A High-Level User Interface for Update and Retrieval in Relational Databases - Language Aspects

Gottfried Vossen and Volkert Brosda

Technical University of Aachen
Lehrstuhl fuer Informatik III
Buechel 29 - 31, D-5100 Aachen
West Germany

Abstract

A high-level user front-end for the experimental relational database system MEMODAX is described It is based on the universal relation model, and it allows querying and updating a database without any knowledge of its conceptual structure We survey the main features of USIL, the Universal Schema Interface Language for MEMODAX, and show how to retrieve data from a database without logical access path specification We additionally demonstrate how to update the database when the user is aware of attributes only We give a description of the syntax and parts of the semantics of USIL indicating that this language is getting close to a natural language interface

1 Introduction

Conventional relational database systems let users access their data through high-level languages that allow querying in terms of the conceptual schema One has to know the attributes and their distribution over base relations For updates, the user again has to remember names for relations in connection with the particular attributes corresponding to them In both cases there is no need for him to be aware of details of the internal database organization, hence he is working on a logical level with physical data independence

Actually, the relational model of data originally aimed at achieving logical data independence as well, which means that retrieval and update may be done without even remembering the conceptual schema A vehicle that has proven useful in this context is the so-called universal relation (UR-) model [MUV], in which the user sees the whole database as if it was just one relation In this model all semantic information about the real world that the database is intended to represent is carried by the attributes alone the user just has to refer to attributes when he is working with the database

Several proposals for the design and implementation of a universal schema interface (USI), as the user front-end based on the UR-model is commonly termed, can be found in the literature, see [MRW] for a survey Well-known examples are AURICAL [CKPS], DURST [BB1, 2], PIQUE [M1, 2, MRSSW], and System/U [KKFGU] Common to all these approaches is the fact that they concentrate on the aspect of querying a database AURICAL and System/U seemingly provide a pure query language, the cited references do not mention any update features of the data manipulation languages, resp The situation looks better for DURST Update operations are allowed in case the operation refers to all attributes of exactly one relation, and in addition insertions are possible if values for some complete join path through the database are provided In [St] it is described how to update PIQUE databases Any update has to maintain the containment condition for associations and is directed against a relation over some association, hence the user has to be aware of the corresponding association structure

of the database and not of the attributes alone

So we see some kind of "contradiction" with respect to logical data independence, since the database schema is usually hidden only for queries One reason for this may be the believe that "updaters" of a database should be more sophisticated than "readers" With the growing dissemination of microcomputers, however, we do not think that this is reasonable any longer The advent of database systems in the office environment, where the typical user like a secretary is of type "ad hoc", forces us to make reading and updating of data possible to any user, a point of view that is also supported in [Sa3]

Therefore, we are developing a USI for our relational database system MEMODAX [KV,V2] that is really "universal" It is based on the UR model, and it combines retrieval and update capabilities in a homogeneous way All that the user has at hand for working with the database is a global set U of attributes that has to satisfy certain requirements (see section 2) For query and for update purposes he always refers to an imaginary relation u over U that may contain null values for unknown information The use of nulls, however, is restricted by so-called update-sets which are "meaningful" portions of U on which defined tuples have to be null-free The answer to a query then is some total projection of u which satisfies certain selection criteria, updating of u is performed in correspondence with the update-sets

In [BV], we have presented the theoretical framework underlying the design and implementation of USIL, the USI Language for MEMODAX that is going to replace the currently running algebraical language Hence, USIL differs from most other USI languages in that it is not a QUEL derivate, the underlying idea is that the user's view consists of a two-dimensional (universal) table (or more often a subtable of it which is a relation over some $X \subseteq U$) and that this table should be accessible by (vertical) projection and by (horizontal) selection This (sub-)table over X acts like a "window" on the database, and it is computed by a so-called window function, denoted [X]

In this paper we describe the language itself We indicate its use by a sequence of examples and end up with a presentation of its syntax and the main definitions of its semantics In detail, we will proceed as follows

Section 2 briefly reviews some necessary terminology and presents a sample database that will be used throughout the paper Section 3 contains an introduction to USIL from a user's point of view We illustrate its simple use when formulating queries or update requests In section 4, we present a BNF-description of the USIL-syntax and parts of the semantics for USIL-queries Section 5 summarizes the main ideas for executing insertions and deletions, which are described in [BV] in more detail The final section 6 summarizes our work and takes a short look at open problems and future research

