K*SQL: A Unifying Engine for Sequence Patterns and XML

Barzan Mozafari Kai Zeng Carlo Zaniolo
Computer Science Department
UCLA
California, USA
{barzan, kzeng, zaniolo}@cs.ucla.edu

ABSTRACT

A strong interest is emerging in SQL extensions for sequence patterns using Kleene-closure expressions. This burst of interest from both the research community and the commercial world is due to the many database and data stream applications made possible by these extensions, including financial services, RFID-based inventory management, and electronic health systems. In this demo, we will present the K*SQL system that represents a major step forward in this area. K*SQL supports a more expressive language that allows for generalized Kleene-closure queries and also achieves the expressive power of the nested word model, which greatly expands the application domain to include XML queries, software trace analysis, and genomics. In this demo, we first introduce the core features of our language in expressing complex pattern queries over both relational and XML data. We overview the architecture of our unifying engine and its user-friendly interfaces. We also present several K*SQL queries from stock market, XML, software trace analysis and genomic applications.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*query processing, relational databases*

General Terms

Languages, Performance, Standardization

Keywords

XPath, SQL, Sequence Queries, Kleene-closure, Pattern Matching

1. INTRODUCTION

There has been much research interest in developing new tools and languages for querying massive collections of data, in order to discover useful patterns in online-user behavior, RFID data processing, asset tracking, weather forecast, fraud detection, and financial data analysis. Several CEP (Complex Event Processing) systems and patterns languages have been proposed [6, 2, 3, 4]. Most pattern languages provide some constructs at least for certain subsets of regular expressions. Several attempts have been made towards supporting pattern expressions in SQL, due to (i) standardization and integration, (ii) appealing characteristics of SQL, e.g. the inherent benefits that come with a relational framework, including its efficiency and amenability to optimization. In this direction, the first proposal, SQL-TS [8], has recently led to the SQL:2003 extension proposal put forth by DBMS vendors and DSMS venture companies, called SQL-MR (SQL Match-Recognize[11]) or Pattern SQL. While this recent SQL change proposal will enable many advanced applications, including temporal queries [10], we argue that by a few minor modifications of the Kstar constructs for SQL, the language will become far more expressive and effective. In fact, we have shown [7] that by simple extensions to SQL, our K*SQL language is at least as expressive as visibly pushdown languages (VPL) and nested words [1]. Nested words and VPLs are recently proposed generalizations of words and tree. These models generalize regular expressions while retaining many of their desirable decision complexity and closure properties [1]. Moreover, the K*SQL language remains highly amenable to efficient implementation for which we have developed special optimization techniques. Nested words have been shown effective also in supporting the search for and the management of data with dual linear-hierarchical structure, such as software programs, genomics, and XML. In particular, we have designed an algorithm to automatically translate all XPath expressions into equivalent K*SQL queries in time linear to the query definition length.

Next, after a brief overview of the K*SQL language, we present an outline of the architecture of K*SQL engine. In Section 4, we provide several examples of advanced application domains of interest that will be used in our demonstration. Finally, we conclude by mentioning the goals and the possible difficulties.

2. A POWERFUL LANGUAGE

Our pattern extensions are meant to be effective on both DB tables and data streams. So, as our first example, let us consider an input stream of RFID readings for parts at certain locations, which can be defined as follows, where along with the ID of the sensor, we store its current temperature, and the type of location the sensor is monitoring, along with the timestamp of the reading.

EXAMPLE 1. *RFID-based monitoring of parts on conveyor belts*

```
CREATE STREAM conbelt (ItemNo  Integer, SensorID  Integer,
                       Temp Integer, LocType char(20),  atTime Timestamp)
ORDER BY atTime, SOURCE 'port4446';
```

In this example `SOURCE 'port4446'` declares the port at which the input data is arriving; `ORDER BY atTime` declares that tuples in our stream are ordered according to their timestamp `atTime`. Thus, the following K*SQL query

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

EXAMPLE 2. *Identifying Cycles*

```
SELECT  A.ItemNo, A.SensorID,
        last(B.atTime) - A.atTime, count(B.*)
FROM conbelt
PARTITION BY  ItemNo
ORDER BY  atTime
AS PATTERN  (A  B+ C)
WHERE  B.SensorID <> A.SensorID  AND  maximal(B)
```

detects items that go around in cycles several times, and reports the location and the length of such cycles.

