Database Programming meets Internet Programming

T. H. Merrett
McGill University School of Computer Science
3480 University Street
Montreal H3A 2A7
Canada
tel: 514 389 3741
fax: 514 398 3883
email: tim@cs.mcgill.ca

May 21, 2002

Abstract

The effect of Internet transmission delays is closely similar to the effect of secondary storage latency on data organization. Database programming languages are designed to accommodate this effect at a high level of abstraction, and should also be suited to Internet programming. Internet programming adds the element of asynchronicity, which is also present in active databases. A database programming language with event handling can be applied in both cases. This paper demonstrates the Internet capability of a database programming language which is adequate for administrative data, expert systems, data mining, and geographical information systems, among other applications. The advantage of such a wide range of capabilities is that a single language and conceptual framework can be used for all forms of information retrieval and decision management. (We fall short of considering the Internet as an enormous database: data organization and indexing problems must still be addressed, and the remote processing capabilities of the Internet must also be included.)

Keywords domain algebra, electronic cash, Internet databases, nested relations, relational algebra.

1 Introduction

Someday, the Internet may be thought of as an enormous database. This could not be a currently conventional database, of course, because of the tremendous variety of the data linked by Internet. How can Internet programming advance conventional databases?

Someday, all the databases that are potentially linked by the Internet may be perceived as a single database. What can database programming offer?

Internet programming has exploded in its short life, and, without often consulting database technology, has proliferated many databases. For instance, the IMAP protocol for electronic mail reception and management is a complicated version of a simple database. Search engines offer new kinds of query capabilities over the predominently text database(s) of the World Wide Web. The Domain Name System itself, fundamental to the conversion between hostnames and IP addresses, is a distributed database which uses novel ideas to compromise between speed and accuracy.

Databases have had a somewhat longer history and a much more commercial culture. This forced standardization which was premature from the point of view of the technology and had the consequence of inhibiting later development of fundamental concepts and causing new functionalities to be clumsily retrofitted onto the standards. The challenges of the Internet, and a move from querying to programming databases, may provide the incentive to clean up the conceptual basis.

This paper uses a simple workflow problem, involving registrations at a conference, to motivate an easy language facility to link databases across the Internet. The database programming language thus extended is already at the level of abstraction, required by its use of secondary storage, that suits Internet latency. It also has the facilities of event handling and relation nesting that the Internet link requires. We show how these facilities fit easily into syntax based on a strong conceptual framework which makes maximum use of a minimal number of ideas.

The most important of these ideas is the "domain algebra", which we discuss in the next section, showing its integration into the relational algebra, and illustrating with the tricky problem in electronic commerce of making change for payments in electronic scrip. In section 3, we explain handlers for update events in the language, and use recursive updates to complete the change-making example. For ease of transmission, we need nested relations, discussed in section 4. Finally, we introduce the Internet extension in section 5 and complete the workflow example.

2 The Domain Algebra

The domain algebra has been around for a couple of decades, but not widely used, which has had unfortunately expensive consequences in terms of programming effort and confusion. It is thus appropriate to review it briefly here, saving illustrative examples for section 2.2, which uses it for electronic cash.

The algebra (which should have been named "attribute algebra") constructs new attributes from existing attributes in two ways. *Scalar* operations work "horizontally" (in terms of the usual table representation of relations) along the tuples. *Aggregation* operators work "vertically" along attributes.

All the normal calculations we might expect to be able to do on *scalar* data have corresponding scalar operators in the domain algebra: arithmetic, logical operations, string operations, the usual mathematical functions, and conditional expressions. Here are some examples, where the variables (*den, qty, over*, etc.) are attributes. Each of these examples will be used later in the paper.

$den \times qty$

$(over/den)$ **min** qty

if day=`"Sun"` **then** `"Su"` **else**
  if day=`"Mon"` **then** `"M"` **else**
  if day=`"Tue"` **then** `"T"` **else**
  if day=`"Wed"` **then** `"W"` **else**
  if day=`"Thu"` **then** `"Th"` **else**
  if day=`"Fri"` **then** `"F"` **else** `"S"`

A domain algebra statement is distinguished from a relational algebra statement syntactically, and it is helpful to give that syntax here. The attribute resulting from the first expression, above, can be named *qty1* using

let *qty1* **be** $(over/den)$ **min** qty;

Special uses of the scalar domain algebra are renaming

let *qty* **be** *qty1*;

and creating constant attributes

let *serial* **be** `"000000"`;

We will omit the **let .. be** in the remaining examples of this subsection, showing only domain algebra expressions.

The *aggregation* operations of the domain algebra are of two kinds, *reduction*, and *functional mapping*, and these each break into two subcategories. We consider first reduction, and its subcategories, *simple reduction* and *equivalence reduction.* These generalize the five aggregation functions of SQL, which are written

red + **of** A // sum values of attribute A
red + **of** 1 // count
red min of A // find least value of attribute A
red max of A // find greatest value of attribute A

and the average is a scalar division of two aggregations

(**red** + **of** A)/**red** + **of** 1

The same statistics can be calculated for groups of tuples that form equivalence classes determined by having the same value for a specified attribute or set of attributes, say G, H.

equiv + **of** A **by** G, H

equiv + **of** 1 **by** G, H

equiv min of A **by** G, H

equiv max of A **by** G, H

A count we shall make use of in totalling electronic payments is

equiv + **of** 1 **by** *den*

The advantage of this notation is that it can be used to express any aggregate or statistic we can define, such as a standard deviation, without limiting the programmer to a pre-defined set of functions. Not only +, **min** and **max** can be used, but any associative and commutative operator, such as multiplication or, as we shall see later, the natural and outer joins of the relational algebra.

Functional mapping, and its group-by variant, partial functional mapping, lifts the restriction to associative and commutative operators, thus allowing subtraction, division, etc., by permitting a selected (sequence of) attribute(s) to induce an ordering on the tuples of the relation.