2 Preliminaries

We assume the reader to be quite familiar with the brass tacks of relational database theory and with the nature of relational languages, at least on the level of [M1, U1] In this section we briefly clarify our terminology

Let $X = \{A_1, \ldots, A_m\}$ be a finite set of attributes (which we sometimes abbreviate as $A_1A_2 \ldots A_m$) with associated domains, and let $dom(X) = \bigcup_{A \in X} dom(A)$ Next let F be a finite set of functional dependencies (FDs) over X of the form $f: L \to R$, $L, R \subseteq X$, which are used to express special integrity constraints and mainly keys An update p (also called object in [Sc]) over X is a subset of X that constrains the use of null values, as is shown in a moment A triple $R = (X, F, P)$, where P is a finite set of updates, is called relation schema A (valid) relation r over R is a finite set of tuples $\mu: X \to dom(X) \cup \mathbb{N}$, where $\mathbb{N}$ is a countable, possibly infinite set of indexed null values of type "value exists, but is presently unknown" [Sa2], such that

(1) $(\forall A \in X)\ \mu(A) \in dom(A) \vee \mu(A) \in \mathbb{N}$,
(2) r satisfies F,
(3) $(\forall \mu \in r)\ (\exists p \in P)\ (\forall A \in p)\ \mu(A) \notin \mathbb{N}$,
(4) each two nulls in r have different indices (unless equality is enforced by some FD),
(5) r is free of subsumed (redundant) tuples

A database schema is a pair $\mathfrak{D} = (\mathfrak{R}, \mathfrak{P})$ where $\mathfrak{R}$ is

a finite set of relation schemas, $\mathcal{R} = \{ R_i = (X_i, F_i, P_i) \mid i = 1(1)k \}$, and $\mathcal{P}$ is the set of all updates that are supported by $\mathcal{D}$ A database d over $\mathcal{D}$ is a finite set of relations, exactly one over each $R_i \in \mathcal{R}$

We postulate that any $\mathcal{D}$ under consideration satisfies the universal relation schema assumption (URSA) stating that the attributes in $\mathcal{D}$ carry all the semantic information about the specific real world situation that is modeled by $\mathcal{D}$ Hence, attribute names have to be globally unique so that they alone can be used to refer to data in the database In addition, we require the unique role assumption (URA) to hold, meaning that among any set of attributes there is at most one connection

Example 1 Consider the following data requirements for housing estates For a collection of houses in a colony and for the flats in those houses an agency wants to store data concerning houses, represented by street name S and street number Sn, each house has a certain number N of flats, and for each individual flat there is a floor number Fl, an area A, a rent R, a current tenant T and a manager M Hence the universe under consideration is

$U = \{S, Sn, N, Fl, A, R, T, M\}$

The following FDs are required to hold

$F = \{S\ Sn \to N,\ T \to S\ Sn\ Fl\ A\ M,\ S\ Sn\ Fl \to T\ R\ A,$
$S\ A \to M\ R\}$

For the moment we define two updates only
$P = \{T\ S\ Sn,\ A\ R\}$

So we are given a UR schema $R_U = (U, F, P)$, only the U-component of which will be seen by the users Obviously it is unreasonable to store a corresponding universal relation u to which queries and updates will refer explicitely Instead, we replace R_U as usual by a suitably chosen $\mathcal{D}$ and store relations over the resulting $\mathcal{R}$ By "suitable" we mean that $\mathcal{D}$ is the result of an application of the synthesis algorithm [BDB, V1] to R_U, which we have extended in [BV] in such a way that the user-specified updates from P now trigger the algorithm so that they become embodied into $\mathcal{D}$ The main results of our new synthesis proposal are as follows

(1) If f is an FD over U with attribute A on its right side and with a non-redundant left side L, then LA is also considered as an update (since L may be regarded as some "key" and A as a property of that key)

(2) For each "original" update $p \in P$, there is a key included in $\mathcal{D}$,so that a restricted kind of losslessness can be guaranteed

The desirable properties that any $\mathcal{D}$ should have (and that our algorithm yields) are the following

(i) $\bigcup_{i=1}^{k} X_i = U$,

(ii) $(\bigcup_{i=1}^{k} F_i)^+ = F^+$, and every $f \in F_i$ is a key-dependency for R_i,

