CS++: Reinventing Computer Science (for Secondary Storage)

T. H. Merrett[*]
McGill University, Montreal, Canada

March 11, 2005

Abstract

Secondary storage ("SS") offers a significantly different memory organization from RAM, which most of computer science is geared to support. This forces a revision of data structures and algorithms on one hand and of programming language on the other. Algorithms become simpler, but in the frequent situation that the data structure sizes exceed RAM capacity, even polynomial complexity becomes intractable unless it is sub-quadratic. Languages are obliged to abstract over looping, which presents opportunities for programming at a much higher level than allowed by the "von Neumann bottleneck" that restricts most languages geared for RAM.

This paper describes the challenges of developing data structures and language which operate at the level demanded by SS. We look at some benefits of abstracting over looping: enormous reductions in code for building large systems, intrinsic parallelization, and easy incorporation of the Internet. We note a diversity of SS applications, including expert systems, data mining, and semistructured data, as well as conventional organizational management. We will see that Computer Science SS also has benefits for ordinary RAM programming, and will touch on some data structure ideas as illustration.

Keywords Algorithms, Data structures, Programming language, Secondary storage

1 Introduction

Four decades of research and commercialization have been dedicated to perfecting systems to support data which are large, persistent and shared. These are all characteristics of secondary storage. The "++" in the title of this paper is intended to represent "SS": *CS-SS, computer science for secondary storage*.

Secondary storage serves this role because it is cheap. Prices have been falling for primary memory ("RAM" or "random-access memory"), but SS prices have dropped faster. The technical cost of this monetary economy per bit is that SS requires totally different data organization than does RAM.

The central difference is "latency", the time it takes to *locate* the data to be processed. What is important is that this is a *relative* difference: it is relative to the time needed to *process* the data, once found. This difference can be quantized by the *access-transfer ratio*, which is the number of bytes that might have been processed during the time it took to locate the data for processing. An ideal definition of RAM gives it an access-transfer ratio of 1. The typical SS access-transfer ratio today is close to 1 million. (Remarkably, as SS technology improves, i.e., becomes cheaper, the access-transfer ratio increases: fifteen years ago, it was closer to ten thousand.)

This degree of relative latency applies not only to SS but also to networks of computers. Even RAM meets a small relative latency (in comparison with caches, for example), although this access-transfer ratio is now only about ten. Thus, the experience of secondary storage research and practice, which seriously encountered the problem first, is directly applicable in a much broader domain.

[*]Copyleft ©T. H. Merrett, 2005

The significant consequence of large relative latency is that data, once found, must be transferred in considerable quantities, called "blocks" or "pages", preferably of the order of the access-transfer ratio itself, but certainly of thousands of bytes. This is entirely to reduce the per-byte overhead of handling the data. It means immediately that, to avoid inefficiencies of the order of the access-transfer ratio, or at least of the block size, data must be organized to be completely processed the one time the block it is on is transferred to RAM.

This concern with data organization leads to data structures and algorithms which are unique to SS. They often resemble their RAM cousins, but are costed differently, of course, in terms of numbers of accesses rather than in terms of byte operations. They are often simpler than the alternatives developed for RAM, only because of their more severe constraints. Thinking about SS data structures also happens to have on occasion provided the key to RAM data structure problems.

Data structure differences are reflected in programming language differences. The most important consequence of needing large blocks of data is that these force the language designer to abstract over looping. Only thus can the programmer avoid writing code which would work fine in RAM but would make the repeated accesses to the same block that must be avoided on SS. This immediately moves an SS programming language to a much higher level of abstraction than is found in "high-level" programming languages built for RAM.

Some of this higher level of abstraction is seen in database query languages, but they, while suggestive, are limited to expressions of various kinds of logic. A programming language, on the other hand, must support procedural abstraction, data abstraction, typing, recursion, and, yes, even looping, among other capabilities.

This paper provides an overview of selected topics from data structures, databases, and programming. The overview is deep in that, where we illustrate with code, we give the full code, but superficial in that we do not attempt to teach the language in which the code is written. That would take more space than we have, and is available elsewhere (notably [**?**]). What we do here is *start* a number of topics, many of them major research areas with years of work already accomplished by many workers. Our purpose is to show how these topics can be simplified and approached directly through the above abstractions, which are fuelled by the demands of secondary storage. This is the sense in which secondary storage has reinvented computer science.

The next section discusses four examples from algorithms and data structures for secondary storage: variable multidimensional arrays; finding all substrings (in a genomics example); variable-resolution maps; and lossless data compression. The third section looks at seven examples significantly ameliorated by language supporting programming at a very high level: software engineering of large projects made suddenly much smaller; automatic parallelization (of simple algorithms for numerical linear algebra); an inference engine for an expert system; processing the datacube for classification mining of data; dealing with marked-up text and semistructured data; and database distribution via Internet.

2 Algorithms and Data Structures

Secondary storage forces us to think in terms of transferring large amounts of data and to make full use of all this data, once transferred. We could show the simplifications to familiar algorithms and data structures, such as sorting or hashing, which arise from this new thinking, but the following four examples should make the point.

2.1 Variable multidimensional arrays

A problem which went unsolved until approached from the fresh perspective of secondary storage is how to add new data to a multidimensional array, for example, a new row or column. For one dimension, we just append the new data to the end of the array. For higher dimensions, since the array still must be mapped to a one-dimensional set of storage addresses, the addition is more problematical. Here is an example in two dimensions. (Two dimensions crack the problem: arrays of more than two dimensions can be treated by a straightforward generalization.)

A Leontieff matrix of an economy is a good example where a fairly large amount of data is represented as a two-dimensional array, which may need to be changed from time to time. Here is a fictitious example in which Charles, Fred and Harry supply clothing, food and housing, respectively, to themselves and the others, and which must be extended at a later date to accommodate Pete, who provides power. (We show total earnings and total expenses for each of these agents, just to illustrate which way the matrices are directed, although they are not part of the stored data.)

out\in	C	F	H	*earned*
C	2	1	3	6
F	2	2	3	10
H	2	3	4	9
spent	6	6	10	

out\in	C	F	H	P	*earned*
C	2	1	3	3	9
F	2	2	3	3	10
H	2	3	4	2	11
P	1	3	5	1	10
spent	7	9	15	9	

The original matrix might be stored in memory, in row-major order, as

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CC	CF	CH	FC	FF	FH	HC	HF	HH							

in which CF represents the value in row C (Charlie) and column F (Fred), and so on.

The conventional representation of the matrix augmented by P (Pete), also in row-major order, would be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CC	CF	CH	CP	FC	FF	FH	FP	HC	HF	HH	HP	PC	PF	PH	PP

but note the displacement of original elements FC, FF, FH, HC, HF, and HH by the new elements CP and FP.

Instead of the formula $a = i + 4j$, which gives the address, a, in terms of row i and column j for the second row-major representation, we can find an alternative which does not displace any of the elements of the original. We do this by adding *axial arrays* which store base addresses for each row and for each column, say *rowbase*(i) and *colbase*(j). These simply store the addresses, respectively, for the 0th column and the 0th row. Then the element address for the expanded array is the original address, $a = i + 3j$, for any original element, or

max(*rowbase*(i),*colbase*(j)) + the other one, i or j

for the new elements.