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 `ItemNo` 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 example above we order the tuples by their chronological order. Similar to SQL, the `DESC` keyword can be added for descending order. This clause is required for stored tables, while for continuous queries over data streams the `ORDER BY` clause will be omitted in where the order follows from the very declaration of the stream, such as that in Example 1.
- The `AS PATTERN` clause defines the sequential pattern that we are searching for. In Example 2, A, B and C refer to *consecutive* tuples. Variable `B` is said to be a *group variable*, i.e., it can match with more than one tuple, while `A`, `C` are called *singleton*. These variable names can be used in the `WHERE` predicates to express the relationships between the matched tuples.

Here, maximal(B) denotes that we will remain in the B+ state until this fails–i.e., until the input satisfies the condition that its SensorID is the same as the SensorID of A. C.SensorID = A.SensorID. In the absence of the maximal predicate, the default behavior is to return all the matches, namely any number of occurrence for any of the stars in the pattern as long as all the given predicates are satisfied.

Running Aggregates. In addition to the actual columns in the tuples, K*SQL provides virtual columns containing the value of continuous (cumulative) aggregates for group variables. Thus, in Example 1, `B.avg(Temp)`, and `B.max(Temp)` would denote the current running average value and max value of `Temp` for the `B`$+$ tuples, up and including the current one. Moreover, `B.count()` is the running count of the tuples in `B+`; so, if the current tuple is the j^{th} tuple, then `B.count()` $=$ `j`.

Final/Blocking Aggregates. In addition to continuous running aggregates, the more traditional blocking aggregates of SQL are available on group variables. Thus, `avg(B.Temp)` and `max(B.Temp)` denote the average and max computed on all the tuples in `B+`, while `count(B.`$*$`)` denote total count of these tuples. It is important to observe that a function such as `avg(B.Temp)` is blocking, with respect to the input sequence, while `B.avg(Temp)` is nonblocking. Moreover, in this second function, `B` is a free variable (as per λ Calculus), while there is no free variable in `count(B.`$*$`)`. The differences between these two kinds of aggregates are clear from a syntactic viewpoint, and significant in terms of semantics, implementation and optimization of our patterns.

The WHERE Clause. The `WHERE` clause contains simple predicates, possibly combined by logical connectives . While singleton variables can be used freely, simple predicates can at most contain one free group variable (multiple occurrences of the same free group variable are fine). The reason for this restriction becomes obvious once we search for a pattern such as $\ldots$ `A`$^+ \ldots$ `B`$^* \ldots$ in a sequence of tuples containing the numeric attribute `myatt`. Then, the simple predicate `A.myatt` $<$ `B.myatt` is ambiguous since it is not clear whether this inequality is to be satisfied for all pairs of tuples in `A` and `B`, or for some pairs of tuples in `A` and `B`, or for some tuple in `A` and all tuples in `B`, or vice-versa. To avoid these problems *at most one free group variable* is allowed in any simple predicate, and this limitation extends to the continuous virtual aggregates associated with the group variables. Thus, `A.count()` $<$ `B.count()`, `A.avg(myatt)` $<$ `B.sum(myatt)` or `A.sum(myatt)` $<$ `B.myatt` are all disallowed. On the other hand, `A.avg(myatt)` $<$ `B.myatt` or `B.first(myatt)` $<=$ `A.myatt` each contains only one free group variable and are therefore fine.