(iii) $\mathcal{P} = \bigcup_{i=1}^{k} P_i \cup \bigcup_{i=1}^{k} \{ X_i \} \cup P$, and for each $p \in \mathcal{P}$ there exists some $f \quad L \to R \in F_i$ (for some i) such that L is a key for p

Example 1 (cont) For R_U given above, our design procedure yields the following $\mathcal{D}$

$R_1 = (S\ Sn\ N,\ \{S\ Sn \to N\},\ \{S\ Sn\ N\})$
$R_2 = (S\ Sn\ Fl\ A\ T,\ \{S\ Sn\ Fl \to A,\ S\ Sn\ Fl \to T,$
$T \to S,\ T \to Sn,\ T \to Fl\},$
$\{S\ Sn\ Fl\ A,\ S\ Sn\ Fl\ T,\ T\ S,$
$T\ Sn,\ T\ Fl\})$
$R_3 = (S\ A\ M\ R,\ \{S\ A \to M,\ S\ A \to R\},\ \{S\ A\ M,\ S\ A\ R\})$
$R_4 = (R\ A,\ \{R\ A \to R\ A\},\ \{R\ A\})$

Furthermore,

$\mathcal{P} = \{T\ S\ Sn\} \cup \bigcup_{i=1}^{4} \{X_i\} \cup \bigcup_{i=1}^{4} P_i$

Note that a lot of updates has been added to the original P, and that it is still possible to update a relation "according" to its schema The flexibility of storing data that stems from the use of updates should become clear from the following sample database

r_1

S	Sn	N
Park Lane	10	4
Traf Square	5	15
Park Lane	7	3

r_2

S	Sn	Fl	A	T
Park Lane	10	1	70	Hart
Traf Square	3	δ_2	δ_1	Jones
Traf Square	δ_4	2	δ_3	James

r_3

S	A	M	R
Park Lane	70	King	δ_5
Traf Square	20	δ_6	250

r_4

R	A
250	20
260	30

(Note that it may happen that there exist tuples - such as (Traf Square, δ_4, 2, δ_3, James) in r_2 - which are defined on a union of updates, this is even considered as a valid relation in our context)

Next we remind the reader of some basic operations on relations, which we will use in subsequent sections Selection (σ) looks for tuples satisfying certain conditions, projection (π) drops columns and after that removes duplicates, and (natural) join ($\bowtie$) concatenates relations via equal values for common attributes We will use total projection (π^*) if the result should contain total tuples only

Another property of $\mathfrak{D}$ that will be of crucial importance in our further developments is that every database d over $\mathfrak{D}$ should possess a so-called representative instance, denoted rep(d), which is defined as follows

$$rep(d) = sub(chase_F(\bigcup_{i=1}^{k} pad_U(r_i)))$$

In this definition, "pad" stands for padding out base relations with (new) indexed nulls so that they get "width" |U|, "$chase_F$" denotes the chase-process [M1] for FD-rules, and "sub" means that the result is subsumption-free [Sa2]

Example 1 (cont) Let $d = \{r_1, r_2, r_3, r_4\}$ be as above, then rep(d) is the following universal relation

S	Sn	N	Fl	A	T	M	R
Traf Square	5	15					
Park Lane	7	3					
Park Lane	10	4	1	70	Hart	King	
Traf Square	3				Jones		
Traf Square			2		James		
Traf Square				20			250
				30			260

where open positions represent different nulls

As [Sa1, 2], we consider rep(d) as the correct representation of the contents of d The main reason is that the user has a global view of the database at any time that he is working with it (i e , at the design and at the manipulation time) This global view is the basis for the definition of the semantics of any weak-instance window function, of which rep is a special case Computational windows, however, take the actual database schema into account when defining a window-semantics, which may lead to ambiguities in answers to queries Furthermore, rep(d) has several properties which are considered desirable in the context of the UR model

Theorem 1 Let d be a database, for which rep(d) exists, $X \subseteq U$, and let $[X] = \pi^*_X(rep(d))$ Then

(i) [MRW] $[X] \supseteq \pi^*_X([Y])$ if $X \subseteq Y$

(i e , [X] satisfies the containment condition)

(ii) [BV] if d (over $\mathfrak{D}$) possesses a (pure) universal instance u, then for any $X \subseteq U$ there is a relation r over X such that $r = \pi_X(u) = \pi_X(rep(d))$

