM generates the operator mesh (as described in Section 3.3) before the actual query processing. The entire mesh is maintained in main memory throughout the lifespan of the query. FM allows optimization by *demand-driven operator execution*, an inter-operator messaging system that eliminates ineffective joins. Specifically, many join operators may execute without any prospect of forming MTJNT, because joins at higher levels lack input from their right child. In Figure 3.8, 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 output from 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 operator has wasted CPU cycles and memory, but not contributed anything 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 pointless, e.g., if j_4 happens to lack right input.

Under demand-driven operator execution, every 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 the operator cannot produce output (case i), or its output would not be consumed (case ii). Figure 4.1 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 must exchange messages regarding their state to ensure that all d and r flags are up-to-date. Particularly, a join operator communicates changes in its state (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.

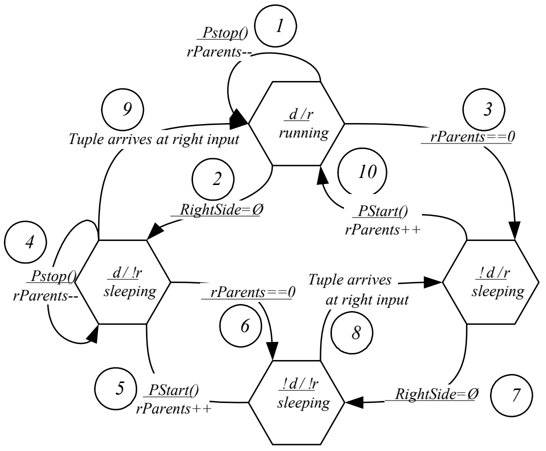


Figure 4.1 States and transitions for join operators

Assume the operator tree in Figure 3.8, 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). Now assume that $T\{k_1\}$ dries up. Consequently, j_3 adjusts its r flag, goes to sleep and informs j_2; j_2 decreases its counter (*rParents* = 0), halts (Transition 3), and calls j_1.*Pstop*. This operator also finds all its parents sleeping, and likewise halts.

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 in increasing order of t_{start}, and joined against those of the opposite input that have a smaller t_{start}. Before receiving tuples, the newly awaking join has to ensure that its left input buffer is up-to-date. After all, the left child may be currently 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 running example, consider that in Figure 3.8 the only sources with output are $S\{k_1\}$, $T\{\}$, $V\{k_2\}$, $U\{k_3\}$ and $T\{k_2, k_3\}$. The only join operators[6] currently running are j_4 and j_7. However, j_4 does not generate results because its left input is empty (since j_3 is sleeping). Now assume that $T\{k_1\}$ begins producing output, causing j_3 to adjust its r flag, wake up (Transition 9), and call j_2.*Pstart*. This operator restarts and informs j_1. Consequently, all joins, with the exception of j_5 and j_6, are running again. Note that demand driven operator execution is not restricted to S-KWS queries; the proposed method can be adapted to arbitrary join trees and benefit other complex data stream applications.

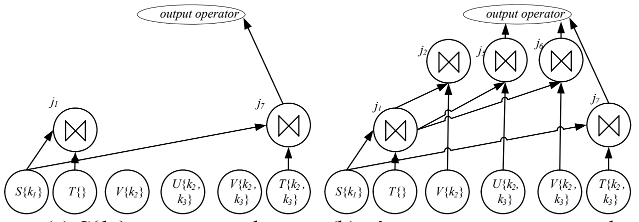
As shown in the experimental evaluation, FM combined with demand driven operator execution is very fast for most problem settings. Furthermore, the overhead of mesh initialization is small, particularly in comparison to the duration of long queries. However, FM has also some drawbacks. First, query processing is delayed until the mesh is complete. For certain applications, this delay may not be acceptable. Second, the size of the mesh can exceed the available main memory, especially if there are multiple active queries in the system. Third, given the potentially huge mesh, for certain settings the computation cost can be high, even with demand driven operator execution. These drawbacks motivate the second approach that adapts the mesh size dynamically according to the stream characteristics.

4.2 Partial-Mesh

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

[6] The output operator is always running.