For example,

fun + **of** 1 **order** *serial*

is an attribute which counts a set of tuples (representing, say, bills of all one denomination) in order of their serial numbers, and

fun + **of** 1 **order** *serial* **by** *den*

does the same for the bills of each denomination.

(If we would like an attribute which numbers bills alternately 1 and 0 in this order, we would use the noncommutative and nonassociative subtraction operator, as in

fun − **of** 1 **order** *serial*.)

2.1 Actualization: Relational Algebra

Apart from the improved flexibility over SQL, the foregoing domain algebra seems pretty trivial. It involves, however, a significant subtlety, which is responsible for the considerable intellectual simplification it permits in tackling complex problems. This is that every domain algebra operation is completely independent of relations; no relation is referred to in any expression of the domain algebra. The attributes produced by the domain algebra are *virtual*. So, of course, to populate them with data, they must be *actualized* in the context of some relation or other. This can be done by the relational algebra, with no syntactic extensions.

It may be helpful to think of a domain algebra atatement as the definition of a parameterless function, whose name is the identifier following **let** and whose body is the expression following **be**. No execution results when the language interpreter encounters the domain algebra statement;

execution is postponed until the newly defined attribute is used somewhere in the relational algebra.

The advantage of the independence of the domain algebra from relations is that we can think separately about the two aspects of any problem which involves both relational operations and calculations on the attributes. It also enables us to implement the domain algebra in any syntax which already contains relational operations, such as SQL, without syntactic modifications.

However, anticipating the need in nested relations for relational operators, such as natural join, to appear explicitly in **red** and **equiv** expressions of the domain algebra, we are going to have to alter the formulation of SQL. We discuss this now, because the relational algebra is, as we said, the context in which virtual attributes are actualized. We need a syntax which makes explicit the unary and binary operators of the relational algebra, and which clearly distinguishes expressions from statements. We call this "programming notation", to distinguish it from SQL which, as a query language, was never intended for programming.

The most straightforward means of actualizing a virtual attribute created by the domain algebra is through *projection*. Here is an example, using first SQL then programming notation.

Suppose we have a relation

CashRegister(*den, serial*),

which gives the denominations and the serial numbers for each bill in the cash register, and the domain algebra

let *qty* **be equiv** + **of** 1 **by** *den*;

let *qty1* **be** (*over*/*den*) **min** *qty*;

Then *qty* and *qty1* can be actualized, together with the other relevant attributes, using

select *den, qty, qty1* **from** *CashRegister*

The programming notation for the projection is almost the same

[*den, qty, qty1*] **in** *CashRegister*;

and there is no apparent motivation for the change. But let us go on to the next step we will have to take in the next section, namely to rename *qty1* back to *qty*.

let *qty* **be** *qty1*;

We need two projections, one after the other, because if we try to actualize the virtual *qty* in a relation which already has a *qty* there would be ambiguity in the result. In SQL, we would have to write this

select *den, qty*

from

select *den, qty1*

from *CashRegister*;

while programming notation is tidier

[*den, qty*] **in** [*den, qty1*] **in** *CashRegister*;

So far, the notational differences are not a big deal. Let us go on to *selection*, which SQL writes

select <attribute list>

from <relation>

where <tuple conditional expression>

and programming notation writes

[<attribute list>] **where** <tuple conditional expression> **in** <relational expression>

This rearrangement of the SQL order puts the <relational expression> last so that compound expressions are more easily further articulated.

Programming notation raises the possibility that a result may have *no* attributes. We now assert that a *nullary* relation serves as a Boolean, with the empty state interpreted as `false` and the non-empty state as `true`. We can therefore also use the nullary projection,

[] **in** <relational expression>

which is true if the value of the relational expression contains any tuples, and can be pronounced "something in <relational expression>".

We proceed to *binary* operators, the joins, which programming notation writes as infix expressions of explicit operators. This also allows compound expressions to be built up, and several types of join to be defined. We use three in this paper, extensions to relations of the set operations of intersection, union, and difference.

Programming notation is

<relational expression> <join operator> <relational expression>

or, if the join attributes must be named explicitly because their names are not common to the two operands,

<relational expression> [<attribute list> <join operator> <attribute list>] <relational expression>

The natural join generalizes set intersection, and we write the operator as **natjoin**. The similar generalization of set union gives the outer join, which we call **union**. Finally, set difference becomes **diff**, the difference join. Since in the paper we mainly use these latter two as ordinary set operators, we do not explain the extensions, which require null values in the case of **union**.

We can discuss the difference in notation in terms of a natural join between a relation *Pay*(*registrant, paid*) and a relation *Service*(*registrant, numdays*). (This pairs up all registrants in relations showing how much they paid and how many days they plan to attend.) SQL uses an expression which closely resembles its unary operators,

select *

from *Pay*, *Service*

where *Pay.registrant* = *Service.registrant*

While this makes explicit that equality across relations between the attributes of the names, *registrant* and *client*, is tested, it does not make explicit the higher-level idea that a natural join is being described. The programming notation, *Pay* **natjoin** *Service*

uses the **natjoin** operator so that we can explicitly ascribe all the known properties of the natural join to this expression.

The apparent greater generality of SQL joins is only apparent: programming notation can always use a selection operator to change the join condition from equality to something else. On the other hand, useful operations such as outer and difference joins cannot be expressed using only the above SQL.

So far, we have looked at *expressions* only of the relational algebra. They may be combined into compound expressions of arbitrary complexity (although it is not usually practical to write more than a join or two, and a couple of selection-projections in a single expression). We must say how relational expressions can be written into statements of the programming language. There are two kinds of statement.

The *assignment* statement creates a new relation, or overwrites an old one, from the value of a relational expression, just as assignment statements in any imperative programming language do. The syntax is

<identifier><− <relational expression>;

The domain algebra and all the relational algebra operators described so far, except for the assignment operator, are *functional*, in the technical sense that the same operands always give the same results, and there are no side-effects. Functional programming is very elegant and avoids most of the opportunities a programmer has for error in non-functional languages. Database programmming, however, can be *purely* functional only by copying entirely every relation one wishes to "change", and this is prohibitively expensive. So we need non-functional **update** operations, which have the side-effect of changing a relation in place.

