Logic Programming

Optimization

Michael Genesereth
Computer Science Department
Stanford University
This lecture:
Materialization
Differential Update
Conceptual Reformulation of Dynamic Logic Programs
Materialization
Example

Ruleset

\[ s(X, Y, Z) : \neg r(X) \wedge r(Y) \wedge r(Z) \]
\[ r(X) \quad \neg p(X, Y) \wedge p(Y, Z) \]

Computation Cost for \( s \)

- \( r \) computed multiple times
- for \( n=2 \), database accesses = 1460
- for \( n=3 \), database accesses = 68130

where \( n \) is the number of objects

Storage for \( p \)

- \( O(n^2) \) facts stored
Ruleset

\[ s(X,Y,Z) :- r(X) \land r(Y) \land r(Z) \]
\[ r(X) :- p(X,Y) \land p(Y,Z) \]

Computation Cost for \( s \)

for \( n=2 \), database accesses = 8
for \( n=3 \), database accesses = 27

Storage for \( s \)

\( O(n^3) \) in worst case
Example with $r$ Materialized

**Ruleset**

\[
\begin{align*}
\text{s}(X,Y,Z) & :- \, \text{r}(X) \, \land \, \text{r}(Y) \, \land \, \text{r}(Z) \\
\text{r}(X) & :- \, \text{p}(X,Y) \, \land \, \text{p}(Y,Z)
\end{align*}
\]

**Computation Cost for s**

- For $n=2$, database accesses = 14
- For $n=3$, database accesses = 39

where $n$ is the number of objects

**Computation Cost for r**

- For $n=2$, database accesses = 20
- For $n=3$, database accesses = 90

**Storage for r**

$O(n)$ in worst case
Conversion process. (1) Rename in all rules defining view to be materialized and replace in all other rules. (2) Write transition rules to relate old and new name.

**Ruleset**
\[ s(X,Y,Z) :- r(X) \& r(Y) \& r(Z) \]
\[ r(X) :- p(X,Y) \& p(Y,Z) \]

**Renaming**
\[ s(X,Y,Z) :- rr(X) \& rr(Y) \& rr(Z) \]
\[ r(X) :- p(X,Y) \& p(Y,Z) \]

**Transition Rules**
\[ \text{add}(p(U,V)) \& rr(X) ==> \neg rr(X) \]
\[ \text{del}(p(U,V)) \& rr(X) ==> \neg rr(X) \]
\[ \text{add}(p(U,V)) \& r(X) ==> rr(X) \]
\[ \text{del}(p(U,V)) \& r(X) ==> rr(X) \]
Ruleset

\[
\begin{align*}
    r(X) & : = p(X,Y) \land p(Y,Z) \\
    \text{add}(p(U,V)) \land rr(X) & \implies \neg rr(X) \\
    \text{del}(p(U,V)) \land rr(X) & \implies \neg rr(X) \\
    \text{add}(p(U,V)) \land r(X) & \implies rr(X) \\
    \text{del}(p(U,V)) \land r(X) & \implies rr(X)
\end{align*}
\]

Old Base Data: \{p(a,b), p(b,a), p(c,d)\}

Old Materialized Data: \{rr(a), rr(b)\}

Changes: \{\text{del}(p(a,b)), \text{add}(p(d,c))\}

Negative Update: \{rr(a), rr(b)\}

Positive Update: \{rr(c), rr(d)\}

New Base Data: \{p(b,a), p(c,d), p(d,c)\}

New Materialized Data: \{rr(c), rr(d)\}
Differential Update
Conversion process. (1) Rename in all rules defining view to be materialized and replace in all other rules. (2) Write transition rules to relate old and new name. (3) *Remove rules defining the materialized view and replace with differential rules relating changes to base relations and changes to views.*
Ruleset

\[ s(X,Y,Z) :- r(X) & r(Y) & r(Z) \]
\[ r(X) :- p(X,Y) & p(Y,Z) \]

Differential Version

\[ s(X,Y,Z) :- rr(X) & rr(Y) & rr(Z) \]
\[ \text{add}(p(X,Y)) & p(Y,Z) & \neg\text{del}(p(Y,Z)) \implies rr(X) \]
\[ p(X,Y) & \neg\text{del}(p(X,Y)) & \text{add}(p(Y,Z)) \implies rr(X) \]
\[ \text{add}(p(X,Y)) & \text{add}(p(Y,Z)) \implies rr(X) \]

Exercise for reader: What are the deletion rules?
Ruleset

\[
\text{add}(p(X,Y)) \land p(Y,Z) \land \neg \text{del}(p(Y,Z)) \implies \text{rr}(X) \\
p(X,Y) \land \neg \text{del}(p(X,Y)) \land \text{add}(p(Y,Z)) \implies \text{rr}(X) \\
\text{add}(p(X,Y)) \land \text{add}(p(Y,Z)) \implies \text{rr}(X)
\]

Old Dataset: \(\{p(a,b), p(b,a), \text{rr}(a), \text{rr}(b)\}\)