Consequently, when a user queries the imaginary u over U, his query will in our approach be answered with respect to rep(d) (see section 4) The crux here is that the existence of rep(d) does not come for free It may happen that the chase-process produces FD-violations if applied to $\bigcup_{i=1}^{k} pad_U(r_i)$ Therefore update-operations have to be performed very carefully such that an already existing rep(d) is not get lost We will briefly return to this point in section 5, for the moment, we restrict ourselves to noticing that we are able to guarantee the existence of rep(d) even if d undergoes updates Details can be found in [BV]

3 USIL from a User's Point of View

In this section we give an introductory survey of USIL features We do so by presenting a sequence of examples of sligthly increasing difficulty, which will all refer to the housing estates database from section 2 We briefly turn to retrievals first The material in that subsection is not totally new, what should mainly be noticed is the

algebraic character of USIL queries The reason for that is that USIL evolved from relational algebra which is currently running in the MEMODAX system, while other approaches such as [BB1, 2, KKFGU, M1, M2, MRSSW, MRW, U2] primarily focus on QUEL-derivates After that we turn to updates and exemplarily show how to accomplish this with USIL as well

3 1 Quering a database

The simplest query a user may pose to an "ordinary" relational database is one which can be expressed by an SPJ-expression that "connects" two relations via a join, selects certain tuples from the result or from the operands and finally projects onto certain attributes.

Example 2 Consider the query q_1 "Print the number of flats in the house where Hart lives " Remembering from example 1 that houses in our sample database are represented by the pair (S,Sn), a relational algebra formulation of q_1 is

(1) $\pi_N(r_1 \bowtie \pi_{S\ Sn}\ \sigma_{T='Hart'}(r_2))$

Having the USI at hand, this query can only refer to u = rep(d), an imaginary relation over the universe U of attributes Since there is neither a need nor a possibility to write down a join path, the expression above reduces to

(2) $\pi^*_N(\sigma_{T='Hart'}(u))$

(We see from (2) that actually there was no need in (1) for explicitely including $\pi_{S\ Sn}$ in the expression although this may reduce the amount of tuple transfers internally) Note that in both cases the result is "N = 4"

An important notion that should be introduced at this point is that of a mention set [M1, 2, MRSSW], which was originally introduced for tuple variables and is here "extended" to queries Informally, let q be an English language query, then the mention set of q, denoted men(q), is the set of all attributes mentioned in q (if these attributes belong to the database)

Hence, $men(q_1) = \{N, S, Sn, T\}$ So we actually need not consider the whole relation u in (2), but it is sufficient to look at $\pi^*_{men(q_1)}(u)$, since all relevant information is contained in this projection already We immediately arrive at a straightforward strategy for the UR-formulation of a great variety of queries The user first formulates his query in a natural language such as English, and then determines the mention set of it Next he calls upon the system to generate a corresponding "view", which in our context is a projection of u onto men(q), this step is called RETRIEVE in the USIL-syntax, abbreviated "R " (see section 4) To this view he finally applies selections and a (total) projection yielding the desired answer

As an aside we mention that we use total projection for several reasons Firstly, we agree with [BB1, Sa2] in that if the user refers to the attributes in men(q) or a subset of this set, then he is not interested in incomplete information on this subuniverse (remember from the end of section 2 that queries will be answered with respect to rep(d)) Another reason is that although indexed nulls are used logically, non-indexed ones (represented by always the same symbol) are stored physically by MEMODAX for efficiency reasons

The query formalism described so far will be sufficient for most of the "everyday work" with a database There are cases, however, in which we are forced to use additional power of our language Readers who are familiar with tuple relational calculus and its adaptions to the USI-context will remember that queries usually involve one tuple variable ranging over the universal relation But in certain cases it is necessary to introduce a second tuple variable which ranges over u as well A typical example is

