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 only the seven tuples of Figure 1.1. Given the R-KWS query $q := \{\text{Tarantino}, \text{Travolta}\}$, the system returns two results: $t_1 \times t_2 \times t_3$ and $t_1 \times t_6 \times t_7 \times t_8$. 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.

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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Example of relational KWS}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sql_examples.png}
\caption{1.2 SQL statements for two results}
\end{figure}

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.

The high computation and memory overhead is due to 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 that is transparent to the schema and its potential changes. 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 σdirector.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 represents a tuple of S, t one of T, etc. Two tuples (e.g., s1, t1) are connected in G, iff their relations (S, T) are connected in the relational schema and the tuples satisfy the corresponding join conditions. Keywords \( k1, k2, k3 \) are noted next to the tuples in which they occur, e.g., \( k1 \) and \( k2 \) exist in \( v1 \).

The result of an R-KWS query can be defined using the concept of Minimal Total Join Network (MTJNT) [HP02]. A query \( q = \{k1, ..., km\} \) is an MTJNT if it is a connected acyclic component of the data graph G that is (i) total, i.e., it contains all keywords \( k1, ..., km \), 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 \( q = \{k1, k2, k3\} \). Intuitively, only MTJNT are valid R-KWS results. For instance, \( (v1, t1, u1) \) does not constitute an MTJNT since \( u1 \) is redundant; the minimal result is \( (v1, t1, u2) \). Similarly, \( (v2, t2, u2) \) is not total as it does not include keyword \( k1 \).

---

1 For ease of presentation, we often denote an MTJNT as a sequence of nodes.
There are various approaches for R-KWS processing. One family of systems, such as Discover [HP02, HGP03], DBXplorer [ACD02] and Mrugayti [SJO01], 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 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; $K \subseteq \{k_1, \ldots, k_n\}$. 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 $(v_3, t_2, u_3)$, 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 $T_{\text{max}}$ nodes. $T_{\text{max}}$ 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 $\sigma_{k_1 \wedge k_2 \wedge k_3}$ produces all tuples in the sub-relation $V(k_1, k_2)$, whereas $\sigma_{\neg k_1 \wedge k_2 \wedge \neg k_3}$ 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., $V(k_1, k_2)$ and $T\{}$) are shared between trees for several CN (e.g., the first and third in Figure 2.2).

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 $t_f$. 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 minimality.

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, BP04, 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 [ACC03], targeted mainly at processing sensor data, (ii) TelegraphCQ [CCD03], 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 [BBD02, 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 after $w$ 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 $[s.t_{\text{start}}, s.t_{\text{end}}]$ between its arrival $s.t_{\text{start}}$ and (the implicit or explicit) deletion time $s.t_{\text{end}}$. 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 $c.t_{\text{start}} = \max(s.t_{\text{start}}, t.t_{\text{start}})$ and $c.t_{\text{end}} = \min(s.t_{\text{end}}, t.t_{\text{end}})$.

---

2 The concept was introduced [HP02], but had not been named. We use the term expanded schema for easier reference.
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 [HVYY06], 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 [HVYY06] 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 originate are sufficiently correlated. The correlation between streams is not necessarily unique keys; e.g., there may exist two records in the same cash register.

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 \( t_{\text{start}} \), 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 \( <\text{product-id}, \text{price}, \text{time}> \) 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, if 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 each time instant \( \tau \), the instantaneous data graph \( G(\tau) \) contains a node for each tuple \( s \) that is alive at \( \tau \). Two tuples are connected, if 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 \( \tau \). The MTJNT \( (v_1, t_1, u_2, v_3, t_2, u_4) \) and \( (u_1, t_3, s_2, t_4, v_2, u_s) \) 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 \( (u_1, t_3, s_2, t_3, v_2, u_s) \) continues as an element of \( R(\tau = 10) \). Results are produced in ascending \( t_{\text{start}} \) order.

