

From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences

Barzan Mozafari Kai Zeng Carlo Zaniolo
Computer Science Department
University of California at Los Angeles, California, USA
{barzan, kzeng, zaniolo}@cs.ucla.edu

ABSTRACT

There is growing interest in query language extensions for pattern matching over event streams and stored database sequences, due to the many important applications that such extensions make possible. The push for such extensions has led DBMS vendors and DSMS venture companies to propose Kleene-closure extensions of SQL standards, building on seminal research that demonstrated the effectiveness and amenability to efficient implementation of such constructs. These extensions, however powerful, suffer from limitations that severely impair their effectiveness in many real-world applications. To overcome these problems, we have designed the K*SQL language and system, based on our investigation of the *nested words*, which are recent models that generalize both words and trees.

K*SQL extends the existing relational sequence languages, and also enables applications from other domains such as genomics, software analysis, and XML processing. At the same time, K*SQL remains extremely efficient, using our powerful optimizations for pattern search over nested words. Furthermore, we show that other sequence languages and XPath can be automatically translated into K*SQL, allowing for K*SQL to be also used as a high-performance query execution back-end for those languages. Therefore, K*SQL is a unifying SQL-based engine for sequence and XML queries, which provides novel optimization techniques for both.

1. INTRODUCTION

There is much interest in extending relational query languages with Kleene-* (Kstar) constructs for matching complex patterns of events in data streams and stored sequences. The power and flexibility of Kstar constructs for SQL, which were introduced in [28, 29], have recently attracted the attention of DBMS vendors and DSMS start-up companies, leading to the recent SQL-MR proposal for their inclusion into the SQL standards [36]. This is hardly surprising, given Kstar's proven effectiveness in application areas as diverse as stock market and auction monitoring [30], publish-subscribe systems [12], RFID-based inventory management [7], click stream analysis [29], and electronic health systems [16, 20]. In financial services, for instance, a brokerage customer may be interested in a sequence of stock trading events that represent a new market trend. In RFID-based tracking and monitoring, applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

may want to track valid paths of shipments and detect anomalies in the supply chains.

In this paper we show that, in spite of the many success stories mentioned above, we have only begun to explore the variety of new applications made possible by the Kstar constructs. We introduce a new language and system, called K*SQL, that reaches well beyond existing proposals to provide:

1. A unifying framework. Many query languages have been proposed, each designed for a different domain. K*SQL is powerful enough to express and support efficiently both set and sequence queries on both relational and XML data, residing in the database or flowing in as a data stream. The many domain-specific languages previously proposed for various combinations of the above retain their validity and desirability in their own application realm, but because of its superior query optimization technology and expressiveness, K*SQL can be used to support and extend them—e.g., XPath 2.0 can be efficiently implemented by a simple translation into equivalent K*SQL queries [24]. Thus, even when programmers prefer to continue to write their XML queries in XPath, they will still benefit from the performance improvement brought by K*SQL as a query execution backend.

Furthermore, K*SQL provides a natural query language for *nested words*—a recently proposed model from the field of formal verification[6], which generalizes both words and tree structures. To the best of our knowledge, this is the first *database query language* proposed for this very powerful and useful data model.

2. More complex patterns. In addition to supporting new data models, the power of K*SQL allows it to match more powerful patterns on standard relational sequences and streams. These are critical in advanced applications, such as stock analysis, RFID processing and trajectory mining. For instance, many real-world sequence queries that involve nested Kstar patterns are not expressible in current languages, such as the proposed SQL-MR standards [36]. Also, when data is embedded in XML, which is a common practice with data exchange over the internet, K*SQL can express sequence queries that are not expressible in XPath 1.0 or 2.0.

We achieve these goals through the following contributions:

1. We study the formal properties of sequence extensions for SQL by incrementally extending our query language to support pattern matching over (i) bounded regular expressions, (ii) regular expressions, and finally (iii) regular expressions over nested words. This methodology allows us to characterize the expressiveness of K*SQL, and compare it to other existing languages (Section 3).

2. Based on our study of expressiveness, we carefully design a natural extension of SQL that provides versatility and ease-of-use, while minimizing syntactic additions (Section 2).

3. We develop extensive optimization techniques for K*SQL, including generalizations of the KMP [19] algorithm to the case

of nested words and visibly push-down words (Section 4).

4. We implement and validate our optimization techniques on well-known benchmarks and real-world data (Section 5).

5. We provide compilation algorithms and tools for automatic translation of several (e.g., XPath, SASE+ [15]) languages into K*SQL, thus allowing for both (i) code-base migrations and (ii) the use of the proposed optimizations also as a *back-end query execution engine* when users prefer those languages as an interface (Section 3.1, Appendix D).

The paper is organized as follows. We briefly introduce the basic syntax of K*SQL through examples in Section 2, followed by a summary of our complexity results in Section 3. We highlight our main algorithms for implementation and optimization of K*SQL in Section 4 which are empirically validated in Section 5. We review the related work and conclude in Sections 6 and 7, respectively.

2. K*SQL BY EXAMPLES

K*SQL extends the syntax of a previous SQL-based sequence language (SQL-TS [28]) with a few but powerful constructs. Thus, we first use a simple example that could also be expressed in most of existing languages, before considering examples involving our extensions. Similar to [36], our pattern extensions are meant to be effective on both DB tables and data streams. So, as our first example, let us consider a DB table containing recent Nasdaq stock transactions (we discuss data streams later):

EXAMPLE 1. *A table with Nasdaq transactions.*

```
CREATE TABLE NasdaqTable (seller Varchar(20), buyer Varchar(20),
    stockName Varchar(8), shares Integer, price Integer,
    datetime Timestamp)
```

Here, `price` is the price per share. As an example, consider the following well-known query from stock market analysis:

EXAMPLE 2 (**Double-bottom or ‘W’ pattern**). *Find those stocks whose price has formed a W-shape. That is, the price has been going down to a local minimum, then rising up to a local maximum and then again, decreasing to another local minimum, and finally, followed by another rise. The starting price should be at least 50. (See Fig. 1).*

```
SELECT S.stockName, D.avg(price) AS runningAvg,
    avg(D.price) AS finalAvg, last(D.price) AS finalPrice
FROM NasdaqTable
    PARTITION BY stockName
    ORDER BY datetime
AS PATTERN (S A* B+ C+ D+)
WHERE S.price >= 50 AND S.price > first(A).price
    AND A.price < prev(A).price
    AND B.price > prev(B).price
    AND C.price < prev(C).price
    AND D.price > prev(D).price AND maximal(D)
```

The above is a typical K*SQL query. The semantics are based on ‘immediately follows’ relationship between ordered tuples. Thus, the syntax is very similar to SQL, except that we have sequential semantics:

- The `PARTITION BY` clause splits the tuples according to their `stockName` value, as if they were separate streams.
- The `ORDER BY` clause defines how the tuples in each partition should be ordered, e.g., in the above example we order the transactions in their chronological order. Similar to SQL, the `DESC` keyword can be added for descending order.
- The `AS PATTERN` clause defines the sequential pattern that we are searching for. In Example 2, `S`, `A`, `B`, `C` and `D` refer to

consecutive tuples. Variable `S` is *singleton* and matches with exactly one tuple, while the other variables are *group variables* (or Kstar variables): `*` allows for arbitrary repetition, while `+` requires a repetition of at least 1. These variable names can be used in the `WHERE` predicates to express the relationship between these tuples.

K*SQL supports both *running aggregates* (e.g. `D.avg(price)`) as well as *final (a.k.a. blocking) aggregates* (e.g., `avg(D.price)`). K*SQL also supports the four typical sequence modifiers, namely `first`, `last`, `prev` and `next` which can be applied to group variables. In K*SQL, `maximal(D)` denotes that we will remain in the `D+` state until this fails—i.e., until the price is no longer increasing. In the absence of the `maximal` predicate, the default behavior is to return all the matches, namely any number of successive occurrences satisfying the predicates.

As mentioned, Example 2 could be expressed in most of the previously proposed languages as well, modulo minor variations in keywords and syntax. K*SQL uses the same constructs for both stored tables and data streams; however, the `ORDER BY` clause will be omitted in continuous queries on data streams where the order follows from the very declaration of the stream, such as that in Example 3. For instance, in our system, an input stream of Nasdaq transactions can be defined as follows:

EXAMPLE 3. *A stream of Nasdaq transactions.*

```
CREATE STREAM Nasdaq (seller Varchar(20), buyer Varchar(20),
    stockName Varchar(8), shares Integer,
    price Integer, datetime Timestamp)
ORDER BY datetime SOURCE 'port4446';
```

In this example `SOURCE 'port4446'` declares the port at which the input data is arriving; `ORDER BY datetime` declares that tuples in our stream are ordered according to their timestamp `datetime`. In the absence of such declaration, the data stream is assumed ordered by its arrival order. But in either case, continuous queries assume and maintain this order, and thus our K*SQL queries over data streams do not contain any explicit `ORDER BY` clause—which will therefore be omitted in the rest of this paper.

Due to space constraints, we only briefly covered the basics of K*SQL syntax so that the reader can follow our examples and understand some of the convenient and more expressive constructs that K*SQL provides¹.

Our next order of business is to allow nested Kstars in the definition of patterns. Although it requires only a minor syntactic extension, nested Kstars significantly improve the usability and expressiveness of our language.

