ON THE EVALUATION OF RECURSION IN (DEDUCTIVE) DATABASE SYSTEMS BY EFFICIENT DIFFERENTIAL FIXPOINT ITERATION

Ulrich Güntzer, Werner Kiessling‡, Rudolf Bayer

Institut für Informatik, Technische Universität München, FRG
EARN: G11H11.DM0TUI1S
‡ Present affiliation: MAD Intelligent Systems, München

Abstract

Based on matured database technology the paper provides new insights into efficient ways to evaluate recursive deduction rules. We show how the forward-chaining approach to deduction can flexibly be married with goal-directed aspects of best/easiest-first strategies. From the natural fixpoint semantics of recursion we develop generally applicable differential iteration schemes that efficiently compute the fixpoint. Surprisingly the well-known Warshall-algorithm gets disclosed as a descendant of this class of algorithms. Performance measurements suggest the former as well as systolic Δ-algorithms with linear fixpoint equation as candidates for incorporating a transitive closure operator in databases. As a next important step towards integration of database technology and logic programming we suggest to profit from the standard features of concurrency control and transaction management by effectively using them for the synchronization of parallel deductions.

1. Introduction

Logic programming with Horn clauses provides a powerful specification tool that is known to have also an operational interpretation and thus can be used as an executable specification language [22]. Historically, the first implementation efforts were undertaken for Prolog(see, e.g., [11]), following the interpretative approach [14] with a theorem-prover using SLD-resolution [24]. Though much work was and is still being invested into developing fast Prolog implementations the results are still unsatisfactory, wrt. running time and even more storage requirements. (The violation of a clear declarative programming style by Prolog's cut and dependency on rule ordering and clause ordering apparently results from those inefficiencies.)

As relational database technology has matured over the past years, a strong interest is emerging recently to combine logic programming and standard advanced database systems [19,30]. The background for this intended coupling is the close relationship between relational algebra [12], deductive databases [14], and Horn clauses [24,5]. The fundamental result for this theoretical affinity is reported in [28,3], namely that operational proof-theoretic semantics (as represented by Prolog's SLD-resolution), model-theoretic semantics (smallest Herbrand model) and fixpoint semantics (smallest fixpoint) coincide for Horn clause programs. While non-recursive rules map straightforwardly into relational algebra expressions [26], the power of recursive rules goes beyond its capabilities [2,10] due to the unability to express the fixpoint operator.

The alternative to the interpretative approach is the compiled approach [14], which aims to bridge the gap between logic programming and relational databases. Hereby the accesses to the extensional database are to be done by set-oriented relational algebra operations, instead of Prolog's tuple-wise interaction with the database. Within the compiled approach itself one can distinguish between two methods. The first is embodied by the work of -e.g.- [9,17], or the top-down capture rule approach of [27]. The second method (called iteration on results in [13]) tries to exploit the fixpoint semantics directly by designing efficient algorithms for computing a fixpoint iteratively; see, e.g., the bottom-up capture rules of [27]. It is essential to see that in either method of the compiled approach all sophisticated database optimization techniques developed over the past decade [18] are fully applicable.

The main topic of this paper is to design and study efficient evaluation algorithms for recursive rules from the point of view of the natural fixpoint semantics. This problem is solved by defining a class of Δ-iteration algorithms ([4;20,25]contain first applications). They can be integrated in both methods of the compiled approach mentioned above as well. In section 2 several iteration paradigms are identified that in principle allow us to tailor the forward-chaining/breadth-first approach according to the application requirements in the sense of best/easiest-first deduction strategies. This is an essential feature in any deduction system, if the fixpoint is infinite or very large such that the user wants to see only a particularly interesting or easily computable part of it. Section 3 introduces several Δ-iteration schemes and provides the basic result that the database approach to deduction can *efficiently* incorporate concepts like best/easiest-first computation. For the transitive closure problem it is revealed that the well-known Kleene/Warshall algorithm [29] is a special variant of the more widely applicable Δ-iteration. In section 4 we show how simple it is to implement and optimize Δ- algorithms and we provide performance measurements that show their effectiveness. Section 5 points out an interesting correlation between control requirements for parallel deductions and standard concurrency control and transaction management in vanilla database systems. Finally, section 6 summarizes our results and provides a proposal how to effi-

CH2407-5/87/0000/0120$01.00 ©1987 IEEE

ciently integrate a transitive closure operator in databases. Then databases will be more suitably equipped to efficiently solve such important practical problems as, e.g., the bill of materials explosion.

2. Relaxation of Systolic Control

Let us study the fixpoint approach more closely by the familiar ancestor predicate:

$$Ancestor(x, y) \leftarrow Parent(x, y)$$

$$Ancestor(x, y) \leftarrow Ancestor(x, z) \wedge Parent(z, y)$$

The $\wedge$-operation of Horn clauses in combination with identical variables (here z) corresponds to the join operation of relational algebra (here with join predicate Ancestor.y = Parent.x) followed by a projection (here to x, y). The fixpoint semantics of Horn clauses is the smallest fixpoint Y of a monotonic (and even continuous) mapping F_P that may be associated uniquely with a set P of Horn clauses. In our example F_P is defined as the relational algebra equation F_P(Ancestor) = Ancestor $*$ Parent + Parent, where $*$ denotes the mentioned join-projection expression.

Our goal is to investigate very efficient techniques to compute the smallest fixpoint Y of a continuous function F. Y may in principle be approached from below by a sequence of sets $(\mathbf{S}_t)_{t \geq 0}$, where $\mathbf{S}_0 := \emptyset$ and $\mathbf{S}_t := F(\mathbf{S}_{t-1})$ for $t \geq 1$. Since F is monotonic, it is true that $\mathbf{S}_t \subseteq \mathbf{S}_{t+1}$. This also shows that the computation of $\mathbf{S}_{t+1}$ by evaluating the expression $F(\mathbf{S}_t)$ is highly redundant: it recomputes all the $\mathbf{S}_t$ again.

In preparation of our main results in section 3 we introduce some terminology concerning different types of iteration schemes. The iteration scheme

begin S := $\emptyset$; *while* F(S) $\neq$ S *do* S := F(S) *end*

delivers the fixpoint Y, if it is finite; otherwise it is exhausting Y. We call this type of iteration ***systolic,*** because there is a very strict discipline in enlarging $\mathbf{S}_t$ to the next version $\mathbf{S}_{t+1}$, corresponding to breadth-first forward chaining. But there are many situations where this strict control is not optimal or even not feasible. Fortunately a rather general "As-You-Please"-theorem proved in [15,16] allows us almost any other type of iteration as long as it is fair. Therefore some useful non-systolic types of iteration shall be considered and their correctness proved here without using the general theorem to make the paper self-contained.

Let us call an iteration ***sloppy,*** if not all, what should have been computed during the actual iteration step, has in fact been generated, i.e., if $S_{t+1} \subset F(S_t)$. ($A \subset B$ denotes $A \subseteq B$ and $A \neq B$. Also, for systolic iteration we use bold $\mathbf{S}_t$'s, whereas for other iteration types we use normal S_t's.) But sloppiness should not go this far as to lose tuples generated in previous steps, i.e., we require $S_t \subseteq S_{t+1}$ for all t. The consideration of sloppiness may be worthwhile for several reasons; e.g., to make the iteration process fault-tolerant or to be able to apply an best/easiest-first strategy. (Of course one has to worry whether those tuples that have been neglected are recaptured automatically or what to do about them.) Sloppy iteration consists in partial evaluation of $F(S_t)$; closely related is the concept of ***partitioned*** evaluation of $F(S_t)$, corresponding to deliberated and controlled sloppiness.

