Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data†

(Extended Abstract)

Seymour Ginsburg and *Xiaoyang Wang*
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{ginsburg,xywang}@pollux.usc.edu

Abstract

A family of sequence operations (rs-operations), based on pattern matching and including most of the "natural" operations on sequences, is introduced. In order to apply rs-operations to calculus-like query languages, a logic about sequences (SL) is defined by converting rs-operations to special predicates. To illustrate the applicability of our concepts to database queries, rs-operations and SL are used in an algebra and a calculus, respectively, over an extended relational data model containing sequences.

1 Introduction

It is generally accepted that sequences (or lists) are useful in many database applications [4, 14, 15]. Because of this, "new-generation" database systems, e.g., EXODUS [6], Galileo [3], O_2 [5, 8] and Vbase [12], usually support "sequenced data." In order to query the sequenced data, these systems require appropriate sequence operations. However, the sequence operations in the systems are usually chosen in an *ad hoc* manner. Also, the essential properties of the selected operations, such as "expressiveness," "completeness" and "independence," are not well understood. A major underlying cause for this situation may be the lack of a unifying theoretical mechanism for defining and studying most, if not all, of the desired operations. The purpose of this paper is to show that "pattern matching" of a simple kind can be used as such a mechanism to specify and investigate most of the "natural" sequence operations.

Sequence operations in database query languages tend to be "high-level" in nature (in contrast to those in programming languages such as C, Lisp and Prolog). For example, in the EXCESS algebra of EXODUS [17], HEAD and SUBARRAY are employed to obtain the head and a subinterval of a sequence, respectively. Such operations specify results of "processes" rather than the processes themselves. The specification of the results of such high-level sequence operations can be viewed as a type of "pattern matching." For instance, let u be a sequence of length at least 1. Clearly, one of the sequences in the regular set§ $\alpha_1\alpha_2^*$, say $\alpha_1\alpha_2^k$, is of the same length as u. Therefore, $\alpha_1\alpha_2^k$ "matches" u and α_1 "matches" the first element of u. Thus, $\alpha_1\alpha_2^*$ can be "used" to retrieve (by "pointing at" α_1) the first element of u, i.e., the HEAD operation on u.

The above pattern matching mechanism can be found in text editors "vi" and "emacs," and the AWK programming language [2] (usually in UNIX systems). Both editors and AWK use regular expressions in their search-and-substitute commands. In these commands, a pair of special symbols retrieve a portion of the matched sequence. The sequence operations introduced in this paper are similar to, but more powerful than, this "retrieving" mechanism.

Pattern matching is used extensively in text processing, e.g., [1], and (although in a different manner) in information retrieval systems, e.g., [11]. One paper employing pattern matching in database queries is [13], where regular patterns serve as "maskings" in an extended NF^2 query language to deal with sequences (as well as sets). However, the roles of pattern matching in

†This research was supported in part by the National Science Foundation under the grants CCR-8618907 and IRI-8920930 and the Air Force Office of Scientific Research under the grant 89-0244.

11th Principles of Database Systems/6/92/San Diego, CA
© 1992 ACM 0-89791-520-8/92/0006/0293...$1.50

§We assume that the reader is familiar with the standard concepts related to regular sets [10].

these systems are as conditions rather than operations.

In this paper, a formal treatment of sequence operations, based on pattern matching, is initiated. It is intended as a unified approach towards specifying and studying operations on sequenced data. As a first step, a family of operations is defined based on a simple pattern matching mechanism with regular sets as "pattern languages." This family (i) includes most of the "natural" sequence operations and (ii) is easy to extend. Also, the family can be characterized by a type of mechanical device called "generic a-transducer" (described in the full paper).

Many database systems provide two ways of querying the stored data: one through an algebraic language and the other through a calculus-like language. The calculus provides a high-level user interface while the algebra gives a procedure interpretation of the queries expressed in the calculus. It turns out that rs-operations can be used not only as algebraic operations (in algebraic query languages) but also as predicates over sequences (in calculus-like query languages). To illustrate this, a logic system (SL) using rs-operations is presented. The use of rs-operations in algebraic query languages and calculus-like query languages (through SL) is exemplified in an extended relational data model.

The current presentation is an extended abstract of the full paper. The rest of this extended abstract is arranged as follows. In Section 2, rs-operations are defined and their properties exhibited. In Section 3, a sequence logic (SL) is introduced. To illustrate the use of the rs-operations and SL in database queries, an extended relational data model with sequences is introduced in Section 4. Also, an algebraic and a calculus-like query language are presented on the data model.