2.1 Nested Kstars

The ‘W-shape’ pattern of Example 2 consists of two ‘V-shape’ patterns. However there are many more complex queries that involve nested Kstars. For instance, consider the following example from stock analysis, known as *uptrend falling wedge* pattern².

EXAMPLE 4 (**Wedge pattern**). *Find those stocks whose price fluctuates as a series of ‘V-shape’ patterns, where in each ‘V’ the range of the fluctuation becomes smaller, and eventually, the price rises up to higher than its starting point.*

Since each ‘V’ sub-pattern is itself two Kstars, say X^+Y^+ , we need to somehow express the *arbitrary number of repetition* of this sub-pattern with a nested Kstar, say $(X^+Y^+)^*$. Next questions are

¹For the formal syntax and semantics of K*SQL see Appendix A and [22].

²<http://www.chartpatterns.com/wedges.htm>

then how to clearly express the complex conditions on such patterns, and how to run them efficiently? The following K*SQL query is the answer to this query.

```
SELECT first(first(Z).X).stockName,
       first(first(Z).X).price AS startPrice,
       E.price AS finalPrice
FROM Nasdaq
  PARTITION BY stockName
  ORDER BY datetime %Optional for streams
  AS PATTERN ((Z: X+ Y+)+ E)
WHERE Z.X.price < prev(Z.X).price
  AND Z.Y.price > prev(Z.Y).price
  AND max(Z.price)-min(Z.price) <
  max(prev(Z).price)-min(prev(Z).price)
  AND first(first(Z).X).price < E.price
```

Here, K*SQL goes beyond SASE+ [15], SQL-TS and SQL-MR by supporting nested Kstars. Next, we briefly explain the new features introduced in the query above.

Aliases. As shown in this Example, K*SQL allows the use of aliases for subpatterns: $(Z : X^+Y^+)$ defines Z as an alias for the sequence X^+Y^+ , one for each ‘V’-phase in the example considered. Now, $(Z : X^+Y^+)^+$ denotes one or more occurrences of Z. We have thus moved from patterns consisting of linear sequences to patterns consisting of sequences of sequences. K*SQL allows for any depth of nested Kstars, e.g. here the depth is 2. Perhaps two main reasons why previous languages did not support nested Kstars were (i) they would lead to ambiguity in the aggregates and, (ii) they require much more complex optimizations. Use of aliases in K*SQL overcomes the former obstacle, as described next.

Aggregates on nested Kstars. K*SQL assumes that each instance of Z has virtual attributes whose values are derived from the instances of X^+ and Y^+ occurring in this instance of Z, e.g. $\min(Z.price)$ is the minimum price among the X’s and Y’s of the current repetition of Z. We could also calculate the running and final averages of the falling prices in the current Z by $Z.avg(X.price)$ and $avg(Z.X.price)$, respectively. Also, $\max(\text{prev}(Z).price)$ refers to the maximum price in the previous repetition of Z.

Similarly, the running aggregate `first` is available on Z, with unchanged semantics, i.e. $Z.first(Y.price)$ denotes the sequence of the rising prices of the first Z, while $first(first(Z).X).price$ returns the price of the first tuple of X in the first Z.

Therefore, the K*SQL syntax for nested Kstars is powerful and unambiguous, and only requires the user to assign a new alias variable to each compound Kstar³, i.e. a Kstar expression consisting of more than one variable. In fact, even though partial optimizations for nested Kstars were proposed in [18], they did not allow aggregates on such constructs due to the ambiguity that such combinations would cause. Thus, K*SQL syntax achieves the unambiguity while allowing aggregates on nested Kstars, mainly through the aliases and the simple semantics introduced above. Efficiency concerns are addressed in Sections 4 and 5.

So far we have only considered relational data, but in practice, many data streams are embedded in XML tags, as XML allows for generality, and usability of data exchange over the Internet. For instance, stock/financial transactions are often encoded and published as XML streams. Thus, a next natural question is ‘whether and how a sequence language can query such data’? And if possible at all, ‘what types of XML data and queries can be expressed in our language’? Next, we answer these questions for K*SQL.

³No alias is required for simple Kstars, since B+ is viewed as equivalent to $(B : _ _ +)$, where $_ _$ is the anonymous variable, as in `Datalog`.

2.2 Linear-Hierarchical Data

Consider the following DTD for an XML schema:

```
<!DOCTYPE company [
<ELEMENT company (name, (transaction)*)>
<ELEMENT transaction (price, buyer, date)>
<ELEMENT name (#PCDATA)>
<ELEMENT price (#PCDATA)>
<ELEMENT buyer (#IDREF)>
<ELEMENT date (#PCDATA)>
]>
```

Throughout this paper, we use SAX-3 [37] representation of XML⁴, a slightly modified version of the famous SAX API. Thus, every XML is processed as a stream of SAX events represented by triplets (type, token, value). The order in which these triplets appear in the sequence reflect their pre-order traversal position in the document. By having a unique tag name for the root element (‘company’ in this example), we can easily extend the same format even to represent a stream of several XML documents with the same schema⁵. The following is the beginning portion of an XML document, within a stream that consists of the XML documents for several Nasdaq companies:

```
(type, token, value)
...
106: ('open', 'company', -),
107: ('open', 'name', -),
108: ('text', 'IBM', -),
109: ('close', 'name', -),
110: ('open', 'transaction', -),
...
```

Here, the numbers represent the relative position of each tag within the stream of company XMLs. Assume that the stock transactions under each company appear according to their date attribute. Thus, a transaction occurred earlier has an open tag with a smaller position number. We begin by searching the same ‘W’-shape pattern as in Example 2, but this time from XML data.

EXAMPLE 5. *The K*SQL query below, returns all the W-shape stocks in a given Nasdaq XML document stream (See Figure 1).*

```
SELECT C.token as CompanyName,
       first(Z.first(X.G.token)) as price1,
       first(Z.last(X.G.token)) as price2,
       first(Z.last(Y.K.token)) as price3,
       last (Z.last(X.G.token)) as price4,
       last (Z.last(Y.K.token)) as price5,
FROM NasdaqStream
AS PATTERN ( A B C D (Z: (X: E F G H I6 J)*
                    (Y: E F K H I6 J)*
                    )2 L
)
WHERE A = open('company')
  AND B = open('name')
  AND D = close('name')
  AND E = open('transaction')
  AND F = open('price')
  AND H = close('price')
  AND J = close('transaction')
  AND X.G.price <= prev(X.G).price
  AND Y.K.price >= prev(Y.K).price
  AND L = close('company')
```

Syntactic shorthands. Note that for XML documents, the tuples are processed according to their appearance order, and hence

⁴Any relational format for pre-order traversal of the stored/streaming XML file(s) is acceptable.

⁵If the document tokens are out-of-order, a fourth column can be used for `documentId`.

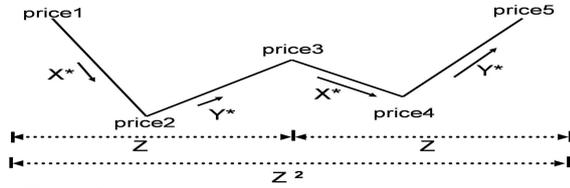


Figure 1: A double-bottom or W-shape stock pattern.

the ORDER BY clause for XML queries is omitted⁶. Here, open() and close() are merely convenient shorthands to recognize open or close tags, e.g. $B = \text{open}(\text{'name'})$ could be replaced by a condition that $B.\text{type} = \text{'open'}$ AND $B.\text{token} = \text{'name'}$. Here, we also used the notation I^6 as a shorthand for the repetition, i.e. $IIIIII$; likewise $(Z : \dots)^2$ stands for $(Z : \dots)(Z : \dots)$. Also observe that, due to their similar definitions, variables E, F, G, H, I and J, are used under both X and Y, and thus we can use the path notation $X.E$ or $Y.E$ to refer to one or the other. When their path notation is missing, the parser disambiguates them by duplicating their predicates for each subpattern that they appear in (see [22]).

Query explanation. In the query above, the first part of the pattern, namely ABCD, parses the $\langle \text{company} \rangle \langle \text{name} \rangle \text{somename} \langle / \text{name} \rangle$ header. Next, we use Z to alias the definition for a 'V'-shaped pattern and use Z^2 to capture a 'W'. In each 'V', the falling and rising phases are defined by X^* and Y^* , respectively. To recognize each occurrence of X, we use four variables, EFGH, recognizing $\langle \text{transaction} \rangle \langle \text{price} \rangle \text{somePrice} \langle / \text{price} \rangle$, which are followed by I^6 , where I's act as wildcards to skip the next six tags, namely $\langle \text{buyer} \rangle \text{Name} \langle / \text{buyer} \rangle \langle \text{date} \rangle \text{someDate} \langle / \text{date} \rangle$, and so on. Here, J and L refer to the corresponding close tags for transaction and company. The rest is obvious (consider the W-shape pattern in Figure 1).

Limitations of other languages. This example illustrates the power of nested Kstars (with aliasing) in K*SQL. Kaghazian et al. [18] also allow nested Kstars but they do not support aggregates on such expressions. However, the hierarchical aliasing in K*SQL allows us to select subsequent occurrences of X and then compare the G prices within and so on. On the other hand, expressing queries such as Example 2 in XPath is often difficult⁷ if not impossible⁸ (e.g., for an extension of XPath with Kstar see [32]).

