Are Databases a Special Case of Programming Languages?

T. H. Merrett
McGill University

1. Chronology
2. Start with Databases —
   abstracting tuples, loops, grouping, & ordering
3. On to Programming Language —
   functions and "computations";
   data abstraction and nesting;
   object orientation and instantiation

Chronology I Secondary Storage

... fixed length

1978 Static Multipaging

$S = N \ldots 2N$ $\quad\quad$ $< T >= 1 \ldots 2$

1982 Dynamic Multipaging

$S = N \ldots 2N$ $\quad\quad$ $< T >= 1 \ldots 2$

1983 Z-Order

$S = N$

1984 Tidy functions

$S = N \ldots 2N$ $\quad\quad$ $< T >= 1 \ldots 2$

... variable length

1994 Tries for text

$S \approx 3N$ $\quad\quad$ $< T >=\mid result \mid$

... fixed length

1994 Tries for maps

$S \approx 0.1N$ $\quad\quad$ $< T >=\mid result \mid$

1996 Tries for orthogonal range queries

$S \approx 0.1N$ $\quad\quad$ $< T >=\mid result \mid$

Chronology II
Programming Languages

	P.L.	**D.B.**
1975	iterator	relation
1984	recursion	fixed point
1992	scope	multidatabases
1992	process syncr.	selector
1992	reflection	metadata
1993	inheritance	join
1994	aggregation	join
1994	function	relation
1995	A.D.T.	nesting
1996	events	active
1997	objects	instantiation
199?	type	constraint

Applications Considered or Coded in this Talk

- inference engine
- polygon area and centroid
- uniform acceleration
- rotation in Euclidean and hyperbolic 2-space
- concept hierarchy mining
- predictive modelling for D.S.S.
- events in active databases
- complex objects and nesting to arbitrary depths
- objects with state: bank accounts
- the o-o n-body problem: gas molecules

Start with Databases

Main ideas:

- Abstract over looping
- Treat attributes independently of relations

5–13

Start with Databases

- Relational algebra:
  extend slightly
  - Natural join corresponds to set intersection.
    So extend set union, difference, etc. similarly.
  - Division corresponds to superset comparison.
    So extend subset, empty intersection, etc. similarly.
  - Projection & selection combine & quantify to give *QT-selector*

- Domain algebra:
  introduce operations on attributes
  (independently from relations)
  - "horizontal" or intra-tuple
  - "vertical" or aggregation

(We call the result "Relix".)

(Relix − SQL) =

no tuples, only relations

Difference I

iteration $\longrightarrow$ high level ops

Difference II

grouping $\longrightarrow$ high level ops

Difference III

ordering $\longrightarrow$ high level ops

Difference I — no iterators

a) Relational Algebra

StuCour(*Stu*	*Cour*	*Asst*	*Exam*)
Joe	Algol	45	42
Sue	Algol	47	38
Joe	Pascal	43	42
Sue	Pascal	45	42

- T-Selector is a loop:
  JoeCour <−[*Cour*,*Asst*,*Exam*]
  where *Stu*= `"Joe"` **in** *StuCour*

JoeCour(*Cour*	*Asst*	*Exam*)
Algol	45	42
Pascal	43	42

Difference I — no iterators

b) Domain Algebra

JoeCour(*Cour*	*Asst*	*Exam*)
Algol	45	42
Pascal	43	42

- Horizontal operations are a loop:
  let *Final* **be** *Asst* + *Exam*

- Vertical operations are a loop:

let *Avg* **be** (**red** + **of** *Final*) / (**red** + **of** 1)

JoeMark <−[*Avg*] **in** *JoeCour*

JoeMark(*Avg*)
86

Difference II — grouping

a) Domain Algebra

StuCour(*Stu*	*Cour*	*Asst*	*Exam*)
Joe	Algol	45	42
Sue	Algol	47	38
Joe	Pascal	43	42
Sue	Pascal	45	42

- Vertical operations support grouping:
  let *Total* **be equiv + of** *Final* **by** *Cour*

TotMark <−[*Cour*,*Total*] **in** *StuCour*

TotMark(*Cour*	*Total*)
Algol	172
Pascal	172

Difference II — grouping

b) Relational Algebra

- Generalizing *division*, plus *recursion*, gives a one-line *inference engine*. (*Horn* is rulebase; *Facts* are known facts.)