2 Rs-operations

We start by defining some preliminary notions. A *sequence* v of length $n > 0$ over a nonempty set A of elements is a mapping from $\{1, \ldots, n\}$ to A, and is usually written as $a_1 \cdots a_n$ where $a_i = v(i)$ for each $1 \leq i \leq n$. The symbol ε represents the sequence of length 0 (i.e., the *empty sequence*). Sequences are denoted by u, v and w etc., possibly with subscripts. Given a sequence v, $|v|$ denotes the length of v. For each set A of elements, let $A^* = \{v | v(i) \in A \text{ for all } 1 \leq i \leq |v|\} \cup \{\varepsilon\}$.

We now informally describe a "merging process." Suppose we have a sequence $w = \alpha_1\alpha_1\alpha_2\alpha_1\alpha_2$ (called a *pattern*) of special symbols α_1 and α_2. Intuitively, this sequence gives the name "α_1" to positions 1, 2 and 4, and "α_2" to positions 3 and 5. Now let u_1 and u_2 be two sequences. A sequence u is a "merging" of u_1 and u_2 according to w if the subsequence of u formed by the elements at the α_i-positions is u_i for $i = 1, 2$. Thus, $u = abcde$ is a merging of abd and ce according to w since the elements of u at the α_1 positions (i.e., positions 1, 2 and 4) is abd, and the elements at the α_2-positions (i.e., positions 3 and 5) is ce. Figure 1 below illustrates the above merging.

In order to formally define the above merging process, we assume a fixed infinite alphabet $\mathbf{\Sigma}_\infty$ (whose elements are denoted by a, b, etc., possibly subscripted) and a fixed, countably infinite set of special symbols $\mathbf{V}_\infty = \{\alpha_i | i \geq 1\}$. For each $n \geq 1$, we will use V_n to denote the set consisting of the first n elements of $\mathbf{V}_\infty$, i.e., $V_n = \{\alpha_1, \ldots, \alpha_n\}$. We also define a special mapping on sequences. For all sequences w in $\mathbf{V}_\infty^*$ and u in $\mathbf{\Sigma}_\infty^*$, with $|w| = |v|$, and element α in $\mathbf{V}_\infty$, let $\pi_2\sigma_{\$1=\alpha}(w \otimes u) = u(i_1) \cdots u(i_k)$, where $1 \leq i_1 \leq \cdots \leq i_k \leq |w|$, $w(i_j) = \alpha$ for each $1 \leq j \leq k$ and $w(i) \neq \alpha$ for each i in $\{1, \ldots, |w|\} - \{i_1, \ldots, i_k\}$.

We are now able to formally define the notion of a merger.

Definition Let $n \geq 1$ be a positive integer and W a subset of V_n^*. Then the construct $[\![W]\!]_n$ is called an *n-ary (sequence) merger*. For subsets $L_1, \ldots, L_n$ of $\mathbf{\Sigma}_\infty^*$, let $[\![W]\!]_n(L_1, \ldots, L_n) =$

$$\{u \in \mathbf{\Sigma}_\infty^* | \exists w \in W \forall 1 \leq i \leq n (\pi_2\sigma_{\$1=\alpha_i}(w \otimes u) \in L_i)\}.$$

The mapping defined thereby is called an *(n-ary) merger mapping.*

As an example, let $W = (\alpha_1\alpha_2)^*$, $L_1 = \{ab, abcd\}$ and $L_2 = \{cd, ef, cdef\}$. Then $u = acbd$ is in $[\![W]\!]_2(L_1, L_2)$ since ab and cd are in L_1 and L_2, respectively, and u is the merging of ab and cd according to $\alpha_1\alpha_2\alpha_1\alpha_2$ in W. Similarly, it is easy to see that $aebf$, $acbdcedf$ are also in $[\![W]\!]_2(L_1, L_2)$. Therefore, $[\![W]\!]_2(L_1, L_2) = \{acbd, aebf, acbdcedf\}$, i.e., the set consisting of the "perfect" shuffles of the sequences from L_1 and L_2.

Now consider the "inverse" of a merger.