The next question is whether these constructs (i.e., nested Kstars plus aliasing and aggregates) are also capable of querying XML data with *recursive schemas*?, as a recursive nature can represent a serious challenge for a relational sequence language.

```
<familyroot id="31602">
  <son name="John">
    <son name="Brian">
    </son>
    <son name="Bob">
      <son name="Paul">
      </son>
    </son>
    <daughter name="Alice">
    </daughter>
  </son>
</familyroot>
```

Figure 2: Sample XML document for ancestry information.

XML with recursive schema. Consider the tiny ancestry XML in Figure 2, in which, for example, a son can contain other sons to

⁶The order of appearance of the tags in the XML, is referred to as the total 'document order' in XPath 2.0 and XQuery.

⁷The 'W'-shape query is expressible in XPath but is hard to write, read and optimize (see Appendix E).

⁸For examples, see Appendix E.

an arbitrary depth. Now consider the following example.

EXAMPLE 6. For an ancestry XML (e.g., the one in Figure 2), return the names of all those sons whose father is named 'John'.

Such queries are very easy to write in XPath, here:

```
//son[name = "John"]/son/@name
```

Current Kstar languages cannot express such queries simply because they cannot determine (i) how many intermediate (son)'s they should skip before reaching all the sons of John, and (ii) they cannot detect that, e.g., Paul is Bob's son and not Brian's, say, e.g., by considering their depth in the XML. To overcome these limitations for recursive structures, K*SQL supports a simple but powerful aggregate, called isElement. The following K*SQL query is equivalent to the XPath expression above:

```
SELECT Y.value as sonNames
FROM AncestryRelation
AS PATTERN (A X N* B Y N* C N* D)
WHERE A = open('son')
      AND X.type = 'attribute' AND X.token = 'name'
      AND X.value = 'John' AND isElement(N)
      AND B = open('son')
      AND Y.type = 'attribute' AND Y.token = 'name'
      AND C = close('son') AND D = close('son')
```

In K*SQL, isElement() is a built-in function that is internally implemented using a stack which evaluates to true on every tuple, until a violation of well-nestedness occurs, at which point, it evaluates to false. For the example above, when a new tuple is assigned to state N^* , it is added to the stack if its token is an open tag. But if the new tuple's token is a close tag, we check if the top of the stack is its corresponding open tag. If yes, we pop it, and otherwise there is a stack violation, and the tuple will be passed to the next state, e.g. B. For tokens that are neither open nor close tags, we do not touch the stack but remain in N^* depending on the query mode, e.g., in maximal mode, we stay in N^* until a stack violation occurs, but in all-match mode, we consider all the options non-deterministically. In Appendix C, we explain how, in K*SQL, isElement() is implemented in a generic form (i.e., not limited to XML or its specific SAX representation).

Query explanation. Here, each time a $\langle \text{son} \rangle$ tag is found (element A), the X element checks its name attribute, the N^* elements skip the well-nested elements to ignore the intermediate children of the current node. Since the default setting is non-deterministic, at some point, the automaton will follow the B element instead of N, and if it is another $\langle \text{son} \rangle$ tag, the automaton will proceed with the rest of the pattern. Once all possible traces of this automaton are explored (either success or failure), the first element (i.e., A) will be moved forward until the next $\langle \text{son} \rangle$ is found, and the same process is repeated.

Nested structures other than XML. The capabilities of K*SQL, in querying data with both sequential and hierarchical structures, is not limited and specific to XML. In fact, K*SQL provides for pattern matching over nested words as well as over words. Nested words were originally proposed for static program analysis [4, 6], but can model other dual linear-hierarchical structures as well. XML represents only one example of such data. Procedural programming traces and genomic data are other examples. A brief background on these notions can be found in Appendix B. Interestingly, in Section 3.3, we prove that K*SQL can query any data that can be modeled using nested words (or visibly pushdown words [5], a closely related notion). The examples in this paper were chosen from the XML domain due to the importance of XML and its familiarity to

the database community, and also its long history of rich languages in the field. However, we emphasize that our constructs are not specific to XML or its particular SAX representation. A brief explanation on the application of K*SQL for other domains such as program traces and RNA sequences are provided in Appendix C.

Many interesting questions arise at this point: Can K*SQL express all XPath queries? What is the true expressive power that our built-in `isElement` construct brings to K*SQL? How does K*SQL compare with other existing sequence languages? What is the query evaluation complexity in K*SQL? What if we allow aggregates? How can we optimize K*SQL queries and ensure efficiency? The next two sections address these questions.

3. EXPRESSIVE POWER

In this section, we briefly present our main results on the expressiveness of K*SQL, and compare it to other existing models and languages. The proofs are in Appendices D, F or [22].

3.1 K*SQL vs. XPath

Core XPath 2.0 [33] represents a fragment of XPath that is complete for First Order (FO) logic over trees [34]. In Appendix D, we prove the following theorem:

THEOREM 1. *For every Core XPath 2.0 query, there is an equivalent K*SQL query.*

Moreover, we show later (Theorem 5) that K*SQL is as expressive as VPLs which are equivalent to monadic second order (MSO) logic over nested words [26]. Thus, K*SQL is strictly more expressive than Core XPath 2.0, which is in turn strictly more expressive than Core XPath 1.0 [34]. Appendix E further elaborates on the limited expressivity of XPath for sequence queries.

In Appendix D, we provide a simple constructive proof⁹, that shows we can algorithmically construct an equivalent K*SQL query for any given Core XPath expression¹⁰. We have implemented this translation algorithm as a utility tool with a user-friendly interface [24] to help migrate old XPath code bases to our K*SQL system. Using this tool, loyal XPath programmers can still write their queries in XPath and compile them into K*SQL to benefit from the optimization techniques developed for sequence queries, i.e. K*SQL can also act as an efficient query execution backend for XPath.

3.2 K*SQL vs. Other Sequence Languages

In this section, we study the expressiveness of K*SQL and compare it with other sequence languages. We first disallow aggregates in these query languages, to differentiate between the real power of the language core itself from that brought about by aggregates. In Appendix G, we briefly address the effect of allowing aggregates on the complexity and expressiveness.

While a full query language can return additional information about the matches, in order to simplify the presentation, here we only consider the decision version of these query languages, i.e. the select clause returns a ‘TRUE’ answer when a match is found. Thus, the language membership for a given query is the *decision* of whether the input sequence satisfies the pattern described by the PATTERN construct and the WHERE conditions (details in [22]).

⁹While we could also derive Theorem 1 from Theorem 5, we inductively prove the former in Appendix D, since it gives us a linear-time algorithm for intuitive translation of XPath queries.

¹⁰Our actual implementation [24] also translates extra features that are not part of Core XPath but are included in most industrial implementations of XPath 1.0 and 2.0, e.g. aggregates, arithmetic, etc.

For a query language L , and for a given alphabet Σ , we use $\mathcal{D}(L)$ to denote the class of all the decision problems that can be encoded/expressed as a query written in L running on a sequence of input symbols from Σ .

A hierarchy of constructs. Here, we start from a restricted version of K*SQL, then incrementally add back its main constructs, leading to the following hierarchy of languages: **K*SQL1**: when we don’t allow any of `isElement`, nested Kstars, or query composition; **K*SQL2**: when we allow nested Kstars but no `isElement` or query composition; and finally **K*SQL3**: where both nested Kstars and `isElement` are allowed but no query composition¹¹. This hierarchy has enabled us to (i) analyze the effect of these critical constructs on the usability of the language, (ii) decide on what extensions are needed for expressiveness and which ones are only syntactic sugar or help the optimizer, and finally, (iii) compare with other existing languages while providing insights on a unified approach to querying both words and nested words.

LEMMA 2 (K*SQL1). *Let $A \subseteq \Sigma^*$. The following statements are equivalent:*

1. $A \in \mathcal{D}(K*SQL1)$.
2. $A \in \mathcal{D}(SQL-MR [36])$ without query composition).
3. $A \in \mathcal{D}(SASE+ [15])$ restricted to its ‘strict contiguity’ and ‘partition contiguity’ modes, and without query composition).
4. $A \in \mathcal{D}(Cayuga [11])$.

In our technical report [22] we formalize the class of the languages above using a ‘bounded’ NFA (i.e., contains no loops) where the transitions between the states are labeled with regular formulas. Moreover, SQL-TS [28] becomes a strict subset of $\mathcal{D}(K*SQL1)$, due to the lack of non-determinism in SQL-TS. The SASE+ language runs in different modes for the matching condition: *strict contiguity*, *partition contiguity*, *skip till next match*, and *skip till any match*. The latter two modes increase the expressiveness of SASE+. Using these modes, SASE+ (under query composition) is equivalent to NFA^b [15] which is in turn equivalent to class of regular languages (when the predicates are regular [15, 22]). This, and the following lemma lead us to Theorem 4.

LEMMA 3 (K*SQL2). *Let $A \subseteq \Sigma^*$. The following statements are equivalent:*

1. A is recognizable using a regular expression (RE).
2. $A \in \mathcal{D}(K*SQL2)$.

THEOREM 4. $\mathcal{D}(K*SQL2)$ is equal to $\mathcal{D}(SASE+)$ with query composition).

From SASE+ to K*SQL2. The translation of SASE+ queries into K*SQL2 is simple. The PATTERN and WHERE clauses of SASE+ are analogous to K*SQL2. However, SASE+ supports *skip till next match* and *skip till any match* in its query modes, whereby irrelevant tuples in the middle of a match are skipped. To emulate these modes in K*SQL2, we use wildcards and nested Kstars as follows: every SASE+ pattern X^* is replaced with $(A : B X C)^*$ in K*SQL, where B and C are wildcards, thus allowing arbitrary tuples between consecutive X’s.