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

- Expands to a 50-line inference engine in a 200-line expert system shell, Relixpert (1987; 1 man-month; subsumes commercial shells AUGMENT, VP-Expert).

Difference III — ordering
a) Domain Algebra

- Here is a planimeter for geographical areas, using Stokes' Theorem

let X' **be par succ of** X **order** *Seq* **by** *Group*;
let Y' **be par succ of** Y **order** *Seq* **by** *Group*;
let *Area* **be equiv** $+$ of $(X \times Y' - Y \times X')/2$
  by *Group*;

- Similar calculations can find centroid, slope of an area in 3d, etc.

What kinds of programming can be done in a language whose data structures support possibly very large sets of related items, and whose operations abstract over loops?

- structured data processing
- geometry and maps — spatial data
- temporal data
- expert system shells
- text data
- mining
- active databases
- images?
- sound?
- warehousing?
- mediation?

On to Programming Language I

Main idea:

Unify functions and relations

Topics

- "Computations"
- Combining computations
- Database mining
- Predictive modelling
- Active databases

14–25

Computations

- Unify base concepts of DB and PL
- In mathematics, functions $\subseteq$ relations
- Let's generalize PL "function"

comp *Sum*(A, B, C) **is**

$$C <\!- A + B$$

alt $$A <\!- C - B$$

alt $$B <\!- C - A;$$

- And let's invoke it by relational algebra

$T <\!-\ [C]$ **where** $A = 2$ & $B = 2$ **in** *Sum*

$T(C)$	$R(A$	$B)$	$S(A$	B	$C)$
4	5	2	5	2	7
	6	3	6	3	9

$S <\!-$ *Sum* **ijoin** R

Computations (cont.)

- a limited form of constraint programming
- think of "**comp**" as "compressed relation": the relation whose extent is all values allowed by the constraint
- so we can make sense of invocations by other joins, e.g., **ujoin**, **djoin**, **sjoin**, $\subseteq$, etc.
- we will also need a top-level invocation, a.k.a. *procedure*
- *Sum*(**in** A, **in** B, **out** C) or *Sum*(**out** A, **in** B, **in** C)

New Horizons from Computations

Computations encourage us to think in terms of sets of related equations.

This introduces some amusements which people who like abstractions may already have formalized.

- Consider the join of two computations

- (the *and* of two constraints).

- An implementation might hold this join as a *view*

- to be applied by invoking one then the other.

Combining Computations

Try uniform acceleration:

$$d = d_0 + v_0 t + \frac{1}{2} a t^2$$

$$v = v_0 + at$$

- These generate 4×5 pairs
- but the nine pairs on v_0, a and t give only three different results
- (which require simultaneous algebraic solutions)
- and only the pairs giving (d, v) and (d_0, v) can be solved in any order.
- There are $\begin{pmatrix} 6 \\ 2 \end{pmatrix}$ solutions to two equations in six variables
- but (d, d_0) cannot be determined.

Combining Computations (cont.)

The matrix shows when this can be implemented as two computations joined, in some order, with actual parameters provided by some input data.

	a	t	v_0	d	d_0
a	—	*alg*	*alg*	$\uparrow$	$\uparrow$
t	*alg*	—	*alg*	$\uparrow$	$\uparrow$
v_0	*alg*	*alg*	—	$\uparrow$	$\uparrow$
v	$\leftarrow$	$\leftarrow$	$\leftarrow$	*any*	*any*

- — means that the equations are overdetermined.
- *alg* means algebraic rearrangement is needed.
- $\uparrow$ means solve first for the variable above, then for the variable to the left.
- $\leftarrow$ means solve first for the variable to the left, then for the variable above.
- *any* means equations can be solved in any order.

New Horizons from Computations (cont.)