Definition Let $[\![W]\!]_n$ be an n-ary merger and $1 \leq i \leq n$. Then the construct $[\![W]\!]_n^{-1}$ is called a (*sequence*)

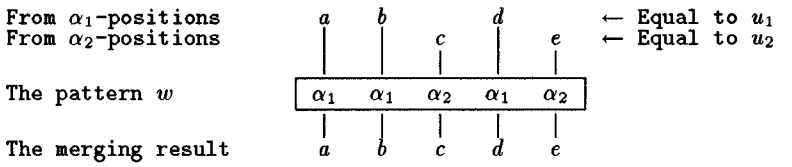

Figure 1: The merging of abd and ce according to $\alpha_1\alpha_1\alpha_2\alpha_1\alpha_2$.

extractor. For each subset L of Σ^*_∞, let $[\![W]\!]_n^{-1}(L) =$

$$\{u_1 | \exists u \in L \exists u_2, \ldots, u_n (u \in [\![W]\!]_n(\{u_1\}, \ldots, \{u_n\}))\}.$$

The mapping defined thereby is called an *extractor mapping*.

To illustrate, let $W = \alpha_1\alpha_2^*$ and $L = \{abc, defgh\}$. Then $[\![W]\!]_2^{-1}(L) = \{a, d\}$, i.e, the set consisting of the first element of the given sequences.

Each sequence merger and extractor defined above consists of an arbitrary set W of patterns. Obviously, the mergers and extractors thus defined are very powerful, and may be hard to compute and/or represent. For practical purposes, the set W should be tractable. In this paper, W will be restricted to the "regular sets." There are two major reasons for this: Regular sets describe most of the natural patterns encountered in practice; and one of their representations, namely "regular expressions," is easy to use in query languages. We now formally define the central notion of the paper.

Definition A *regular sequence operation*, or *rs-operation*, is either a merger $[\![W]\!]_n$ or an extractor $[\![W]\!]_n^{-1}$, where W is regular.

Henceforth, all mergers and extractors are assumed to be rs-operations.

We now present several examples of rs-operations and their compositions.

Examples In the following, u and v are assumed to be sequences in Σ^*_∞.

(1) $[\![\alpha_1^*\alpha_2^*]\!]_2(\{u\}, \{v\}) = \{uv\}$.

(2) $[\![\alpha_1^k\alpha_2^*]\!]_2^{-1}(\{u\})$ is the prefix of u of length k, $[\![\alpha_2^*\alpha_1^k]\!]_2^{-1}(\{u\})$ is the suffix of u of length k, and $[\![\alpha_1^*\alpha_2^*]\!]_2^{-1}(\{u\})$ is the set of all prefixes of u.

(3) $[\![(\alpha_1 \cup \alpha_2)^*]\!]_2^{-1}(\{u\})$ is the set of all subsequences of u.

(4) Let $\mathrm{SL}(u, v) = [\![(\alpha_1\alpha_2)^*]\!]_2^{-1}([\![(\alpha_1\alpha_2)^*]\!]_2(\{u\}, \{v\}))$. Obviously, $\mathrm{SL}(u, v) = \{u\}$ if $|u| = |v|$, and $\mathrm{SL}(u, v) = \emptyset$ otherwise.

(5) Let $\mathrm{Half}(u) = \mathrm{SL}(\mathrm{Prefix}(u), [\![(\alpha_1\alpha_2)^*]\!]_2^{-1}(\{u\}))$, where $\mathrm{Prefix}(u) = [\![\alpha_1^*\alpha_2^*]\!]_2^{-1}(\{u\})$. It is easily seen that $\mathrm{Half}(u)$ returns the first half of u if u is of even length, and $\mathrm{Half}(u)=\emptyset$ if u is of odd length. For example, $\mathrm{Half}(abcd) = \{ab\}$.

Using the above examples, it is easy to see that all the sequence operations defined in [9] can be simulated by our rs-operations.

Since regular sets are defined by "mechanical devices" (i.e., finite state automata), it is natural to seek some similar devices to describe the rs-operations. Indeed, a type of transducers, called "generic a-transducers," is used in the full paper to characterize rs-operations. Because of the length limitation, generic a-transducers are omitted in this extended abstract.

It is shown in the full paper that the set of mergers (extractors, respectively) are closed under composition. The question arises as to whether there exists a finite subset of mergers (extractors and rs-operations, resp.) which yields all merger (extractors and rs-operations, resp.). In order to answer these questions, the decomposition of the rs-operations is studied.