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 \( \tau + 1 \), 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 \( \tau \) also contains \( k \) at time \( \tau + 1 \). Similarly, since edges cannot change, if \( r \) is connected at \( \tau \), it is also connected at \( \tau + 1 \). Hence, if \( r \) is total and minimal at \( \tau \), it is also total and minimal at \( \tau + 1 \). □

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 \( t_{\text{start}} \)), 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: \( t_{\text{end}} = \max(n.t_{\text{start}}) + w \) and \( t_{\text{end}} = \min(n.t_{\text{start}} + w) \), where \( n \) are the component tuples of \( r \). For example, \( (v_1, t_1, u_2) \) in Figure 3.1 is output at \( \tau = 5 \), in combination with the lifespan \( [5, 10) \). In the positive-negative model (where \( n.t_{\text{end}} \) 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](image)

### 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. Furthermore, the algorithm contains elaborate pruning conditions that abort an expansion as soon as it becomes clear that it cannot lead to a new CN.

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., $S[k_3].kbit = 010 = 4$ and $S[k_1, k_3].kbit = 101 = 5$. The node order nid is a lexicographic combination of sid and kbit, e.g., $S[k_1, k_3].nid = T[k_1].nid$ and $S[k_1, k_3].nid < S[k_1, k_3].nid$. 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 nroot 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 $q = S[k_1]$ and the corresponding tree for $n_{root} = S[k_1]$. Note that nroot may actually be a terminal node of the CN, in which case it only has one child.

![Figure 3.3 A CN and its interpretation as a tree](image)

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 $T[j]$. 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. $T[j]$ 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 $T[j]$ in Figure 3.3 would lead to another valid CN with two nodes for $T[j]$. This CN could be generated twice, by adding either the first, or the second occurrence of $T[j]$ as the left child of $S[k_1]$. Such duplicates can be avoided by observing the lexicographic order of sub-trees root at the problematic nodes, but, for simplicity, we omit the details in pseudo-code of Figure 3.4.

---

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 $k_i$ 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 lefmost 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](image)

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 $T_{\{t\}}$, 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 $T_{\{t\}}$. 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](image)

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 $n_{id}$. 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 $|SR|\cdot 2|K|$ 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 $n_{roots}$ 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 $j_{1}(S_{\{k_1\}} \cdot M_{\{t\}})$ is shared by $(S_{\{k_1\}}, T_{\{}}, V_{\{k_2\}}, T_{\{k_1\}}, U_{\{k_1\}}), (S_{\{k_1\}}, T_{\{}}, U_{\{k_2\}, k_3})$ 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](image)
subsequent joins due to expansion of \( t \). The algorithm is initialized with \( t_{\text{stop}} \) pointing to the source of \( t_{\text{root}} \). For instance, the mesh of Figure 3.8 at first contains only \( S \{ k_1 \} \). When \( T \{ \} \) is visited by \( \text{CNGen} \), \( j_1 \) 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, \( \text{CNGen} \) 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 \( S \{ k_1 \} \setminus T \{ \} \)) and source \( U \{ k_2, k_3 \} \).

The entire operator mesh has \( |S| \cdot 2^{|K|} \) leaves/sources, one for each node of the extended schema. The maximum depth of the mesh is \( T_{\text{max}} + 1 \) 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, \( j \text{-left-buffer} \) and \( j \text{-right-buffer} \). In our system, these buffers are replaced by the output buffers of the child operators, e.g., in Figure 3.8, \( j_1 \text{-out} \) replaces \( j \text{-left-buffer} \) and \( j \text{-left-buffer} \). 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 \( t_{\text{start}} \) 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 \( V \{ k_2 \}, U \{ k_3, k_1 \} \) and \( V \{ k_2, k_3 \} \) are empty. None of the joins \( j_2, j_3 \), 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 \( V \{ k_2 \}, U \{ k_3, k_1 \} \) 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 \( j_6 \) 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](image-url)
calling \(j_1.P_{\text{stop}}\). 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 \(j_3\)). When \(V[k_2, k_1]\) stops producing output, \(j_2\) halts, and \(j_3\) 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 (\(r\text{Parents} = 0\)), halts (Transition 3), and calls \(j_1.P_{\text{stop}}\). This operator also finds all its parents sleeping, and likewise halts.