This approach comes from a problem in secondary-storage data representation in multiple dimensions in which the array elements are pages which cluster data in, say, two dimensions, but store to a one-dimensional addressing scheme. For such pages, we already have axial arrays, which index the data on the pages for fast retrieval. It is an easy step from here to *rowbase, colbase*, and so on.

The approach works because we can think of the array being built up by adding "slabs" of elements to each face. These additions have a history, and the history is captured by *rowbase* and *colbase*. Here for our example is the extended Leontieff matrix, in which column 3 was added first, starting at address 9, and then row 3 was added, starting at address 12.

	j	**0**	**1**	**2**	**3**
i		**0**	**1**	**2**	**9**
0	**0**	**0**	**1**	**2**	**9**
1	**3**	**3**	**4**	**5**	**10**
2	**6**	**6**	**7**	**8**	**11**
3	**12**	**12**	**13**	**14**	**15**

In this illustration, *rowbase* is the column of numbers, `0, 3, 6, 12`, beside the column of i indices, *colbase* is the row of numbers, `0, 1, 2, 9`, below the row of j indices and the numbers in the boxes are the addresses of the elements. The boxes give the history if we stipulate that the array must be rectangular at every stage. The illustration shows the original 3-by-3 matrix and only two stages of addition. In fact, the whole matrix could be grown from a 1-by-1 by a succession of stages, and we could dispense with the row-major addressing altogether. Furthermore, a third dimension could be added and its growth captured by a third axial array, and so on for any number of dimensions. References: [**?**, **?**].

2.2 Finding all substrings

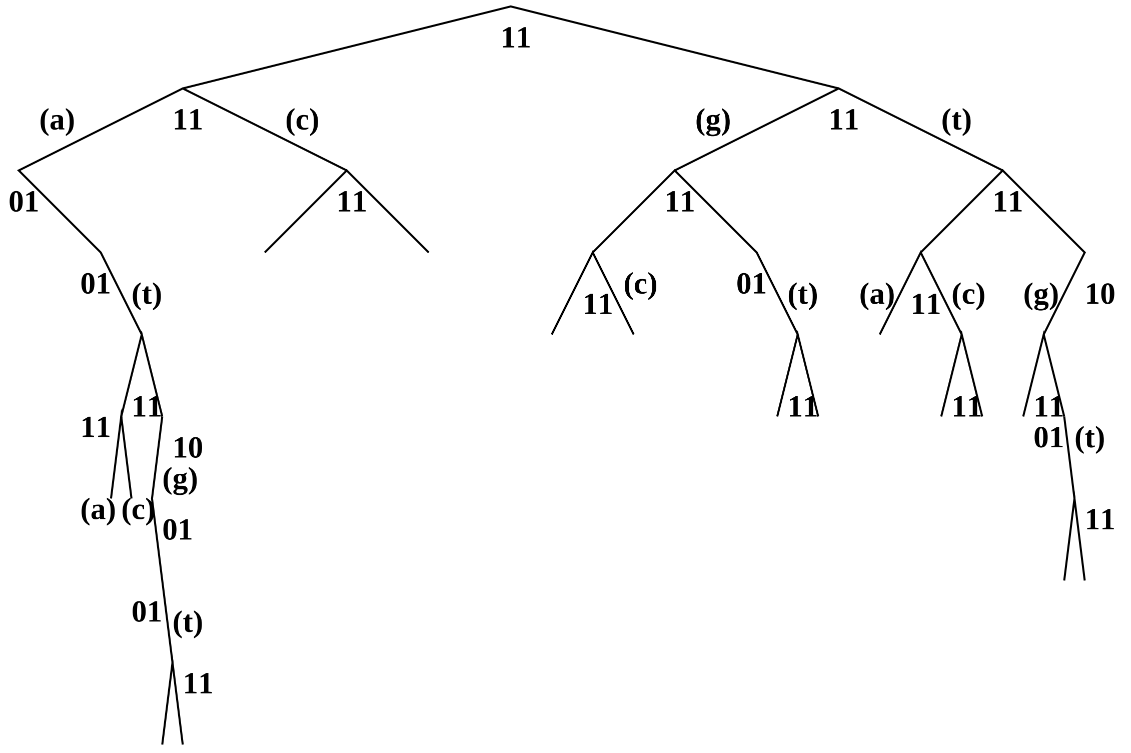
To find *all* substrings of a sequence of n characters effectively requires a search of $\mathcal{O}(\backslash^{\in})$ characters, namely the original n characters plus $n-1$ characters starting at the second, $n-2$ starting at the third, and so on. A basis for such searches was proposed in 1968 [**?**], building on the "trie" structure introduced nine years earlier [**?**]. Tries make possible both sublinear search time for substrings and massive data compression, but both these advantages required that tries be exploited for secondary storage.

Trie methods have been used with great success for data in the form of text, but it is currently topical to consider an example from genomics. Here are 16 base letters from the Mycobacterium tuberculosis genome, starting at codon 729.

`atgtcatatgtgatcg`

Here, `a` and `g` stand for the purines, adenine and guanine, respectively, and `c` and `t` stand for the pyrimidines, cytosine and thymine, respectively: these bases occur in DNA in pairs, `at` and `gc`, and we have shown only one side of the above DNA fragment. For various reasons, we will encode these four letters in binary as `a = 00, c = 01, g = 10` and `t = 11`.

With this encoding, `atgtcatatgtgatcg` occupies 32 bits, and the 16×17/2 = 136 letters representing every possible starting point in this string require 272 bits. Here is a trie representation which requires only 174 bits.



The 174 bits include the original string and the following 142 bits (shown without breaks).

```
111111011111110100010000111011011110110010110011111100011011111111000001
000100100001100110100101001000101001100011101000001001000110000000000111
```

These are easier to explain laid out as follows,

```
11
11 11
01 11 11 11
01 000100 001110 11 01 11 10                      4,14
11 001011 001111 11 000110 11 11                 11,15,6
11 10 000010 001001 000011 001101 001010 01     2,9,3,13
000101 001100 01 11                                 5,12
01 000001 001000                                     1,8
11
000000 000111                                        0,7
```

where the six-bit groups are also shown translated into decimal.

These 16 six-bit groups are `00` followed by the locations in the original string of each of the 16 substrings that run from some intermediate address all the way to the end of the string. The two-bit groups represent the nodes of the trie: `11` for a node with two descendents, `01` for a node with only a right descendent, and `10` for a node with only a left descendent. The trie encodes bitstrings by the simple rule: `0` means go left; `1` means go right. Thus we see that a path of two left edges represents `00` or `a`, a path of a left then a right edge means `01` or `c`, and so on.