A merger (extractor, resp.) is said to be *decomposable* if it is equivalent to a composition of some other mergers (extractors, resp.). The first result is

Theorem 1 *For each $n \geq 3$, there exists an n-ary merger which is not decomposable.*

Proof. (*sketch*) For each $n \geq 3$, let

$$W_n = (\alpha_1\alpha_2 \cup \alpha_2\alpha_3 \cup \cdots \cup \alpha_{n-1}\alpha_n)^*.$$

It can be shown (proof omitted) that $[\![W_n]\!]_n$ is not decomposable for each $n \geq 3$. □

A corollary of the above theorem is

Theorem 2 *There is no finite set of mergers which yields all mergers by composition.*

The decomposability of extractors is more complicated because we need to get rid of the "trivial" decompositions. First, we have the following notions.

For each subset W of $\mathbf{V}_\infty^*$ and $k \geq 1$, let

$$W_{<k} = \{w \in W | |w| < k\}$$

and $W_{\geq k} = \{w \in W | |w| \geq k\}$. let $\sim_k$ be the relation on the set of all extractors defined by $\mathcal{E}_1 \sim_k \mathcal{E}_2$ if $\mathcal{E}_1(L) \cap (\mathbf{\Sigma}_\infty^*)_{\geq k} = \mathcal{E}_2(L) \cap (\mathbf{\Sigma}_\infty^*)_{\geq k}$ for each subset L of $\mathbf{\Sigma}_\infty^*$. For extractors $\mathcal{E}_1$ and $\mathcal{E}_2$, let $\mathcal{E}_1 \sim \mathcal{E}_2$ if $\mathcal{E}_1 \sim_k \mathcal{E}_2$ for some $k \geq 0$. For each extractor $\mathcal{E}$, $\{\mathcal{E}\}_\sim$ denotes the set $\{\mathcal{E}' | \mathcal{E}' \sim \mathcal{E}\}$. An extractor $\mathcal{E}$ is said to be a *pseudo-identity* if $\mathcal{E}$ is in $\{[\![\alpha_1^*]\!]_1^{-1}\}_\sim$.

Definition Let $\mathcal{E}$ be an extractor. A list $\mathcal{E}_1, \ldots, \mathcal{E}_m$ of extractors is said to be a *decomposition* of $\mathcal{E}$ if

$$\mathcal{E}(L) = \mathcal{E}_m(\cdots \mathcal{E}_1(L))$$

for each subset L of $\mathbf{\Sigma}_\infty^*$. A decomposition $\mathcal{E}_1, \ldots, \mathcal{E}_m$ of $\mathcal{E}$ is said to be *trivial* if either $\mathcal{E}_i$ is a pseudo-identity or $\mathcal{E}_i \sim \mathcal{E}$ for some $1 \leq i \leq m$.

We are now ready to show the following:

Theorem 3 *There is no finite set of extractors which yields all extractors by composition.*

Proof. (*sketch*) Let $\mathcal{E}^{(i)} = [\![\alpha_1^i \alpha_2 \alpha_1^+]\!]_2^{-1}$ for each $i \geq 0$. It can be shown (proof omitted) that if $\mathcal{E}$ is in $\{\mathcal{E}^{(k)}\}_\sim$ for some $k \geq 0$ and $\mathcal{E}_1$, $\mathcal{E}_2$ is a decomposition of $\mathcal{E}$, then either $\mathcal{E}_2 \sim \mathcal{E}$ or $\mathcal{E}_2$ is a pseudo-identity. It is then easily seen that each decomposition of $\mathcal{E}^{(i)}$ is trivial for each $i \geq 0$.

Suppose there exists a finite set $\mathcal{F} = \{\mathcal{E}_1, \ldots, \mathcal{E}_m\}$ of extractors such that each extractor is a composition of some extractors in $\mathcal{F}$. Thus, $\mathcal{E}^{(k)}$ is a composition of extractors in $\mathcal{F}$ for each $k \geq 0$. It can be shown (proof omitted) that there exists $k \geq 0$ such that

$$\{\mathcal{E}^{(k)}\}_\sim \cap \mathcal{F} = \emptyset.$$

Since $\mathcal{E}^{(k)}$ is a composition of extractors in $\mathcal{F}$ and each decomposition of $\mathcal{E}^{(k)}$ is trivial, it follows that (proof omitted) $\{\mathcal{E}^{(k)}\}_\sim \cap \mathcal{F} \neq \emptyset$. This is a contradiction. □

Furthermore, we have the following result about the rs-operations in general.