In summary, in the absence of aggregates, and once we allow query compositions, from the previous languages, SQL-MR [36] (using all match mode), SASE+ [15] (using its ‘skip till any match’ mode) become equivalent to K*SQL2. Also, Cayuga under query composition is contained in K*SQL2. This containment is strict, if the class of DSPACE[log n] problems are strictly contained in NSPACE[log n]. This is due to Theorem 4 and complexity results from [13], showing that Cayuga is a subset of DSPACE[log n] and can express some complete problems in this class.

¹¹Query composition does not add to the expressiveness of K*SQL [22].

3.3 Monadic Second Order Logic

While the unified support and optimization of sequence and XML queries represent a significant result, that is ready for commercial deployment, even higher level of expressive power and more exciting applications can be envisioned with the approach proposed in this paper. In fact, the expressive power of K*SQL can be formally characterized in terms of a recently proposed model, called Visibly Pushdown Languages (VPL), and thus, K*SQL can query other hierarchical structures besides XML, such as procedural traces and genomic data (e.g., see Appendix C).

Similar to regular languages, VPLs can be recognized by two equivalent representations: Visibly Pushdown Automata (VPA) and Visibly Pushdown Effects (VPEs)[26]. Also, VPLs are equivalent to languages definable in Monadic Second Order logic with a matching relation μ , a.k.a. MSO $_{\mu}$ [5]. For background on VPL and VPE, and the proof of the following theorem see Appendix F.

THEOREM 5 (K*SQL3). *K*SQL3 can express all Visibly Pushdown Expressions, and therefore can recognize all Visibly Pushdown languages and nested words.*

4. OPTIMIZATION

Here we briefly cover some of the core ideas that we have developed for the optimization of K*SQL. Optimization of non-deterministic queries, other run-time optimizations, caching and indexing are discussed in details in [23].

4.1 Compile-time Optimization

At the compile time, we perform two important steps: query rewriting, and pre-calculating several offline matrices which are used by the optimization engine at run-time.

4.1.1 Query Re-writing

The compiler translates the K*SQL query into a special VPA (Visibly Pushdown Automata) where the transitions are made based on the predicates of the WHERE clause, and the states correspond to the pattern variables [23]. The K*SQL parser categorizes the predicates into three types: **Context Free (CF)**, **Running Context Sensitive (RCS)**, and **Final Context Sensitive (FCS)**. In summary, running predicates (i.e., CF and RCS) are preconditions which are assigned to the states, and are evaluated upon examining each tuple for that state, while final predicates (i.e., FCS) are postconditions which are assigned to the outgoing edges, and are examined only upon leaving a state. CF predicates are those predicates whose latest results can be cached in our in-memory history structure (part of the run-time system). For instance, the results of predicates that involve aggregates, are considered context sensitive (they depend on the assignment of more than one tuple), and thus, are not cached.

In a naive implementation, an impossible match with a Kstar may not be detected until the end of the input window, i.e. when the post-conditions are finally checked. To avoid this, we re-write the FCS predicates into an equivalent form, by splitting them into a running part (weaker version) and a final one (the stronger condition). This way, the running part serves as a precondition and prunes many unpromising attempts earlier on, even before the end of the input is reached. For example, $\max(B.price) = 18$ is equivalent to $B.\max(price) \leq 18$ AND $\max(B.price) = 18$ while the first conjunct in the latter form, is RCS and hence, can be checked as a precondition. Analogous rewritings are possible for min and count and even for more complex postconditions involving a combination of these aggregates. Another important case of such query re-writing applies to our nested constructs. For instance, $isElement(B)$ is split into two separate checks: (i) the stack for B must be empty in the end, and (ii) the stack for B must stay valid

at all times. Thus, while $isElement$ is by its nature a context sensitive postcondition, it is translated into FCS and RCS parts. The compile-time optimizer adds all these weaker preconditions to the WHERE clause, in order to optimize the execution.

4.1.2 Off-line Optimization Matrices

K*SQL infers an implication graph [30] from the WHERE clause to capture the implications between different parts of the pattern. In order to optimize the pattern search, several offline tables are pre-calculated which will later guide the pattern search at run-time. We briefly mention the more important ones (P_j refers to the j 'th element of a given pattern P):

- $Jump[j]$: How far should the pattern be shifted to the right, if a mismatch occurs on P_j .
- $Next[j]$: The earliest position in the pattern that we need to check for a match, once we shift the pattern by $Jump[j]$.
- $NETB$ (Not Even Try Before): A table to infer and remember the earliest position before which we should not attempt any matches. This is mainly used for $isElement$ where the distance between an open and its close tag is used to skip many unpromising tuples, as soon as a mismatch occurs.

The calculation of $NETB$ is described in [23] and the first two are similar to [30, 18] (with some corrections).

4.2 Optimization for Nested Constructs

The main construct of K*SQL for querying hierarchical structures, is the $isElement$. The K*SQL system applies several compile-time optimizations for this construct. For run-time optimizations of the nested constructs, we have developed another algorithm, called $VPSearch$ which generalizes the Knuth-Morris-Pratt algorithm [19] to the case of pattern matching over visibly push-down words. We have also designed another algorithm, called *Nested KMP* [23], which applies to nested words. The input¹² is considered a nested word, when it is preprocessed (e.g., indexed or annotated) in such a way that, for every open tag, the position of its corresponding close tag can be found in $O(1)$ time, otherwise it is considered a visibly pushdown word, i.e. we need a stack to parse the matching tags. Trivially, Nested KMP achieves a greater level of optimization than $VPSearch$. Since such preprocessing of XML data is not always feasible (e.g., in data streams or in on-line applications), we only present $VPSearch$ which applies to the more general case where no pre-annotation of the input is assumed (for Nested KMP see [23]).

Assume that we are searching for pattern $P = \langle a \rangle b \langle a \rangle \langle c \rangle b \langle /c \rangle \langle /a \rangle \langle /a \rangle$. Failing to recognize the hierarchical structure, any word search algorithm will consider $\hat{\Sigma} = \{a, \langle a \rangle, \langle /a \rangle, b, \langle b \rangle, \langle /b \rangle, c, \langle c \rangle, \langle /c \rangle\}$ as the alphabet. For instance, KMP [19] or OPS [30] will start scanning the input from left to right, until a mismatch occurs, as shown in the example of Figure 3, where the first failure is when P_4 mismatches with T_4 (step *I*). Using their prefix functions, KMP/OPS shift the pattern by 2 positions, and since $Next[4] = 2$, their next comparison will be between T_4 and P_2 (step *II*). After the second failure, since $Next[2] = 1$, those algorithms compare T_4 with P_1 (step *III*), and only after the third failure, they finally move the input pointer to T_3 .

However, by exploiting the hierarchical structure, we could avoid most of these unnecessary checks. In fact, by analyzing the pattern P , we knew a priori the distance of each open tag from its close tag. For instance, for P_1 , this distance is 7 (since it matches with P_8) while for the second $\langle a \rangle$ this distance is 4. Thus, after the first mismatch in step *I*, we could immediately infer that T_3 is an open

¹²Formally, in a *nested word*[6] the hierarchical structure is explicit, but in a *visibly pushdown word*[5] it is implicit.

i	1	2	3	4	5	6	7	8	9	10	11
T	$\langle a \rangle$	b	$\langle a \rangle$	$\langle /a \rangle$	\dots						
P	$\langle a \rangle$	b	$\langle a \rangle$	$\langle c \rangle$	b	$\langle /c \rangle$	$\langle /a \rangle$	$\langle /a \rangle$			
(I)				\uparrow							
(II)			$\langle a \rangle$	b	$\langle a \rangle$	$\langle c \rangle$	b	$\langle /c \rangle$	$\langle /a \rangle$	$\langle /a \rangle$	
(III)				\uparrow							
				\uparrow							

$$T: \langle a \rangle b \overbrace{\langle a \rangle \langle /a \rangle}^1 \dots$$

$$P: \underbrace{\langle a \rangle b \langle a \rangle \langle c \rangle b \langle /c \rangle \langle /a \rangle \langle /a \rangle}_7$$

Figure 3: KMP versus VPSearch for pattern matching against visibly pushdown words.

tag that closes after 1 tuple, and thus can never match with either of P_1 or P_3 . This would allow us to skip the next two checks (steps II, III) and immediately resume the search from T_7 . Note that KMP/OPS were not able to skip those checks, since they only look at the equality of the symbols but not at the hierarchical edges.

The observation made above is the main idea behind the VPSearch algorithm where we use a 2-dimensional prefix array instead of the KMP’s 1-D prefix. The full description of the algorithm is provided in [23]. In summary, when the implication graph for a given query is complete, the VPSearch achieves the same linear-time optimality for nested words as KMP does for words. The memory complexity is $O(d)$ where d is the maximum depth of the given XML.

5. EXPERIMENTS

The goal of our experiments is to study the amenability of K*SQL queries to efficient execution. Thus, we compare the efficiency of XML queries written in K*SQL to those run on the state-of-the-art XPath/XQuery engines. We also study the effectiveness of our optimization on the execution time, as well as the contribution of each of our optimization techniques to the overall performance.

We have implemented the parser, optimizer and the run-time query execution engine for K*SQL, all in Java. For data I/O and storage, we use the Stream Mill [7] API which is an extensible DSMS, providing access methods for both stored and streaming data.