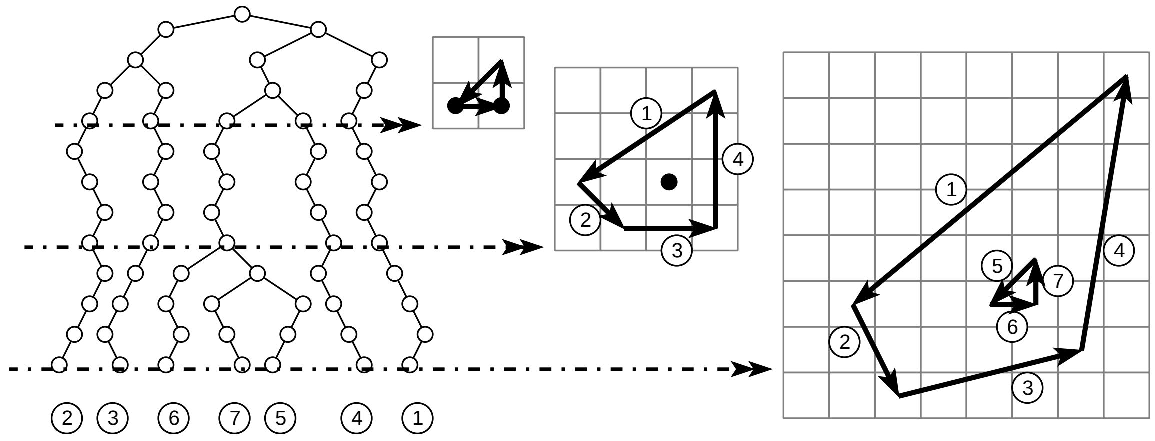
Thus if we are searching for the substring `ca` = `0100`, we follow the trie branching left, right, left, .. and we would branch left after this except the trie ends at address `4`, so we look that up in the original string and find the substring `catatgtgatcg`, which starts with `ca` as requested.

Any such search can be carried out (with a little bookkeeping to do what our eyes do to follow the trie) on the 142 bits we first showed above. Everything so far would work fine in RAM, but there is a problem: the search on the 142 bits is linear, not sublinear. Secondary storage comes to the rescue by breaking the trie into pages, in a manner we have no space to show here, and recording a synopsis of the bookkeeping, in two additional integers per page. This allows the search to descend a path in a tree of pages without considering any other pages on the same level: the linear search has become logarithmic in the size of the trie, and so $\mathcal{O}(\log \backslash)$ in cost. Reference: [?].

2.3 Variable-resolution maps

It is sometimes preferable to store all the data in the trie, rather than using the trie just as an index to the data. Since a trie node in the above representation needs two bits, the lower parts of the trie, which may be simple paths without bifurcation, can be twice as expensive to store this way than just storing the single bits of the original data. So some compression is lost. But there are compensating advantages, particularly when the data is fixed-length.

Here is a trie representation of a simple diagram, which could be a map in a geographical information system.



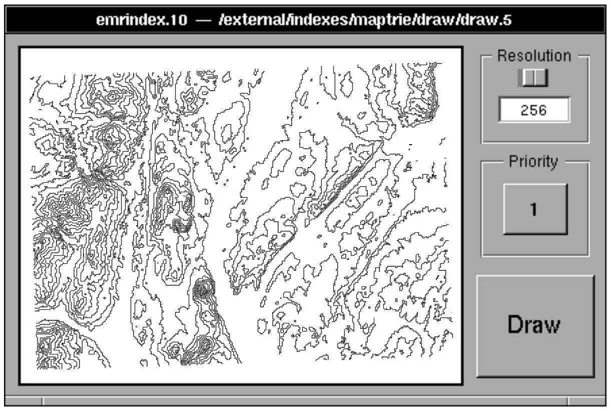
This trie is a "kd-trie", representing data of more than one dimension by cycling the dimension as we descend the levels of the trie. The root node of the trie, since it discriminates whether the first bit is `0` or `1`, splits the data space in half. We can chose this split to be on x, falling between small x-values and large x-values. The next level down splits these subspaces in half in turn, and, in a kd-trie, we could choose these splits to fall between small y-values and large y-values.

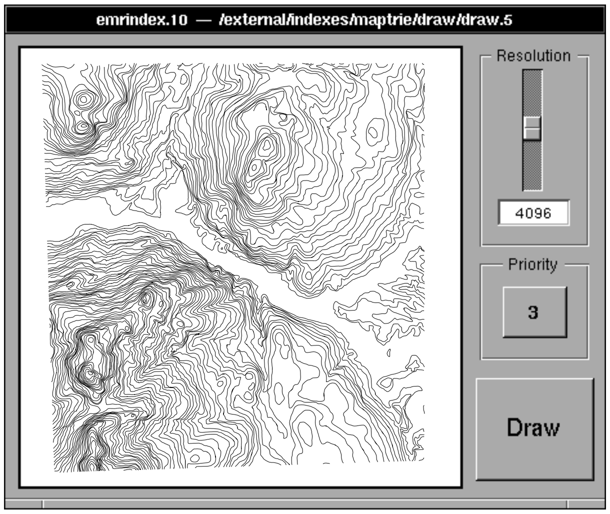
In the diagram shown, what is represented are not 2-D points, but 2-D *edges*. An edge has four coordinates, two for each end-point, and is thus a four-dimensional object. The kd-trie shown is indeed a 4d-trie, the first level for the x-coordinate of one endpoint, the second for the y-coordinate of that endpoint, and the third and fourth for the x and y-coordinates, respectively, of the other endpoint. In this illustration, each coordinate uses three bits, so full resolution is the diagram on the 8-by-8 grid shown on the right, with seven edges corresponding to the seven branches of the trie.

If we truncate the trie to two bits per coordinate, i.e., eight node levels, we see only five branches. These correspond to four edges and a single point into which the last three edges have coalesced. If we truncate to one bit, the diagram becomes the smallest one shown, on a two-by-two grid: the trie has four branches, but the two endpoints of edge 2 coincide so it disappears.

Since on secondary storage the trie is paged, as we discussed in the previous section, truncating it means that we simply do not fetch certain pages from secondary storage. This means that, if we wish to display only an overview of a large diagram such as a map, we do not need to transfer all the data and then filter it to the resolution offered by the display. Instead we fetch only the relevant pages.

For maps and similar diagrams, the trie structure thus allows variable resolution in addition to the ability to display only certain ranges of the map. The left map, below, shows a whole region at low resolutions, while the right map shows the bottom left corner of this region at higher resolution. Using this technique, we could store the whole 1:50,000 topographic series for Canada in a couple of gigabytes, and yet never transfer more than the megabyte or so of data needed for most displays, either to show the whole country in outline or to show smaller parts at appropriate levels of detail. The technique is also useful in transferring images across the Internet, presenting the recipient with an overview first, then increasing the resolution as more transmission time elapses. Reference: [**?**].