Theorem 4 *There is no finite set of rs-operations which yields all rs-operations by composition.*

Proof. (*sketch*) It can easily be shown that a composition of mergers and extractors not equivalent to mergers is not equivalent to a merger. Therefore, the mergers in the set generating rs-operations must generate all mergers. This is a contradiction. □

By the above results, there is no "finite generating set" for mergers and/or extractors. Therefore, if a set $\mathcal{F}$ of sequence operations does have a finite generating set, then $\mathcal{F}$ either contains some operations which are not rs-operations or is a proper subset of the rs-operations. One open problem is to find "interesting" sets of mergers (extractors and rs-operations, resp.) which have finite generating sets.

The previous results also show that in using rs-operations in query languages, we need an infinite number of them to have the power of the rs-operations. In the remainder of this extended abstract, we use all the rs-operations in the query languages.

3 A Sequence Logic: SL

In order to construct calculus-like query languages over databases involving sequences, a sequence logic (SL) is (informally) introduced here. (SL is formally defined in the full paper.)

SL is a first-order logic. The rs-operations are used in SL as special predicates. Specifically, each $[\![W]\!]_n$, where $n \geq 1$ and W is a regular subset of V_n^*, is used as an $(n+1)$-ary predicate and called a *sequence predicate*. Each atomic formula involving $[\![W]\!]_n$ is written in the form

$$t_{n+1} \in [\![W]\!]_n(t_1, \ldots, t_n).$$

The "structures" for SL languages are to "assign" a sequence to each constant and a subset of n-ary tuples (over sequences) to each n-ary non-sequence predicate. The meaning of a sequence predicate $[\![W]\!]_n$ is determined by the mapping defined by the merger $[\![W]\!]_n$. For example, $x \in [\![\alpha_1\alpha_2]\!]_2(x_1, x_2)$ is *true* if ab is assigned to x, a to x_1 and b to x_2, and is *false* if ab is assigned to x, b assigned to both x_1 and x_2.

We now use two examples to illustrate SL.

Examples Sequence logic formulas are used to specify or declare properties of sequences or sets of sequences. The following are two examples. (The symbol "–" in the atomic formulas stands for "there exists some." For instance, $P(-, y)$ is an abbreviation of $\exists x P(x, y)$.)

(1) A sequence can be viewed as a multiset[¶]. Let $x \sqsubseteq y$ denote the formula

$$\forall z(x \in [\![(\alpha_1 \cup \alpha_2)^*]\!]_2(-, z) \wedge \mathrm{EQ}(z) \rightarrow y \in [\![(\alpha_1 \cup \alpha_2)^*]\!]_2(-, z)),$$

where $\mathrm{EQ}(z) = \forall y_1, y_2(z \in [\![\alpha_3^* \alpha_1 \alpha_2 \alpha_3^*]\!]_3(y_1, y_2, -) \rightarrow (y_1 = y_2)))$. It is easily seen that (1) $\mathrm{EQ}(u)$ is true if and only if u is in a^* for some a, and (2) $u_1 \sqsubseteq u_2$ is true if and only if u_1 is a subset of u_2 when viewed as multisets.

Now suppose $\leq$ is a total order relation on basic elements. Let $\mathrm{Sorted}(z) =$

$$(\forall x, y)(z \in [\![\alpha_1^* \alpha_2 \alpha_3 \alpha_1^*]\!]_3(-, x, y) \rightarrow x \leq y).$$

Clearly, $\mathrm{Sorted}(u)$ is true if and only if u is sorted according to $\leq$. Now let $\mathrm{Sort}(x, y)$ be the formula

$$(x \sqsubseteq y) \wedge (y \sqsubseteq x) \wedge \mathrm{Sorted}(y).$$

Intuitively, $\mathrm{Sort}(u_1, u_2)$ is true if and only if u_2 is a result of sorting u_1 according to $\leq$.

(2) At the end of Section 1, the operation Half was expressed using rs-operations. Here, an SL formula is used to describe the property that one sequence is the first half of another. Let $\mathrm{Samelength}(x, y)$ be the formula $- \in [\![(\alpha_1 \alpha_2)^*]\!]_2(x, y)$ and $\mathrm{Prefix}(x, y)$ the formula $y \in [\![\alpha_1^* \alpha_2^*]\!]_2(x, -)$. Since $[\![(\alpha_1 \alpha_2)^*]\!]_2$ represents the operation of "perfect shuffle," it is easy to see that

$$\mathrm{Samelength}(u_1, u_2)$$

is true if and only if u_1 and u_2 are of the same length. It is also easy to see that $\mathrm{Prefix}(u_1, u_2)$ is true if and only if u_1 is a prefix of u_2. Now let $\mathrm{Half}(x, y)$ be the formula

$$\mathrm{Prefix}(x, y) \wedge \exists z(y \in [\![(\alpha_1 \alpha_2)^*]\!]_2(-, z) \wedge \mathrm{Samelength}(x, z)).$$