Let us define ***pushy*** iteration as the opposite of sloppy iteration. Thereby a single iteration step computes more than prescribed by systolic iteration, i.e., $F(S_t) \subset S_{t+1}$. But of course we require that S_{t+1} is still contained within the fixpoint Y; i.e., no wrong extraneous tuples are computed, but some tuples are delivered earlier than with systolic iteration and this may have a snow ball effect. To summarize:

Def. 2.1: An iteration sequence S_t, with $S_0 = \emptyset$, is

systolic, if $F(S_t) = S_{t+1} \quad \forall\, t$

sloppy , if $F(S_t) \supseteq S_{t+1}$ and $S_{t+1} \supseteq S_t \quad \forall\, t$

and $F(S_t) \neq S_{t+1}$ for *at most a finite number of t's*

pushy , if $F(S_t) \subseteq S_{t+1}$ and $S_{t+1} \subset Y \quad \forall\, t$

and $F(S_t) \neq S_{t+1}$ for *at least one t.*

The most important application of pushy iteration is ***transient*** iteration, where some of the newly generated tuples are fed back into S_t and lead then to the production of even more tuples. Algorithms and examples for all these iteration types shall be given in the next section.

This classification of iteration types depends on the relationship between S_{t+1} and $F(S_t)$. An orthogonal classification distinguishes according to how S_{t+1} is computed from S_t. So far we only have considered ***bulk iteration,*** where -e.g.- for systolic iteration S_{t+1} is computed from S_t by applying the operator F to S_t. This has the disadvantage that $F(S_{t-1})$ is de facto evaluated again when elaborating $F(S_t)$, because S_{t-1} is part of S_t and F is monotonic. Therefore it is worthwhile to consider differential iteration schemes where S_{t+1} gets computed from S_t and $\Delta_t := S_t \setminus S_{t-1}$. Such iteration schemes shall be called ***Δ-iteration*** schemes. The bulk iteration shall be used mostly as a yardstick to prove correctness and speed of the more efficient Δ-iteration schemes. Another feature of bulk iteration is that both sloppy and pushy iteration are correct:

Proposition 2.2:

(a) Every sloppy iteration sequence converges to the fixpoint (although at a slower pace in general).

(b) Every pushy iteration sequence converges to the fixpoint (normally at a faster rate).

Proof: see [31].

3. Delta-Iteration

To avoid the inefficiency of redundant computation inherent to bulk systolic iteration it is highly desirable to find an algorithm that computes F($\mathbf{S}_t$) by emphasizing those elements that were newly generated by the most recent iteration step; such elements are commonly called ***deltas*** (Δs). In the sequel we are going to describe two variants of this Δ-iteration method. Both of them use two

sequences of sets $S = (S_t)_{t \geqq 0}$ and $\Delta = (\Delta_t)_{t \geqq 1}$ to approximate the fixpoint Y.

3.1. Δ1–iteration

Here we perform the computation of S and Δ according to the following discrete timing:

$$(S_{t-1}, \Delta_t) \longrightarrow (S_t, \Delta_{t+1}) \quad , t \geqq 1$$

Now we state, how the mentioned algorithm can be defined in a "clever" way for the Δ1–iteration (a justification will be supplied later).

Def. 3.1: $\delta 1(S, \Delta) := F(S + \Delta) \setminus F(S)$

This yields the following iteration scheme to compute the fixpoint $F(\mathbf{S}) = \mathbf{S}$:

Iteration scheme Δ1:

Initialization: $(S_0, \Delta_1) := (\emptyset, F(\emptyset))$
Iteration:

$$(S_{t-1}, \Delta_t) \longrightarrow (S_t, \Delta_{t+1}) := (S_{t-1} + \Delta_t, \delta 1(S_{t-1}, \Delta_t))$$

Invariants for Δ1: (for t ≧ 1)

(a) $F(S_{t-1}) = S_t = \mathbf{S}_t$

(b) $\Delta_t = \mathbf{S}_t \setminus \mathbf{S}_{t-1}$

Proof:
(a) t = 1: $F(S_0) = F(\emptyset) = \mathbf{S}_1$ and also $F(S_0) = F(\emptyset) = S_0 + \Delta_1 = S_1$.

$$\begin{aligned} t \to t+1: \quad F(S_t) &= (Def.\,\delta 1,\ F\ monotonic) \\ \delta 1(S_{t-1}, \Delta_t) + F(S_{t-1}) &= (induction\ hypothesis) \\ \delta 1(S_{t-1}, \Delta_t) + S_t &= (\Delta 1\text{-}scheme) \\ \Delta_{t+1} + S_t &= (\Delta 1\text{-}scheme) \\ S_{t+1} \quad \square \end{aligned}$$

Because of $S_t = \mathbf{S}_t$ we also have $F(S_t) = F(\mathbf{S}_t) = \mathbf{S}_{t+1}$.
(b) $\Delta_1 = F(\emptyset) = \mathbf{S}_1 \setminus \mathbf{S}_0$. Let $t \geqq 1$, then:

$$\Delta_{t+1} = \delta 1(S_{t-1}, \Delta_t) = F(S_{t-1} + \Delta_t) \setminus F(S_{t-1}) = F(S_t) \setminus F(S_{t-1}) = \mathbf{S}_{t+1} \setminus \mathbf{S}_t \quad \square$$

Correctness and termination of Δ1:
The correctness is a direct consequence of invariant (a). Therefore is Δ1 a systolic iteration. Invariant (b) is the halting condition and means that the iteration has approached the smallest fixpoint and gets stationary, when $\Delta_t = \emptyset$ occurs. For a finite fixpoint this must happen after finitely many steps.

Efficient computation of δ1:
(For the following discussion we omit the iteration index t for clarity.)
From the invariants it follows that δ1(S, Δ) = F(S+Δ) \ F(S) = F(S+Δ) \ (S+Δ) holds. This property cannot be expected to hold for arbitrary S and Δ. However, the difference F(S+Δ) \ F(S) may still be well-behaved, which is intuitively clear, because F(S+Δ) \ F(S) can approximately be described as the "derivative of F at S" ($\delta 1(S, \Delta) \approx \Delta \cdot \frac{dF}{dS}(S)$). Thus in many practical applications –concrete examples follow– one can hope to compute the difference F(S+Δ) \ F(S) much cheaper by introducing a proper ***auxiliary function*** Aux1(S, Δ).

Def. 3.2:
Any function Aux1(S, Δ) such that $\delta 1(S, \Delta) = Aux1(S, \Delta) \setminus (S+\Delta)$ for $S = \mathbf{S}_{t-1}$ and $\Delta = \mathbf{S}_t \setminus \mathbf{S}_{t-1}$ is called an auxiliary function (for Δ1).

So Aux1 is by no means uniquely defined, and for sure we are interested to find the "cheapest" one according to the problem to be solved. In the worst case Aux1(S, Δ) = F(S+Δ) satisfies the above definition. Now we are prepared to present an efficient while-program, which implements our Δ1–iteration:

Δ1: S := ∅; Δ := F(∅);
while Δ ≠ ∅ *do begin* aux := Aux1(S, Δ); S := S $\dot{+}$ Δ;
Δ := *aux* \ *S end*

The operator $\dot{+}$ denotes the disjoint union. ($S_{t-1} \cap \Delta_t = \emptyset$ follows immediately from invariant (b)).