Experiments were conducted on a 1.6GHz Intel Quad-Core Xeon E5310 Processor running Ubuntu 6.06, with 4GB of RAM. For complex sequence queries we used real-world datasets including world crude oil prices¹³, a year of historical data for the S&P 500 stocks¹⁴ (125K records), and more than 7.6M NASDAQ records¹⁵ since 1970. For XML, we used well-known benchmarks: Protein Sequence Database¹⁶ (600MB, avg depth 5), Shakespeare plays¹⁷ (8MB, avg depth 6) and XMark [31]. Due to lack of space and the similarity of the results, for each experiment we only report the results on one dataset.

5.1 XML queries in K*SQL

We used the XMark benchmark to compare the execution time of their queries on native XML processors, versus the same queries that were run in K*SQL (using our XPath translation algorithm, Appendix D). We compared against two of the fastest academic and industrial engines, MonetDB/XQuery[10] and Zorba [8], respectively. Since these two engines are written in C/C++, we transformed our java bytecodes into binary executables using Excelsior

¹³Official energy statistics of the US government, www.eia.doe.gov

¹⁴<http://biz.swcp.com/stocks/>

¹⁵http://infochimps.org/dataset/stocks_yahoo_NASDAQ

¹⁶<http://www.cs.washington.edu/research/xmldatasets>

¹⁷<http://www.cafeconleche.org/examples/shakespeare>

JET 7.0. (Natively coded C/C++ algorithms are typically much faster than JET generated binaries). (We have conducted more comparisons against other XPath engines that we could obtain, including XSQ [25] and eXist which are reported in [22]).

Out of the 20 XMark XQuery queries, Q1, Q2, Q5, Q13, Q14, Q15 were easily expressible in XPath. In Figure 4(a), we report the total execution time for these queries, on an XMark dataset of size 57MB. We have also run several sequence queries on Nasdaq transactions (embedded in XML tags). For instance, in Figure 4(a), S1 is the ‘V’-shape query (similar to Example 5) that we ran for 20KB of data (the XPath engines could not easily handle larger data, since the XPath query for finding ‘V’ patterns involves several nested joins). In summary, despite the maturity of the research on XPath optimization, K*SQL achieves a very competitive performance on conventional queries, while for sequence queries involving Kstars (such as S1), K*SQL queries are consistently faster than their XPath counterparts, by several orders of magnitude.

5.2 Query Execution Time

Sequence queries written in K*SQL enjoy a high level of efficiency through the proposed optimization techniques. Depending on the query and input, our optimization can improve the execution time of a K*SQL query by several folds. Due to lack of space, here we only report the results for double-bottom (W-shape) query over the NASDAQ dataset, shown in Figure 4(b). The optimized query runs from 1.5x to 6x times faster, and the gap becomes larger as the number of input tuples increases.

5.3 Number of Backtracks

We further evaluated each part of our optimization techniques, in isolation, to gain better insight on their effect on the execution of K*SQL queries. In Figure 4(c), we report the number of backtracks during the execution of the ‘V’-shape query (i.e., A^+B^+), over Nasdaq transactions, embedded in XML format. Here, we only focus on two main parts of K*SQL optimization for XML queries, namely VPSearch and caching—whereby a compact bitmap retains the result of predicate evaluations on the recent tuples. For this query, on average, caching (which itself uses the implication graph) reduced the number of unnecessary backtracks by 55% (compared to the naive implementation). The contribution of VPSearch to the overall performance of this query is limited (i.e., 16%) due to the low depth (i.e., 3) of the XML structure for Nasdaq transactions which only allows for a few tags to be skipped after each mismatch. However, VPSearch combined with the cache structure reduce the backtracks by 70%. More experiments are reported in [22].

6. RELATED WORK

The original SQL-TS language [29, 28, 30], led to the recently proposed extension of SQL standards called SQL Match-Recognize (SQL-MR) [36] which features K*SQL1 kind of constructs. Simple optimizations of nested Kstars were addressed in [18]. The use of these languages in temporal queries was discussed in [35, 17], and further applications were demonstrated in the recent implementation of SQL-MR in [14].

Another major area that benefits from our proposal is Complex Event Processing (CEP), where pattern matching is a means for discovering complex events. The SASE language [1], was designed for CEP over data streams, and was recently extended in SASE+ [15] which provides a special syntax for allowing (i.e., skipping) irrelevant tuples in between those that match a given pattern (see Section 3.2). Another CEP system is Cayuga [11] that comes with a SQL-like language (called CEL) for expressing queries over event streams. CEL has a FOLD operator, that skips an

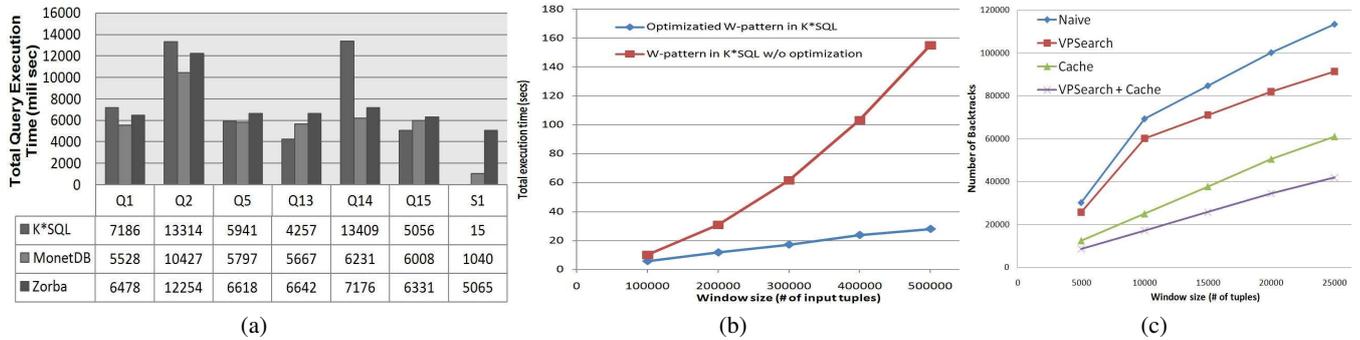


Figure 4: (a) XML queries in K*SQL vs. native XML engines. (b) W-shape pattern in K*SQL: optimized vs. straightforward implementation. (c) Contribution of different parts of the K*SQL optimization on the overall performance.

a-priori unknown number of tuples. However, expressing a pattern with more than one Kstar element requires writing nested queries that are inherently hard to optimize. The patterns expressible in CEL are a subset of those expressible in SQL-MR. The CEDR language [9] also has sequencing operators, but does not support Kstars. A recent system is the Microsoft CEP server[3] which is based on the LINQ language (an extension to .NET, as a built-in query language).

Query automata have been recently proposed [21] for the evaluation of MSO formulas on nested words. The K*SQL system that implements K*SQL language will be demonstrated (as a demo paper) in [24], which also includes its user-friendly interfaces, automatic query translators, and several visualization tools.

7. CONCLUSIONS

In this paper, we propose powerful generalizations for the Kleene-closure constructs that have recently been the focus of much research and commercial interest. Our extensions support more complex pattern queries both on linear sequences and on XML data—in fact the queries supported by XPath are a subset of those supported by our K*SQL language. The paper also introduces powerful query optimization techniques whereby K*SQL can be implemented very efficiently on both relational sequences and hierarchical data such as XML. Having a unified execution engine that efficiently supports different data models and their query languages represents an exciting development for both data bases and data stream management systems. There is also potential for further benefits, given that K*SQL can express Visibly Pushdown Expressions—a powerful generalization of regular expressions that has been successfully applied to software analysis and genomic data. The competitive performance, compared to mature XML technology, achieved by K*SQL is remarkable considering that the latter is still in its infancy and provides greater expressive power than Core XPath 2.0.