In the beginning, the dynamic mesh contains only the $|SR|\cdot 2^{|K|}$ sources. Join operators are created later, as tuples travel up the mesh. For our left-deep operator trees, we define that a join operator must be part of the mesh, iff it has left input. Recall that the operator mesh is composed of $|SR|\cdot 2^{|K\text{-}1|}$ clusters, one for each source containing k_1. Figure 4.2 illustrates the generation of part of the cluster in Figure 3.8. 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 4.2a). Joins producing MTJNT (e.g., j_7) do not construct further parents, but connect to the (permanent) central output operator. On the other hand, when j_1 generates results, it creates its own parents, e.g., j_2, j_5 and j_6 (Figure 4.2b). The new parents directly process their first input, e.g. when j_1 outputs its first tuple t and j_2 is formed, j_2 immediately probes t against $T\{\}$. Results trigger the addition of new joins in the mesh (i.e., the parents of j_2).



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

Figure 4.2 Growing a cluster of operators from $S\{k_1\}$

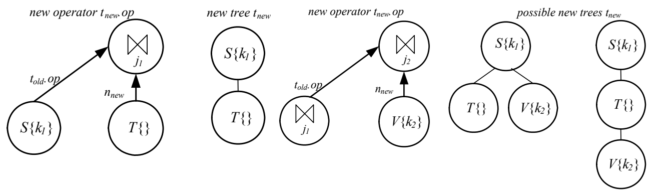
A point that needs clarification is how an operator at an arbitrary level in the mesh determines its direct parents. Recall from Section 3.3 that whenever *CNGen* creates a new tree t_{new} (by adding a node n_{new} to a previous tree t_{old}), a join $t_{new}.op$ is inserted to the operator mesh. The 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} for deciding *which* parents to create. Figure 4.3 illustrates *TreeGen*, an algorithm for reconstructing a tree t_{new}, given its last added operator $t_{new}.op$. The main idea is to check 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.

```
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 $t_{new}$
```

Figure 4.3 The *TreeGen* algorithm

Figure 4.4 explains *TreeGen* by retracing the steps of Figure 4.2. 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 direct parents. The call

TreeGen(j_1) returns the tree t_1 of Figure 4.4a, derived by adding a child $T\{\}$ to the only node $S\{k_1\}$ of t_0. The one-node expansion of t_1 reveals the parents of j_1 (e.g., j_2, j_5, j_6) in the mesh. 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 4.4b). Otherwise ($V\{k_2\}$ is joined with $T\{\}$), t_2 is derived by adding $V\{k_2\}$ to $T\{\}$ (right tree in Figure 4.4b). Note that during the computation of t_2, t_1 must also be reconstructed since intermediate trees are not stored. Keeping all these trees would require a large amount of memory, defeating the purpose of PM.



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

Figure 4.4 Examples of *TreeGen*

Conversely to generating parents, any operator without output destroys its parents, thereby freeing memory. In Figure 4.2b, 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 4.2a. 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. Note that the destruction of parent operators recursively travels up the operator mesh, e.g., if in Figure 4.2b $S\{k_1\}$ dries up, the entire cluster is reduced to its sources.

4.3 Purging Dead Tuples

When a source tuple s is deleted, all join 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. Therefore, we focus on sliding windows, where there are no explicit deletions. In this case, source buffers are ordered by $s.t_{end}$ (since $s.t_{end} = s.t_{start} + w$). These can be purged by simply inspecting the topmost tuples. On the other hand, since the output buffers of joins are not sorted on t_{end} (recall that join results do not expire according to their creation order), deletions may trigger 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.