Example 3 Consider the query q_2 "Print all managers who are also tenants in the colony " In this case we have to check whether or not there is an M-value (in r_3) that also appears in the T-column of r_2 An obvious algebraic formulation is

$$\sigma_{T=M}(\pi_T(r_2) \bowtie \pi_M(r_3)) \quad ,$$

but a selection that is able to compare attribute-values directly is not always present in a DBMS So we have to devise an alternative formulation which uses the rename-function for attributes

$$\pi_T(r_2) \bowtie rename_{M=T}(\pi_M(r_3))$$

When quering through the USI, we are obviously not allowed to express this request simply by $\pi^*_{TM}(u)$,

since URA implies that if the "connection" T M is intended to refer to a tenant in a flat that is supervised by manager m, then no second relationship between T and M is possible in u, but this former connection is established by the specification of the FD T → M with respect to the universal schema Hence, a database-wise navigation is needed, and we still have to use renaming in USIL since the algebraic character of our language does not include tuple variables So the expression above reduces to

$$\pi^*_T(u) \bowtie \text{rename}_{M=T}(\pi^*_M(u)) \quad ,$$

where $\bowtie$ now degenerates to a Cartesian product Again, the answer in both cases is an empty relation over T

3 2 Updating a database

In this subsection we exemplarily compare how to update a database in case we do not have a USI at hand and in case we have it The first three examples refer to the insertion of new tuples which is usually done in correspondence with the defined relation schemas With a USI that is aware of update-sets independent of relation schemas, this process is considerably simplified and becomes much more flexible. The same is true for deletions, which will be shown in the fourth example Note that we have not yet implemented a MODIFY-command, at the moment we leave it to the user to perform this operation by an appropriate delete/insert-sequence

Generally speaking, any insertion or deletion is done in two steps as is described in section 5, and has to be performed in such a way that for the result rep(d) always exists Since we do not require that $\mathfrak{D}$ satisfies the uniqueness condition [Sa2], an insertion or deletion is (internally) not always local with respect to a single base relation

Example 4 Suppose a user wants to insert the fact that Smith lives at Park Lane No 15 Since this information "fits" into r_2 above, he normally will enter a command like "INSERT R_2," and after that the tuple (Park Lane, 15, δ, δ, Smith) (The nulls in this tuple will be converted to δ_7 and δ_8, resp by the system)

In USIL, he again determines the "mention set" first, which now refers to the attributes of the intended insertion, and enters

INSERT(S, Sn, T),

Now the system checks whether or not S Sn T is known as an update (-set) Since this is the case, the user is asked to enter a (totally defined) new tuple over S Sn T, namely

(Park Lane, 15, Smith)

It remains invisible to him which relations are affected by this insertion, although it is not surprising that in this case only r_2 is augmented, since no FD-violations occur when a chase-process is applied to $\bigcup_{i=1}^{4} \text{pad}_U(r_i)$

Example 5 Let the next fact to be inserted be that Smith is living on the fifth floor The analogous process as in the last example now yields in the ordinary case

r_2	S	Sn	Fl	A	T
	Park Lane	10	1	70	Hart
	Traf Square	3	δ_2	δ_1	Jones
	Traf Square	δ_4	2	δ_3	James
	Park Lane	15	δ_7	δ_8	Smith
	δ_9	δ_{10}	5	δ_{11}	Smith

where the first four tuples represent the result from example 4 Using

INSERT(T, Fl),
(Smith, 5),

the system will now apply the FDs T → S to replace δ_9 by 'Park Lane', T → Sn to replace δ_{10} by 15, T → Fl to set δ_7 to 5 and finally S Sn Fl → A to equate δ_8 and δ_{11} Hence r_2 is left with two copies of (Park Lane, 15, 5, δ_8, Smith), one of which is finally discarded

Example 6 Suppose that a user tries to insert (Park Lane, δ, δ, 300) into r_3 In the ordinary case this attempt would fail since the key S A is not free of nulls, a requirement that is observed by most present-day DBMSs With USIL, the request

INSERT(S, R),

would already be answered with the hint that S R is not a valid update, but at the same time with a

presentation of the alternative S A R, which now may or may not be used

It should be mentioned that insertions into non-unique relations are more complicated, for details and for examples see [BV]

We close this section with an example that indicates how to perform a deletion and some of the problems that arise with this operation

Example 7 If a user wants to delete the fact that Hart is living at Park Lane, he normally has to look for a relation in which this information is stored and then to enter a corresponding command, for instance

DELETE R_2 WHERE S ='Park Lane' AND T = 'Hart',

The effect will be that every tuple which matches the delete-condition is dropped from r_2 But it may be suspected that this was not what the user intended Strictly speaking, he did not want to delete the fact that Hart's flat is of area 70 , or otherwise he would have included this in the condition Therefore we believe that the USIL-way of deleting is again more adequate Firstly, the user enters