Before going to sleep, an operator sets a local timestamp \(\text{stopTime} = \text{now}\). When it later wakes up, it processes all tuples from its left and right input that are (i) alive and (ii) arrived after \(\text{stopTime}\). To ensure the correct temporal order of results and to avoid duplicates, tuples are processed in increasing order of \(t_{\text{start}}\), and joined against those of the opposite input that have a smaller \(t_{\text{start}}\). 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 \(\text{leftChild}.P_{\text{start}}\), 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 \(S[k_1]\), \(T[j_1]\), \(V[k_1]\), \(U[k_1]\) and \(T[k_2, k_1]\). The only join operators’ currently running are \(j_1\) and \(j_3\). However, \(j_2\) 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 \(j_2.P_{\text{start}}\). This operator restarts and informs \(j_1\). Consequently, all joins, with the exception of \(j_2\) and \(j_3\), 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.

In the beginning, the dynamic mesh contains only the \(|SR|2^{|K|}\) 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 comprised of \(|SR|2^{|K|-1}\) clusters, one for each source containing \(k_j\). 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 \(j_2\), along with others that are not depicted (Figure 4.2a). Joins producing MTJNT (e.g., \(j_2\)) 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., \(j_2\), \(j_3\), and \(j_4\) (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 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 \(\text{CNGen}\) creates a new tree \(t_{\text{new}}\) (by adding a node \(n_{\text{new}}\) to a previous tree \(t_{\text{old}}\)), a join \(t_{\text{new}}.op\) is inserted to the operator mesh. The left input of \(t_{\text{new}}.op\) is \(t_{\text{old}}.op\) and the right one is the source of \(n_{\text{new}}\). In PM the problem is reversed: we have an operator \(t_{\text{new}}.op\), but we need the corresponding tree \(t_{\text{new}}\) for deciding which parents to create. Figure 4.3 illustrates \(\text{TreeGen}\), an algorithm for reconstructing a tree \(t_{\text{new}}\), given its last added operator \(t_{\text{new}}.op\). The main idea is to check the join condition of \(t_{\text{new}}.op\): if \(n_{\text{old}}\) is the source joined with \(n_{\text{new}}\) then \(t_{\text{new}}\) is generated by adding \(n_{\text{new}}\) as the rightmost child of \(n_{\text{old}}\) in \(t_{\text{old}}\).

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

Figure 4.3 The \(\text{TreeGen}\) algorithm

Figure 4.4 explains \(\text{TreeGen}\) by retracing the steps of Figure 4.2. When \(S[k_1]\) produces its first output, \(\text{TreeGen}(S[k_1])\) 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 \(\text{CNGen}(S[k_1])\), i.e., adding nodes to \(t_0\) according to the rules of Section 3.2. Each parent (e.g., \(j_1, j_2\)) is inserted in the mesh and connected to its left and right inputs. Similarly, when \(j_1 = S[k_1].MT[\{}\}\) starts generating results, it has to create the layer of its direct parents. The call

\[\text{TreeGen}(\text{operator } t_{\text{new}}.op)\]

6 The output operator is always running.
TreeGen(\(j_3\)) returns the tree \(t_1\) of Figure 4.4a, derived by adding a child \(T\{\}\) to the only node \(S[k_1]\) of \(j_0\). 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 \(j_2\) joins with \(V[k_3]\) in \(j_5\). If \(V[k_3]\) is joined with \(S[k_1]\), \(t_2\) is derived by adding \(V[k_3]\) as the rightmost child of \(S[k_1]\) in \(t_1\) (left tree in Figure 4.4b). Otherwise (\(V[k_3]\) is joined with \(T\{\}\)), \(t_2\) is derived by adding \(V[k_3]\) 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.