Eager, illustrated in Figure 4.5, applies a bottom-up method, which resembles the positive-negative approach to deletion. 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 cleans its own output buffer, and informs its parents, should it find expired tuples. *Eager* is memory-optimal since all occurrences of a deleted tuple are removed immediately from all affected operators. However, it can be quite expensive in terms of CPU overhead, 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)
```

Figure 4.5 *Eager* purging

Lazy reduces the CPU cost by removing expired tuples only when they are encountered during join execution. Assume, for instance, that in Figure 3.8 $S\{k_1\}$ 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 probing consists of looping over the buffer and inspecting each tuple for join-ability (nested loop). During the loop, all dead tuples in $j_1.out$ are removed. *Lazy* 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 its dead tuples will continue wasting memory.

For full-mesh query processing, 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. Under demand driven operator execution, j_1 must be sleeping, since all its parents are also sleeping. 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 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. In any case, even if a tuple in the output buffer had expired 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 ($op_{top}.stopTime + w = now$), it is de-queued and its buffer 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. Since removal of outdated buffers is integrated with demand driven operator execution, this optimization is only applicable to FM. *Lazy* purging for the partial mesh cannot have guarantees regarding when an expired tuple is deleted.

4.4 Handling 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 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}. Furthermore, 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. In order to achieve this effect, 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 inserted in l_{rem}. The process terminates when l_{rem} is empty. In contrast to appearances, SR disappearances require no immediate attention and can be performed whenever the system has resources to spare. The above discussion applies to both full and partial mesh approaches. The only difference is that in PM new operators are only created as high as there are data (instead of the entire M_{new}).

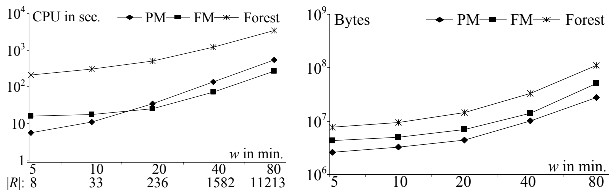
5. EXPERIMENTAL EVALUATION

The proposed algorithms are implemented in C++, following the *Pipes* data stream framework [KS04]. Experiments are performed on a 3.2GHz Dual-Pentium IV with 2 GB of RAM. Due to lack of real datasets, we resort to synthetic data. 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 (its parent and children). An SR has one attribute for each edge, used to evaluate equi-joins with the corresponding neighbor. Results are restricted to at most T_{max} joined tuples. 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* (i.e., the join selectivity). A tuple may contain several different keywords, each with an independent probability *KWF*. We assume a sliding window of *w* minutes, and answer a continuous S-KWS query with |*K*| keywords for the duration of five hours. We investigate peak memory and total CPU as a function of *w*, |*K*|, T_{max}, |*SR*|, *KWF* and *sel*. Table 5.1 illustrates the ranges and the default values (in bold) of these parameters. In each experiment, we vary one parameter and set the remaining ones to their default.

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

Table 5.1 Parameters under investigation

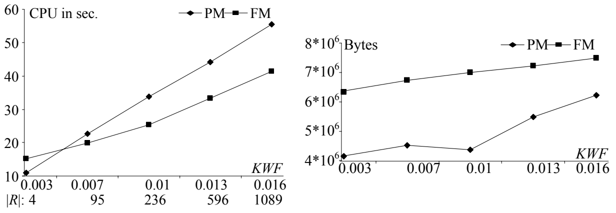