DELETE(T, S),

and the system recognizes T S as an update Then he delivers a (total) tuple over T S representing a fact or a "connection" that is to be deleted, namely

(Hart, Park Lane),

and now all that happens is that in (Park Lane, 10, 1, 70, Hart), the S-value is set to null (Note that this correspond to

MODIFY R_2 WHERE S ='Park Lane' AND T ='Hart',
S = δ,

in an ordinary system)

We have to mention that there are cases in which deletions are not so simple, instead, the deletion of facts in one relation may imply that other facts in different relations must also be dropped, since otherwise an efficient retrieval from the representative instance could no longer be guaranteed For convenience, we mention that we consider deletions as being "inverse" to certain insertions Hence, if d' is the result of one or more deletions of tuples from database d, then it must be possible that state d' is also achievable by a sequence of insertions alone

Furthermore it is presently impossible to delete a set of tuples satisfying a certain condition in one step Refinements of this kind have to be included in further USIL releases

4 USIL-Syntax and Semantics of USIL-Queries

In this section we complete our introductory survey of USIL with a presentation of its syntax in BNF and of parts of its semantics

USIL-Syntax

```
<USIL-command>   = PRINT <rel-expr>, |
                   INSERT <ins-list>, |
                   DELETE <del-list>, |
                   MODIFY <ident> WHERE <cond>,
                          <assign>, |
                   RETURN <ident>, |
                   <ident>  = <rel-expr>, |
                   END,

<rel-expr>       = <subrel-ex> |
                   <subrel-ex> + <subrel-ex> |
                   <subrel-ex> * <subrel-ex>

<subrel-ex>      = ( <rel-expr> ) |<ident> |
                   PROJECTION <rel-expr>
                              ( <attr-list> ) |
                   SELECTION <rel-expr>
                              WHERE <cond> |
                   RETRIEVE ( <attr-list> ) |
                   RENAME <rel-expr>
                           ( <ren-list> ) |
                   SORT <rel-expr>
                          BY ( <attr-list> )

<ins-list>       = <ident> | ( <attr-list> )

<del-list>       = <ident> WHERE <cond> |
                   ( <attr-list> )

<assign>         = <attr-ident>  = <attr-rhs> |
                   <assign> , <assign>

<cond>           = <cond-term> |
                   <cond-term> OR <cond-term>

<cond-term>      = <cond-fac>  |
                   <cond-fac> AND <cond-fac>
```

<cond-fac> = NOT <cond> | (<cond>) |
<attr-ident> <cop> <oper>

<cop> = = | < | > | ≦ | ≧ | ≠

<oper> = <string> | NULL | <number> |
+ <number> |- <number>

<ren-list> = <attr-ident> = <attr-ident> |
<ren-list> , <ren-list>

<attr-rhs> = <attr-expr> | <string> | NULL

<attr-expr> = <attr-term> | ± <attr-term> |
<attr-term> ± <attr-term> |
± <attr-term> ± <attr-term>

<attr-term> = <attr-fac> |
<attr-fac> * <attr-fac> |
<attr-fac> / <attr-fac>

<attr-fac> = <attr-ident> | <number> |
(<attr-expr>)

<attr-list> = <attr-ident> |
<attr-ident> , <attr-list>

<ident> stands for a temporary relation identifier, <attr-ident> for an attribute identifier, each begins with a letter followed by letters or digits <number> stands for an unsigned integer- or real-number, and <string> denotes a character-string that is included in "'" and may not contain an apostrophe

The following abbreviations are allowed
for PRINT, P for PROJECTION,
S for SELECTION, R for RETRIEVE

Excerpt from USIL-Semantics

<< R (<attr-list>) >> = $\pi^*_{\text{<attr-list>}}$(rep(d))

<< S <rel-expr> WHERE <cond> >>
= $\sigma_{\text{<cond>}}$(<< <rel-expr> >>)

<< P <rel-expr> (<attr-list>) >>
= $\pi^*_{\text{<attr-list>}}$(<< <rel-expr> >>)

<< RENAME <rel-expr> (<ren-list>) >>
= << <rel-expr'> >>

where <rel-expr'> is derived from <rel-expr> by applying <ren-list> "+" is "∪", "*" is "⋈"

For the detailed semantics of INSERT and DELETE see [BV] For convenience, we give a USIL-formulation of our sample queries from section 3 1