Traditional blocking aggregates produce no free variables, and therefore predicates such as `sum(A.myatt)` $<$ `sum(B.myatt)`, `count(A.`$*$`)` $<$ `count(B.`$*$`)`, `sum(A.myatt)` $<$ `B.sum(myatt)`, and `last(A.myatt)` $<$ `B.myatt` are all allowed (the first two have no free group variables, while the third and the fourth have one each).

The formal syntax and semantics of K*SQL can be found in [7].

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

Figure 1: Sample XML document for ancestry information.

2.1 Querying XML in K*SQL

Consider the tiny ancestry XML in Figure 1, in which, for example, *son*s can contain other *son*s to an arbitrary depth. In K*SQL system, we use any relational pre-order traversal of the XML tree, such as the SAX-3 [5] representation which is a slightly modified version of the famous SAX API: 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 reflects their pre-order traversal position in the document. The following is a portion of an XML document, within a stream that consists of the XML documents for several ancestries:

```
        ...
106: ('open', 'son', -),
107: ('attribute', 'name', 'Bob'),
108: ('open', 'son', -),
109: ('attribute', 'name', 'Paul'),
110: ('close', 'son', -),
        ...
```

Here, the numbers represent the relative position of each tag within the stream of XML tags. Now given and ancestry in XML format, we have the following XPath query:

EXAMPLE 3. *Return the names of all the sons of 'John'*

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

While these queries are easily expressed in XPath, current sequence languages, such as SQL-TS or SQL-MR, cannot express (i) how many intermediate sons should skipped before reaching the son or its parent, and (ii) match a closing tag with its corresponding open one. To overcome these limitations for recursive structures, K*SQL supports a simple but powerful new constructs called `isElement`, whereby the previous query can now be expressed as follows:

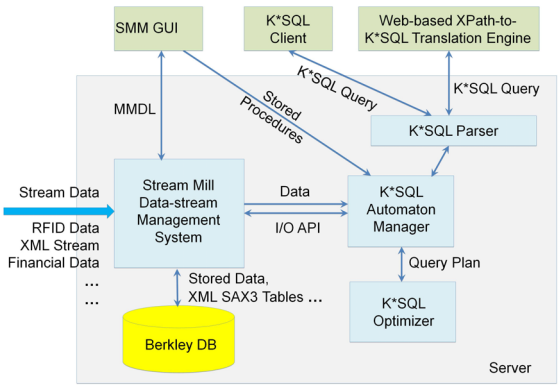

Figure 2: Architecture Overview.

```
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. We will shortly explain how in K*SQL `isElement()` is implemented in a generic form, and is not limited to XML or its specific SAX representation.

Query explanation. Here, each time a $\langle son \rangle$ tag is found (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 C element instead of N, and if it is another $\langle son \rangle$ 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 $\langle son \rangle$ is found, and so on.

3. ARCHITECTURE OVERVIEW

The high-level architecture of K*SQL system is depicted in Figure 2. The K*SQL users can submit their queries via three different interfaces: (i) a web-based client that automatically translates XPath queries into equivalent K*SQL ones, and submits the translated queries to the server, shown in Figure 3. (ii) K*SQL queries can also come in through our client shell, or (iii) through the SMM interface that allows users to write high-level workflows using a library of stored procedures, as shown in Figure 5.

Our K*SQL server has three major modules that have integrated into an extensible DSMS, called Stream Mill Miner (SMM [9]). K*SQL queries once parsed, are turned into raw query plans which are further optimized using our K*SQL query optimizer (we have generalized search optimization techniques such as KMP and OPS to cope with nested-hierarchical data model). The final query plan is an automaton which is executed by the K*SQL automaton manager. K*SQL relies on SMM (which in turn uses Berkely DB) which provides I/O methods for both stored and streaming data.

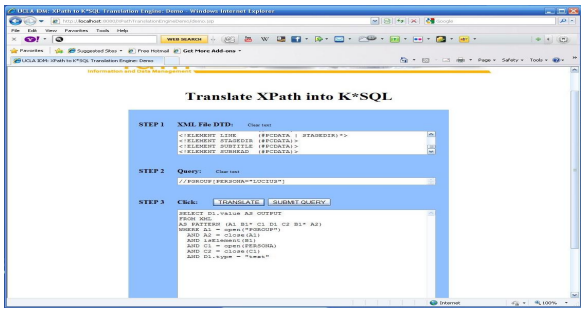

Figure 3: Web-based client for translation of XPath queries into K*SQL.

Stream Mill can handle several data streams at a time, including XML stream. Both stored and streaming XML documents are represented in SAX3 format, both for efficiency and simplicity.

4. DEMONSTRATION DETAILS

4.1 Advanced Applications

Stored Financial Data Analysis.

We will use stored NASDAQ data to allow our demonstration audience to issue their queries to find patterns of their interest. For instance, the following query can detect the so-called 'downward wedge pattern'[1], which is a well-known trend in finance and is often interpreted as a bullish implication for future.

EXAMPLE 4. *Downward Wedge Pattern*