SQL offers three separate commands for this, one for each of insert, change, and delete operations, with the insert command unfortunately differing from the other two by unfortunately working with only one tuple. It is cleaner to make them all relational and to avoid the ambiguity of the word, "update" by using it only to refer to relations, not tuples. It is also cleaner to isolate the update operations from the conditions determining which parts of the relation are affected. The selection-projection and joins already discussed are ample to provide this control.

Here are the three cases of the **update** statement.

update R **add** S;

update R **delete** S;

update R **using** S **change** <statements>;

S is always a relation, and the **using** S clause (which is optional) in the change command uses the natural join of S with R to select the parts of R that will change. The <statements> in this case are usually assignment statements changing values of attributes. They may contain domain algebra expressions on the right hand side. S in the **change** case may also be preceeded by a join operator, if simple **natjoin** is not enough for the task. Of course, S may be any relational expression in all three cases.

We have introduced three binary operators of the relational algebra. We have reviewed the programming notation we will use in the paper for these operators and for the unary selection-projection combination, and we have justified the programming notation as a replacement for the SQL query language for flexibility and generality.

We need only now return to the actualization of virtual attributes, which we have illustrated in the context of projection. *Anywhere* in the relational algebra that an actual attribute can occur, so can a virtual attribute. It can be actualized in a join as well as in a projection; it can be tested in a selection or in a join. Furthermore, anonymous domain algebra *expressions* can often replace the name of an actual attribute, with the exception being any usage that would place an anonymous attribute in the result relation.

Even this exception is partially lifted when we come to nested relations in section 4. For the moment, we can illustrate with extracting the total amount paid from a relation *Payment*(*den, qty*).

Paid <− [**red** + **of** (*den*×*qty*)] **in** *Payment*;

Normally this assignment would make *Paid* a relation, but since the attribute in the projection list has no name (because it is an expression), the relation would have no attribute. Thus it is instead considered to be a scalar value. Its type is the type of the attribute epression, **red** + **of** (*den*×*qty*), presumably integer. Note that this expression defines only one value—the resulting relation would be a singleton; this must be the case or else the scalar would be multi-valued.

2.2 Electronic Cash

This section begins a discussion of how to make change automatically when dealing with electronic cash. Numerous proposals have been made since the late 1980s (for example, Mondex [www.mondex.com], CyberCash [2], DigiCash [www.digicash.com], and Millicent [3]) for a secure method of making cheap, anonymous payments over the Internet. These proposals were made possible by the combination of public-key encryption and digital signature published in 1978 by Rivest, Shamir, and Adelman [6].

We can briefly discuss the issues of electronic payment leading to the need to make change electronically by looking at the advantages and disadvantages of conventional methods of payment. First, for each of c customers to keep an account with each of v vendors gives rise to the "n^2" problem that the total number of accounts is somewhere between c **max** v and $c \times v$. Not only is this unwieldy, but it also involves expensive and potentially unreliable accounting and billing. In a small community, where every vendor knows every customer, there is little chance of forgery in the form of mistaken identity, but unreliability increases with the size of the population.

Second, with credit cards, debit cards, or cheques, a third party is involved, potentially increasing the complication and expense of verifying (and billing) the transaction.

Third, with all the above, there is an extended risk if any one transaction is successfully attacked: the account lies open to further attacks until the owner's money runs out or some upper limit is reached.

A cash transaction is not subject to extended risk. Even if a mugger gets all the cash from your wallet, your loss is limited. Cash avoids the "n^2" problem by using a third party, but the government, unlike a bank, needs only to authenticate the bills and coins when they are made, and needs to do no accounting or billing. Forgery is always possible, and must be especially guarded against with electronic cash (forgery is called the "replay attack" in this context), but is deterred by sufficient skill in making the bills and coins, and especially by serial numbers.

Electronic cash can avoid the expense and extended risk of, say, sending a credit card number across the Internet, but to achieve this must do what physical cash does, namely use bills or coins (called *scrip*) of a fixed set of denominations, authenticated by a reliable third party (which does not need to be a government). Hence the need for making change.

Suppose that payment has been made by a relation

Pay(*den*	*serial*	*scrip*)
100	314159	..
100	271828	..
50	618033	..
20	141421	..
20	282842	..

in which *den* is the denomination of the scrip, *serial* is its serial number, and *scrip* is an attribute containing all this information, the digital signatures of the authenticating third party, and any other data, encrypted or not, deemed necessary to support the scrip. Since this paper defers to the extensive literature for these details, we do not elaborate on *scrip* here, and will, indeed, leave it out of the rest of this example.

It is easy to calculate that we have been paid \$290.

Paid <− [**red** + **of** *den*] **in** *Pay*;

Here is the intermediate result, showing the reduction as an unactualized attribute, and so outside the relational parentheses.

Pay(*den*	*serial*)	**red** + **of** *den*
100	314159	290
100	271828	290
50	618033	290
20	141421	290
20	282842	290

Since the result, 290, is a constant value, the same for every tuple (and of course not stored, because it is virtual), projecting this attribute alone gives a singleton relation. Since the attribute is anonymous, the singleton relation is treated as a scalar, *Paid* = 290.

The *Pay* relation can also be summarized as two \$100 bills, one \$50, and two \$20.

let *qty* **be equiv** + **of** 1 **by** *den*;

Payment <− [*den*, *qty*] **in** *Pay*;

Payment(*den*	*qty*)
100	2
50	1
20	2

Then

Paid <− [**red** + **of** (*den*×*qty*)] **in** *Payment*;

gives the same result as before.

Now suppose what we are paying for is a conference registration for the following four days, at \$69 per day.