3.2. Δ2–Iteration

The timing of the iteration steps is done for this method as follows:

$$(S_t, \Delta_t) \longrightarrow (S_{t+1}, \Delta_{t+1})$$

Def. 3.3: $\delta 2(S, \Delta) := F(S) \setminus F(S \setminus \Delta)$

Iteration scheme Δ2:

Initialization: $(S_1, \Delta_1) := (F(\emptyset), F(\emptyset))$
Iteration:

$$(S_t, \Delta_t) \longrightarrow (S_{t+1}, \Delta_{t+1}) := (S_t + \delta 2(S_t, \Delta_t), \delta 2(S_t, \Delta_t))$$

Invariants for Δ2:

(a) $F(S_t \setminus \Delta_t) = S_t = \mathbf{S}_t \qquad (t \geqq 1)$

(b) $\Delta_t = \mathbf{S}_t \setminus \mathbf{S}_{t-1} \qquad (t \geqq 2)$

Proof:
(a) t=1: $F(S_1 \setminus \Delta_1) = F(\emptyset) = \mathbf{S}_1$ and also $F(\emptyset) = S_1$.

$$\begin{aligned} t \to t+1: \quad F(S_{t+1} \setminus \Delta_{t+1}) &= \\ F((S_t + \delta 2(S_t, \Delta_t)) \setminus \delta 2(S_t, \Delta_t)) &= \\ F(S_t \setminus \delta 2(S_t, \Delta_t)) &= \\ F(S_t \setminus (F(S_t) \setminus F(S_t \setminus \Delta_t))) &= \\ F(\mathbf{S}_t \setminus (F(S_t) \setminus \mathbf{S}_t)) &= \\ F(\mathbf{S}_t) &= \mathbf{S}_{t+1} \quad \square \end{aligned}$$

Thus, we likewise have:
$S_{t+1} = S_t + \delta 2(S_t, \Delta_t) = S_t + (F(S_t) \setminus F(S_t \setminus \Delta_t))$.
By induction assumption and F monotonic we can conclude $S_{t+1} = S_t + (F(S_t) \setminus S_t) = S_t + F(S_t) = F(\mathbf{S}_t) = \mathbf{S}_{t+1}$.
(b) $\Delta_{t+1} = \delta 2(S_t, \Delta_t) = F(S_t) \setminus F(S_t \setminus \Delta_t) = F(\mathbf{S}_t) \setminus \mathbf{S}_t = \mathbf{S}_{t+1} \setminus \mathbf{S}_t \quad \square$

Correctness and termination arguments are analogous to the Δ1–method.

Def. 3.4:
Any function Aux2(S, Δ) such that $\delta 2(S, \Delta) = Aux2(S, \Delta) \setminus S$ for $S = \mathbf{S}_t$ and $\Delta = \mathbf{S}_t \setminus \mathbf{S}_{t-1}$ is called an auxiliary function (for Δ2).

Again, Aux2 is by no means uniquely defined, and our task will be to find the "cheapest" one according to the

problem to be solved. The equivalent while-program, which efficiently implements the Δ2-iteration looks as follows:

Δ2: S := Δ := F(∅);
while Δ ≠ ∅ *do begin* aux := Aux2(S, Δ);
Δ := aux \ S; S := S +˙ Δ *end*

3.3. Correlation between Δ1 and Δ2

Proposition 3.5: $\delta 1(\mathbf{S}_{t-1}, \Delta_t) = \delta 2(\mathbf{S}_t, \Delta_t)$

Proof: The invariants for Δ1 and Δ2 have shown that S_t and Δ_t are identical for Δ1 and Δ2 by showing that S_t is identical to $\mathbf{S}_t$ produced by bulk systolic iteration. This observation suffices to prove this proposition.

Consequence 3.6: Δ1-iteration is preferable to Δ2-iteration, if Aux1 depends on S; if Aux1 = Aux2 and does not depend on S, then Δ2-iteration may be preferable.

Proposition 3.5 tells us that δ1 computes the same values as δ2, however from a smaller set ($S_{t-1} \subset S_t$ in general). Therefore in general Aux1 can be computed more easily than Aux2. On the other hand, if Aux1 = Aux2 and Aux1 does not depend on S, then Δ2 is slightly more efficient as we will see later (sect. 4 on implementation aspects).

3.4. Examples

Example 1: The set of strings $\{1\} \circ A^*$ for some alphabet A.
The logic program

S(1) ←
S(y) ← S(x), A(z), y = x ∘z

yields the fixpoint equation: F(S) = S ∘A + {1}. (Note: F has an infinite least fixpoint.)
Δ1-iteration: (We'll drop again the iteration index t for the subsequent examples.)
δ1(S, Δ) = F(S+Δ) \ F(S) =
((S+Δ) ∘A + {1}) \ (S ∘A + {1}) =
(S ∘A + Δ ∘A + {1}) \ (S ∘A + {1}) = (Δ ∘A) \ (S ∘A + {1}) =
(Δ ∘A) \ (S + Δ)
Thus Aux1(S, Δ) = Δ ∘A is a good choice.
Iteration: S := ∅; Δ := F(∅) = {1};
while Δ ≠ ∅ *do begin* aux := Δ ∘*A; S := S +˙ Δ;*
Δ := *aux* \ *S end*
Since aux contains strings that are longer by 1 than those in S, aux \ S = aux holds. This allows us to make another optimization here:
Δ1: S := ∅; Δ := {1}; *while* Δ ≠ ∅ *do begin* S := S +˙ Δ;
Δ := Δ ∘A *end*

Example 2: Transitive closure
We consider a binary relation R(s, e) containing the edges (s stands for start vertex, e for end vertex) of a directed graph. The operation * gets defined as a special join operation on two binary relations, say, A(as, ae) and B(bs, be) as:

$$A * B := \pi_{as, be}(A \underset{ae = bs}{equijoin} B)$$

(E.g., for a sample graph R = {(1 2), (2 3), (3 4), (2 4), (4 1)} we get R*R = {(4 2), (1 3), (1 4), (2 4), (2 1), (3 1)}.)

Case 1: The logic program

S(x, y) ← R(x, y).
S(x, y) ← S(x, z), R(z, y).

yields the linear fixpoint equation F(S) = S*R + R.
This fixpoint equation essentially corresponds to the previous from example 1. Therefore for the Δ1-iteration our choice is Aux1(S, Δ) = Δ * R. From prop. 3.5 we know that for the Δ2-iteration Aux2 can be chosen as Aux2(S, Δ) = Δ * R too.

Case 2: The logic program

S(x, y) ← R(x, y).
S(x, y) ← S(x, z), S(z, y).

yields the quadratic fixpoint equation F(S) = S*S + R.
Δ1-iteration:

$$\delta 1(S, \Delta) = ((S+\Delta) * (S+\Delta) + R) \setminus (S * S + R) =$$
$$(S * \Delta + \Delta * S + \Delta * \Delta) \setminus (S * S + R) =$$
$$(S * \Delta + \Delta * S + \Delta * \Delta) \setminus (S + \Delta)$$

Thus we choose Aux1(S, Δ) = S*Δ + Δ*S + Δ*Δ.