Change Request: \(\{\text{add}(p(c,d)), \text{add}(p(d,c))\}\)

Negative Update: \(\{\}\\)

Positive Update: \(\{\text{rr}(c), \text{rr}(d)\}\)

New Dataset:

\[
\{p(a,b), p(b,a), p(c,d), p(d,c), \\
\text{rr}(a), \text{rr}(b), \text{rr}(c), \text{rr}(d)\}\]
Differential Relational Calculus for Integrity Maintenance

Levent V. Orman

Abstract — A differential calculus for first-order logic is developed to enforce database integrity. Formal differentiation of first-order sentences is useful in maintaining database integrity, since once a database constraint is expressed as a first-order sentence, its derivative with respect to a transaction provides the necessary and sufficient condition for maintaining integrity. The derivative is often much simpler to test than the original constraint since it maintains integrity differentially by assuming integrity before the transaction, and testing only for new violations. The formal differentiation requires no resolution search, but only substitution. It is more efficient than resolution-based approaches; and it provides a considerably more general solution than previous substitution-based methods since it is valid for all first-order sentences and with all transactions involving arbitrary collections of atomic changes to the database. It also produces a large number of sufficient conditions that are often less strict than those of the previous approaches; and it can be extended to accommodate many dynamic constraints.

Index Terms — Database integrity, integrity maintenance, relational database, first-order logic, formal differentiation.

1 INTRODUCTION

Database integrity constraints are logical statements that define valid states of the database. They must be obeyed by the data at all times to ensure the validity of data. They are critical in maintaining the quality of data and preventing a variety of errors and abuses. Efficient maintenance of integrity is a critical problem since testing the validity of a large number of constraints against a large database, and after each transaction that may change the database state, is an expensive undertaking. Considerable efficiency can be gained by maintaining integrity differentially, by assuming integrity before the transaction, and testing for new violations only [10], [11]. The critical problem in differential integrity maintenance is deriving the test conditions for these new violations. These conditions are often considerably easier to test than the original constraint, but they are also considerably different from the original constraint, and their derivation is not trivial, adding to the cost of integrity maintenance. Moreover, these test conditions are different for every transaction and, although they could be derived and compiled ahead of time for anticipated transactions, ad hoc transactions have even more stringent efficiency requirements—since their test conditions have to be derived in real time after the transaction is received and before it can be authorized [8], [13].

There have been two general approaches to the derivation of differential integrity test conditions. The first approach involves studying the quantifier structure of the constraint, and substituting the update values of the transaction for some constraint variables [1], [6], [10], [18]. This approach is very efficient in deriving the test conditions, but it is severely restrictive. The general problem is the difficulty of finding a procedure for analyzing the structure of arbitrarily complex constraints and arbitrarily complex transactions. Consequently, each implementation has its own severe restrictions. Some fail to derive the test conditions for constraints with complex quantifier structures or for constraints with multiple occurrences of a relation. Some find only one test condition when there are alternative conditions that may be easier to test. For complex transactions with multiple updates, this approach either fails or requires repeated application of the substitution algorithm. It also often derives conditions that need to be tested after the execution of the transaction, requiring an expensive rollback if the integrity test fails. The second approach involves resolution theorem proving where, given the constraint is satisfied before the transaction and the database state is changed by a transaction, one needs to prove that the constraint continues to be satisfied [8], [16], [19]. This approach is very general. It derives a necessary and sufficient test condition for all first-order sentences that satisfy the domain independence requirement, and it can be used with arbitrary nonatomic transactions. This approach also finds some sufficient conditions that may be easier to test; and its tests can be applied before the transaction, eliminating the need for a rollback. The disadvantages of this approach are the need for resolution search that makes it less efficient than substitution, and the need to write complex transition axioms to describe transactions. Both of these problems are partially alleviated for anticipated transactions by performing these tasks ahead of time and compiling the results into update programs; but for ad hoc transactions, real time integrity maintenance remains a problem. A similar approach was described in [15] where a differential relational algebra was developed to compute the changes to a relational algebra expression. This approach is also very general since it
**Ruleset**

\[ s(X) :\neg p(X) \& q(X) \]

**Differentials:**

\[ s^+(X) :\neg p^+(X) \& q(X) \]
\[ s^+(X) :\neg p(X) \& q^+(X) \]
\[ s^+(X) :\neg p^+(X) \& q^+(X) \]
\[ s^-(X) :\neg p^-(X) \]
\[ s^-(X) :\neg q^-(X) \]

**Active Versions:**

\[ p^+(X) \& q(X) \implies s(X) \]
\[ p(X) \& q^+(X) \implies s(X) \]
\[ p^+(X) \& q^+(X) \implies s(X) \]
\[ p^-(X) \implies \neg s(X) \]
\[ q^-(X) \implies \neg s(X) \]
Tic Tac Toe