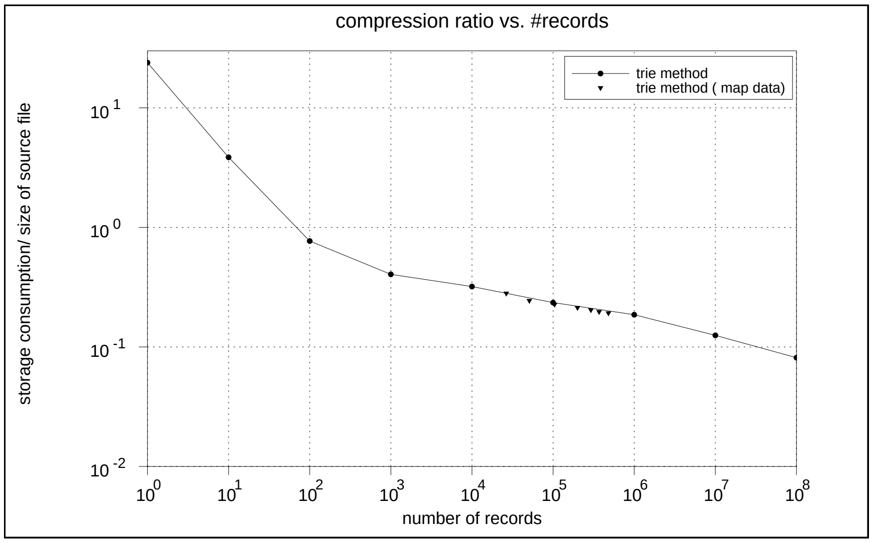
2.4 Data Compression

Because tries, of both sorts discussed in the previous two sections, enable the different data to share storage, they compress the data as we mentioned. Here is a simple argument giving an upper bound of $1 - 2/\lg n$ to this compression, where n is the number of bits in each data item (which we assume to be fixed in length).

If the trie is $h = \lg n$ levels of nodes high, it has up to 2^h branches, which represent 2^h data items of h bits each. These would occupy $h \times 2^h$ bits in raw form. As a trie, this data occupy $2^h - 1$ nodes, and with two bits per node, as above, the trie needs $2(2^h - 1)$ bits. This is a factor $2/h = 2/\lg n$ smaller. As a percentage, the compression is, for various file sizes

n	10^3	10^6	10^9	10^{12}
$2/\lg n$	1/5	1/10	1/15	1/20
lossless compression	80%	90%	93%	95%

Experimentally, the result is not far from this simplistic upper bound:



These and other experimental results can be found in [**?**].

3 Programming Language Principles

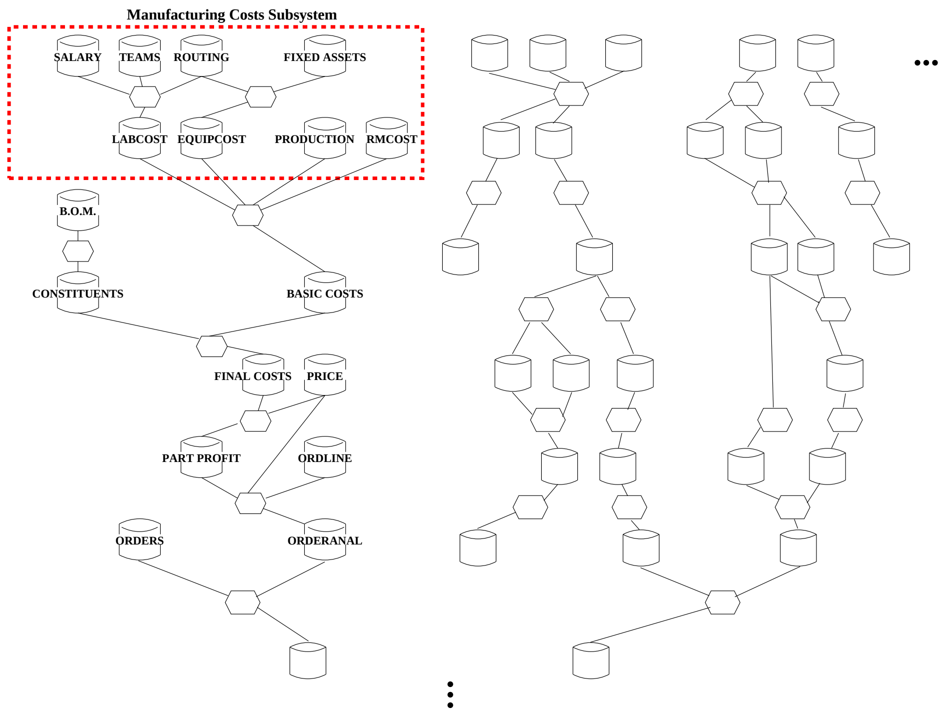
The previous four examples are ways in which secondary-storage considerations have led to the solution of outstanding problems or have had other beneficial side-effects. These solutions and benefits might have happened without thinking in terms of transferring large amounts of data, although they mostly did not.

in the realm of programming, secondary storage obliges us to abstract over looping, and this introduces a much higher level of thinking than is usual. Here are seven examples of the benefits.

3.1 Software engineering

High in the list of challenges faced by software engineering are effective development of the product and maintainability of the result. Both of these challenges would be ameliorated by reducing the size of software. If very much higher-order primitives were available and effective for a wide range of tasks, the number of lines of code would be substantially reduced and the tasks of software building and maintenance become easier. Secondary-storage considerations give us these higher-order primitives.

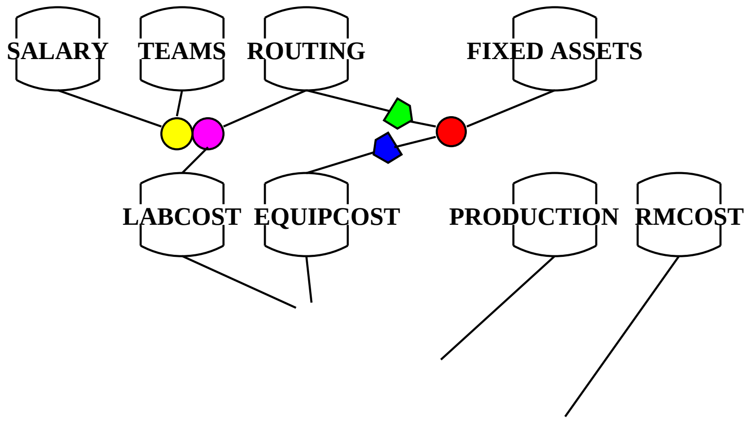
The system diagram of a moderately-sized enterprise might be several times more extensive than the following, in which each little hexagon represents a program of thousands of lines of code.