Here is another pair of equations, rotation in two Euclidean dimensions.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

where c is cosine and s is sine. So, of course,

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

But computations motivate us also to find

$$\begin{pmatrix} x' \\ x \end{pmatrix} = \frac{1}{\sqrt{1-c^2}} \begin{pmatrix} -1 & c \\ -c & 1 \end{pmatrix} \begin{pmatrix} y \\ y' \end{pmatrix}$$

and

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \frac{1}{\sqrt{1-c^2}} \begin{pmatrix} -1 & c \\ -c & 1 \end{pmatrix} \begin{pmatrix} x' \\ x \end{pmatrix}$$

This is the improper Lorentz transformation that gives time reflection. !

From physics to data mining:

computation to climb a concept hierarchy. (ref. Jiawei Han)

```
comp generalize(Src, Rslt, Attr, Thld)
{ let distinct be equiv + of 1 by Attr;
  while [] where distinct > Thld in Src do
  { local tmp <- Src[Attr icomp Low]Hier;
   if [] in tmp then
   { let Attr be High;
     Rslt <- [Attr, ~{ Attr}] in
      [High, ~{ Attr}] in tmp;
   } else
     Rslt <- Src;
  }
}
```

(*Hier*(*Low, High*) holds the concept hierarchy)

Mining (cont.)

For example, with a data relation

Student(*Name*	*Major*	*Born*	*GPA*)
`Anderson`	`hist`	`Vanc.`	`3.5`
`Bach`	`biol`	`Calg.`	`3.7`
⋮	⋮	⋮	⋮

and a concept hierarchy, *Hier*, which subsumes `biol`, etc. under `sci`; `hist`, etc. under `hum`; `Vanc.`, etc. under `BC`; `BC`, etc. under `Can`; `3.7`, etc. under `excel`; and so on, we could deduce

StudentAbil(*Major*	*Born*	*GPA*)
`hum`	`Can`	`excel`
`sci`	`Can`	`excel`
`sci`	`frgn`	`good`

by the invocations

generalize(**in** *Student*, **out** *MajorGen*, **in** *Major*, 3);
generalize(**in** *MajorGen*, **out** *BornGen*, **in** *Born*, 3);
generalize(**in** *BornGen*, **out** *StudentAbil*, **in** *GPA*, 3);

Predictive Modelling for D.S.S.

- Given a relation *PastSales*(*Year, Qty*)
- we can find out last year's sales with
  [*Qty*] **where** *Year* = 1996 **in** *PastSales*
- Suppose we want to predict sales for next year:
  [*Qty*] **where** *Year* = 1998 **in** *FutSales*
- In fact, it would be nice to have
  Sales **is** *PastSales* **ujoin** *FutSales*
- so that we can query *Sales* for *any* year, past or future.

Predictive Modelling (cont.)

Here is the code for the last two, using least squares for the model:

let *yavg* **be** (**red** + **of** *year*) / (**red** + **of** 1.0);
let *qavg* **be** (**red** + **of** *qty*) / (**red** + **of** 1.0);
let *slope* **be**
(**red** + **of** ((*year* − *yavg*)* (*qty* − *qavg*)))
/ (**red** + **of** ((*year* − *yavg*)**2));
let *base* **be** *qavg* − *slope***yavg*;
comp *LinInterp*(*year, qty, slope, base*) **is**
qty <− *slope***year* + *base*;
FutSales **is** *LinInterp* **icomp** *PastSales*;

Active Databases

- Events are system-generated procedure calls

- and event handlers are procedures (computations).

We have built event handlers as parameterless computations for the usual database operations, e.g., **comp** *add:local* for additions to relation *local*.

We could also have *programmer-defined* events which might automatically trigger more than one event-handler computation.

On to Programming Language II

Main idea:

Relational operators all apply to attributes

Topics

- Complex Objects
- Simple Nesting
- Double Nesting
- Deeper Nesting
- Recursive Nesting
- Abstract Data Types

26–33

Complex Objects

- A basic precept of PL is “no 2nd-class types”.

- So values in relations should not be restricted.

- So relations can contain relations: “complex objects”.

- How do we support this in the language?

The domain algebra subsumes the relational algebra:

any attribute type can be relational,
so relational operations are available for attributes.