X

O

X

cell(1,1,x)
cell(1,2,b)
cell(1,3,b)
cell(2,1,b)
cell(2,2,o)
cell(2,3,b)
cell(3,1,b)
cell(3,2,b)
cell(3,3,x)
control(o)
Non-Differential Rules

\[
\text{mark}(J,K) \& \text{cell}(M,N,R) \implies \neg \text{cell}(M,N,R)
\]

\[
\text{mark}(J,K) \& \text{control}(R) \\
\implies \text{cell}(J,K,R)
\]

\[
\text{mark}(J,K) \& \text{cell}(M,N,R) \& \text{distinct}(R,b) \\
\implies \text{cell}(M,N,R)
\]

\[
\text{mark}(J,K) \& \text{cell}(M,N,R) \& \text{distinct}(J,M) \\
\implies \text{cell}(M,N,b)
\]

\[
\text{mark}(J,K) \& \text{cell}(M,N,R) \& \text{distinct}(K,N) \\
\implies \text{cell}(M,N,b)
\]
Differential Rules

\[
\text{mark}(M,N) \land \text{control}(R) \implies \text{cell}(M,N,R)
\]

\[
\text{mark}(M,N) \implies \neg \text{cell}(M,N,b)
\]
### Computational Benefit

Number of playouts per second*

<table>
<thead>
<tr>
<th></th>
<th>Unindexed</th>
<th>Indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-differential</td>
<td>880</td>
<td>1070</td>
</tr>
<tr>
<td>Differential</td>
<td>1340</td>
<td>1780</td>
</tr>
</tbody>
</table>

* Larger is better.

---

* Larger is better.*
Conceptual Reformulation
Materialization and differential update sometimes have computational benefit.

In *some* cases, none of the views may be suitable for materialization.

However it is *sometimes* possible to rewrite programs into equivalent programs that have suitable views.
p(art, bob)
p(art, bud)
p(bob, cal)
p(bob, cam)
p(bud, coe)
p(bud, cory)

Diagram:

```
   art
  /   \\  \\
 b-- bob --c \\n     /   \\  \\
    /     \\
   cal    cam
```

```
   art
  /   \\  \\
 b-- bud --c \\n     /   \\  \\
    /     \\
   coe    cory
```
Ancestor:
\[ a(X,Y) :- p(X,Y) \]
\[ a(X,Z) :- p(X,Y) \land a(Y,Z) \]

Same family:
\[ sf(Y,Z) :- a(X,Y) \land a(X,Z) \]

Computing \( sf \) requires computation of ancestors (join of the ancestor relation with itself), which is \( O(n^2) \), where \( n \) is the number of people.
Worst case analysis for $n$ people:

- Space to store $\mathsf{sf}$ is $O(n^2)$
- Computing an instance of $\mathsf{sf}$ is still $O(n^2)$
Materialization

Worst case analysis for $n$ people:
Space to store $a$ is $O(n^2)$
Computing an instance of $sf$ is still $O(n^4)$
Reformulation

ff(art, bob)
ff(art, bud)
ff(art, cal)
ff(art, cam)
ff(art, coe)
ff(art, cory)

NB: ff is a new relation, not in the original ontology.
Forefather:

\[
\begin{align*}
\text{ff}(X,Y) & \ :- \ p(X,Y) \\
\text{ff}(X,Z) & \ :- \ p(X,Y) \ & p(Y,Z) \\
\text{ff}(X,W) & \ :- \ p(X,Y) \ & p(Y,Z) \ & p(Z,W)
\end{align*}
\]

*can also be defined recursively*

Same Family:

\[
\begin{align*}
\text{sf}(Y,Z) & \ :- \ \text{ff}(X,Y) \ & \text{ff}(X,Z)
\end{align*}
\]

Worst case analysis for \( n \) people:

Space to store \( \text{ff} \) is same as for \( p \)

Computing an instance of \( \text{sf} \) is \( O(n) \) or \( O(\log n) \)
Hex
Axiomatization

redwin :-
    left(X) & redpath(X,Y) & right(Y)

redpath(X,X) :-
    cell(X,red)

redpath(X,Z) :-
    cell(X,red) & next(X,Y) & redpath(Y,Z)

Results

Very expensive if path exists
Can run forever if not
Axiomatization

redwin :-
    left(X) & redpath(X,Y,nil) & right(Y)

redpath(X,X,nil) :-
    cell(X,red)

redpath(X,Z,P) :-
    cell(X,red) & next(X,Y) & ~member(Y,P) &
    redpath(Y,Z,Y!P)

Results
Does not run forever
Can take ~ 1 second to compute in bad cases
Result:
takes <1 millisecond to compute in worst cases
Even Better

Result:

takes <1 millisecond to compute in worst cases
Axiomatization

\[
\text{redwin} :- \\
\text{left}(X) \ & \ \text{rep}(X,K) \ & \ \text{rep}(Y,K) \ & \ \text{right}(Y)
\]

Results

takes less than 1 millisecond in the worst case
additional update cost neglible

Exercise

Write the transition rules for the better still axiomatization.