Example 2 (cont)

P S R (T, N) WHERE T = 'Hart' (N),

Example 3 (cont)

R (T) * RENAME R (M) (M =T),

5 On the Semantics of Updates

As was shown in section 3, updating is a two-step process that consists of a dialogue between the user and the system In the first step the user enters either "INSERT <ins-list>," or "DELETE <del-list>,", and the system checks whether or not the given list is a defined update, i e an element of $\mathfrak{P}$ If not, it presents subsets and supersets of it on which insertions and deletions are allowed, and the user has to choose one of them In case there is some p ∈ $\mathfrak{P}$ that equals the insert- or delete-list, resp , the user may in the second step enter a tuple μ that is totally defined on p, as was shown in examples 4 to 7 This tuple is then checked for "insertability" if an insertion is required, deletions are always possible, but may result in the deletion of more tuples than were intended The reason for both is that update-operations have to assure that resulting databases still possess a representative instance, and that rep(d) can be computed efficiently via so-called minimal extension joins

We do not want to go into the corresponding details here, they can be found in [BV] The key observation on which the execution of update-operations is based is that the existence of rep(d) can be enforced either by the modified foreign key constraint [Sa1] or by the already mentioned uniqueness condition [Sa2] In [BV] we have extended the latter notion in such a way that this condition becomes applicable to update-sets Since it is not clear how to design $\mathfrak{D}$ in such a way that the uniqueness of relation schemas is achieved, we have explored the structure of non-unique schemas and updates, we have derived conditions that exactly tell us how to proceed in case that an insertion refers to a non-unique update

The central point with insertions is that a local chase is performed within the "relevant" base re-

lations, which uses the key dependencies Inter-relational chasing is applied only when the answer to a query is computed, but it has to be sure that this process cannot lay bare a hard FD-violation Since this requirement is fulfilled for databases that are build up with our insert-operation, this latter chasing can be simulated by minimal extension joins (as in [Sa2] for the unique case)

Informally, if the update p mentioned above fits into some unique R_1, then the insertion refers to r_1 only, and system-generated nulls or already known values are added to pad out the new tuple μ so that it gets the desired format If R_1 is not unique, the insertion has to look for "side-effects" that μ may have on other relations, and we have been able to show that these can only occur within a certain "region" around R_1 Hence, the system has to check whether "contradictory" information can be derived within that region, and if this can be excluded, the insertion of μ yields in an "adjusting" of (extension) join paths through that region If p is composed of parts from several relations, a corresponding operation is performed for each such part

For deletions the situation is simpler, but additional deletions may be required since information along an (extension) join path through a region may get lost by the dropping of fact μ

In conclusion, we have proven the following

Theorem 2 Let d be a database that was build up by USIL-insertions or -deletions, and let μ be a tuple that has been inserted but not yet dropped (where μ was user-defined on update p), then the following holds

(i) rep(d) exists,
(ii) $\mu \in \pi_p^*(\text{rep}(d))$

(i e [p] is "almost" faithful in the sense of [MRW])

6 Conclusions and Future Work

In this paper we have given an introductory survey of USIL, a high-level, user-friendly language for performing update and retrieval in a relational database We have demonstrated its use by several examples which should indicate that it is a quite easy exercise to learn how to use this language This point is also supported by our own experience that we have achieved during the implementation of the universal schema interface for our experimental relational system MEMODAX and during a practical course in databases for students at the graduate level [V2]

Within a short time USIL will have replaced the currently running relational algebra of MEMODAX completely, where the only technical problems that have to be solved concern the fact that this system is developed on a micro-computer and that this machine is now getting close to the limits of its capacity

From a theoretical point of view, several other problems remain open

First of all it may be argued that the representative instance in its present form is not the last resort The chase-process that is hidden in our processing strategy is able to derive new knowledge about the outside world, but this knowledge cannot be retrieved from the database, at least in certain cases The most typical example is that two employees have the same manager, but he is presently unknown Neither do they appear in the answer to a query that asks for all employees with the same manager nor is it possible to insert corresponding tuples (with nulls) into the database A first proposal attacking this problem can be found in [U2]

Another problem remains because we allow explicit modifications for temporary relations only, but not for tuples over updates This may become a handicap especially for casual users if they have entered a typo, even in this case a correction has to be performed via a deletion followed by an insertion