First we compare the (i) Full-Mesh (FM), (ii) Partial-Mesh (PM) and (iii) a "forest" approach that executes the operator trees independently, as a function of the window size *w*. FM includes *lazy* purging combined with demand driven operator execution. Recall that these optimizations are not applicable to PM. Figures 5.1a and 5.1b illustrate the total CPU time (in seconds) and the peak memory consumption (in bytes). The output cardinality |*R*| is shown under the x-axis of the chart for CPU cost. While the number of live tuples grows linearly with *w*, the ways in which they can be joined (i.e., the number of edges in the data graph), grows quadratically. The CPU overhead (for evaluating joins) and space consumption (for intermediate and actual results) reflect this observation. As expected, FM is, generally, the best method in terms of CPU cost, and PM in terms of space. The forest approach is consistently inefficient, and excluded from the remaining experiments for better scaling of the diagrams.



(a) Total CPU cost (b) Peak memory

Figure 5.1 Window size *w*

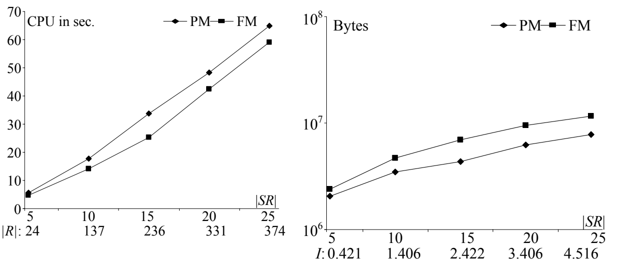
Figure 5.2 studies the effect of keyword frequency KWF. The number of MTJNT, as well as the CPU and memory required for their production increases with KWF. The relative performance of FM and PM is similar to Figure 5.1. The better CPU performance of PM for small values of *KWF* (and *w*) is counter-intuitive since PM has to grow and shrink the mesh at runtime (in addition to processing tuples). As we show in the following experiments, a similar phenomenon exists for highly complex meshes (e.g., large T_{max} and |*K*|). The explanation is that, in these cases, the overhead of the mesh exceeds the actual cost of processing. Specifically, in full meshes, every incoming tuple has to announce itself to all its parent operators. Most of these operators cannot produce output because they lack input from their left child (join). Nevertheless, the looping over all parents (usually several thousand) and the corresponding message exchanges burden the CPU cost.



(a) Total CPU cost (b) Peak memory

Figure 5.2 Keyword frequency *KWF*

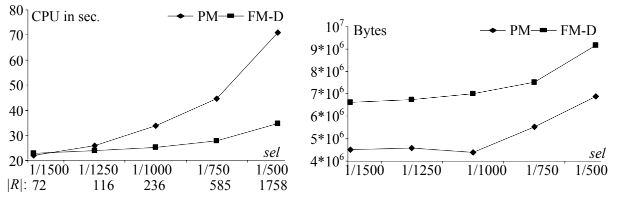
Figure 5.3 shows the cost as a function of the SR cardinality ranging between 5 and 25. Because the number of neighbors for each SR is limited to four, an increasing |*SR*| causes a linearly more complex streaming schema (and corresponding operator mesh), which echoes on the CPU and memory consumption. The duration of the initialization phase (*I*) for FM is depicted below the x-axis of Figure 5.3b (in seconds). As expected, *I* is proportional to the mesh size.



(a) Total CPU cost (b) Peak memory

Figure 5.3 Number of stream relations |*SR*|

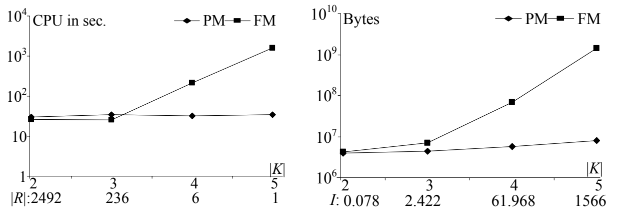
Figure 5.4 investigates the impact of decreasing join selectivity. Raising the likelihood of joining two tuples causes a quadratic increase to the number of edges in a conceptual data graph. This growth is reflected in the number of intermediate results and MTJNT, as well as in the CPU and memory consumption. PM's CPU performance degrades particularly fast because numerous tuples that travel up the operator mesh cause the system to create more join operators. Several of these operators are removed (when they lack input) and re-generated repeatedly through expensive computations.



(a) Total CPU cost (b) Peak memory

Figure 5.4 Join selectivity 1/*sel*

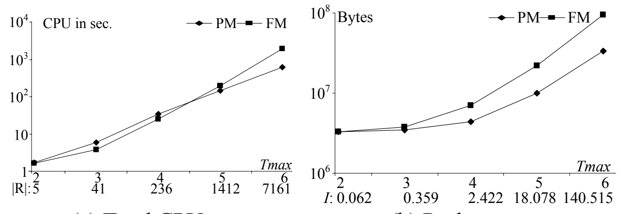
Figure 5.5 depicts the effect of the number of query keywords |*K*|. This parameter has no impact on the number of tuples, or the way they can be joined. However, it causes an exponential growth in the size and complexity of the operator mesh. This phenomenon can be observed from the initialization time to construct the full mesh (see *I* in Figure 5.5b). Three keywords require only 2.5 seconds of initialization, whereas five keywords require almost half an hour. Since most operators in this mesh are commonly idle, they are never created by PM; hence the increasing gap in terms of memory overhead between FM and PM. A similar gap exists also for the CPU cost, as explained in the context of Figure 5.2. Note that queries with more than five keywords are unrealistic for two reasons. First, experience from Web search shows that actual queries rarely exceed four terms. Second, according to our semantics, any increase in |*K*| lowers the likelihood to produce results. For five keywords, we only observe a singe result in five hours.