```
SELECT first(first(X).A).price AS startPrice,
            last(last(X).B).price AS endPrice
FROM NasDaqLogs
     PARTITION BY ticker
     ORDER BY date
     AS PATTERN ( (X: A  B+)+)
WHERE
     A.price <= B.price AND maximal(A) AND
     X.first(A.price) > prev(X).first(A.price) AND
     B.price >= B.price AND
     X.first(B.price) > prev(X).first(B.price) AND
     X.max(A maximal(X) AND
     X.first(A.price) - X.first(B.price) >
     prev(X).first(B.price) - prev(X).first(A.price)
```

XML Queries.

All XPath queries can be translated into equivalent K*SQL queries and benefit from our efficient sequence query optimization, as briefly explained in Section 2. K*SQL can also express complex sequence queries over XML. In this demonstration, users will experience simplicity and power of K*SQL via the first-hand experience of writing both set and sequence queries over XML data.

Genomic Data.

Another appealing area for nested words is genomics. For instance, RNA sequences are not simply long strands of nucleotides. Rather, intra-strand base pairing leads to structures such as the one depicted in Figure 4. The covalent chemical bonds between subsequent nucleotides in each strand can be seen as the primary structure, while the hydrogen bonds between the bases form a secondary structure. Since these bonds do not cross, each RNA sequence can

[1] `http://www.chartpatterns.com/`

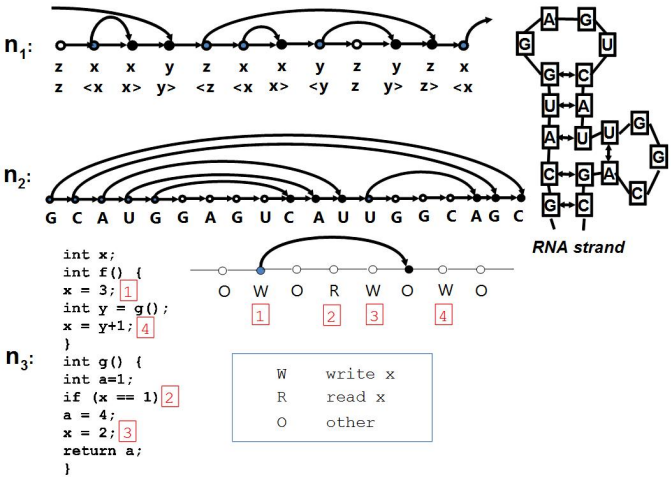

Figure 4: Tiny examples of nested words in different domains: XML, genomics and software analysis.

be modelled as a nested word, e.g. n_2 in the tiny example of Figure 4. In our demonstration we will provide both human and bacterial RNA sequences (from the famous Noncoding RNA database) to allow our audience to interact with K*SQL and visualize the result of their RNA queries to further appreciate the importance of K*SQL for yet another application domain.

Software Analysis.

The initial motivation for nested words has come from the software verification literature [1]. A procedural program consists of several nested function calls and returns, while other instructions (internal positions) form the sequential execution. For instance, given a large corpus of C++ code, with function calls and return positions (or read and write instructions), the programmer can verify several pre and post conditions on function parameters upon calling a function or on their return, and ensure many invariants, such as maximum stack depth at any time, code reachability, and many others,

4.2 Ease-of-Use and Performance