8. REFERENCES

- [1] E. W. 0002, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.
- [2] J. Abrashams and et. al. Prediction of RNA secondary structure, including pseudoknotting. *Nucleic Acids Research*, 18(10):3035, 1990.
- [3] M. H. Ali and et. al. Microsoft cep server and online behavioral targeting. *PVLDB*, 2009.
- [4] R. Alur. Marrying words and trees. In *PODS*, 2007.
- [5] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.
- [6] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, 2006.
- [7] Y. Bai, F. Wang, P. Liu, C. Zaniolo, and S. Liu. Rfid data processing with a data stream query language. In *ICDE*, 2007.
- [8] R. Bamford and et. al. Xquery reloaded. *VLDB*, 2009.
- [9] R. S. Barga and et. al. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.
- [10] P. Boncz and et. al. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD*, 2006.
- [11] A. J. Demers and et. al. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.
- [12] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.
- [13] Y. Diao and et. al. SASE+: An Agile Language for Kleene Closure over Event Streams. Technical report, University of Massachusetts, Amherst, 2008.
- [14] N. Dindar and et. al. Dejavu: declarative pattern matching over live and archived streams of events. In *SIGMOD*, 2009.
- [15] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *ICDE*, 2008.
- [16] L. Harada and Y. Hotta. Order checking in a cpoee using event analyzer. In *CIKM*, 2005.
- [17] C. S. Jensen and R. T. Snodgrass. Temporal query languages. In *Temporal Database Entries for the Springer Encyclopedia of Database Systems*, volume TR-90, 2008.
- [18] L. Kaghazian, D. McLeod, and R. Sadri. Scalable complex pattern search in sequential data. In *CIKM*, 2008.
- [19] D. Knuth, J. Morris Jr, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323, 1977.
- [20] M. Liu and et. al. E-cube: Multi-dimensional event sequence processing using concept and pattern hierarchies. In *ICDE*, 2010.
- [21] P. Madhusudan and M. Viswanathan. Query automata for nested words. In *MFCS '09*, pages 561–573, 2009.
- [22] B. Mozafari and C. Zaniolo. K*sql reference: Syntax, semantics and optimizations. Technical report, UCLA, <http://cs.ucla.edu/~barzan/reports/ksql.pdf>, 2010.
- [23] B. Mozafari, K. Zeng, R. Majumdar, and C. Zaniolo. Optimization for Kleene-Closure Queries Based on Visibly Pushdown Automata. under submission.
- [24] B. Mozafari, K. Zeng, and C. Zaniolo. K*sql: A unifying engine for sequence patterns and xml. In *SIGMOD*, 2010.
- [25] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD*, 2003.
- [26] C. Pitcher. Visibly pushdown expression effects for xml stream processing. In *PLAN-X*, 2005.
- [27] A. Potthoff. Modulo-counting quantifiers over finite trees. *Theor. Comput. Sci.*, 126(1), 1994.
- [28] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *PODS*, 2001.
- [29] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. A sequential pattern query language for supporting instant data mining for e-services. In *VLDB*, 2001.
- [30] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *TODS*, 29(2):282–318, 2004.
- [31] A. Schmidt and et. al. Xmark: a benchmark for xml data management. In *VLDB*, 2002.
- [32] B. ten Cate. The expressivity of xpath with transitive closure. In *PODS*, 2006.
- [33] B. ten Cate and M. Marx. Axiomatizing the logical core of xpath 2.0. In *ICDT*, 2007.
- [34] B. ten Cate and M. Marx. Navigational xpath: calculus and algebra. *SIGMOD Record*, 36(2):19–26, 2007.
- [35] C. Zaniolo. Event-oriented data models and query languages in transaction-time databases. In *TIME*, 2009.
- [36] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern matching in sequences of rows. In *[sql change proposal, march 2007]*, <http://asktom.oracle.com/kyte/row-patternrecognition-11-public.pdf> <http://www.sqlsnippets.com/en/topic-12162.html>, 2007.
- [37] X. Zhou, H. Thakkar, and C. Zaniolo. Unifying the processing of xml streams and relational data streams. In *ICDE*, 2006.

APPENDIX

A. K*SQL SYNTAX

The K*SQL syntax extends the `<simple table>` construct of the SQL:2003 standard. The BNF grammar is provided in Figure 5. The definition of several non-terminal symbols, such as `<identifier>` and `<derived column>`, have been omitted from Figure 5, since they are identical to those in the ANSI/ISO standard¹⁸ for SQL:2003.

The syntax for the additional method invocations that K*SQL supports as built-in functions are as follows. Both `open()` and `close()` methods accept an expression of type string as argument and return a string value. The `isElement()` function accepts a `<column reference>` as argument and return a boolean value.

Note that throughout the paper, for clarity purpose, we have used:

```
<pattern element base> = open/close(<expression>)
```

as a shorthand for:

```
<pattern element base> · XmlColName = open/close(<expression>).
```

Similarly, we have used

```
isElement(<pattern element base>)
```

as a shorthand for:

```
isElement(<pattern element base> · XmlColName).
```

The formal semantics and also the full syntax can be found in [22].

B. BACKGROUND ON NESTED WORDS AND VPA

Nested word [4] is a recently proposed notion from the field of automata [5, 6] that can model data with both sequential and hierarchical structures. Common examples include XML, procedural programming traces or even genomic data [5].

Informally¹⁹, in a nested word there is a sequential ordering among all the elements, while there is a secondary, hierarchical structure which is formed by nested edges between some of the elements, i.e., the edges do not cross. In this sense, nested words generalize both words and ordered trees, and allow both word and tree operations. In a nested word, the elements (a.k.a. positions) are divided into three disjoint sets: (i) call elements, where there is an outgoing hierarchical edge, (ii) return elements, where there is an incoming hierarchical edge, and (iii) internal elements that lack any hierarchical edges. A nested word is allowed to have pending edges, that are incoming edges without any call positions or outgoing edges without return positions. A nested word without pending edges is called *well-matched*. This terminology is owed to the software verification literature [6], where a program consists of several nested function calls and returns, while other instructions (internal positions) form the sequential execution. However, nested words can be also used in several other domains. In Figure 6, n_1 is a nested word that represents a portion of an XML document that is not well-matched (it is still a valid nested word, as none of the edges cross). In Figure 6, white circles are internal positions, while blue and black circles represent calls and returns, respectively.

Another appealing area for nested words is genomics. RNA sequences are not simply long strands of nucleotides. Rather, intra-strand base pairing leads to structures such as the one depicted in Figure 6. The covalent chemical bonds between subsequent nucleotides in each strand can be seen as the primary structure, while the hydrogen bonds between the bases (G&C, A&U) form a secondary structure [2]. Since these bonds do not cross, each RNA

sequence can be modeled as a nested word, e.g. n_2 in Figure 6.

Decision properties. Traditionally, dual structures such as XML have been modeled as ordered trees, and thus, have been queried using tree automata. Various classes of automata over nested words have been defined that have higher expressiveness and succinctness compared to word and tree automata [6]; however, their decision complexity and closure properties are analogous to the corresponding word and tree special cases. For example, regular languages of nested words are closed under union, intersection, complementation, concatenation, and Kleene-* [6]; deterministic nested word automata are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [4].

Difference between nested words and visibly pushdown words.

The input to a Nested Word Automaton (NWA) must come as a word with a *parsed nested structure*, i.e., upon seeing a call position we know its corresponding return position and vice versa. However, in many situations the input is given as word and the nested structure yet needs to be parsed/inferred. For instance, given a streaming XML, we do not know the return positions of the calls, at least during the first scan of the data. Thus, to handle such situations, Alur and Madhusudan [5] have proposed Visibly Pushdown Languages (VPL) where a stack is used to store the pending call and return symbols. VPLs are a subclass of context-free languages that are accepted by Visibly Pushdown Automata (VPA). Here again, the alphabet is split into three disjoint sets of Σ_c, Σ_r and Σ_i and upon reading a call symbol ($a \in \Sigma_c$), the VPA *has to push* on the stack, and upon reading $a \in \Sigma_r$ it *has to pop* the stack. For $a \in \Sigma_i$, the VPA *cannot use the stack*.

C. K*SQL FOR OTHER DOMAINS

As briefly mentioned in Section 2.2, the power of K*SQL in querying linear-hierarchical data is not limited to XML, and its `isElement` construct is not dependant on a particular SAX representation. To see the latter, note that `isElement(B)` is used as a shorthand for `isElement(B.myXmlTag)` where we could replace `myXmlTag` with any other column name under which the original xml tags are stored (same applies to `open()` and `close()`). Also, these constructs are not XML-specific: in general, for any domain that can be represented by nested words, the user only needs to re-define the `open()` and `close()` functions, which are, internally invoked by `isElement()`, and thus, we do not need to re-implement `isElement` for every new domain. For instance, in running static analysis over programming traces, the `open()` function detects a *function call*, while the `close()` detects its corresponding *return statement(s)*. Similarly, in RNA sequences (genomics), intra-strand base pairing occurs between guanine (G) and cytosine (C) pair which can be modeled as corresponding open and close symbols, and so can adenine (A) and uracil (U) pair (see [2, 5] for more on the representation of RNAs as nested words.).

D. PROOF OF THEOREM 1 (ALGORITHM)

Here, we provide a simple constructive proof for Theorem 1, that shows we can algorithmically construct an equivalent K*SQL query for any given XPath expression. We have implemented this algorithm as a utility tool for K*SQL system [24].

Our algorithm starts by rewriting the leftmost axis-step into a K*SQL query. Then, at each step, iteratively, the pattern clause of the existing K*SQL query is updated, depending on the type of the current axis specifier. The predicates on the current level of XML nodes are moved to the WHERE clause of K*SQL, while nested expression patterns are independently translated into K*SQL, which then will be intersected with the answer set of the current K*SQL

¹⁸ISO/ANSI Foundation (SQL/Foundation), <http://www.iso.org>.

¹⁹The formal definitions can be found in [4].