Days(*day*)	**red** + **of** 69
Mon	276
Tue	276
Wed	276
Thu	276

Fee <− [**red** + **of** 69] **in** *Days*;

gives *Fee* = 276.

Now we can begin to make change. Suppose we have a *CashRegister* with the following contents.

CashRegister(*den*	*serial*)	*qty*	(*den*	*serial*)	*qty*
100	628319	5	10	986960	4
100	942478	5	10	197392	4
100	125664	5	10	394784	4
100	251327	5	10	296088	4
100	188496	5	5	101321	3
50	471238	4	5	506606	3
50	785398	4	5	253303	3
50	109956	4	1	177245	9
50	172788	4	1	886227	9
20	159155	7	1	443224	9
20	795775	7	1	125331	9
20	530516	7	1	797885	9
20	397887	7	1	310062	9
20	318310	7	1	146459	9
20	636620	7	1	682784	9
20	106103	7	1	214503	9

First we put the *Pay* received into the *CashRegister*.

update *CashRegister* **add** *Pay*;

Then we figure the overpayment.

Over <− *Paid* − *Fee*;

Since the change will consist of denominations no greater than the overpayment, we make a workspace copy of only this part of the *CashRegister*.

CRcopy <− **where** *den*≤*Over* **in** *CashRegister*;

The manual process of making change consists of reaching into the slots in the cash drawer containing the suitably small denominations, and taking the top bills in the stack (or nearest coins in the heap) until the overpayment has been reduced to zero. Imitating this process would require us to construct stacks or heaps of scrip of each denomination and then work downwards through the denominations figuring how many bills/coins of each denomination to return. It is not hard to model a set of stacks in a relation by creating an attribute which gives the orders of receipt of the bills of each denomination, but since we have the serial numbers of the scrip, we will skip the stack-building and select the low serial numbers instead. The overall cost of the process that follows is the same as the cost of making and popping all the stacks.

In our example, *Over* is \$14, and *CRcopy* consists of all those bills in *CashRegister* of denominations \$10 or less. We start by deciding how many \$10 bills we need to extract, which is given by the integer quotient of $Over/den = 1$. Of course, there may be no \$10 bills in *CRcopy*, or not enough in general, so we must take into account the current *qty* of these bills.

let *qty1* **be** $(Over/den)$ **min** *qty*;

We rank the bills of a given denomination in *CRcopy* according to their serial number.

let *rank* **be fun** $+$ **of** 1 **order** *serial*;

Finally, we select the highest of the denominations in *CRcopy*, perform the above calculations, remove the denomination from further consideration in *CRcopy*, and finally reduce the overpayment by the change made so far.

NextDen <− **where** *den* = **red max of** *den* **in** *CRcopy*;

NextChg <− **where** *rankleqqty1* **in** *NextDen*;

update *CRcopy* **delete** *NextDen*;

Over <− *Over* − [*qty1*×*den*] **in** *NextChg*;

For the example, *CRcopy* now has only denominations of \$5 or less. Here is *NextDen*, with the \$10 bills extracted from *CRcopy*, the virtual attributes shown, and the tuple selected for *NextChg* shown in bold.

NextDen(*den*	*serial*)	*qty*	*qty1*	*rank*
10	197392	4	1	1
10	296088	4	1	2
10	394784	4	1	3
10	986960	4	1	4

This process is now repeated to find that no \$5 bills and four \$1 bills are also to be included in the change. The repeats could be executed by a loop, but we have not discussed looping in the programming notation of the database language. So we will use instead a method of recursive updating, which needs no additional syntax. This requires us to consider event programming, which is also used for active databases and which we shall use later for asynchronous Internet communications.

3 Events and Active Databases

Events in the context of programming languages are not what the word, "event", implies in English. *An event is a system-generated procedure call.* The procedure that is called is commonly known as an *event handler*, although it is in fact merely a procedure with a specially formulated name linking it to the circumstances under which it should be called. [1] For example, when a relation is updated by **add**, **delete**, or **change**, the response can be coded in procedures named, respectively,

[1]This definition precludes syntax for combinations of "events" (such as more than one bank withdrawal per hour, or withdrawals *or* deposits on a certain day). But the definition we give allows us to build a log of such "events", and use ordinary database querying or further event handlers to detect these compound events. The great variety of possibilities one might want to check for makes special syntax counterproductive in any case, given the general query capabilities already available.

with the following syntax.

[**pre**|**post** :] **add** : <*relation*>
[**pre**|**post** :] **delete** : <*relation*>
[**pre**|**post** :] **change :** <*relation*>:<*attribute*>

where **pre** means that the procedure is to be invoked before executing the update, and **post** (the default) means call the procedure afterwards.

For example, each time the conference secretariat updates the *registrants* relation with a new *registrant*, the total number of banquet attendees is to be calculated and sent automatically to the caterer.

update *registrants* **add** *registrant* ;
comp *post* :*add* :*registrants* () **is**
{ **let** *sumBanq* **be red** + **of** *banq* ;
  let *To:* **be** `"caterer@confCenter.com"`;
  let *From:* **be** `"secretariat@confCenter.com"`;
  let *Subject:* **be** `"new tally for Banquet"`;
  Mail <– [*To:, From:, Subject:, sumBanq*] **in** *registrants* ;
}

where we imagine the email thenbeing automatically sent.

To explain the keyword, **comp**, and its syntax, we must discuss procedures. A procedure in a conventional programming language, if it has no side-effects, is a relationship among its parameters. Some of these are provided as input, and some are output from the procedure. Since this is just what a selection (see section 2.1) on a relation does, we already have the concepts and the syntax for invoking procedures. In fact, this way of thinking generalizes the conventional procedure, because one relation can be used in many selections, each with different sets of input and output parameters. The resulting relational generalization of procedures gets a new name: *computation.*

3.1 Computations

A computation can be thought of as a compressed relation, in which the relationship is given not explicitly by data but implicitly by code. For example, a currency converter is a relationship among *fromAmt, toAmt,* and *rate* given by

comp *ConvertCurr*(*rate, fromAmt, toAmt*) **is**
    toAmt <– *rate*×*fromAmt*
alt *fromAmt* <– *toAmt*/*rate*
alt *rate* <– *toAmt*/*fromAmt* ;

where the **alt** keyword separates the various alternative evaluations of the relationship that are permitted.