What we might like to do with this is represented by this symbolic extension of the part shown above in the dashed box:

EQUIPCOST ⟵ ⬟ **(** ⬟ **ROUTING** 🔴 **FIXED ASSETS)**

LABCOST ⟵ **SALARY** 🟡 **TEAMS** 🟣 **ROUTING**



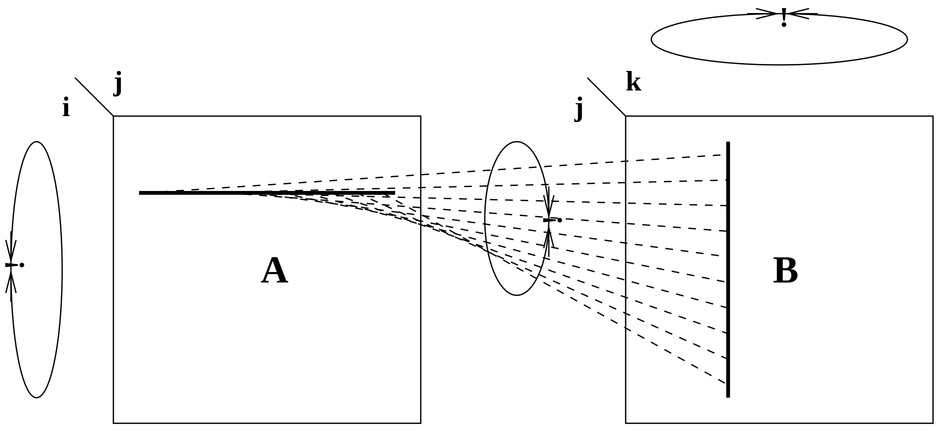
In the diagram part, the hexagons have been replaced by combinations of a small number of operators distinguished by their shapes: binary operators (two inputs and one output) are shown as circles; unary operators (one input and one output) are five-sided polygons in different orientations.

The upper two lines above give the same structure as the diagram, but in the form of statements built up of expressions. The effect is to turn one program into two binary operators and a second program into an expression with two unary and one binary operator. If we can devise a small set of different operators of sufficient flexibility to support the whole enterprise, then whole programs can be replaced by expressions or a few high-level statements.

The operators shown are indeed the selection/projection and the joins of the relational algebra [**?**, **?**].

3.2 Parallel algorithms

In addition to simplifying system construction by reducing program size, the abstraction over looping offered by high-level programming operators simplifies parallelization. Here is a schematic for matrix multiplication, which normally requires three loops nested inside each other.



In fact, each of these loops can be performed in any order, and so they are shown with a →!← label. If the code for matrix multiplication were written in a low-level language such as Java or C++, the loops would be explicit and a clever compiler would be needed to "parallelize" the code for running on multiple processors. This parallelization would undo the order explicitly specified by the programmer who wrote the loops.

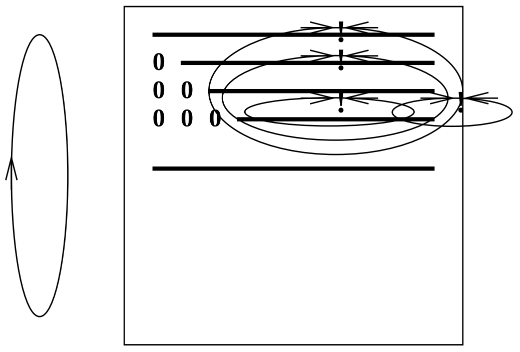
In a language which abstracts over looping, no order is ever specified. Here is matrix multiplication code for matrices represented as relations, $A(i,j,a)$ and $B(j,k,b)$.

```
let $ab$ be equiv + of $a \times b$ by $i, k$;
$AB$ <−[$i, k, ab$] in ($A$ natjoin $B$);
```

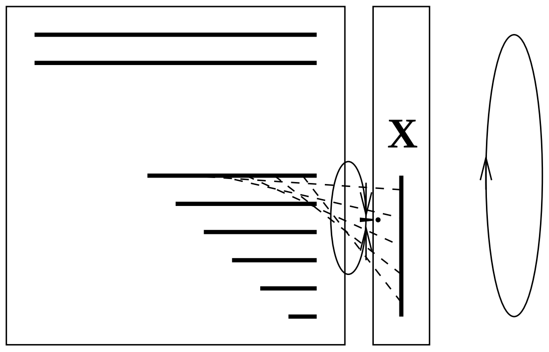
The second line is relational algebra, which takes the natural join of A and B, then projects the result on i, k and a calculated attribute, ab. The first line is "domain algebra", which calculates ab as a sum of $a \times b$ over j (i.e., grouped by i and k).

The domain algebra is orthogonal to the relational algebra, in a way which has recently become familiar through "aspect-oriented" programming but is more precise: domain algebra operations are carried out independently of any relation and this independence is an important vehicle to save intellectual work. The domain and relational algebras are orthogonal aspects of secondary-storage programming. Statements of the domain algebra are syntactically differentiated from relational algebra statements by the **let** construct.

Note that no loop constructs are needed in the matrix multiplication example, because all three loops are order-independent. Gaussian elimination provides a more subtle example, in which one loop in each step (triangulation and back-substitution) must be ordered. These loops must be explicit in the code, while the loops nested within them (two for triangularization and one for back-substitution) have no preferred order and may be implicit.



Triangularization



Back−substitution

Here is the code, where $A[i,j]$ includes the matrices A and B (in its last column) for the equation $AX = B$.

Triangularization

```
let $a'$ be $a$; let $a''$ be $a$;
for $row$ <− 1 to [red max of $i$] in $A$
{   $A'$ <−[$j, a'$] where $i$=row and $j$ >row in $A$;
    $A''$ <−[$i, a''$]where $j$=row and $i$ >row in $A$;
    let $aa$ be $(a' \times a'')/A$[row,row];
    update $A$ delete where $i$ >$row$ and $j \leq row$ in $A$;
    update $A$ change $a$ <− if $i \leq row$ then $a$ else $a - aa$ using [$i, j, aa$]
        in ($A''$ natjoin $A'$)
}
```

The first two lines in the loop pick out the current row and column. The first update set to zero the column below the diagonal. The second update modifies all the other elements below the current row, hence incorporating the two inner loops.

Back-substitution

```
relation $X(j, x)$;
let $ax$ be equiv + of $a \times x$ by $j$;
for $row$ <− [red max of $i$] in $A$ to 1 by −1
{   $AX$ <− [$ax$] in ($A$ natjoin $X$);
```

```
        let $x$ **be** $(X[row,$**red max of** $i+1] - ax)/A[row,row]$;
        let $j$ **be** $row$;
        update $X$ **add** $[j,x]$ **in** $AX$
    }
```
The loop repeatedly takes the natural join of X with the current row, and this incorporates the inner loop.

3.3 Expert systems

A language, motivated by the bulk processing which secondary storage requires, does not abstract over all loops but judiciously over certain loops. The Gaussian elimination example showed explicit looping where the order of evaluation needed to be controlled by the programmer. Recursion is a powerful alternative to explicit loops.

A classical Prolog-like example defines an ancestor to be a parent or the parent of an ancestor. The relation *ancestor*(*Sr, Jr*) is derived from the relation *parent*(*Sr, Jr*):

ancestor **is** *parent* **union** *parent*[*Jr* **natcomp** *Sr*]*ancestor*;

The **is** "assignment" implements the *view* mechanism of databases, which in this case happens to be a recursive view: *ancestor* is defined in terms of itself.

A similar construct gives a one-line inference engine.