$\langle \text{simple table} \rangle$	\leftarrow	$\langle \text{sequence query spec} \rangle \langle \text{query specification} \rangle \langle \text{table value constructor} \rangle \langle \text{explicit table} \rangle$
$\langle \text{sequence query spec} \rangle$	\leftarrow	SELECT $\langle \text{seq select list} \rangle$ $\langle \text{from clause} \rangle$ PARTITION BY $\langle \text{column reference} \rangle$ $\langle \text{order by clause} \rangle$ $\langle \text{pattern clause} \rangle$ $\langle \text{where clause} \rangle$
$\langle \text{seq select list} \rangle$	\leftarrow	$\langle \text{derived column} \rangle [, \langle \text{derived column} \rangle \dots]$
$\langle \text{pattern clause} \rangle$	\leftarrow	AS PATTERN ' (' $\langle \text{pattern} \rangle$ ')'
$\langle \text{pattern} \rangle$	\leftarrow	$\langle \text{atomic pattern} \rangle [\langle \text{pattern} \rangle] \langle \text{compound pattern} \rangle [\langle \text{pattern} \rangle]$
$\langle \text{atomic pattern} \rangle$	\leftarrow	$\langle \text{pattern element} \rangle [\langle \text{pattern repetition} \rangle]$
$\langle \text{compound pattern} \rangle$	\leftarrow	$(\langle \text{pattern element} \rangle : \langle \text{pattern list} \rangle) [\langle \text{pattern repetition} \rangle]$
$\langle \text{pattern element} \rangle$	\leftarrow	$\langle \text{identifier} \rangle$
$\langle \text{pattern repetition} \rangle$	\leftarrow	$+ * \langle \text{unsigned integer} \rangle \{ \langle \text{unsigned integer} \rangle : \langle \text{unsigned integer} \rangle \}$ $ \{ : \langle \text{unsigned integer} \rangle \} \{ \langle \text{unsigned integer} \rangle : \}$
$\langle \text{pattern list} \rangle$	\leftarrow	$\langle \text{pattern} \rangle [\langle \text{pattern list} \rangle]$
$\langle \text{column reference} \rangle$	\leftarrow	$\langle \text{pattern element base} \rangle ' . ' \langle \text{column name} \rangle$
$\langle \text{pattern element base} \rangle$	\leftarrow	$\langle \text{pattern element} \rangle \langle \text{pattern element base} \rangle ' . ' \langle \text{pattern element} \rangle$ $ (\text{PREV} \text{NEXT} \text{FIRST} \text{LAST}) ' (' \langle \text{pattern element base} \rangle ')'$

Figure 5: Formal syntax for K*SQL. The starting rule for K*SQL is $\langle \text{sequence query spec} \rangle$, which extends the $\langle \text{simple table} \rangle$ construct of SQL:2003.

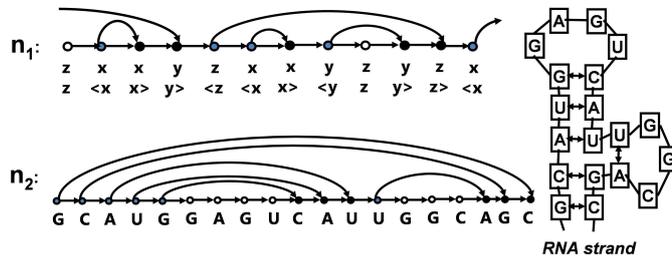


Figure 6: Tiny examples of nested words in different domains: XML and genomics.

Axis	:=	self child parent descendant ancestor following preceding following_sibling preceding_sibling
NameTest	:=	QName *
Step	:=	Axis::NameTest Axis::NameTest[NodeExpr]
PathExpr	:=	Step PathExpr/Step PathExpr union PathExpr PathExpr intersect PathExpr PathExpr except PathExpr
NodeExpr	:=	PathExpr not NodeExpr NodeExpr and NodeExpr NodeExpr or NodeExpr

Figure 7: Syntax of Core XPath 1.0 combined with 2.0.

query.

To focus on the navigational fragment of XPath, in the following, we use the syntax of core XPath 1.0 [34] combined with XPath 2.0 [33]. This syntax is presented in Figure 7. To further simplify the discussion, we also omit the ‘reference’ and ‘for loop’ of XPath 2.0, as they can be trivially emulated in K*SQL using variables and conjunctions, respectively.

In core XPath, the start production is PathExpr. We inductively translate a PathExpr into an equivalent K*SQL query. Whenever the production rule is ‘union’, ‘intersect’, or ‘except’ we rewrite the expression into separate paths, and then inductively, translate each path expression independently; in the end we use respectively union operator $|$, K*SQL intersection, and negation of the predicates to combine the sub-queries. Therefore, we only need to concentrate on the PathExpr/Step production. As our induction hypothesis we assume that we know how to translate the first

$k - 1$ steps of the given expression from the left, into an equivalent K*SQL query, as follows:

```

SELECT  $X_{i_1}, \dots, X_{i_k}$ 
FROM XmlStream
ORDER BY tokenId
AS PATTERN (( $X_1 : A_1 \dots E_j^* \dots (X_2 : A_2 \dots B_2)^{t_2} \dots B_1$ ) $t_1$ 
 $\dots E_{j'}^* \dots (X_3 : A_3 \dots B_3)^{t_3}$ )
WHERE where_clause

```

After translating each step, the pattern clause consists of a list of well-nested elements, i.e. $(X_i : A_i \dots B_i)$ or E_j^* where A_i and B_i are corresponding open/close tags and E_j^* is a well-nested element. The t_i 's denote the occurrence of their element, i.e. whether they are a star element ($t_i = *, +$) or a simple singleton ($t_i = 1$). The select clause outputs a subset of X_i 's, such that the selected tuples are precisely the XML tags that correspond to the XPath expression upto the current Step. For the base case of $k = 1$, the select and where clauses are empty and the pattern clause consists of a simple (E_0^*) . Now, assuming that we have an K*SQL query in the format above that is equivalent to the first (leftmost) $k - 1$ steps of the given PathExpr, we show how to construct a new K*SQL query that is equivalent to the first $k > 0$ steps. Depending on the Axis of the k 'th step, we have the following cases for Axis::NameTest (node filter [NodeExpr] is addressed separately):

self: If the NameTest is a QName, for every X_i in the current select clause, we add the following predicate to the where clause: $A_i = \text{open}(QName)$. When the NameTest is ‘*’ we do not need to change the where clause.

child: For every X_i in the current select clause, we replace it with all of its ‘immediate children’ in the current pattern definition, as follows: an immediate child of X_i is defined as either an $E_j^{t_j}$ or an $(X_j : A_j \dots B_j)^{t_j}$ that appears between A_i and B_i without being enclosed in any sub-patterns of X_i . For the immediate children of X_i that are of form X_j , we just add X_j to the select clause, while for the immediate children of form $E_j^{t_j}$, we first replace them with the new²⁰ pattern $E_{j_1}^* (X_j : A_j E_{j_2}^* B_j)^{t_j} E_{j_3}^*$, and then we add the new X_j to the select clause. Trivially, in the where clause we declare all the new E, A and B variables as isElement, open

²⁰In this proof, whenever we add new variables to the pattern clause we assure that the new variable names are different from the existing names.

and close, respectively. In the end, we remove the original X_i and also duplicate X_j 's from the select clause. The where clause is also updated to reflect the NameTest requirement, similar to the 'self' axis above.

parent: For every X_i in the current select clause, we replace it with its 'immediate parent' in the current pattern definition, as follows: an immediate parent of X_i is defined as the first upper level ($X_j : A_j \cdots B_j$) that encloses X_i . If such X_j does not exist for a given X_i (i.e., when X_i is the root element) we eliminate X_i from the select clause without adding any new variables. Otherwise, we replace X_i with X_j and update the where clause appropriately to reflect the NameTest requirement for A_j . Duplicate X variables are removed from the select clause to avoid identical outputs.

descendant (ancestor): For every X_i in the current select clause, we replace it with its 'descendants' ('ancestors'), as follows: a descendant (ancestor) of X_i is defined as either an $E_j^{t_j}$ or an ($X_j : A_j \cdots B_j$) ^{t_j} (for ancestor, it can be only of form ($X_j : A_j \cdots B_j$) ^{t_j}) that is enclosed between A_i and B_i (for ancestor, X_j should be enclosing X_i definition). For the descendants of X_i that are of form X_j , we just add X_j to the select clause, while for the descendants of form $E_j^{t_j}$, we first replace them with the new pattern $E_{j_1}^*(X_j : A_j E_{j_2}^* B_j)^{t_j} E_{j_3}^*$, and then we add the new X_j to the select clause. (For ancestors, we simply replace X_i with all its ancestor X_j 's.) Trivially, in the where clause we declare all the new E , A and B variables as isElement, open and close, respectively. In the end, we remove the original X_i and also duplicate the X_j 's from the select clause. The where clause is also updated to reflect the NameTest requirement for A_j .

following sibling (preceding sibling): For every X_i in the current select clause, we replace it with its 'next' ('previous'), as follows: next (previous) of X_i is defined as the variable that immediately follows (precedes) the definition of X_i in the pattern clause, and has the same immediate parent as X_i . If such a variable does not exist, we simply remove X_i from the select clause. If the next (previous) is of form X_j , we just add X_j to the select clause, while for variables of form $E_j^{t_j}$, we first replace them with the new pattern ($X_j : A_j E_{j_1}^* B_j$) ^{t_j} (for previous, we replace $E_j^{t_j}$ with $E_{j_2}^{t_j}(X_j : A_j E_{j_1}^* B_j)$), and then we add the new X_j to the select clause. Trivially, in the where clause we declare all the new E , A and B variables as isElement, open and close, respectively. In the end, we remove the original X_i and also duplicate X_j 's from the select clause. The where clause is also updated to reflect the NameTest requirement for A_j .

following (preceding): For every X_i in the current select clause, we replace it with its 'rights' ('lefts'), as follows: right (left) of X_i is defined as any variable that follows (precedes) the definition of X_i in the pattern clause. If no such variable exists, we simply remove X_i from the select clause. If the right (left) is of form X_j , we just add X_j to the select clause, while for variables of form $E_j^{t_j}$, we first replace them with the new pattern $E_{j_1}^{t_j}(X_j : A_j E_{j_2}^* B_j) E_{j_3}^{t_j}$, and then we add the new X_j to the select clause. Trivially, in the where clause we declare all the new E , A and B variables as isElement, open and close, respectively. In the end, we remove the original X_i and also duplicate X_j 's from the select clause. The where clause is also updated to reflect the NameTest requirement for A_j .