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 \(j_2\) 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 \(s_{\text{start}}\) (since \(s_{\text{start}} = s_{\text{end}} + w\)). These can be purged by simply inspecting the topmost tuples. On the other hand, since the output buffers of joins are not sorted on \(t_{\text{end}}\) (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, \textit{eager} and \textit{lazy}, for removing tuples from the output buffers of join operators.

\textit{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. \textit{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.

\textit{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 \(V[k_3]\), \(U[k_5, k_3]\) or \(V[k_3, k_1]\) arrives, it is probed against \(j_1_{\text{out}}\). The probing consists of looping over the buffer and inspecting each tuple for join-ability (nested loop). During the loop, all dead tuples in \(j_1_{\text{out}}\) are removed. \textit{Lazy} incurs minimal CPU overhead, but provides no guarantee regarding when a dead tuple is removed. If \(V[k_3]\), \(U[k_5, k_3]\) or \(V[k_3, k_1]\) dry up, \(j_1_{\text{out}}\) will not be purged and its dead tuples will continue wasting memory.

For full-mesh query processing, we combine \textit{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 \(j_i\) with output, whose parents \((j_3, j_5, j_6)\) have no right input. Under demand driven operator execution, \(j_i\) 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 \(o\) halts, its output buffer may contain live tuples that cannot be expunged since \(o\) may wake-up soon. However, after \(o\) sleeps for \(w\) seconds, its entire output has expired, and its buffer can be discarded. On the other hand, if \(o\) 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 \(o\) halted, it cannot remain in the system for more than \(2w\) after its expiration.

In order to monitor outdated buffers, \textit{lazy} maintains a doubly linked list \(Q\) of sleeping operators. If an operator \(o\) halts, an entry \(e = \langle op, \text{stopTime}\rangle\) is appended to \(Q\). Additionally, \(o\) keeps a pointer to \(e\). A continuous process watches \(Q\)'s head. When the topmost operator \(o_{\text{top}}\) (the first to halt in \(Q\)) has been sleeping for \(w\) (\(o_{\text{top}, \text{stopTime}} + w = \text{now}\)), it is dequeued and its buffer cleared of all content. Should an operator wake up before it is dequeued, 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. \textit{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 \(S_{\text{new}}\) at time \(t_{\text{new}}\) introduces \(2^k\) new nodes in the expanded schema and produces an equal number of source operators. Let \(M_{\text{old}}\) (\(M_{\text{new}}\)) be the operator mesh before (after) \(t_{\text{new}}\). Directly switching from \(M_{\text{old}}\) to an empty \(M_{\text{new}}\) 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 $t_{rem}$ 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 SRs (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 $[1, 10^5]$. Two tuples of neighboring SRs 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range &amp; Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W$</td>
<td>5, 10, 20, 40, 80 minutes</td>
</tr>
<tr>
<td>$</td>
<td>K</td>
</tr>
<tr>
<td>$T_{max}$</td>
<td>2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>$</td>
<td>SR</td>
</tr>
<tr>
<td>$KWF$</td>
<td>0.003, 0.067, 0.01, 0.013, 0.016</td>
</tr>
<tr>
<td>$</td>
<td>sel</td>
</tr>
</tbody>
</table>

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.

5.2. Keyword Frequency $KWF$

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.

![Figure 5.2](image)

Figure 5.2 Keyword frequency $KWF$
idle, they are never created by PM; hence the increasing gap in seconds of initialization, whereas five keywords require almost mesh (see can be observed from the initialization time to construct the full graph. This growth is reflected in the number of intermediate results and (iii) their generation requires more CPU. In contrast to |K|, increasing T\textsubscript{max} also causes an exponential growth to the number of results.

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.

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\textsubscript{max}). For all other cases, the method of choice is PM, especially when there are concurrent queries competing for the system memory.

Figure 5.3 Number of stream relations |SR|

Figure 5.4 Join selectivity 1/\textit{K}

Figure 5.5 Number of keywords |K|

Figure 5.6 Limit T\textsubscript{max} of nodes per MTJNT

Figure 5.7 Effects of demand driven operator execution (D)

Figure 5.8 Effects of lazy purging (L)
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