Clearly, $\mathrm{Half}(u_1, u_2)$ is true if and only if u_2 is of even length and u_1 is the first half of u_2. □

[¶]A *multiset* is a set having possible duplicate elements. For example, $\{a, a, b\}$ and $\{a, a, b, b\}$ are both multisets. Let s_1 and s_2 be multisets. Then s_1 is a subset of s_2 (in the multiset sense) if, for each a, a occurs in s_2 at least as often as it occurs in s_1. For instance, $\{a, a, b\}$ is a subset of $\{a, a, b, b\}$.

4 An Extended Relational Data Model

In this section, we extend the "standard" relational data model to include sequences. Using the rs-operations, we then construct two query languages over the extended relational data model.

To motivate the discussion, consider the following:

Example Figure 2 describes the tour schedules of a travel agency. For each tour, the number in column TOUR_NO is its identification and the number in column COST its price. For each tour, the list in column CITY specifies the cities to be visited, and the lists in columns ARRIVAL and DEPARTURE show the arrival and departure dates of these cities. Note that the order in which the cities are to be visited is significant. □

To formally model such tables as in Figure 2, let $\mathcal{U}$ be a non-empty set of elements called *atoms* (sometimes called *atomic values*). Atoms are usually denoted by a and b etc., possibly subscripted. For each $n \geq 0$, a mapping t from $\{1, \ldots, n\}$ to $\mathcal{U}^*$ is called an *(n-ary) sequence tuple*, abbreviated *tuple*, and is customarily written in the form of $(u_1, \ldots, u_n)$, where $u_i = t(i)$ for each $1 \leq i \leq n$. Thus, $(ab, cbbc, bcab)$ is a 3-ary tuple. For each $n \geq 0$ and finite set I of n-ary sequence tuples, (n, I) is called an *(n-ary) s-instance* and abbreviated I when n is understood. For example, $(2, \{(a, b)\})$ and $\{(ab, cbbc), (bba, abca)\}$ are both binary s-instances. Clearly, the table in Figure 2 (ignoring the column names) is a 5-ary s-instance.

Finally, let $\mathcal{R}$ be a nonempty set of elements called *s-relation names*. Let *arity* be a mapping from $\mathcal{R}$ to the positive integers. For each R in $\mathcal{R}$, the integer $arity(R)$ is called the *arity of* R. Each finite subset of $\mathcal{R}$ is called an *s-database scheme* and usually denoted by D, possibly subscripted. Each mapping I_D from an s-database scheme D to the set of all s-instances, where $I_D(R)$ is an $arity(R)$-ary s-instance for each R in D, is called an *s-database instance (of D)*.

The data model defined above is a simple and natural extension of the relational data model [7]. We now define an algebraic query language, called "s-algebra," over s-database instances. S-algebra is essentially the

TOUR_NO	CITY	ARRIVAL	DEPARTURE	COST
356	New York	3/14/90	3/16/90	1004
	Miami	3/16/90	3/20/90	
456	Los Angeles	3/18/90	3/20/90	1409
	Santa Barbara	3/20/90	3/22/90	
	San Francisco	3/22/90	3/27/90	
556	San Francisco	3/21/90	3/23/90	699
	Los Angeles	3/23/90	3/29/90	

Figure 2: Tour schedules.

relational algebra [7, 16] with rs-operations used to deal with sequences. The first use of rs-operations is in the "merger reconstructions." Formally, let I be an n-ary, $n \geq 1$, s-instance. For each list $\xi = i_1, \ldots, i_k$ of $k \geq 1$ numbers in $\{1, \ldots, n\}$ and each k-ary merger $[\![W]\!]_k$, let $[\![W]\!]^\xi(I)$, called a *merger reconstruction (of I)*, be the $(n+1)$-ary s-instance

$$\bigcup_{t \in I}\{(u_1, \ldots, u_n, u_{n+1}) | u_i = t(i) \text{ for } 1 \leq i \leq n \text{ and } u_{n+1} \text{ is in } [\![W]\!]_k(\{u_{i_1}\}, \ldots, \{u_{i_k}\})\}.$$

For example, $[\![\alpha_1^*\alpha_2^* \cup \alpha_2^*\alpha_1^*]\!]^{2,3}(\{(ab, bcd, a)\}) = \{(ab, bcd, a, bcda), (ab, bcd, a, abcd)\}$ (since $bcda$ and $abcd$ are in $[\![\alpha_1^*\alpha_2^* \cup \alpha_2^*\alpha_1^*]\!]_2(\{bcd\}, \{a\})$) and $[\![\alpha_1^*\alpha_2^*]\!]^{3,3}(\{(a, b, c)\}) = \{(a, b, c, cc)\}$.