Δ2-iteration: As can be derived easily from prop. 3.5, it holds that δ2(S, Δ) = δ1(S \ Δ, Δ). Therefore

$$\delta 2(S, \Delta) = ((S \setminus \Delta) * \Delta + \Delta * (S \setminus \Delta) + \Delta * \Delta) \setminus S =$$
$$(((S \setminus \Delta) + \Delta) * \Delta + \Delta * ((S \setminus \Delta) + \Delta)) \setminus S =$$
$$(S * \Delta + \Delta * S) \setminus S$$

Our choice for Aux2 will be then Aux2(S, Δ) = S*Δ + Δ*S.
Note: Aux2 appears to be simpler here than Aux1, contradicting consequ. 3.6. But this is a deception, because with a closer look one recognizes that the term Δ*Δ is present twice, i.e., once more than for Aux1.

3.5. Interdependence of sloppy iteration and Δ-iteration

Now we want to show by an example that sloppy execution of one single iteration step during Δ-iteration may ruin the whole iteration. This stands in marked contrast to the general bulk iteration, which reaches the fixpoint in spite of many sloppy iteration steps (recall sect. 2).

Proposition 3.7: Sloppy execution of Δ-iteration is incorrect.

We show this by our example 1 with F(S) = S ∘A + {1} for A = {0, 1}, i.e., strings in {0, 1}* that start with 1. Because of $\Delta_{t+1} = \Delta_t \circ A$ and $\Delta_1 = \{1\}$ we have Δ_t = {ω; |ω| = t and ω starts with 1}. We know that $\mathbf{S}_n = \sum_{1 \leq t \leq n} \Delta_t$ = {ω; |ω| ≦ n and ω starts with 1}. Take an arbitrary sloppy sequence. Then there is some n≧0 such that we have applied the Δ-iteration correctly up to the n-th step (i.e., $S_n = \mathbf{S}_n$), however for the (n+1)-th step we sloppily compute Δ_{n+1}. Then in S_{n+1} there are some strings of length n+1 missing. As for Δ_{n+2}, Δ_{n+3}, ... only strings of length ≧ n+2 will be generated this loss can never be made up by the Δ-iteration. □

This proposition means that it would be fallacious that also for a sloppy iteration Δ = ∅ is sufficient for correct termination. But it should not be read as: control for Δ-iteration may not be relaxed or strict systolic control must

always be enforced. The proposition only says that losses during one iteration step must be taken into account during one of the following iteration steps by a suitable repair measure, e.g., in the way described now: Let Δ_t be partitioned into two parts, $\Delta_t := \Delta_t^n \dot{+} \Delta_t^r$. (n and r mean "nice" and "rest", resp.)

For $\Delta 1$ it then holds: $F(S_{t-1}) = S_{t-1} \dot{+} \Delta_t = S_{t-1} \dot{+} \Delta_t^n \dot{+} \Delta_t^r$. It also holds:

$$F(S_{t-1} + \Delta_t) = F((S_{t-1} + \Delta_t^n) + \Delta_t^r) =$$

$$\delta 1(S_{t-1} + \Delta_t^n, \Delta_t^r) \dot{+} F(S_{t-1} + \Delta_t^n) =$$

$$\delta 1(S_{t-1} + \Delta_t^n, \Delta_t^r) \dot{+} \delta 1(S_{t-1}, \Delta_t^n) \dot{+} F(S_{t-1}) =$$

$$\blacktriangleright \quad S_{t-1} \dot{+} \Delta_t^n \dot{+} \Delta_t^r + \delta 1(S_{t-1}, \Delta_t^n) \dot{+} \delta 1(S_{t-1} + \Delta_t^n, \Delta_t^r)$$

This equation can be interpreted as follows:

(1) Partition Δ_t into Δ_t^n and Δ_t^r such that Δ_t^n receives the best/easiest portion and Δ_t^r the rest.

(2) $(S_{t-1}, \Delta_t^n) \longrightarrow (S_{t-1} + \Delta_t^n, \delta 1(S_{t-1}, \Delta_t^n))$
/ use the best/easiest part first /

(3) $(S_{t-1} + \Delta_t^n, \Delta_t^r) \longrightarrow (S_{t-1} + \Delta_t, \delta 1(S_{t-1} + \Delta_t^n, \Delta_t^r))$
/but don't forget the rest /

The above equation ▶ now just proves that with $S_t = S_{t-1} \dot{+} \Delta_t$ and $\Delta_{t+1} = \delta 1(S_{t-1}, \Delta_t^n) \dot{+} \delta 1(S_{t-1} + \Delta_t^n, \Delta_t^r)$ this proceeding is correct. Moreover, (2) and (3) could be executed by some sort of pipelining mechanism. Recognizing that the above partitioning into "nice" and "rest" describes an easiest/best-first strategy, we can state:

Proposition 3.8: Sloppy Δ-iteration in combination with best/easiest-first strategies is correctly feasible.

3.6. Partitioned $\Delta 2$-iteration

For convenience we restrict ourselves to $\Delta 2$-iteration. It can be advisable to decompose the work of the operator Aux2 into different parts, e.g., to exploit parallelism or to use an easiest- or best-first strategy or to take advantage of some inherent partitions of S. Therefore we assume that Aux2 can be partitioned into n monotonic operators $Aux2_k$ ($1 \leq k \leq n$), such that

$$\blacktriangleright \quad Aux2(S, \Delta) = \sum_{1 \leq k \leq n} Aux2_k(S, \Delta)$$

for all S, Δ such that $\Delta \subset S$, where $\sum$ denotes set-theoretical union.

Of course this partition can only be useful, if the operators $Aux2_k$ are considerably simpler than Aux2 itself. Let us consider our running example, the transitive closure R^+ of a binary relation R with attributes s and e.

(1) First we consider linear iteration according to
F(S) = S∗R + R.

As we have seen Aux2 can be chosen as Aux2(S, Δ) := Δ∗R. For notational convenience assume that V := $\pi_s(R) + \pi_e(R)$ equals {1, ..., n}; i.e., we label the nodes in the graph represented by R with numbers from 1 to n. Define $\Delta^k := \sigma_{e=k}(\Delta)$ for k=1, ..., n; i.e., Δ^k contains exactly those edges from Δ ending in point k. This partition of Δ causes a partition of Aux2 as follows: Put $Aux2_k(S, \Delta) := \Delta^k * R$. Then $Aux2_k$ is monotonic and we have

$$\blacktriangleright \quad \sum_{1 \leq k \leq n} Aux2_k(S, \Delta) = \sum_{1 \leq k \leq n} \Delta^k * R = (\sum_{1 \leq k \leq n} \Delta^k) * R = \Delta * R = Aux2(S, \Delta)$$

(2) Next we consider quadratic iteration with
F(S) = S∗S + R.