NewFacts **is** *Facts* **union** [*Concl*] **in** (*NewFacts*[*Concl* $\supseteq$ *Ante*]*Horn*)

This applies to the Horn clauses, *Horn*(*Rule#,Ante,Concl*), and the initial facts, *Facts*(*Concl*). A simple example successively deduces that the animal being examined is a bird and then that it is a duck:

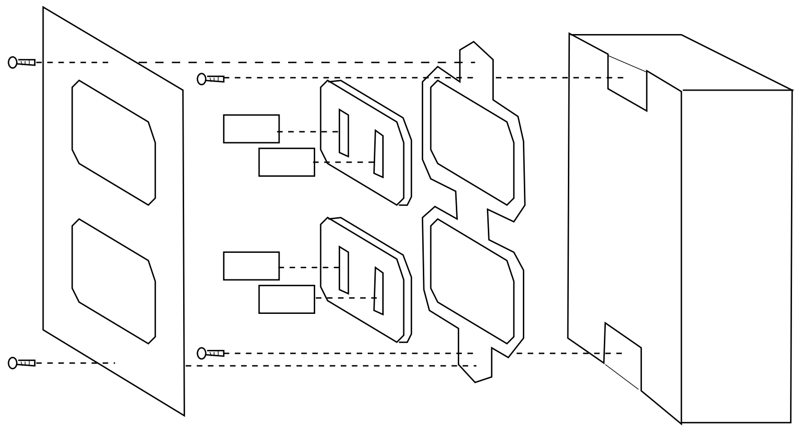
[*New*]*Facts*	*Horn*			
(*Concl*)	(*Rule#*	*Ante*	*Concl*	)
`lays eggs`	1	`lays eggs`	`is bird`	
`has feathers`	1	`has feathers`	`is bird`	
`swims`	2	`flies`	`is bird`	
——	2	`is not mammal`	`is bird`	
`is bird`	3	`is bird`	`is duck`	
——	3	`swims`	`is duck`	
`is duck`	3	`is brown`	`is duck`	
	4	`is bird`	`is duck`	
	4	`swims`	`is duck`	
	4	`is green`	`is duck`	
	4	`is red`	`is duck`	
	5	`is duck`	`migrates`	
	5	`is not tame`	`migrates`	

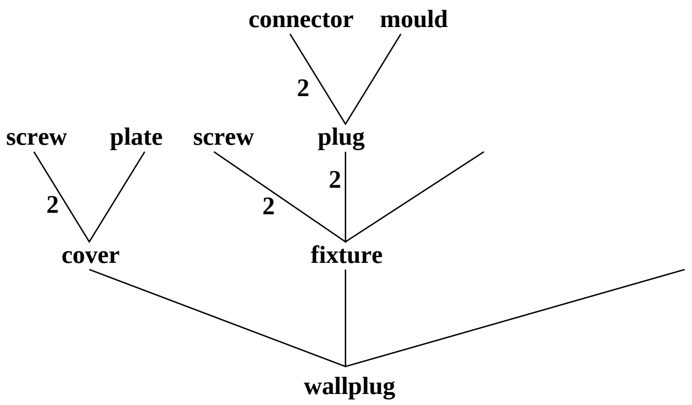
Reference [?] expands this one-line inference engine to fifty, in a 200-line expert system shell, which emulates a couple of commercial expert systems and adds some useful features.

3.4 Bill of materials

Including domain algebra adds still more power to the recursive relational algebra, and the independence of the two formalisms from each other make the thinking particularly straightforward. Here is a bill of materials example which needs a couple of pages of code in textbook Prolog, a language with much higher-level abstractions than popular imperative languages.

A bill of materials is a description of a manufactured product in terms of its components, such as the wallplug shown. Typical bill-of-material processing would find out from this description total quantities of raw materials, for instance that the wallplug needs four screws and four connectors.





We can represent the bill of materials by $\mathit{PartOf}(A, S, Q)$, standing for *assembly, subassembly* and *quantity*, respectively, and containing tuples such as (`plug, connector, 2`) and (`fixture, plug, 2`). The code must find the transitive closure of the tree or DAG, which relational algebra recursion does as for *ancestor* and *NewFacts*, above. It must also calculate the cumulative quantities while doing the recursion, and this can be expressed orthogonally in the domain algebra.

```
let $A'$ be $A$; let $S'$ be $S$; let $Q'$ be $Q$;
let $Q''$ be equiv + of $Q \times Q'$ by $A, S'$;
let $Q'''$ be $Q + Q''$; let $Q$ be $Q'''$;
$\mathit{Explo}$ is $[A, S, Q]$ in $[A, S, Q''']$ in $(\mathit{PartOf}\ [A, S$ union $A, S']$
    $[A, S', Q'']$ in $(\mathit{Explo}\ [S$ natjoin $A']\ [A', S', Q']$ in $\mathit{PartOf}))$;
```

3.5 Object-orientation

The most important idea behind object-oriented programming is instantiation. “Objects” are encapsulated states with associated code, and instantiation is a programming abstraction which creates instances from a template “class”. In low-level languages, this is done one object at a time by an explicit operator, usually **new**. In a language motivated by secondary storage we abstract from individual objects to the class as a whole, just as we abstract from individual tuples, or records, to the relation as a whole. Thus we must be able to instantiate many objects at once. Here is how we can do this with a join operator.

The example is a bank account class, modelled as a persistent first-class procedure [**?**] (the procedure itself persists on secondary storage, so that it can serve as a library, common to many programs; procedures are first-class data types, meaning in particular that they can be returned as parameters, so procedures serve as “methods” as well as as the class).

```
proc bankAccount (Balance, Deposit) is
state $BAL$ intg
```

```
{    proc Deposit(dep) is
     {    BAL <- BAL + dep};
     proc Balance(bal) is
     {    bal <- BAL;
     BAL <- 0
}
```

Here, *Deposit* and *Balance* are the two methods of the class *bankAccount*, which has *BAL* as its state, initialized to zero. Since *BAL* must be instantiated for every account, we have a set of accounts.

relation *accts*(*acctno, client*) <− {(1729, "Pat"),(4104, "Jan")};

We instantiate by joining the class, *bankAccount*, with the *accts* relation,

Accounts <− *accts* **natjoin** *bankAccount*;

giving a new relation with a hidden attribute, *BAL*.

(*accno*	*client*	*Balance*	*Deposit*	[*BAL*])
1729	Pat			[0]
4104	Jan			[0]

Note that the join produces a result, as always, whose attributes are the union of the attribute sets of its operands: *Balance* and *Deposit* are constant attributes, having the same value for every tuple, and this value is their respective representations as methods.

Since *BAL* is hidden (and so shown as [*BAL*]), we can access it only through the methods. Here is how we update Jan's *BAL* by depositing 100$.

update *Accounts* **change** *Deposit*(100) **using where** *acctno*=4104;

Reference: [?].

Inheritance is a second object-oriented idea: given classes, which instantiate to sets of objects, we may want to consider subsets with special properties. Here is a class of interest-bearing accounts which we will make a subclass of *bankAccount* and which thus will inherit its methods.

```
proc interest(Interest) is
state BAL intg;
{    proc Interest(int) is
     {  BAL <- BAL × (1 + int/100.0)};
}
```

(We repeat the declaration of *BAL* because it is needed syntactically and, unconventionally, we have not specified the inheritance yet.)