Although *ConvertCurr* is code and we call it a computation or **comp**, it must always be thought of as a relation. That way, we do not need any new syntax to invoke it. A selection-projection will do:

Amount <− [*toAmt*] **where** *rate*=`0.65` **and** *fromAmt*=`446.15` **in** *ConvertCurr*;

gives `290.00` in *Amount*(*toAmt*). Similarly,

AnotherAmount <− [*fromAmt*] **where** *rate*=`0.65` **and** *toAmt*=`290.00` **in** *ConvertCurr*;

gives `446.15` in *AnotherAmount*(*fromAmt*), and

TheRate <− [*rate*] **where** *fromAmt*=`446.15` **and** *toAmt*=`290.00` **in** *ConvertCurr*;

gives `0.65` in *TheRate* (*rate*).

We see that not every possible combination of "input" and "output" attributes is permitted: no provision has been made for calculating two of the attributes from a third, for example (because the output would need an infinite number of tuples). This limitation introduces *type*s of computations (relations), which match **alt**-blocks to permitted selection-projection operations.

To make the notation less cumbersome and more like procedures in ordinary programming languages, syntactic sugar can be used for these very special kinds of selection-projection. The above three invocations become, respectively

Amount <− *ConvertCurr*[`0.65,446.15`];

AnotherAmount <− *ConvertCurr*[`0.65,,290.00`];

TheRate <− *ConvertCurr*[`,446.15,290.00`];

Computations may also participate in joins, so if we had a relation, *unconvertedAmount*(*fromAmt, rate*), with many amounts and possibly different conversion rates, we could convert all the amounts using their respective rates with a single statement,

Amount <− *ConvertCurr* **join** *unconvertedAmount*;

(This is an old idea from the Peterlee Relational Test Vehicle [9].)

3.2 Recursive Updates

Event handlers are parameterless computations with special names, as explained and illustrated just before section 3.1. Because they have no parameters, event handlers have no **alt** keywords (that is, only one **alt**-block).

If the event handler uses the same update that caused it to be invoked, we have what can be called recursive updating. We use this idea to continue our example of making change.

In that example, at the end of section 2.2, we created *NextChg* with one ten-dollar unit of scrip out of the $14 that we needed to create in change. Then we said that recursive updating (or a loop) would make the rest of the change. To do this, we need a relation to accumulate the change, *GiveChg*, and an update statement.

update *GiveChg* **add** *NextChg*;

which puts the scrip for the change into *GiveChg*.

Now we put all the code we wrote into an event handler for just this same update.

```
comp add :MakeChg() is
{ let qty1 be (Over/den) min qty;
  let rank be fun + of 1 order serial;
  NextDen <- where den = red max of den in CRcopy;
  NextChg <- where rank≤qty1 in NextDen;
  update CRcopy delete NextDen;
  Over <- Over − [qty1×den] in NextChg;
  update GiveChg add NextChg;
}
```

Since the last statement is a further **update .. add** to *GiveChg*, the handler will be called again and the process will repeat for the next denomination and the new value of *Over*. The recursion stops when the update makes no further change.

This stopping condition causes a problem in the very next iteration in this example. *GiveChg* now contains the $10 bill with serial number `197392` and *Over* is `4`. Since *NextDen* will hold the three $5 bills and *Over*/*den*=0, *qty1* is 0 and there will be no tuples in the $5 *NextChg* to update *GiveChg*. This constitutes a null update (no change to the target relation) and the process will stop without adding the four $1 bills to *GiveChg*.

Accordingly, we must test for this situation and force an update even for the $5 case. The update we will make is to include a dummy bill with serial number `000000`. The test requires conditional statement syntax, and the idea that a relation with no attributes is a boolean, with value **false** if it is empty and **true** otherwise.

```
if ![] in NextDen
{ let serial be "000000";
  NextDen <- [den, serial] in [den] where den = red max of den in CRcopy;
}
```

This code can be inserted into the **comp** just before the last, update, statement. The recursion will now stop only when *CRcopy* runs out of denominations.

We must also start the recursion. We do a fake update and add a $0 bill.

```
relation Start(den, serial) <- {(0, "000000")};
update GiveChg add Start;
```

The sequence of values for *Over, NextDen*, and *GiveChg* that results from this process applied to our *CashRegister* for the example is

Over	*NextDen*(*den*	*serial*)	*GiveChg*(*den*	*serial*)
14	10	197392	0	000000
	10	296088		
	10	394784		
	10	986960		
4	5	101321	0	000000
	5	253303	10	197392
	5	506606		
0	1	125331	0	000000
	1	146459	10	197392
	1	177245	5	000000
	1	214503		
	1	310062		
	1	443224		
	1	682784		
	1	797885		
	1	886227		
			0	000000
			10	197392
			5	000000
			1	125331
			1	146459
			1	177245
			1	214503

The zero serial numbers must be removed

update *GiveChg* **delete where** *serial*="000000" **in** *GiveChg*;

and the bills must be taken from *CashRegister*

update *CashRegister* **delete** *GiveChg*;

4 Nested Relations

In our discussion so far, the hypothetical registrant has had to provide two relations to the conference secretariat, which we have processed in section 2.2, above. These are *Pay*(*den, serial, scrip*) (from which we have omitted the *scrip* attribute for brevity), and *Days*(*day*). To make the Internet transactions cleaner, we would like to package these relations, together with other data such as the registrant's name and accompanying persons, in a single construct. Nested relations provide this construct.