Adding node filters. In navigational XPath [34], node expressions are used as node filters, with an existential semantic, i.e. $R[N]$ is the subset of nodes satisfying path expression R from which node expression N evaluates to at least one node. Thus, for translating a path expression $R[N]$, we apply the process above to translate R first, then by appending N to R we have another path expression that can be similarly translated into a separate query

in K*SQL, which then will be added as a conjunct. When the node expression contains 'not' we first negate the pattern (through its where clause) and then add it a conjunct; Similarly, for node expressions with 'or'/'and', we use disjunctive/conjunctive sub-queries, accordingly.

For instance, for translating $R[N_1 \text{ or } N_2]$ we will have:

```
SELECT select_clause_for_R
...
WHERE where_clause AND (
  EXISTS (K*SQL query for R/N1)
  OR EXISTS (K*SQL query for R/N2))
```

E. XPATH FOR SEQUENCE QUERIES

XPath is strictly subsumed by K*SQL. Core XPath 2.0 represents a fragment of XPath that is complete for First Order (FO) logic for trees [34]. From Theorem 5 we know that K*SQL is as expressive as VPLs which are equivalent to monadic second order (MSO) logic over nested words [26]. Thus, K*SQL is strictly more expressive than Core XPath 2.0.

Optimization of sequence queries in XPath/XQuery. While there are MSO queries over XML that cannot be expressed in Core XPath 2.0 (e.g., modulo counting [27] such as returning every 4th tag), and FO queries that cannot be expressed in Core XPath 1.0 (see [34] for an example), in practice, the main deficiency of XQuery and XPath in expressing sequence queries lies in the inevitable complexity of such queries, which compromises their optimization and readability. For instance, consider the following simple sequence query over XML:

EXAMPLE 7. For the following stock data xml, find the decreasing sequences of consecutive close prices, with length at least 1.

```
<Stocks>
<Stock close="0.98"/>
<Stock close="0.95"/>
...
</Stocks>
```

Below is a possible way of writing this query, which clearly exemplifies the limited room for optimizations of such complex queries in XPath/XQuery²¹:

```
<results>{
  for $t1 in doc("mydoc.xml")//Stock
  return <result><head>{$t1/@close}{
    for $t4 in $t1/following-sibling::Stock
    let $x:=(for $x in $t1/following-sibling::Stock
              where $x<<$t4 return $x)
    where $t4/@close<=$t1/@close
    and (every $t2 in $x satisfies
          $t2/@close<=$t1/@close and
          $t2/@close>=$t4/@close)
    and (every $t2 in $x, $t3 in for $x in
          $t2/following-sibling::Stock
          where $x<<$t4 return $x
          satisfies $t2/@close>=$t3/@close
          and $t3/@close>=$t4/@close)
    return <tail>{$t4/@close}</tail>
  }</head></result></results>
```

This situation becomes significantly worse if we want to search for several Kstar patterns. For instance, in [22], we have expressed the 'V'-pattern query (similar to Example 2) in XPath and XQuery using double negations and nested queries resulting in an extremely complex expression which cannot be easily optimized (e.g. see the performance of query S1 in Section 5.1). However, such queries can be easily represented as a regular expression in K*SQL (see Example 5).

²¹None of the available XQuery engines were able to execute this query on any XML document larger than a few kilobytes.

F. FROM VPE TO K*SQL

Background on Visibly Pushdown Expressions. The class of visibly pushdown languages (VPL) has been proposed [5] as embeddings of context-free languages that is rich enough to model data with hierarchical relations (such as XML, software analysis, and RNA) and yet is tractable and robust like the class of regular languages. Visibly pushdown automata (VPA) recognize VPLs, where the input symbol determines when the stack should be pushed or popped.

Pitcher [26] generalized the notion of regular expressions for representing VPLs, called Visibly Pushdown Expressions (VPE). VPEs represent another equivalent notion for VPLs: every VPL can be expressed as a VPE, and every VPE can be translated into a monadic second order logic (MSO) over a nested relation, and there exists a VPA that accepts the same language that that VPE expresses. Below is the formal definition of a VPE:

The symbol patterns used in a VPE are defined as follows (where Σ_c , Σ_r and Σ_i are the set of call, return and internal symbols, respectively):

$p ::=$	a	(symbols, $a \in \Sigma_c \cup \Sigma_r \cup \Sigma_i$)
	$p + p$	(union)
	$\neg p$	(complement)
	\sim_c	(wildcard for Σ_c)
	\sim_r	(wildcard for Σ_r)
	\sim_i	(wildcard for Σ_i)

In the following, we use the abbreviation $p_1 \& p_2$ to denote $\neg(\neg p_1 + \neg p_2)$. Also, P_c refers to all symbol patterns of the form $\sim_c \& p$, and so on. Thus, a well-matched VPE (denoted as T) is defined as:

$T ::=$	ϕ	(empty set)
	$()$	(empty sequence)
	p	(symbol pattern where, $p \in P_i$)
	$p_1[T]p_2$	(element, $p_1 \in P_c, p_2 \in P_r$)
	$T.T$	(concatenation)
	$T + T$	(union)
	$T \& T$	(intersection)
	A	(VPE variable)
	T^*	(repetition)

And finally, below is the grammar for VPEs:

$S ::=$	T	(Well-nested VPE)
	p	(symbol pattern)
	$S.S$	(concatenation)
	$S \oplus S$	(overlapped concatenation)
	$S + S$	(union)
	$S \& S$	(intersection)
	S^*	(repetition)

Here, the \oplus operator insists that the last symbol of the first string is the same as the first symbol of the second string, e.g. $a \oplus a.b = a.b$, but $a \oplus (b + c)$ denotes an empty language. Next, we show how our K*SQL3 language can encode any arbitrary VPE.

F.1 Proof of Theorem 5

PROOF. We prove this by induction, with the base case being the expression of the symbol patterns.

Expressing symbol patterns (SP). An arbitrary symbol a in K*SQL3 is a simple pattern A with a predicate $A = a$. The union of two SPs A and B can be written as $A|B$ in the pattern clause with

the disjunction of their predicates in the where clause. The complement of p is derived by negating the predicates of the K*SQL3 query for p . Wildcards for calls, returns and internal symbols can be encoded using simple checks, e.g. $A = c_1 \text{ OR } \dots \text{ OR } A = c_k$ for all $c_i \in \Sigma_c$ and so on.

Expressing well-matched VPEs. Empty sets and sequences are trivial. SPs $p \in P_i$ are derived by encoding p inductively, and then adding a conjunctive predicate to enforce that all the symbols are internal. For $p_1[T]p_2$, once we recursively encode p_1, T and p_2 , we append their patterns and conjunct their predicates. Note that according to our induction assumption, p_1 and p_2 are guaranteed to be made of open and close tags, i.e. using predicates. Concatenation is encoded by first renaming all the variables such that the two K*SQL3 queries do not share any variables. Then, we append the pattern parts of the queries and conjunct their predicates. Intersection, VPE variables and repetition (a.k.a. Kstar) are directly supported by K*SQL3.

Expressing arbitrary VPEs. T and p can be encoded by our induction assumption. Concatenation, union, intersection and repetition are encoded similarly to their well-matched counterparts. Note that K*SQL3 does not require the pattern to be well-nested, e.g. a check for an open tag does not have to be accompanied by a corresponding check for its close tag. The overlapped concatenation, $S_1 \oplus S_2$, will be encoded as follows. We rename all the variables of the K*SQL3 patterns for S_1 and S_2 , to assure that they do not share any variable names. Assume that the first variable of S_2 is v_2 and the last variable of S_1 is v_1 . We conjunct the predicates of the K*SQL3 queries for S_1 and S_2 , and append their patterns. We then add the following predicate to the resulting K*SQL3 query as a conjunctive term: $\text{last}(v_1) = \text{first}(v_2)$. \square

G. AGGREGATES AND COMPLEXITY

Similar to other practical query languages, K*SQL also allows certain aggregates to appear in the predicates. For instance, SASE+ allows any associative aggregation operation with an identity element and an NC¹ iterated multiplication algorithm. Once we allow the same set of aggregates in K*SQL and all of the languages discussed in Section 3.2, we will achieve similar complexity results.

For instance, the ordered graph reachability problem (called oREACH) can be expressed in a simple SASE+ query (using its ‘skip till any match’ mode) without using any aggregates [13]. Thus, according to Theorem 4, K*SQL2 can also express oREACH which is NSPACE[log n]-complete. However, even after allowing the aforementioned class of aggregates in K*SQL2, the new language will be still contained in NSPACE[log n], since similarly to SASE+, the formulas can be simulated in NC¹, where for the aggregates we perform a partial-prefix computation. Similarly for K*SQL3, after allowing such aggregates, the new language will be still contained in NSPACE[log n] (see [22] for details).

Thus, in summary, both K*SQL2 and K*SQL3, once enhanced with predicates that have aggregate functions discussed above, can express a subset of NSPACE[log n] including some problems that are complete for NSPACE[log n].

ACKNOWLEDGEMENTS

We would like to thank Rupak Majumdar for his insightful comments, Alexander Shkapsky for his help with improving this manuscript, Hetal Thakkar and Yijian Bai for the Stream Mill system, Nikolay Laptev and Hamid Mousavi for their their help on the integration of the K*SQL in Stream Mill, and Vincenzo Russo for the GUI. This work was supported in part by NSF-IIS award 0705345.