Example (Jaeschke & Schek, 1982):

Books(*Authors*	*Title*	*Price*	*Descript*)
`A1, A2`	`T1`	`P1`	`D1, D2`
`A2`	`T2`	`P2`	`D1, D2`
`A1`	`T3`	`P1`	`D1, D2, D3`

Which books by A2 are described by both D1 and D2?

[*Title*] **where** *Authors*$\supseteq$ {`(A2)`} **and** *Descript*$\supseteq$ {`(D1), (D2)`} **in** *Books*

- The two $\supseteq$ operators look like set comparisons, but they are σ-joins (division) from the relational algebra applied to attributes in the select clause.

Example (Deshpande & Larson, 1987):

Employee

(*ENo*	*Name*	*Children*			*Training*	)
		(*Name*	*DoB*	*Sex*)	(*CNo*	*Date*)
105	John	Jane	800510	F	314	791010
		Eric	821005	M	606	810505
					714	820620
123	Anne	Maria	751112	F	315	810613
					423	820711
153	Bruce				314	791010
205	Ian	Bob	701016	M	314	791010
		Steve	750115	M		

Find employees without children:

[*ENo*] **where not** ([] **in** *Children*)

in *Employee*

(Note the projection of the attribute, *Children*.)

Find employees who took course 314:

[*ENo*] **where** ([] **where** *CNo*=314 **in** *Training*)

in *Employee*

(Note the selection of the attribute, *Training*.)

Double nesting (cont.)

Find children of employees who took course 314:
[*Children*] **where** ([] **where** *CNo*=314 **in** *Training*)

in *Employee*

(Note the result is still a nested relation.)

Find names of those children:

let C **be** [*Name*] **in** *Children*;

or

let C **be** *Children.Name*

[*C*] **where** ([] **where** *CNo*=314 **in** *Training*)

in *Employee*

Deeper Nesting

EFaculty								
(*Dept*	*Profs*							)
	(*Name*	*Sal*	*Offc*	*Students*				)
				(*Name*	*Offc*	*Courses*		)
						(*Num*	*Grd*	)
CS	Pat	55	312	Kim	300	612	A	
						617	A-	
				James	300	612	B+	
						617	A-	
	Sue	42	322	Chris	328	506	A	
	Luc	48	319	Kim	300	612	A	
						617	A-	
EE	Pat	55	312	Kim	300	612	A	
						617	A-	
				James	300	612	B+	
						617	A-	
	Denis	52	412	Joe	425	530	B+	
			Peter	426	531	A-		

Find DEPTs with two A grades:

let *A1* **be red** + **of** 1;

let *S2* **be red ujoin of** [*Name*,

[*A1*] **in where** *Grd*="A" **in** *Courses*]

in *Students*;

let *A2* **be red** + **of** A1;

Ans <− [*Dept*] **where** *Prof.S2.A2*=2 **in**

EFaculty;

Recursive Nesting

Assembly					
(*Id*	*Assembly*				*Cost*)
	(*Id*	*Assembly*		*Cost*)	
		(*Id*	*Cost*)		
A	A1	A11	6	3	4
		A12	5		
	A2			2	

Find total costs
(A11 6; A12 5; A1 14; A2 2; A 20):

let *SumCost* **be red** $+$ **of** *CalCost*;
let *CalCost* **be if** *Assembly*$=\mathcal{DC}$ **then** *Cost*
    else *Cost* $+$ *Assembly.SumCost*;
Ans $<-$ [*Id*, *CalCost*] **in** *Assembly*;

The NEST and UNNEST operators are too unruly and unimportant to be implemented except in terms of other primitives which we need anyway.

- syntax in the domain algebra to name a set of attributes;

- a way of suppressing unneeded names of nested relations;

- **red ujoin** and **equiv ujoin** in the domain algebra, which we would have anyway as part of the subsumption of the relational algebra.

Operators on ABSTRACT DATA TYPES are supported by computations whose arguments are at once attributes and relations.

On to Programming Language III

Main idea:

Instantiate with domain and relational algebras

Topics

- Instantiation
- Unary Operators on Objects
- Binary Operators on Objects

34–40

Instantiation

- When a computation has a *state*, as for a counter or a flipflop, different uses of the computation may require different *instantiations* of the state.