A third problem is due to the unique role assumption that we have made in section 2 With this assumption it is impossible to distinguish, for instance, between "supplies", "stores", and "needs" in a supplier-part-connection This directly carries over to one of our directions for future work If one thinks of an English language query, the mention set is derived from the nouns of it, as was demonstrated in example 2

But what about verbs and the other components of such a sentence ? We think that work is needed on the relationship between USIs and a natural language database access, because the former seems to be a quite adequate prerequisite for the latter Consider, for instance, the problem of missing joins when queries are incompletely or superficially specified in the RENDEZVOUS system [Ch] This problem is no longer existing in the presence of a USI, since the system now knows from its window function how to build join paths internally The syntax of the target language is considerably simpler than that of an ordinary database language, so that this relationship deserves further research

References

[BB1] J Biskup, H H Bruggemann Universal Relation Views A Pragmatic Approach, Proc VLDB 9, 1983, 172 - 185

[BB2] J Biskup, H H Bruggemann Eine Datenbanksprache fur eine Universalrelationen-Sicht, unpubl manuscript, Univ of Dortmund 1983

[BDB] J Biskup, U Dayal, P A Bernstein Synthesizing Independent Database Schemas, Proc ACM SIGMOD Conf 1979, 143 - 152

[BV] V Brosda, G Vossen Update and Retrieval in a Relational Database through a Universal Relation Interface, Techn Rep No 101 Techn Univ of Aachen 1984, Extended Abstract appears in Proc ACM SIGACT-SIGMOD PODS 4, 1985

[Ch] C L Chang Finding Missing Joins for Incomplete Queries in Relational Data Bases, IBM Research Report RJ 2145, San Jose 1978

[CKPS] H -H Chen, S.M Kuck, J Peterson, Y Sagiv A User's Manual for AURICAL A Universal Relation Implementation via Codasyl, Techn Rep UIUCDCS-R-82-1114, Univ of Illinois at Urbana-Champaign 1982

[KV] U Kamper, G Vossen The MEMODAX Relational Database System - on the current state of development, Angewandte Informatik 26, 1984, 152 - 163 (in German)

[KKFGU] H F Korth, G M Kuper, J Feigenbaum, A van Gelder, J D Ullman System/U A Database System Based on the Universal Relation Assumption, ACM TODS 9, 1984, 331 - 347

[M1] D Maier The Theory of Relational Databases, Computer Science Press 1983

[M2] D Maier Universal Scheme Interfaces and the Theory of Window Functions, Proc TIDB 2, 1984, 105 - 130

[MRSSW] D Maier, D Rozenshtein, S Salveter, J Stein, D S. Warren: Toward Logical Data Independence: A Relational Query Language Without Relations, Proc ACM SIGMOD Conf 1982, 51 - 60

[MRW] D Maier, D Rozenshtein, D S Warren Windows on the World, Proc ACM SIGMOD Conf 1983, 68 - 78

[MUV] D Maier, J D Ullman, M Y Vardi On the Foundations of the Universal Relation Model, ACM TODS 9, 1984, 283 - 308

[Sa1] Y Sagiv Can we use the Universal Instance Assumption without using Nulls?, Proc ACM SIGMOD Conf 1981, 108 - 120

[Sa2] Y Sagiv A Characterization of Globally Consistent Databases and their Correct Access Paths, ACM TODS 8, 1983, 266 - 286

[Sa3] S Salveter A Transportable Natural Language Database Update System, Proc ACM SIGACT-SIGMOD PODS 3, 1984, 239 - 247

[Sc] E Sciore The Universal Instance and Database Design; Ph D Dissertation, Princeton Univ 1980

[St] J Stein Data Definition and Update in the Association-Object Data Model, unpubl manuscript, SUNY, Stony Brook, 1982

[U1] J D Ullman Principles of Database Systems, Computer Science Press, 2nd ed 1982

[U2] J D Ullman Universal Relation Interfaces for Database Systems, Proc IFIP 1983, 243 - 252

[V1] G Vossen On an Efficient Implementation of the Synthesis Algorithm for Relational Database Design, Angewandte Informatik 24, 1982, 493 - 503 (in German)

[V2] G Vossen A Practical Course in Databases

with dBase II and MEMODAX, Techn Rep No 103 (in German), Techn Univ of Aachen 1985