Nested relations are relations whose attribute values may themselves be relations. Thus, to work with them, we need a formalism to manipulate relation-valued attributes (and to create new relation-valued attributes from existing ones). The domain algebra, with relational operations incorporated into it, is the required formalism.

With the domain algebra and a cleaned-up relational algebra, we get nested relations for free (syntactically, that is: the implementation is more difficult, although everything we are about to say can be built with non-nested relations). The linguistic step is to subsume the relational algebra into the domain algebra.

Although nested relations add nothing whatsoever to the *functionality* of the "flat" relational and domain algebras, which is what we have discussed so far, they do at times simplify our *thinking*, and so have a rightful place in secondary storage programming. They also repair an aesthetic inegality of the data types in the programming language: numbers, strings, and other scalar data types have hitherto had privileges that relations have not, namely the ability to be included as values in relational tuples. Nesting gives relations these privileges, too.

Here are two tuples of a nested relation containing sample registration data, including the *Pay* and *Days* relations as attributes. We see that *Days* is a *unary* relation (only one attribute), illustrating the special case of a *set*-valued attribute. We also show *Name*, which is a *singleton* relation (only one tuple), illustrating the special case of a *tuple*-valued attribute. *Pay* is a normal relation, with more than one tuple and more than one attribute.

Regis

(*Name*		*Days*	*Pay*	)
(*last*	*other*)	(*day*)	(*den*	*serial*)
Mert	Tim	Mon	100	314159
		Tue	100	271828
		Wed	50	618033
		Thu	20	141421
			20	282842
Wang	Yan	Tue	100	628317
		Wed	50	424263

Since we must now consider the possibility of making change simultaneously for two registrants, *Paid, Fee*, and *Over* can no longer be "top-level" scalars, independent from any relation, but must be attributes each with a value for each tuple of *Regis*. Similarly, relations such as *CRcopy, GiveChg, NextChg*, and *NextDen* can no longer stand alone, but must be nested within *Regis*.

For the scalars, this is accomplished by changing the assignment statements into domain algebra.

let *Paid* **be** [**red** + **of** *den*] **in** *Pay*;

let *Fee* **be** [**red** + **of** 69] **in** *Days*;

let *Over* **be** *Paid* − *Fee*;

In the first two of these statements, the level is lifted from the nested relations, *Pay* and *Days*, to ordinary attributes, *Paid* and *Fee*. These do not have ("sub"-) attributes, because the attribute expressions in the projections creating them do not pass on any name. This just repeats for nested relations the use of anonymity to raise the level from relation to scalar that we saw in section 2.1.

The third statement is just scalar domain algebra to replace the top-level scalar calculation that we did in section 2.2.

Similarly, assignments to relations must be converted to the corresponding domain algebra. For example,

let *CRcopy* **be where** $den \leq Over$ **in** *CashRegister*;

would[2] create a new virtual attribute, the nested relation *CRcopy* from the top-level relation *CashRegister*. *CRcopy* becomes a copy of the lower denomination bills from *CashRegister* in each tuple of the nesting relation.

Creating nested *NextDen* would be similar, and would also make identical copies in each tuple. The nested *NextChg*, however, would differ in each tuple, because *qty1* depends on *Over*, which is different for each registrant ($14 for `Mert`, and $12 for `Wang`, as we will see).

let *NextChg* **be where** $rank \leq qty1$ **in** *NextDen*;

The update to *CRcopy* in the event handler would have a nested equivalent which updates the nesting relation, *Regis*, at the top level.

update *Regis* **change update** *CRcopy* **delete** *NextDen*;

Here we see an **update** .. **change** used to execute **update** in turn on a nested relation. The update to *GiveChg* would similarly be a two-level update.

Having described the simplest modification of the change-making code to extend it to nested relations, we now backtrack to the bug that this approach would have introduced. When we made identical copies of *CRcopy* and *NextDen* for each tuple of *Regis*, we ran the risk of making change for each registrant with the same bills: both `Mert` and `Wang` would be sent the same $10 bill, namely the one with the smallest serial number, `197392`, by the naive extension of the top-level code to nested relations.[3]

It is beyond the scope of this paper to consider copy-proofing data such as scrip, and we do

[2]This statement introduces a bug, not in general, but for this example, which requires that scrip may not be copied. See three paragraphs ahead; but until then we carry on, to illustrate the valid augmentations to the language to support nesting.

[3]This bug is a consequence of the intrinsic parallelism of relational operations, which a good relational programming language will retain—but not use inappropriately.

not address the issues of confounding an infinitely skilled malevolent programmer. In what follows, we suppose a highly skilled benevolent programmer. To consider a moderately skilled benevolent programmer, who might make change with duplicated bills inadvertently, we note that we have been dealing only with *den*ominations and *serial* numbers and not with the scrip itself, which we can suppose to be copy-proof, so that the replay "attack" need not wait until the issuer of the scrip finally checks serial numbers to detect the error.

The change we make to our top-level approach is to consider each denomination of change for *all* registrants at once. We rank the available bills of this denomination by their serial numbers, as before. We accumulate the numbers of bills of this denomination needed for each registrant, and we select the bills to pay to each by comparing bill rank with accumulated numbers. Here are the details for the \$10-bill change, looking first at *CashRegister*, then at *Regis*, then combining the extracts from each.

CRcopy and *NextDen* are top-level extracts from *CashRegister* as in our original, top-level calculation. We will need an attribute, *Den*, for the denomination, which we did not need before.

CRcopy <− **where** *den*≤*Over* **in** *CashRegister*;
let *Den* **be** [**red max of** *den*] **in** *CRcopy*;
NextDen <− **where** *den* = *Den* **in** *CRcopy*;
let *rank* **be fun** + **of** 1 **order** *serial*;

NextDen(*den*	*serial*)	*rank*
10	197392	1
10	296088	2
10	394784	3
10	986960	4

Den has the value 10.