- The ability to make many instances, as for all molecules of a gas, was the original motivation of the object-oriented approach.

- Relations are geared to support large numbers of instances, but deal with sets of instances, not individual "objects".

- So we need an instantiation mechanism which does not need to say **new** for each individual instance.

- We also need to be able to *hide* the state of each instance.

Instantiate by domain algebra actualization.

e.g., bank accounts

comp *ba*(*deposit, balance*) **is**
{ **state** *bal* **init** 0;

  deposit <− **comp**(*dep*) **is**
  { *bal* <− *bal* + *dep*
   alt
   dep <− *bal* − **old** *bal*;
  } /* deposit */

  balance <− **comp**(b) **is**
  { b <− *bal*;
  } /* balance */

} /* ba */

Given

$$\begin{array}{rl} \textit{accts}(\textit{ac\#} & \textit{client}) \\ 1 & \texttt{joe} \\ 2 & \texttt{sue} \end{array}$$

instantiate *ba* as an attribute

let *ba1* **be** [*deposit, balance*] **in** *ba*;
accounts <− [*ac#, client, ba1*] **in** *accts*;

accounts

(*ac#*	*client*	*ba1*				)
		(*deposit*	*balance*	[*bal*	*oldbal*])	
1	joe	..	..	0	-	
2	sue	..	..	0	-	

- [*bal oldbal*] are hidden. This state is instantiated for each account in *accts*.

- *deposit, balance*, the computations, are constant attributes

Binary Operators

- What if we need interactions between two (or more) objects of the same type?
- This is the *n-body problem* in object-orientation.

- For example, adding two numbers should not be asymmetrical, as in sending one in a message to the other.

Gas Molecules: *hitWall* is unary; *bang* is binary.

comp *molecules*(*initial, hitWall, getvel, putvel, ..*)
{ ...

  hitWall <− **comp**(*which*) **is**
  { **case** *which*:
   { "x": *xvel* <− −*xvel*;
    "y": *yvel* <− −*yvel*;
    "z": *zvel* <− −*zvel*;
  } } /* hitWall */

} /* molecules */

The binary operations are defined outside the class, with parameters.
For instance, *bang* collides two molecules (of equal mass), swapping their velocities.

bang <− **comp**(*mol1*, *mol2*, *mola*, *molb*) **is**
{ **local** *xv1*, *yv1*, *zv1*, *xv2*, *yv2*, *zv2*;
  mol1.getvel(*xv1*, *yv1*, *zv1*);
  mol2.getvel(*xv2*, *yv2*, *zv2*);
  mol2.putvel(*xv1*, *yv1*, *zv1*);
  mol1.putvel(*xv2*, *yv2*, *zv2*);
} /* bang */

This is all invoked on a relation of molecules.

gas <− **for** *mol#* **from** 1 **to** 10000 **par** *mol#*;

Binary Operators on Objects (cont.)

let *mol1* **be** [*initial, hitWall, getvel, putvel, ..*]
  in *molecules*;
let *mol2* **be** [*initial, hitWall, getvel, putvel, ..*]
  in *molecules*;
binaryMols $<-$ ([*mol#1, mol1*] **in** *gas*) **ijoin**
  [*mol#2, mol2*] **in** *gas*

update *binaryMols* **change**
  bang(**in** *mol1*, **in** *mol2*, **out** *mol1*, **out** *mol2*)
  using where *close* **in** (*binaryMols* **ijoin** *near*);

(*near* is another binary computation which inputs *mol1* and *mol2* and outputs the boolean, *close*.)

Conclusions

- – All the above was done with
  – Relational algebra
    * μjoins (intersection, union, etc.)
    * σjoins (subset, superset, etc.)
    * T-selectors (select, project)

  – Domain algebra
    * horizontal operations
    * vertical operations (**red**, **equiv**, **fun**, **par**)

  – Views & Recursion
  – Computations & Events

- – Abstract over looping
  – Better programmer productivity: 2 o.m.?
  – Exhaust the relational algebra
  – Separate the domain algebra
  – Driven by
    * Applicability (usefulness)
    * Elegance (generality)