(a) Total CPU cost (b) Peak memory

Figure 5.5 Number of keywords |*K*|

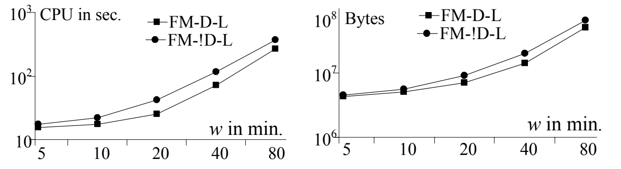
T_{max} has a similar impact to |*K*|: it does not influence the number of tuples or how they can be joined, but leads to an exponential growth of the mesh. Mesh creation for T_{max} = 6 exceeds two minutes, compared to less than one second for T_{max} = 3. Increasing both T_{max} and |*K*| simultaneously can cause mesh initialization to take several hours. Following the mesh size, CPU and memory also grow fast, since (i) the mesh requires more storage, (ii) there are more intermediate results and (iii) their generation requires more CPU. In contrast to |*K*|, increasing T_{max} also causes an exponential growth to the number of results.



(a) Total CPU cost (b) Peak memory

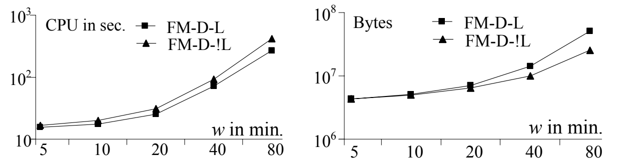
Figure 5.6 Limit T_{max} of nodes per MTJNT

The last set of experiments measures the benefits of demand driven operator execution and *lazy* purging for FM as a function of *w*. FM-D-L signifies that both optimizations are applied. The absence of an optimization is denoted with symbol "!". Specifically, !L implies *eager* purging. As shown in Figure 5.7, demand driven operator execution reduces both the CPU time and the memory consumption. The space savings are due to the avoidance of intermediate results that cannot lead to actual output. On the other hand (Figure 5.8), *lazy* also reduces CPU cost, but increases space consumption because of tuples that remain in the system after their expiration.



(a) Total CPU cost (b) Peak memory

Figure 5.7 Effects of demand driven operator execution (D)



(a) Total CPU cost (b) Peak memory

Figure 5.8 Effects of lazy purging (L)

Summarizing the evaluation, both FM and PM achieve large performance gains with respect to the independent execution of operator trees. FM is faster than PM for most settings. However, it incurs significant space overhead, and may be outperformed by PM for problems that involve highly complex meshes. In addition, a query can be processed only after the mesh initialization completes. Given the above, FM is preferable for queries with long duration (where the initialization cost is amortized) and small meshes (i.e., few keywords, low T_{max}). For all other cases, the method of choice is PM, especially when there are concurrent queries competing for the system memory.

6. CONCLUSION

This paper is the first to propose keyword search on relational data streams. S-KWS has several advantages over structured query languages, most notably, ease of use and ability to retrieve information without knowledge of the schema. At the same time, it presents considerable challenges compared to keyword search for static relational data. In particular, S-KWS is more intricate than R-KWS, because it has to perform additional tasks that are specific to data streams (e.g., handle result expirations) and is at the same time subject to streams' unpredictability and sudden changes. Furthermore, the search space is vast, since all possible combinations of keyword occurrences must be considered. In contrast to R-KWS, this space cannot be pruned, but must be fully monitored during the entire query lifespan.

We face these challenges through a series of contributions. We present the first duplicate-free algorithm that enumerates all possible candidate networks and prunes current expansions at the earliest possible stage. The resulting forest of operator trees answers an S-KWS query correctly, but inefficiently. Going one step further, we integrate the individual trees into a single mesh of shared operators. Finally, we present two highly optimized, query processing techniques. FM builds the full mesh in an initialization phase, so that at runtime system resources are dedicated to tuple processing. PM does not require initialization, but dynamically grows and shrinks the operator mesh at runtime. The relative performance of these techniques is evaluated by an extensive set of experiments.