The "extractor reconstructions" of s-instances are defined similarly. Specifically, let I be an n-ary, $n \geq 1$, s-instance. For each integer k, $1 \leq k \leq n$, and extractor $[\![W]\!]_2^{-1}$, let $[\![W]\!]^{-k}(I)$, called an *extractor reconstruction (of I)*, be the $(n+1)$-ary s-instance

$$\bigcup_{t \in I}\{(u_1, \ldots, u_n, u_{n+1}) | u_i = t(i) \text{ for } 1 \leq i \leq n \text{ and } u_{n+1} \text{ is in } [\![W]\!]_2^{-1}(\{u_k\})\}.$$

For example, $[\![\alpha_1^*\alpha_2^*]\!]^{-2}(\{(ab, bc, a)\}) = \{(ab, bc, a, \varepsilon), (abc, bc, a, b), (abc, bc, a, bc)\}$.

The operations "union," "intersection," "difference," "cross product" and "projection" over s-instances are exactly the same as those in the relational algebra. The "selection" operations use the following "selection conditions:"

1. $\gamma_1 = \gamma_2$ is a selection condition if γ_i $(i = 1, 2)$ is in $\mathcal{U}^*$ or is of the form $\$j$,
2. $(C_1 \vee C_2)$, $(C_1 \wedge C_2)$ and $(\neg C_1)$ are all selection conditions if both C_1 and C_2 are selection conditions.

Using the above six operations plus the merger and extractor reconstructions, we can define "s-algebra expressions" similar to the relational algebra (formal definitions omitted). Each n-ary s-algebra expression over D represents a mapping from each s-database instance over D to an n-ary s-instance.

Turning to the calculus-like query language, we convert SL formulas as queries. Formally, an *s-calculus query over the s-database scheme D* is a construct of the form

$$T = \{(x_1, \ldots, x_n) | F\},$$

where F is a formula of SL with relation names in D as predicates, and $x_1, \ldots, x_n$ are the free variables in F. The "value" of an s-calculus query over D on an s-instance I_D of D, denoted $T[I_D]$, is defined similar to the relational calculus.

To illustrate s-algebra and s-calculus, we formulate some specific queries over the s-instance in Figure 2.

Examples Suppose R is a 5-ary relation name whose first column corresponds to TOUR_NO, second column CITY, third column ARRIVAL, fourth column DEPARTURE and fifth column COST (cf. Figure 2). The following are some queries addressed to R. The answer given for each query is the value of the query over the s-instance shown in Figure 2.

(1) "Print the numbers of those tours whose second city is Atlanta." Expressed in s-algebra:

$$\pi_1 \sigma_{\$6 = \text{"Atlanta"}} [\![\alpha_2\alpha_1\alpha_2^*]\!]^{-2}(R)$$

and in s-calculus:

$$\{(x) | \exists y(R(x, y, -, -, -) \wedge y \in [\![\alpha_1\alpha_2\alpha_1^*]\!]_2(-, \text{"Atlanta"}))\}.$$

The answer is the set {("356")}.

(2) "Give the numbers and costs of those tours which visit Los Angeles and later San Francisco." Expressed in s-algebra:

$$\pi_{1,5}\sigma_{\$6=\text{"Los Angeles, San Francisco"}}[\![(\alpha_1 \cup \alpha_2)^*]\!]^{-2}(R)$$

and in s-calculus:

$$\{ (x,y) | \exists z(R(x,z,-,-,y) \wedge z \in [\![(\alpha_1 \cup \alpha_2)^*]\!]_2(-, \text{"Los Angeles, San Francisco"})) \}.$$

The answer to this query is the set {("456", "1409")}. □

S-calculus is a very powerful language, perhaps too powerful. Indeed, we have:

Theorem 5 *There exists an s-calculus query T over an s-database scheme D such that it is undecidable to determine for an arbitrary s-instance I_D whether $T[I_D]$ is empty.*

Analogous to relational calculus, a computable subset of s-calculus, called "safe s-calculus," is defined below.

In order to define the notion of safe s-calculus, the "active domain" of each s-calculus query is first presented.