Here Aux2 can be chosen as Aux2(S, Δ) = Δ∗S + S∗Δ. Define ${}^k\Delta := \sigma_{s=k}(\Delta)$, i.e., ${}^k\Delta$ contains all the edges of Δ starting in k. S^k and ${}^kS$ are defined analogously. Using these two partitions of S and Δ we can define $Aux2_k(S, \Delta) := \Delta^k * {}^kS + S^k * {}^k\Delta$. Then one sees immediately that all these operators are monotonic and condition ▶ is satisfied:

$$\sum_{1 \leq k \leq n} Aux2_k(S, \Delta) = \sum_{1 \leq k \leq n} (\Delta^k * {}^kS + S^k * {}^k\Delta) = \Delta * S + S * \Delta = Aux2(S, \Delta)$$

If one has several processors available they can elaborate $Aux2_1, \ldots, Aux2_n$ in parallel. Moreover, a further interesting application of partitioned iteration is ***transient*** partitioned iteration, which shall be discussed now.

3.7. Transient partitioned $\Delta 2$-iteration

Let $Aux2_1, \ldots, Aux2_n$ be a partition of the operator Aux2 into monotonic components. When $Aux2_1$ has finished its job, in general it has produced some new tuples that can be taken into account by component $Aux2_2$, which in turn can feed its results into the parameters of $Aux2_3$ and so on. Formally this transient partitioned iteration scheme gets defined as follows, assuming that the equation Aux2(S, Δ) \ S = F(S) \ S not only holds for S = $\mathbf{S}_t$ and Δ = $\mathbf{S}_t \setminus \mathbf{S}_{t-1}$ as required in def. 3.4, but also is valid for all S and Δ such that $\Delta \subset S$ and $F(S \setminus \Delta) \subset S$. This is true, e.g., for the two functions Aux2 from the previous section.

Def. 3.9:

(a) $ST_0(S, \Delta) := S; \qquad \Delta T_0(S, \Delta) := \Delta$

(b) for k = 1, ..., n:

$$Aux2T_k(S,\Delta) := Aux2_k(ST_{k-1}(S, \Delta), \Delta T_{k-1}(S, \Delta))$$

$$ST_k(S, \Delta) := ST_{k-1}(S, \Delta) + Aux2T_k(S, \Delta)$$

$$\Delta T_k(S, \Delta) := \Delta T_{k-1}(S, \Delta) + Aux2T_k(S, \Delta)$$

Def. 3.10:

(a) $Aux2T(S, \Delta) := \sum_{1 \leq k \leq n} Aux2T_k(S, \Delta)$

(b) $\delta 2T(S, \Delta) := Aux2T(S, \Delta) \setminus S$

Comments: $ST_k(S, \Delta)$ and $\Delta T_k(S, \Delta)$ are the transient versions of S and Δ, resp., updated by the k-th partition. $Aux2T_k$ is monotonic, because $Aux2_k$ is so.

Using the operator $\delta 2T$ for incrementing S yields the following

Iteration scheme $\Delta 2T$:

Initialization: $(S_1, \Delta_1) := (F(\emptyset), F(\emptyset))$

Iteration:

$$(S_t, \Delta_t) \longrightarrow (S_{t+1}, \Delta_{t+1}) := (S_t + \delta 2T(S_t, \Delta_t), \delta 2T(S_t, \Delta_t))$$

Invariants for $\Delta 2T$: ($t \geq 1$)

(a) $\Delta_t \subseteq S_t$, $\Delta_t \cap S_{t-1} = \emptyset \quad (S_0 := \emptyset)$

(b) $F(S_{t-1}) = F(S_t \setminus \Delta_t) \subseteq S_t$

(c) $\mathbf{S}_t \subseteq S_t \subseteq Y$

Proof: see 31.

Invariants (b) and (c) tell us that $(S_t)_{t \geq 1}$ is a pushy iteration sequence converging to Y. During a single iteration step starting with S_t (and Δ_t) all tuples belonging to $F(S_t)$ and possibly more –due to transience of S and Δ– are computed, but no extraneous tuples are produced. Y is reached if and only if $\Delta_{t+1} = \emptyset$. Thus we have verified:

Proposition 3.11:
The transient iteration scheme Δ2T is correct. It delivers the least fixpoint, if it terminates with $\Delta = \emptyset$, else the sequence S_t approximates the fixpoint at least as fast as systolic iteration.

Remarks:
(1) The n operators $Aux2T_k$ may be parallelized to some extent. They can start their work simultaneously, only their finishing of their jobs must be sequentialized to make sure that $Aux2T_k$ can take into account what $Aux2T_{k-1}$ has produced.
(2) The formal speeding up of the iteration Δ2T as compared with Δ2 may be paid for by more duplicates because of bigger Δs. Therefore it still remains to be shown that the transient method is useful.

We are now ready to transform Δ2T into an (unoptimized) equivalent while-program.

```
Δ2T:  S := Δ := F(∅);
   while Δ ≠ ∅ do
     begin Aux2T := ∅; S_old := S;
      for k:=1 to n do
        begin aux := Aux2_k(S, Δ); S := S + aux;
         Δ := Δ + aux; Aux2T := Aux2T + aux end;
        Δ := Aux2T \ S_old
     end
```

Transitive closure revisited:
We return to the quadratic equation F(S) = S∗S + R for the transitive closure and want to compute it by the above program Δ2T. We make the assumption (to be justified below) that one single execution of the body of the while-loop is sufficient, i.e., yields already the fixpoint. Then because of the initialization S := Δ := F(∅) = R we see that Δ = S holds during the for-loop. Then Δ and Aux2T and then aux are no longer needed. From the introductory motivation for the partitioned Δ-iteration we know $Aux2_k(S, S) = S^k * {}^k S = \sigma_{e*k}(S) * \sigma_{s*k}(S)$.
Then the program for Δ2T could be simplified to:
W: S := R; *for* k:=1 *to* n *do* $S := S + \sigma_{e*k}(S) * \sigma_{s*k}(S)$;
This is WARSHALLS's algorithm [29] in the notation of relational algebra! It is not hard to prove that Warshall's algorithm delivers the transitive closure within one loop for k from 1 to n. Since W can be transformed back into Δ2T, also Δ2T may stop after finishing the first for-loop, satisfying our assumption. Thus we have:

Proposition 3.12:
WARSHALL's algorithm is a special case of transient Δ2-iteration for the fixpoint equation F(S) = S∗S + R, needing only one (transient) iteration step.

Remarks:
(1) This same result can also be found starting out with Δ1-iteration.
(2) Because this algorithm comes through with a single iteration step, the Δs are no longer visible. Nevertheless Warshall's algorithm is a genuine descendant of the family of Δ-procedures as the following situation shows: Assume that the transitive closure has been computed and the base relation is enlarged by new facts coming in (a typical situation for the applications we have in mind). Then the transitive closure of the enlarged relation is computed by ***one*** further iteration step of a suitable transient Δ-iteration.

4. Implementation and Performance Results

Our experimental logic programming system that is currently under development has the gross architecture of fig. 4.1.
For reasons of fast prototyping –and others– we decided to augment conventional LISP with the power of relational algebra. Basically R-LISP consists of a fast implementation of the relational algebra on a INTERLISP system (currently on a SIEMENS 7536/BS2000). Of course, a connection from R-LISP to a standard database system with a relational algebra interface can be established at will. The Δ-algorithms become part of the rule optimizer. It should be clear that the generation of the while-programs can be fully mechanized: the fixpoint equation F can be generated automatically from the Horn clauses, the auxiliary functions can be gained by symbolic evaluation according to the distributive laws for relational algebra. R-LISP contains also the so-called Δ⁺-operators in an optimized form as described now.