We define *intaccts* as a subset of *accts* and we instantiate *interest* on *intaccts* in the same way as before.

relation *intaccts* (*acctno, intrate*) <− {(4104, 3)};
InterestAccounts <− *intaccts* **natjoin** *interest*;

Then we introduce a new keyword (the only new keyword in the whole treatment of object orientation) to specify the inheritance.

InterestAccounts **isa** *Accounts*;

(This keyword, **isa**, is implemented as a join, and says merely that any future reference to *InterestAccounts* will be taken to mean *InterestAccounts* **natjoin** *Accounts*.)

Finally, the bank can calculate and apply the interest for all interest-bearing accounts.

update *InterestAccounts* **change** *Interest*(3) **using** *intaccts*;

This gives

(*accno*	*client*	*intrate*	*Balance*	*Deposit*	*Interest*	[*BAL*])
1729	Pat	−			−	[0]
4104	Jan	3				[103]

3.6 Data mining

Classification data mining uses "training data" to build a "theory" which can subsequently be used to classify any unanticipated input. For instance, we might want to learn from set of example weather conditions when it is a good idea to pick cotton: rainy, still days are good, independent of whether the temperature is hot or mild; sunny, humid and windy days are not good; and so on through a number of samples shown below.

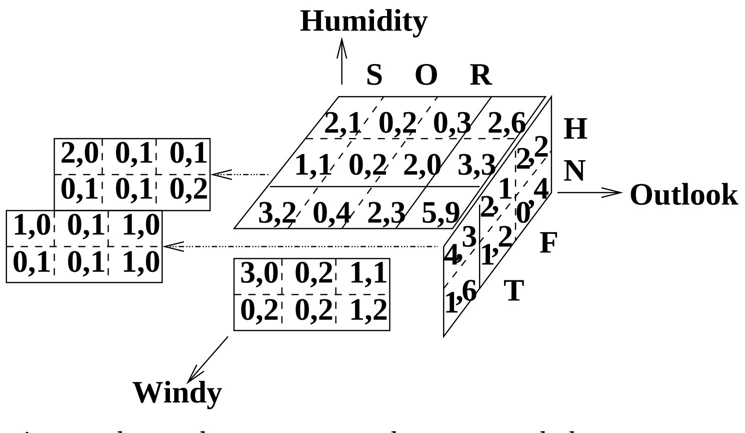
Training

(*Outlook*	*Humidity*	*Windy*	*N*	*P*)
`sunny`	`high`	`f`	`2`	`0`
`sunny`	`high`	`t`	`1`	`0`
`sunny`	`normal`	`f`	`0`	`1`
`sunny`	`normal`	`t`	`0`	`1`
`overcast`	`high`	`f`	`0`	`1`
`overcast`	`high`	`t`	`0`	`1`
`overcast`	`normal`	`f`	`0`	`1`
`overcast`	`normal`	`t`	`0`	`1`
`rain`	`high`	`f`	`0`	`1`
`rain`	`high`	`t`	`1`	`0`
`rain`	`normal`	`f`	`0`	`2`
`rain`	`normal`	`t`	`1`	`0`

(The `N` and `P` are negative and positive evaluations of the conditions—no evaluation can be both—and the `2` entries are in tuples that represent two samples in which temperature, not shown, is either high or mild. Since these pairs of samples have outcomes independent of temperature, the temperature does not figure in the final theory.)

This training data is a classic example illustrating "decision tree" theories, which are grown from decision nodes that minimize information, i.e., surprise, at each step. We do not build such a tree here but discuss the basic construct needed to build it, the "datacube".

The datacube for this training data summarizes the `N, P` pairs in three dimensions, one for the *Outlook* attribute, one for *Humidity*, and one for *Windy*. The twelve training tuples shown give a $3 \times 2 \times 2$ array of pairs, shown to the left below.



The right of the picture shows the aggregates that are needed to compute the information that the decision tree will minimize. These form faces to the datacube, as we sum the numbers in the `N, P` pairs along each of the three dimensions.

For this special case, it is easy to use the domain algebra to calculate the datacube in a single loop over the three dimensions. (The inventor of the relational algebra said that such processing was not intended for database systems, and coined the term "OLAP" (on-line analytical processing) to separate such transcendental operations from ordinary, transaction processing. But he was thinking of SQL-like *query* languages, not of a secondary-storage *programming* language.) To construct and process general datacubes, with arbitrary attributes in any number of dimensions, we need new capabilities in the form of attribute metadata. Here is the code, including such syntax, for building a general datacube.

```
let N be totN;
let P be totP;
domain attr attribute;
relation AllAttribs(attr) <- AttribsOf Training;
                //Outlook, Humidity, Windy, N, P
relation ClassAttribs(attr) <- {(N), (P)};
relation TotAttribs(attr) <- {(totN), (totP)};
PropAttribs <- AllAttribs diff ClassAttribs;
LoopAttribs <- PropAttribs;
while [] in LoopAttribs
{    Attrib <- pick LoopAttribs;
     update LoopAttribs delete Attrib;
     let eval Attrib be "ANY";
     let totN be equiv + of N by (PropAttribs diff Attrib);
     let totP be equiv + of P by (PropAttribs diff Attrib);
     update Training add [AllAttribs] in
          [PropAttribs diff Attrib union TotAttribs] in Training;
}
```

This code adds tuples to *Training* with the placeholder "ANY" marking the attribute(s) that have been aggregated. The metadata syntax allows expressions, on unary relations on an attribute of type **attribute**, to be used where constant attribute lists were used before.

From this datacube, the decision tree analysis is easily done. In addition, simpler special-case classification methods, such as "one-rule" and Bayesian, also follow from the datacube. Reference: [**?**].

3.7 Semistructured data

The eXtensible Markup Language, XML, is the best-known embodiment of semistructured data. Semistructured data is more flexible than data usually represented by flat relations with their fixed attributes. Semistructured systems support, among other things, indefinite nesting, optional, multiple and alternative attributes, internal linking, and even alternative types. It does this by embedding the schema, which describes the data, into the data itself. It is not true, however, that semistructured data cannot be fully represented or processed relationally. There is, indeed, considerable advantage to doing so, because it includes semistructured data under the same terms as all other kinds of data, so all forms of data can be integrated within one framework.

We start with a marked-up text in a format which is even more flexible than XML: it allows text outside the tags in addition to tagged values. (This format could be called xML, with x standing for G, SG, HT, X, etc., all the related kinds of markup language.)

```
<Person>
  <Name>Ted</Name> married
  <Family><Conj>Alice</Conj> in
   <Wed>1932</Wed>. Their children,
    <Children><Name>Mary</Name> (<DoB>1934</DoB>) married
      <Family><Conj>Alex</Conj> in <Wed>1954</Wed>
        :
      </Family>
      :
    </Children>
  </Family>
</Person>
```

This is a partial display of the following text, the rest of whose markup is self-evident.

> Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James and Jane had Tom in 1961 and Sue in 1962).