S-KWS enables an entire class of querying tasks and novel applications. One direction for future work concerns additional functionality in current techniques. In particular, a user may wish to receive only the top-*k* results for any time instant, in which case we should incorporate ranking mechanisms in the query processing methods. Another interesting setting involves long running KWS queries on a combination of stream and static relational data. Going back to the example of Section 1, some of the tables (e.g., *director*) may be stored in the DBMS, while tuples of other tables (e.g., *movie*) arrive continuously from distributed sources. Processing in this environment would require integration of S-KWS and R-KWS methods.

ACKNOWLEDGMENTS

This work was supported by grant HKUST 6184/05E, from Hong Kong RGC. We would like to thank George Fakas for initiating discussions that led to this paper and Vagelis Hristidis for providing useful advice.

REFERENCES

[ABW06] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2): 121–142, 2006.

[ACC+03] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S. B. Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2): 120–139, 2003.

[ACD02] Agrawal, S., Chaudhuri, S., Das, G. DBXplorer: A system for keyword-based search over relational databases. *ICDE*, 2002.

[AH00] Avnur, R., Hellerstein, J. M. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

[BHP04] Balmin, A., Hristidis, V., Papakonstantinou, Y. ObjectRank: Authority-based keyword search in databases. *VLDB*, 2004.

[BBD+02] Babock, B., Babu, S., Datar, M., Motwani, R., Widom, J. Models and Issues in Data Stream Systems, *PODS*, 2002.

[CCD+03] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Raman, V., Reiss, F., Shah, M.A. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *CIDR*, 2003.

[CDHW04] Chaudhuri, S., Das, G., Hristidis, V., Weikum, G. Probabilistic ranking of database query results. *VLDB*, 2004.

[CKKS05] Cohen, S., Kanza, Y., Kimelfeld, B. Sagiv, Y. Interconnection semantics for keyword search in XML. *CIKM*, 2005.

[DEGP98] Dar, S., Entin, G., Geva, S., Palmon, E. DTL's DataSpot: Database exploration using plain language. *VLDB*, 1998.

[FJL+01] Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A., Shasha, D. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record*, 30(2): 115–126, 2001.

[GO03] Golab, L., Öszu, T.M. Issues in Data Stream Management. *SIGMOD Record*, 32(2): 5–14, 2003.

[GSBS03] Guo, L., Shao, F., Botev C., Shanmugasundaram J. XRANK: Ranked keyword search over XML documents. *SIGMOD*, 2003.

[HBN+01] Hulgeri, A., Bhalotia, G., Nakhe, C., Chakrabarti, S., Sudarshan, S. Keyword search in databases. *IEEE Data Engineering Bulletin* 24(3): 22–32, 2001.

[HGP03] Hristidis, V., Gravano, L., Papakonstantinou, Y. Efficient IR-style keyword search over relational databases. *VLDB*, 2003.

[HP02] Hristidis, V., Papakonstantinou, Y. DISCOVER: Keyword search in relational databases. *VLDB*, 2002.

[HPB03] Hristidis, V., Papakonstantinou, Y., Balmin, A. Keyword proximity search on XML graphs. *ICDE*, 2003.

[HVVY06] Hristidis, V., Valdivia, O., Vlachos, M., Yu, P. Continuous Keyword Search on Multiple Text Streams. Poster, *CIKM*, 2006.

[IMS+06] Irmak, U., Mihaylov, S., Suel, T., Ganguly, S., Izmailov, R. Efficient Query Subscription Processing for Prospective Search Engines. *USENIX Annual Technical Conference*, 2006.

[KS04] Kramer, J., Seeger, B. PIPES: a public infrastructure for processing and exploring streams. *SIGMOD*, 2004.

[LYMC06] Liu, F., Yu, C., Meng, W., Chowdhury, A. Effective Keyword Search in Relational Databases. *SIGMOD*, 2006.

[SJ01] Sarda N.L., Jain, A. Mragyati: A system for keyword-based searching in databases. TR *CoRR cs.DB/0110052*, 2001.

[SW05] Su Q., Widom, J. Indexing relational database content offline for efficient keyword-based search. *IDEAS*, 2005.

[WLK04] Wheeldon, R., Levene, M., Keenoy, K. DbSurfer: A search and navigation tool for relational databases. *Annual British National Conference on Databases*, 2004.

[YG99] Yan, T. W., Garcia-Molina, H. The SIFT information dissemination system. *ACM Transactions on Database Systems*, 24(4): 529–565, 1999.