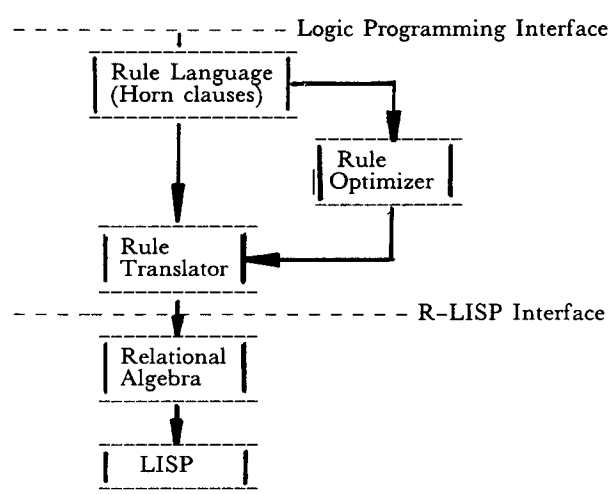

Fig. 4.1: Logic programming system on R-LISP.

Δ⁺-operators
Taking a look at the while-programs for Δ1, Δ2 and Δ2T one can recognize that the following sequence of set-operations must be evaluated repeatedly:

Δ1: $\{ +^{*} ; \setminus \}^{*}$; Δ2: $\{ \setminus ; +^{*} \}^{*}$; Δ2T: $\{\{+ ; + ; +\}^{*} ; \setminus \}^{*}$

Thus we are faced with the problem how to efficiently implement such a series of union and difference operations. For Δ1, Δ2 the costly operation is \, which is $O(N^2)$, if implemented carelessly. However, if a fast access path on S was available, then the cost for one \ would be reduced to $O(N \log N)$ for a balanced tree index or even $O(N)$ for a hash index with a luckily chosen hash function. The point now is that even in the absence of such a suitable access path it certainly will pay to create one on-the-fly, when we start the iteration. This consideration leads us to the following efficient **Δ⁺-operators**.

Constituents of the Δ1⁺-operator:
Δ1⁺-init: << S := ∅; initialize access path I_S *on S* >>
Δ1⁺-iterate: begin aux := Aux1(S, Δ); S := S $\dot{+}$ Δ;
Δ := aux \\ S *end*

The operator $\dot{+}$ inserts Δ-elements via I_S into S (no duplicate test must be done); I_S is updated accordingly. The operator \\ is an efficient difference, using I_S for a fast membership test in S.

Constituents of the Δ2⁺-operator:
Δ2⁺-init: << *S := Δ := F(∅); create access path* I_S *on S* >>
Δ2⁺-iterate: begin aux := Aux2(S, Δ); S := S +- aux;.
Δ := aux *end*

The operator +- checks via I_S, whether an aux-element x is in S (duplicate test); if not, x is inserted and I_S is updated accordingly, otherwise x gets removed from aux. In this way we directly insert those aux-elements into S immediately, which would have been posted there exactly one round later in the un-optimized Δ2-scheme. This saves one traversal of I_S.

To design an efficient $\Delta 2T^+$-operator we need some additional considerations. First, we transform the Δ2T-while-program by pushing the \ into the inner for-loop as follows:

```
Δ2T:   S := Δ := F(∅);
 while Δ ≠ ∅ do
   begin Aux2T := ∅;
     for k:=1 to n do
       begin aux := Aux2_k(S, Δ); S := S +̇ (aux \ S);
                 Δ := Δ +̇ (aux \ Δ);
               Aux2T := Aux2T + (aux \ S) end;
       Δ := Aux2T
   end
```

Observing that the insertion of elements from aux, which lie in S \ Δ, into Δ makes no sense, we arrive at the following efficient $\Delta 2T^+$-operator:

Constituents of the $\Delta 2T^+$-operator:
$\Delta 2T^+$-init: << Create access path I_S *on S := F(∅)* >>
$\Delta 2T^+$-iterate:
for k:=1 *to* n *do*
begin aux := $Aux2_k(S, \Delta)$; *S := S +- aux;*
Δ := Δ $\dot{+}$ aux; Aux2T := Aux2T $\dot{+}$ aux end

Fast Warshall implementation
To shown that the additional complexity of the transient methods -as far as the description is concerned- does not lead to increased use of time or space, we study the Warshall algorithm more closely. Obviously it is a bad idea to implement it straightforwardly by n separate selection and join operations, particularly if the selections are performed by exhaustive relation scans. Having disclosed Warshall's algorithm as a transient Δ-iteration the following optimization is clear now:
Basically there are two ways to accelerate the selection operation:
(1) Produce 2 identical versions of the base relation R(s, e) and sort one w.r.t. to the first and one w.r.t. the second component. Then maintaining two pointers on each version will suffice to efficiently handle the selections. Insertions of newly produced tuples are done in an order-preserving way. Thus only one simultaneous scan over both transient versions of R is required.
(2) Create two access paths on s and e, resp. Selections are done efficiently via them, the same holds for insertion of new tuples. We chose this second approach and used hashing. We came up with this choice, because no deletions are to be performed (F is monotonic).

Further refinements:
- The join-operator * can be replaced by a Cartesian product x. This gives a slight improvement, because x works faster here than * (all tuples in $\sigma_{e*k}(S)$ and $\sigma_{s*k}(S)$ participate in the result).
- Another speedup can be gained as follows: Instead of partitioning R into n parts, where n is the number of nodes of the underlying graph G, it is sufficient to restrict the iteration to ni steps, where ni is the number of inner nodes of G; i.e., nodes that have an incoming and outgoing edge.

Benchmark tests
All Δ-iteration schemes and the Warshall algorithm were implemented on top of R-LISP. We'll provide now the performance results for the transitive closure problem, both for the linear fixpoint equation
F(S) = S*R + R (l) and for the quadratic F(S) = S*S + R (q) . We present results for the 6 algorithms listed in table 4.2. They are applied to various ***test-graphs*** documented in [31]; for the performance results with these graphs see the appendix, table A.1 for a preliminary version of R-LISP and table A.2 for a tuned one.

Interpretation of performance results:
There are several interesting observations to state:
(a) The primitive methods bear their name right. There are huge gains by the Δ-methods.
(b) The pure greedy way to compute the transitive closure by quadratic iteration is bad, even with Δs; the reason is the production of too many duplicates. Consequently, whenever there are several possibilities of specifying an algorithm with Horn clause one should select that with the smallest Δs expected.

name	comments
L-P	linear F, no Δs (Primitive)
Q-P	quadratic F, no Δs (Primitive)
Q-Δ1	quadratic F, Δ1
L-Δ1	linear F, Δ1
L-Δ2T	linear F, Δ2T
FW	Fast Warshall

Table 4.2: Tested iteration algorithms for transitive closure.

(c) L-Δ2T versus L-Δ1 (L-Δ2 performed pretty identical to L-Δ1): The transient Δ-methods seem to lose against the straight Δs for this transitive closure problem.
(d) Comparing the results in table A.2 and A.3, one can recognize the influence of the relative overhead of the underlying relational algebra operations. So, depending on the performance of a particular relational algebra interface and on the structure of the input data, L-Δ2 (or L-Δ1) or FW is the candidate of choice for implementing a transitive closure operator. However, implementing both of them and keeping them in a method library of alternative algorithms makes sense. An intelligent facility for selecting a good algorithm as described in [23] could make the right choice automatically, if supplied with the proper knowledge.