We will allow our audience to write their own sequence and XML queries, using our several user-friendly interfaces that provide code assistance. First, for users who are more familiar with XPath than with a sequence query language, we will allow them to use our web-based client (see Fig. 3) that translates arbitrary XPath queries into equivalent K*SQL that can be efficiently executed on our server. Second, our more experienced audience can use the K*SQL client shell to directly write K*SQL queries and see the visualization of the query output. Finally, our third interface is the SMM [9] GUI that allows users to write high level work-flows, which will use our built-in library of stored procedures, as shown in Fig. 5. Moreover, we will have other native XPath engines so that our visitors can compare the performance of K*SQL on their translated queries with those run on ad-hoc XML databases. Thus, they can experience first-hand that using K*SQL, as a unifying engine with more powerful queries does not come at the price of efficiency.

5. CONCLUSION

The K*SQL system presented here supports a powerful and unifying new language for expressing complex sequence patterns over both relational and XML; K*SQL is effective on both stored and

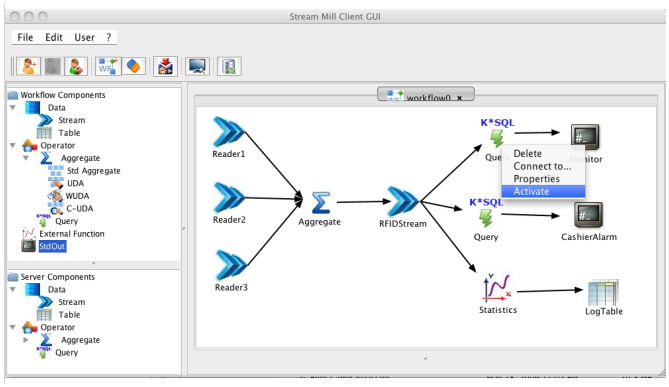

Figure 5: Defining SMM work-flows with K*SQL queries.

streaming data. Since it is highly amenable to optimization, K*SQL not only serves as a unifying framework for all these combinations, but can be also used as an efficient query execution backend, whereby XPath queries can be translated and run in K*SQL. We have devised several user-friendly interfaces including a tool for automatic translation of XPath queries into our K*SQL language, so that the users can gain first-hand experience at writing queries in K*SQL. We will present several real-world examples drawn from different applications, such as genomics, XML, stocks, and software analysis.

Acknowledgements: We would like to thank Yijian Bai and Hetal Thakkar for the Stream Mill system, Vincenzo Russo for the GUI, Nikolay Laptev and Hamid Mousavi for their invaluable help in extending Stream Mill for K*SQL. This work was supported by NSF-IIS award 0742267.

6. REFERENCES

[1] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, 2006.

[2] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[3] M. H. A. et. al. Microsoft cep server and online behavioral targeting. *PVLDB*, 2009.

[4] N. D. et. al. Dejavu: declarative pattern matching over live and archived streams of events. In *SIGMOD*, 2009.

[5] X. Z. et. al. Unifying the processing of xml streams and relational data streams. In *ICDE*, 2006.

[6] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *ICDE*, 2008.

[7] B. Mozafari and C. Zaniolo. K*sql reference: Syntax, semantics and optimizations (ucla technical report), 2009.

[8] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *PODS*, 2001.

[9] H. Thakkar, B. Mozafari, and C. Zaniolo. A data stream mining system. In *ICDM*, 2008.

[10] C. Zaniolo. Event-oriented data models and query languages in transaction-time databases. In *TIME*, 2009.

[11] 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/tkyte/row-patternrecogniton-11-public.pdf http://www.sqlsnippets.com/en/topic-12162.html*, 2007.