We define *Paid, Fee* and hence *Over* as virtual attributes for *Regis*, as above. We also need *overCum*, the cumulative number of bills of this denomination to be given in change, and *overPast*, its predecessor. We extract the last name of each registrant by a level-change reduction (the *Name* nested relation is a singleton, so the reduction just gives back the single value), and we assume that this last name identifies the registrant uniquely.

let *lastName* **be** [**red min of** *last*] **in** *Name*;
let *Paid* **be** [**red** + **of** *den*] **in** *Pay*;
let *Fee* **be** [**red** + **of** 69] **in** *Days*;
let *Over* **be** *Paid* − *Fee*;
let *overCum* **be fun** + **of** *Over*/*Den* **order** *lastName*;
let *overPast* **be if** *lastName*=**red min of then** 0 **else**
fun pred of *overCum* **order** *lastName*;

RegisChange <− [*lastName*, *Over*] **in** *Regis*;

Regis					
()*lastName*	*Paid*	*Fee*	*Over*	*overCum*	*overPast*
Mert	290	176	14	1	0
Wang	150	138	12	2	1

Now we create *NextChg*, nested inside *RegisChange*, by selecting bills from *NextDen* whose ranks are between *overPast* and *overCum*.

let *NextChg* **be where** *overPast*<*rank* **and** *rank*≤*overPast* **in** *NextDen*;

This will automatically extract the correct numbers of different bills from *NextDen* for each registrant, and will not exceed the total number of bills available in *NextDen*. We must reduce each value of *Over* by the amount we have just put in *NextChg*.

let *Over1* **be** *Den*×[**red** + **of** 1] **in** *NextChg*;

RegisChange				
(*lastName*	*Over*)	*NextChg*		*Over1*
		(*den*	*serial*)	
Mert	14	10	197392	4
Wang	12	10	296088	2

Finally, we accumulate the change we have just made for bills of this denomination into a nested relation, *GiveChg*, which we originally created using

relation *Chg*(*den*, *serial*);

GiveChg<− [*lastName*, *Chg*] **in** *Regis*;

Here is the update.

update *GiveChg* **change update** *Chg* **add** *NextChg*;

We also update *Over* in *RegisChange*.

update *RegisChange* **change** *Over* <− *Over1*;

GiveChange			*RegisChange*	
(*lastName*	*Chg*	)	(*lastName*	*Over*)
	(*den*	*serial*)		
Mert	10	197392	Mert	4
Wang	10	296088	Wang	2

To make change for all denominations, the above code is embedded in an event handler on updates to *GiveChg*, and an initial update made to *GiveChg*. Artificial updates must be made to *GiveChg* when no change is needed for a given denomination. Finally, the contents of *GiveChg* are removed from the *CashRegister*.

update *CashRegister* **delete** [**red union of** *Chg*] **in** *GiveChange*;

Here, we see the importance of being able to name the relational operators. If we had no name for the **union** operator, we would not be able to use it in the **red**uction, to extract from *GiveChange* all the bills we have given out in change.

5 Internet Database Programming

The Internet offers several paradigms and associated protocols at the application level which might offer bases for Internet database programming. We could start at the transport layer, a level below applications, with TCP/IP [8], but the need to deal with port numbers makes this perhaps too low-level. We could start much higher with distributed objects and CORBA [4], but the essence of object-orientation is the idea of state, while statelessness is central to making Internet applications robust against link and host failures. We could use any of the file transfer protocols [8], or extend them to one of the distributed file systems (such as UNIX United [1], Sun's Network File System [7], or the Andrew File System [5]). Or we could be less elaborate and take an approach which is more widely familiar, such as email or the Universal Resource Locator (URL) of the World Wide Web.

Adapting the URL idea requires the fewest changes in syntax and semantics, and is adequate for the example application we have been developing in this paper. Without rejecting other possibilities, for the time being we focus on this approach.

The examples we have already used in this paper to illustrate database programming concepts are components of a conference registration system. Here is a simplified workflow, which we will subsequently simplify still further so that we can treat it fully in this section.

- Secretariat
  1. set up registration (web)site
  2. advertise conference
  3. receive joint conference, hotel registrations
  4. send acknowledgement to conference registrant
  5. send registration to hotel
  6. send payment to bank
- Registrant
  1. register on joint conference, hotel website
  2. receive acknowledgement from conference
  3. receive confirmation from hotel
- Hotel

    1. receive registration from secretariat
    2. send confirmation to registrant

- Bank
    1. receive payment

The database needed to support this workflow has components such as the following.

- Secretariat
    - *Registrants* (*Name, Address, Days, Events, Accompanying, Paid, Diet, Nights*)
- Hotel
    - *Guests*(*Name, Dates, Occupancy, Smoking, guestComments*)
- Bank
    - *Accounts* (*Name, Balance*)

We have already seen parts of these, and know that they contain further, nested, relations.

We limit discussion to only two agents, the secretariat and the registrant, since what we show for them extends easily to further interactions with hotel and bank. The secretariat, located at `sec.conf.org`, sets up in directory `/home/externalactivities/hicss/public_aldat` an empty relation

$$
\begin{array}{lllll}
\mathit{Regis} & & & & \\
(\ \mathit{Name} & & \mathit{Days} & \mathit{Pay} & \ ) \\
(\mathit{last} & \mathit{other}) & (\mathit{day}) & (\mathit{den} & \mathit{serial})
\end{array}
$$

with permissions set for public appending, to receive registrations. They also set up in directory `/home/externalactivities/hicss/regis2002` the (simplified)

$$\mathit{Registrants}(\mathit{Name} \quad \mathit{Days} \quad \mathit{Paid})$$

also initially empty. Then they advertise the conference.