Let $T = \{(x_1, \ldots, x_n)|F\}$ be a query over the s-database scheme D and I_D an s-database instance over D. Then the *active domain of T over I_D*, denoted $adom(T, I_D)$, is the set

$\{a \in \mathcal{U} | a$ appears in F or in $I_D(R)$ for some R in $F\}$.

Let $adom^k(T, I_D) =$

$$\{u | u \text{ in } adom(T, I_D)^* \text{ and } |u| \leq k\}$$

for each $k \geq 1$.

The active domain of a query over an s-database instance consists of all elements used in the formula and all elements in the s-instances of the relation names appearing in the formula. Intuitively, the answer set of a query over an s-instance should consist of only these elements, i.e., no "new" elements should be "invented." Furthermore, the answer set should not contain arbitrary long sequences. These observations lead to the notion of "safe s-calculus queries." First though, the following technical term is needed. An SL formula F_1 is said to be a *subformula* of an SL formula F if either (i) $F_1 = F$, (ii) $F = (F' \vee F'')$ and F_1 is a subformula of either F' or F'', or (iii) $F = \neg F'$ or $F = \exists x F'$ and F_1 is a subformula of F'.

We are now ready for the notion of safe s-calculus.

For each $k \geq 1$, an s-calculus query

$$T = \{(x_1, \ldots, x_n)|F\}$$

over the s-database scheme D is said to be *k-safe* if T satisfies both of the following two conditions for each s-database instance I_D:

(1) If $(u_1, \ldots, u_n)$ is in $T[I_D]$, then u_i is in $adom^k(T, I_D)$ for each $1 \leq i \leq n$.

(2) If $(\exists x F_1)$ is a subformula of F and $x, x_1, \ldots, x_m$ are the free variables in F_1, then that $(u, u_1, \ldots, u_m)$ is in $T_1[I_D]$, where $T_1 = \{(x, x_1, \ldots, x_m)|F_1\}$ and u_i is in $adom^k(T, I_D)$ for $1 \leq i \leq n$, implies that u is in $adom^k(T, I_D)$.

A query T is said to be *safe* if it is k-safe for some $k \geq 1$. The collection of all safe-s-calculus queries is called *safe s-calculus*.

Similar to the equivalence of the relational algebra and the safe relational calculus [16], we have

Theorem 6 *Safe s-calculus and s-algebra are equivalent in expressive power.*

The above notion of safe s-calculus is defined by the means of a semantical restriction. In the full paper, a different notion of safe s-calculus, depending on a set syntactic conditions, is presented and shown to be equivalent to s-algebra.

5 Conclusion

A theoretical study is initiated on the sequence operations used in database query languages. Specifically, a set of special sequence operations (rs-operations) is introduced and shown to be readily applicable to database query languages.

Acknowledgment

The authors wish to thank Paris Kanellakis for bringing our attention to the AWK programming language.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language.* Addison-Wesley, 1988.

[3] A. Albano, L. Cardelli, and R. Orisini. Galileo: A strongly typed language for complex objects. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database manifesto. In *Proceedings of the International Conference on Deductive and Object Oriented Databases*, 1989.

[5] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Database Programming Languages: 2nd International Workshop.* Morgan-Kaufmann, Inc., June 1989.

[6] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. of SIGMOD International Conferenece on Management of Data*, pages 413–423, 1988.

[7] E. F. Codd. A relational model for large shared data banks. *Communications of ACM*, 13(6):377–387, 1970.

[8] O. Deux et al. The O_2 system. *Communications of ACM*, 34(10):34–48, October 1991.

[9] S. Ginsburg and C. Tang. Canonical forms for interval functions. *Theoretical Computer Science*, 54:299–313, 1987.

[10] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley, 1969.

[11] D. Metzler and S. Haas. The constituent object parser: Syntactic structure matching for information retrieval. *ACM Trans. on Information System*, 7(3):292–316, 1989.

[12] Ontologic, Inc. *Vbase Technical Overview*, version 1.0 edition, March 1987.

[13] P. Pistor and R. Traunmueller. A database language for sets, lists and tables. *Information Systems*, 11(4):323–336, 1986.

[14] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievments and opportunities. *SIGMOD Record*, 19(4):6–22, 1990.

[15] The Committee for Advanced DBMS Function. Third-generation database system manifesto. *SIGMOS Record*, 19(3):31–44, 1990.

[16] J. D. Ullman. *Principles of Database and Knowledge-base Systems.* Computer Science Press, 1988.

[17] S. Vandenberg and D. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, 1991.