5. Practically Feasible Relaxation of Systolic Control

This section is concerned with the question, how to synchronize mutually recursive rules systems, if non-systolic Δ-iteration is to be used. Let us study the simplest mutually recursive system

S1 = F1(S1, S2)
S2 = F2(S1, S2).

easily be figured out: reduce the idle times as much as it is correctly permissible. In principal, there are two strategies to achieve this:
(a) As soon as the fastest Aux-process terminates, interrupt all other processes, do the Δ⁺-processing and start over with a new iteration round. Obviously, this is a sloppy iteration, which we have shown to be incorrect in combination with Δs, unless special counter measures are taken. So we quit this possibility here and study the alternative.
(b) The time until the slowest Aux-process terminates is utilized by (otherwise idle) Aux-processes in doing some extra -i.e. pushy- work; e.g. Aux1 might grasp what Aux2 has generated up to this point. We know already that this strategy is correctly feasible also in the presence of Δs; however, we still have to point out efficient ways to synchronize the different processes when accessing the shared data objects S1, S2, D1 and D2 (in the above example). One could think of synchronization techniques like communicating sequential processes, use of object versions or timestamps and so on. The crucial point in these considerations is the question of practical feasibility and efficiency, though. This then carries us directly to our solution proposal for this difficult problem.

The database approach to parallel deduction:
The concurrency control problem for database systems was the subject of stormy research activities until recently and belongs to one of the best-understood topics in the database area nowadays; see, e.g., [7,6]. Thus, we would benefit a lot, if we succeed in applying standard concurrency control methods to schedule our special parallel application. The most-widely implemented consistency criterion for parallel transactions is ***serializability*** (also called *degree-3 consistency*). Though scheduling parallel transactions this way is known not to produce the theoretically maximum amount of parallelism in any case, it certainly integrates database technology and logic programming in an important and effective way as the following proposition shows.

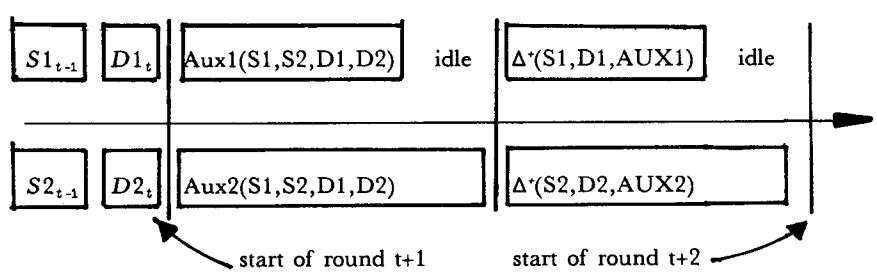

Fig. 5.1: Systolic control and Δ-iteration.

Strict systolic control governs the iteration process as sketched in fig. 5.1. Observe that actually there are two synchronization points, the first after the slowest Aux-computation terminates and the second after the slowest Δ⁺-computation terminates. A relaxation of such a strict systolic control can serve two purposes, namely to pursue some goal-directed iteration (this will not be our issue here) or to increase the throughput produced by systolic control. From the above diagram the plan to reach this can

Proposition 5.2: Enforcing serializability for parallel Δ-iteration guarantees a pushy iteration.

Proof:
The parallel execution of a set $\{T_1, T_2, \cdots, T_n\}$ of transactions is serializable, if $\{T_1, T_2, \cdots, T_n\}$ produces the same effect as some serial execution $\{T_{p(1)}, T_{p(2)}, \cdots, T_{p(n)}\}$, where p is a permutation of {1, ..., n}. For our fixpoint computation by Δ-iteration the

role of a transaction T_i is taken over by the Δ-iteration process for the i-th rule of the rule system. Serializability now means that $T_{p(i)}$ takes into account all that has been generated by $T_{p(i-1)}$ and so on. The bottom line of such a schedule is that $T_{p(1)}$ produces the same as for strict systolic serial execution schedule, whereas all others might produce somewhat more than prescribed by systolic iteration; i.e., we get a pushy iteration, which we have shown to be correct. □

In practice, several competing ***lock protocols*** are available to implement the desired form of concurrency control. For the above sample system, read locks on S1, S2, D1, D2 (or pieces of them) would be required by the two Aux-processes and write locks on S1, D1 and S2, D2 by the respective Δ^+-processes. With the proper choice of the lock granularity one can govern the amount of wanted potential parallelism. While the performance impacts of those methods have been investigated for general-purpose transaction schemes (see, e.g., [21]), there might be novel techniques for this type of application here, mainly because fixpoint iteration does not necessarily require the restrictive consistency criterion of serializability. Besides this, the collection of parallel iteration processes resembles very closely the notion of *long, nested transactions*; see, e.g., [8]. Therefore there will be a benefit from the various development efforts in this particular database area for the implementation of parallel deduction, for sure, and possibly vice versa.

6. Conclusions

We tackled the deduction problem from a database view: fixpoint iteration by forward chaining from the base relations, using monotonic relational algebra operators, is an alternative to other deduction or refutation methods as, e.g., the SLD-resolution mechanism of Prolog. The following iteration paradigms were identified: systolic, pushy, sloppy, iterations, the latter with respect to best-first or easiest-first strategies. Performance reasons dictate to use Δ-iteration methods instead of bulk iteration. In detail, Δ1- and Δ2-iteration were defined; Δ1 was shown to be mostly superior to Δ2. As a sort of drawback for Δ-iteration schemes we proved that a sloppy execution for them is incorrect. On the other hand Δ-iteration schemes allow a controlled sloppy calculation w.r.t. to some best/easiest-first strategy. The method of transient partitioned iteration was introduced, concretely for Δ2, and shown to yield a correct pushy iteration. As a discovery of some theoretical interest Warshall's algorithm for transitive closure was disclosed as a transient partitioned Δ2-iteration with a quadratic fixpoint equation. All iteration schemes were implemented using efficient Δ^+-operators and performance tests were conducted with the following outcome: Fast Warshall and Δ2-iteration for the linear fixpoint equation turned out to be candidates to implement a transitive closure operator in database systems. From the discussion on an efficient implementation it became clear that this transitive closure operator can conceptually be visualized as one ***transient self-join.*** (For other proposals see, e.g., [1]).

In summary, because of its general applicability Δ-iteration is an excellent optimization technique to evaluate recursive rules in Horn clause form. As a rule of thumb, the rules should be designed such that the corresponding fixpoint equations produce small Δs. We also exemplified the application of Δ-iteration to parallel deduction. Hereby, as a further cornerstone of integrating database technology and logic programming we suggest a promising database approach to parallel deduction: synchronization enforcement by standard database concurrency control.

Acknowledgment:
We like to thank Helmut Schmidt for his precise reading of the manuscript and for suggesting several improvements.

Literature:

[1] Aiello, L.; Cecchi, C.: *Adding a Closure Operator to the Extended Relational Algebra: A Further Step towards the Integration of Data Base Techniques and Logic Programming,* Proc. Workshop on KBMS, Crete, June 1985.

[2] Aho, A.V.; Ullman, J.D.: *Universality of Data Retrieval Languages,* Proc. 6th ACM Symp. on Princ. of Progr. Lang., 1979, pp. 110 -120.

[3] Apt, K.R.; Van Emden, M.A.: *Contributions to the Theory of Logic Programming,* J. ACM, 29, 3, July 1982, pp. 841 - 862.