For now, we ignore the text outside the tags, collapsing the present discussion to conventional XML. We wish to convert the marked-up text to a relation, one which is not only nested but recursively nested.

```
PERSON
(Name    FAMILY                                              )
         (Conj   Wed    CHILDREN                             )
                        (DoB   Name   FAMILY                 )
                                      (Conj   Wed   CHILDREN )
                                                    (DoB   Name  )
Ted      Alice   1932   1934   Mary   Alex    1954  1956   Joe
                        1935   James  Jane    1960  1961   Tom
                                                    1962   Sue
```

The code to make this conversion involves a new operator, **mu2nest**, which works at only one level of nesting. To complete the conversion, we introduce recursion into the domain algebra.

let *FAMILY* **be** [*Conj, Wed, CHILDREN*] **mu2nest** *Family*;
let *CHILDREN* **be** [*DoB, Name, FAMILY*] **mu2nest** *Children*;
PERSON <− [*Name, FAMILY*] **mu2nest** *Person*;

(This example actually involves *mutual* recursion between *FAMILY* and *CHILDREN*.)

The self-reference must always be one level down. Thus *FAMILY* refers to *CHILDREN* as an attribute, and *CHILDREN* refers to *FAMILY* as an attribute.

Such recursive domain algebra can be used for queries, with no new syntax. But it is handy to have a syntactic sugar for the common special cases. This takes the form of *path expressions* linking relations with their attributes down the nested structure. The following example queries show simple paths, paths using regular-expression operators, and paths in the condition of a **where** clause.

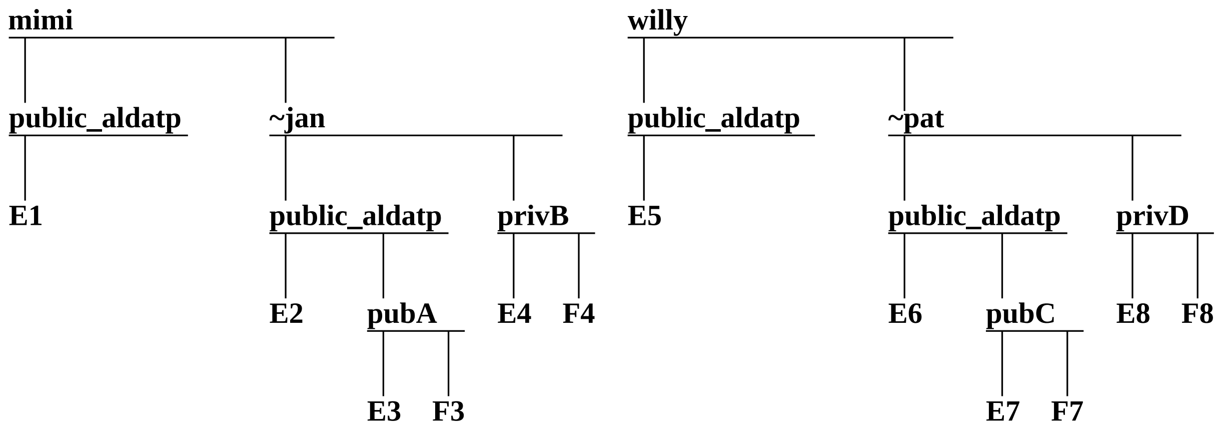
PERSON/Name	`Ted`
PERSON/FAMILY/CHILDREN/Name	`Mary, James`
PERSON/FAMILY/CHILDREN/FAMILY/CHILDREN/Name	`Joe, Tom, Sue`
*PERSON/(./)*Name*	`Ted, Mary, James, Joe, Tom, Sue`
Name **where** *FAMILY/Conj*=`"Alice"` **in** *PERSON*	`Ted`

The first query is a projection, since *Name* is a simple attribute. The second and third queries collapse three and five levels respectively. The fourth query uses the Kleene star to express recursion: it returns the union of all levels that contain the *Name* attribute. The fifth query uses the same path-expression syntactic sugar with a different meaning: the selection condition is raised from a lower level. Reference: [**?**].

3.8 Databases distributed by Internet

The same path-expression syntax we have just introduced can be used in a completely different setting to link multidatabases (on one host) and distributed databases (on different hosts, connected by Internet). For both, this provides a kind of assembler language which does not address the central issues of transparency, etc., but simply offers a naming convention to make reference to database resources at various locations.

The protocol, which could be called the aldat protocol, aldatp, mimics HTTP by designating special directories to hold publically-accessible database resources, including relations and persistent procedures. Here is a sketch of directories on two hosts, mimi and willy, with two owners, Jan and Pat, and database elements, Ei and Fi, which are either public or private.



Any resource in the current directory of a database system may be referred to simply by its name. Any resource elsewhere requires the extended name, beginning with the protocol header, *aldatp://*. This naming convention is universal, and will locate any element which the operating system permissions allow.

F4 <− aldatp://mimi/∼jan/pubA/E3;
aldatp://mimi/∼jan/pubA/F3 <− E2;
aldatp://willy/∼pat/pubC/{*F7 <− E7*};

The first statement is executed from Jan's private directory, *privB*, and assigns to *F4* the value of *E3* in Jan's public directory, *pubA*, under Jan's special aldatp directory, *public_aldatp*. Like *public_http*, this directory is not named in the query, but provides an effective root directory for aldatp access.

The second statement runs in Jan's *public_aldatp* directory, and copies *E3* in Jan's public directory, *pubA*. The third statement can run anywhere, and executes an assignment in Pat's public directory, *pubC*.

To come closer to the flavour of distributed databases, here is a semijoin to join *E3*(A, B) in Jan's *pubA* with *E7*(B, C) in Pat's *pubC*, where the statement is being executed. Note how the statement reaches over to Jan's *pubA* to get *E3*, then reaches back to Pat's *pubC* to project and get *E7* for the first join. The second join with the full *E7* is the last operation, performed locally in *pubC*.

{*aldatp://mimi/∼jan/pubA/*{*E3* **natjoin**
  aldatp://willy/∼pat/pubC/{[*B*] **in** *E7*}} **natjoin** *E7*

Reference: [?]

4 Conclusion

Secondary storage differs from primary memory by requiring orders of magnitude greater seek time than retrieval time, which in turn requires data to be stored and transferred in large blocks. For storage and processing data structures and algorithms, this imposes strong use-all constraints to avoid re-fetching the same block. For programming language, it imposes abstraction over looping, which raises the level of programming significantly above the level of common languages.

We have shown many areas of computer science which are simplified by this new thinking: object orientation, parallel programming, artificial intelligence and networking, among others. We have shown simplification in a sample of applications: numerical analysis, bioinformatics, geographical information systems, and semistructured systems, among others.

We have not discussed many other areas, such as event programming and concurrency (which are not the same thing). Future work involves locating where further new insights are most likely to arise. This includes areas which we reckon can be reduced to the approach in this paper but do not yet know exactly how. In rough order of decreasing certainty, these areas might include: data visualization and what might be called "relational graphics" (which we are currently working on); constraint databases; peer-to-peer cooperative work; and agent programming.

5 Acknowledgements

We are indebted to the Natural Science and Engineering Research Council of Canada for support under grant OGP0004365. The Quebec and Canadian taxpayers have covered salaries, grants and scholarships. Over 100 graduate supervisees have done the work, funded by many private sources as well as public.