At `mcgill.ca`, in file `/home/profs/mert/public_aldat`, a conference registrant creates

Regis				
(*Name*		*Days*	*Pay*	)
(*last*	*other*)	(*day*)	(*den*	*serial*)
Mert	Tim	Mon	100	314159
		Tue	100	271828
		Wed	50	618033
		Thu	20	141421
			20	282842
Wang	Yan	Tue	100	628317
		Wed	50	424263

from the template provided by the secretariat:

Regis <− **edit** `aldat://sec.conf.org/~hicss/`*Regis*;

where **edit** allows interactive access to the tuples of the operand relation, returning its value when the interactive session is closed. Then he sends it to the secretariat.

update `aldat://sec.conf.org/~hicss/`*Regis* **add** *Regis*; [4]

We see in these two exchanges the incorporation of a standard URL (with the prefix, `aldat`, standing for *al*gebraic *dat*a language) to access remote relations. Here, the client is at `mcgill.ca` and the server at `sec.conf.org`.

The sites for client and server are reversed for the next exchange, which is the acknowledgement, with the change that was made. The secretariat calculates the change, using the methods described above, notably in section 4, and includes it in an acknowledgement.

`aldat://mcgill.ca/~mert/`*GiveChange* <− *GiveChange*;

(together, if desired, with an indication of the days registered for). We can suppose that both client and server Internet software is included in the database programming language, together with secure transmissions supported by public/private key cryptography. (Encrytping for transmission must not inadvertently undo any encryption used in the scrip.)

(The secretariat will have used similar path labels locally to link the two directories,

`/home/externalactivities/hicss/public_aldat` and

`/home/externalactivities/hicss/regis2002`:

`../public_aldat`, for instance.)

We have not addressed how to cause the acknowledgement to follow receipt of the registration form and payment. The exchange is *asynchronous* in that a registration can be received at any time (within the deadline), unpredictable by the secretariat. This is what event handlers were

[4]Since we now intend that the scrip is included in the transmission, this statement actually makes an illegal copy of the scrip. The modification needed is a form of selection-projection which *removes* tuples from the source relation. The keyword **in** is changed to **from**. This is beyond the scope of this paper, but would modify this statement to .. **add from** *Regis*.

originally intended for: we put the code that calculates change and sends the acknowledgement into the handler

post :**add** :*Regis* ()

for the event of updating *Regis*.

The reader can now extend the exchanges illustrated here to a full-scale conference registration system, including hotel and bank agents, or to any electronic commerce, or to a distributed database system.

A couple of points may be added, extending the discussion somewhat beyond the scope of the paper, but not beyond the implemented capabilities of the database programming language it is based upon. First, beyond asynchrony managed by event handlers, we have assumed the relations to be present at the remote site when they are needed locally. To incorporate asynchronous *delay*, in case the local site must wait while the remote site creates the needed relation, or puts the needed tuples into it, we use a modified selection-projection with the keyword **where** changed to **when**. This has the effect of blocking, in the case that the original selection-projection (with **where**) would return an empty result, until the source relation is changed so that the result is no longer empty.

The second development allows the transfer from a remote site of a relation which has not been declared locally, or even whose attributes are locally unknown. This involves transmitting the attributes, too, using *attribute metadata*, which has previously been found useful in the database programming language for data mining applications. We do not elaborate here.

As a final point, since computations are relations, the URL extension to relation names works also for computations, giving us remote procedure calls with no additional syntax.

6 Conclusion

We have introduced a URL-based name extension to a database programming language which gives it collaborative and distributed capability over the Internet with no changes in syntax or semantics apart from the new structure in names.

We use handlers for update events to manage the asynchronicity of the exchanges with remote sites.

We use nested relations to package complex transfers into a single name.

We express the change-making problem of electronic commerce with vendor- and customer-independent scrip as a recursive update, mediated by event handlers.

All of this depends heavily on the domain algebra and its actualization in a programming (as opposed to query) notation for the relational algebra. Our discussion of these facilities implicitly answers the question that would occur to most conventional readers (why not use some combination of existing database and Internet facilities, such as SQL plus JDBC plus Java?). Although the scope of the paper does not permit elaboration of the full potential of an integrated database programming language (for example, for administrative data, expert systems, data mining, and geographical information systems), it is clear why we should prefer a single conceptual and syntactic framework for our programming over two or three forcefully juxtaposed separate frameworks.

Most of the database/Internet programming facilities used in this paper are implemented and running in one or more research systems. The major exception is the URL naming structure, and this will be running in time for the HICSS conference.

Future work includes considering the Internet as an enormous database. Although adequate data structures and remote procedure calls both arise easily out of our approach, data organization and indexing problems must still be addressed, and the remote processing capabilities of the Internet must also be included.

7 Acknowledgements

We are indebted to the Natural Science and Engineering Research Council of Canada for support under grant OGP0004365. This work was motivated by work on Internet support for *spatial* decision management, supported by the Networks of Centres of Excellence program through the GEOIDE Project, GEODEM. Zongyan Wang is building database URL support into the *relix* implementation of the language.

References

[1] J. P. Black, L. F. Marshall, and B. Randell. The architecture of UNIX united. *Proceedings of the I.E.E.E.*, 75(5):709–18, May 1987.

[2] S. Crocker, B. Boesch, A. Hart, and J. Lum. Cybercash: Payments systems for the Internet. In R. Coggeshall, editor, *INET'95 Hypermedia Proceedings*, page C2, Honolulu, Hawai'i, June 1995. Internet Society, www.isoc.org. http://info.isoc.org/HMP/PAPER/181/.

[3] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The MilliCent protocol for inexpensive electronic commerce. 1995.

[4] Object Management Group. Common object request broker architecture. 1992.

[5] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[6] R. Rivest, A. Shamir, and L. M. Adelman. On digital signatures and public key cryptosystems. *Communications of the A.C.M.*, 21:120–6, 1978.

[7] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *USENIX Conference Proceedings*, Berkeley, CA, 1985. USENIX Association.

[8] W. R. Stevens. *TCP/IP Illustrated Volume 1 The Protocols.* Addison-Wesley, Reading, Mass., 1994.

[9] S. J. P. Todd. The Peterlee relational test vehicle—a system overview. *I.B.M. Systems Journal*, 15(4):285–308, 1976.