[4] Bayer, R.: *Query Evaluation and Recursion in Deductive Database Systems,* Techn. rep., TUM-I8503, Techn. Univ. München, Inst. für Inf., Febr. 1985.

[5] Bayer, R.: *Database Technology for Expert Systems,* in: Intern. GI-Kongress 85, Wissensbasierte Systeme, Informatik Fachberichte 112, Oct. 1985, pp. 1 - 16.

[6] Bernstein, P.A.; Goodman, N.: *Concurrency Control in Distributed Database Systems,* ACM Computing Surveys, 13, 2, June 1981, pp. 185 - 221.

[7] Bayer, R.; Heller, H.; Reiser, A.: *Parallelism and Recovery in Database Systems,* ACM ToDS, 5, 2, 1980, pp. 139 - 156.

[8] Bancilhon, F.; Kim, W.; Korth, H.: *A Model of CAD Transactions,* Proc. VLDB, Stockholm, Aug. 1985, pp. 25 - 33.

[9] Chang, C.: *On Evaluation of Queries Containing Derived Relations in a Relational Database,* in Advances in Databases, Vol. 1, Gallaire, Minker, Nicolas (eds.), 1981, pp. 235 - 260.

[10] Chandra, A.K.; Harel, D.: *Horn Clause Queries and Generalizations,* J. of Logic Prog., 1, 1, 1985, pp. 1 - 15.

[11] Clocksin,W.; Mellish, C.: *Programming in Prolog,* Berlin: Springer 1981.

[12] Codd, E.F.: *A Relational Model of Data for Large Shared Data Banks,* CACM 13, 6, pp. 377 - 387, June 1970.

[13] van Emde Boas, G. and R.: *Storing and Evaluating Horn-Clause Rules in a Relational Database,* IBM J. Res. and Dev, 30, 1, Jan. 1986, pp. 80 - 91.

[14] Gallaire, H.; Minker, J.; Nicolas, J.-M.: *Logic and Databases: A Deductive Approach:* ACM Computing Surveys, 16, 2, pp. 153 - 186, June 1984.

[15] Güntzer, U.; Bayer, R.: *Control for Iterative Evaluation of Recursive Rules in Database* Systems", Techn. rep., TUM-I8513, Techn. Univ. München, Inst. für Inf., July 1985.

[16] Güntzer, U.; Kiessling, W.; Bayer, R.: *Evaluation Paradigms for Deductive Databases: from Systolic to As-You-Please,* Techn. Rep. TUM-I8605, Techn. Univ. München, March 1986.

[17] Henschen, L.J.; Naqvi, S.A.: *Compiling Queries in Recursive First-Order Databases,* J. ACM, 31, 1, 1984, pp. 47 - 85.

[18] Jarke, M.; Koch, J.: *Query Optimization in Database Systems,* ACM Comp. Surveys, 16, 2, June 1984, pp. 111 - 152.

[19] Jarke, M.; Clifford, J.; Vassiliou, Y.: *An Optimizing PROLOG Front-end to a Relational Query System,* Proc. ACM-SIGMOD, Boston, June 1984.

[20] Jarke, M.; Linnemann, V.; Schmidt, J.W.: *Data Constructors: On the Integration of Rules and Relations,* Proc. VLDB, Stockholm, Aug. 1985, pp. 227 - 240.

[21] Kiessling, W.; Pfeiffer, H.: *A Comprehensive Analysis of Concurrency Control Performance for Centralized Databases,* 4th Intern. Workshop on Database Machines, Grand Bahama, March 1985, Springer, pp. 277 - 298.

[22] Kowalski, R.: *Predicate Logic as a Programming Language,* Proc. IFIP, Amsterdam 1974, North-Holland Publ., pp. 569 - 574.

[23] Kung, R.; et alia: *Heuristic Search in Data Base Systems,* Memo No. UCB/ERL M84/58, Univ. of Cal., Berkeley, July 1984.

[24] Lloyd, J.W.: *Foundations of Logic Programming,* Springer 1984.

[25] Linnemann, V.; Schmidt, J.W.; Jarke, M.: *Integrated Fact and Rule Management Based on Relational Technology,* Workshop on KBMS, Crete, June 1985.

[26] Reiter, R.: *Deductive Question Answering on Relational Databases,* Gallaire and Minker, Eds. Plenum, N.Y., pp. 149 - 178.

[27] Ullman, J.D.: *Implementation of Logical Query Languages for Databases,* ACM-ToDS, 10, 3, Sept. 1985, pp. 289 - 321.

[28] Van Emden, M.H.; Kowalski, R.A.: *Semantics of Predicate Logic as a Programming Language,* J. ACM, 23, 4, Oct. 1976, pp. 733 - 742.

[29] Warshall, S.: *A Theorem on Boolean Matrices,* J. ACM, 9, 1962, pp. 11 - 12.

[30] Zaniolo, C.: *The Representation and Deductive Retrieval of Complex Objects,* Proc. VLDB, Stockholm, Aug. 1985, pp. 458 - 469.

[31] Güntzer, U.; Kiessling, W.; Bayer, R.: *On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration,* Techn. Rep. TUM-I8604, Techn. Univ. München, Inst. f. Inform., 1986.

Appendix:

Table A.1: Performance results, transitive closure, times in CPU-sec.

SIZE	L-P	Q-P	Q-Δ1	L-Δ2T	L-Δ1	FW
16	2.7	3.0	2.4	1.2	1.1	0.8
17	1.7	1.9	1.4	1.1	0.9	0.7
25	3.4	4.2	3.7	1.6	1.5	1.1
60	7.9	10.5	7.4	3.4	3.8	2.4
380	229.3	>...	228.9	29.3	32.7	21.3
307	147.3	241.8	134.4	22.9	23.8	13.1
470	293.1	582.7	291.9	41.1	45.4	27.1
98	15.4	13.5	10.5	7.8	5.9	4.8
258	54.6	62.3	39.1	21.9	17.1	13.2
36	2.9	2.9	2.7	1.9	1.8	2.1
228	33.5	39.1	29.3	16.0	14.8	13.3
136	13.9	14.2	11.2	9.2	8.1	8.8
465	606.9	293.7	153.8	41.4	42.1	16.3
1225	>...	>...	>...	112.0	117.3	44.4

Table A.2:
Performance results for tuned relational algebra system.

SIZE	L-P	Q-P	Q-Δ1	L-Δ2T	L-Δ1	FW
16	0.5	0.6	0.4	0.2	0.2	0.1
17	0.3	0.4	0.2	0.1	0.1	0.1
25	0.7	0.8	0.6	0.3	0.2	0.2
60	1.4	1.8	0.9	0.6	0.5	0.5
380	49.5	92.9	44.6	8.6	5.0	9.6
307	28.3	47.6	23.3	5.5	3.2	3.9
470	58.0	120.7	51.5	11.1	5.8	9.6
98	2.5	2.5	1.3	0.9	0.6	0.5
258	8.3	11.4	4.5	3.0	1.7	1.2
36	0.5	0.6	0.3	0.2	0.2	0.2
228	4.8	8.2	3.0	1.9	1.4	0.9
136	2.2	3.6	1.3	0.7	0.7	0.5
465	109.2	55.0	23.0	8.3	4.6	4.0
1225	>...	308.4	106.4	30.1	15.0	14.1