

JOIN ORDERING

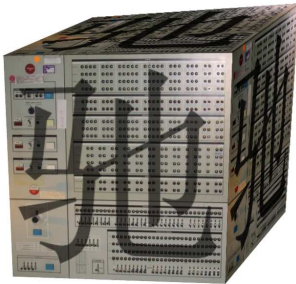
FOR CONSTRAINT HANDLING RULES

Leslie De Koninck and Jon Sneyers
K.U.Leuven, Belgium



CHR Workshop
September 2007

驰



1 Introduction

- CHR
- The problem
- Significance
- Approach

2 Cost model

- Static approximations
- Dynamic approximations

3 Optimization

- Join graphs
- Acyclic join graphs
- Cyclic join graphs

4 Conclusion

Constraint Handling Rules [Frühwirth 1991]

3/22

- ▶ High-level language extension (of Prolog/Java/C/...)
- ▶ Multi-headed committed-choice guarded rewrite rules
- ▶ Originally: designed for writing constraint solvers
- ▶ Increasingly: general-purpose programming language
- ▶ Refined operational semantics [Duck et al 2004]:
 - ▶ activate constraints depth-first, left-to-right
 - ▶ search for matching rules by trying occurrences in textual order

The join ordering problem

4/22

- ▶ *Join*: finding matching partners for a given active occurrence
 - ▶ common approach: nested loops
 - ▶ using indexes to take equality guards into account
- ▶ *Ordering*: finding a (good) order in which to do the lookups
- ▶ E.g. consider the active occurrence part/2 in the rule $\text{part}(A,I), \text{delta}(T,A,X), a(A,I,X) \setminus b(J,T) \Leftrightarrow b'(J,T)$.
3 partner constraints, so $3! = 6$ possible join orders:

<pre>foreach(delta(T,A,X)) { foreach(a(A,I,X)) { foreach(b(J,T)) { call(b'(J,T)) } } }</pre>	<pre>foreach(a(A,I,X)) { foreach(delta(T,A,X)) { foreach(b(J,T)) { call(b'(J,T)) } } }</pre>	<pre>foreach(b(J,T)) { foreach(delta(T,A,X)) { foreach(a(A,I,X)) { call(b'(J,T)) } } }</pre>	...
--	--	--	-----

The join ordering problem: significance

5/22

- ▶ Early CHR systems allowed only single and two-headed rules
⇒ at most 1 partner constraint, so nothing to order
- ▶ Hence, old CHR programs (e.g. `1eq`) use at most 2 heads
- ▶ More recent programs have more heads:

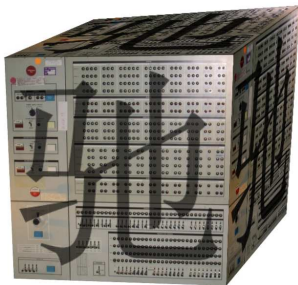
Name	1	2	3	4	5	6	Total
EU Car Rental	5	4	2	5	2		18
Hopcroft	10	9	4	1			24
Monkey & Bananas	1	7	15	2			25
RAM Simulator	1	2	3	5	2		13
Timed Automaton		11	10	3			24
Type Inference	26	48	13	6	4		97
Well-founded Semantics	3	25	8	4	1	2	43
Total	46	106	55	26	9	2	244

The join ordering problem: significance (2)

6/22

- ▶ Why care about join order?
Wrong join order often means wrong time complexity!
- ▶ Why not just leave it to the programmer? (and simply use textual order)
 - ▶ *Programmer should not worry about “low-level” details*
 - ▶ *Multi-headed rules may have many active occurrences, so there is not always a way to arrange the heads such that textual order is optimal*
 - ▶ *Sometimes the optimal order depends on dynamic properties of the store*

- ▶ Current implementations do static join ordering based on ad-hoc heuristics
- ▶ We propose a more precise cost model
- ▶ We also consider dynamic join ordering
- ▶ Typical information/time conflict:
 - ▶ Statically (at compile time) we can spend much time on finding the optimal order, but we have little information
 - ▶ Dynamically (at runtime) we have all information, but no time
- ▶ However, in some cases there is no single optimal join order, so dynamic ordering is needed to obtain correct complexity



1 Introduction

- CHR
- The problem
- Significance
- Approach

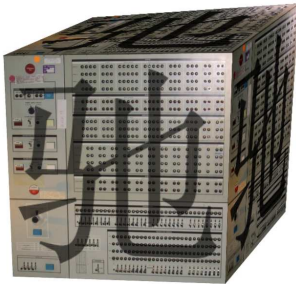
2 Cost model

- Static approximations
- Dynamic approximations

3 Optimization

- Join graphs
- Acyclic join graphs
- Cyclic join graphs

4 Conclusion



- 1 Introduction
 - CHR
 - The problem
 - Significance
 - Approach
- 2 Cost model
 - Static approximations
 - Dynamic approximations
- 3 Optimization
 - Join graphs
 - Acyclic join graphs
 - Cyclic join graphs
- 4 Conclusion

Finding a partner constraint

10/22

- ▶ Most rules have many implicit guards: e.g.
 $\text{part}(A,I), \text{delta}(T,A1,X), \text{a}(A2,I2,X2) \setminus \text{b}(J,T2)$
 $\Leftrightarrow A==A1, A==A2, I==I2, X==X2, T==T2 \mid \text{b}'(J,T).$
- ▶ Using indexes on (combinations of) constraint arguments, we find constraints satisfying these equality guards in constant time (amortized, w.h.p.).
- ▶ In general: guard $G = G_{\text{eq}} \wedge G_{\star}$, where
 - ▶ partners satisfying the a priori guard G_{eq} can be found instantly (thanks to indexing)
 - ▶ current implementations: only equality guards
 - ▶ could add e.g. comparison guards like \leq using search trees
 - ▶ to find partners satisfying the a posteriori guard G_{\star} , you need to explicitly test all candidates

Cost model

11/22

- ▶ Total cost of a join is the sum of the a priori sizes of the partial joins:

$$\sum_{j=1}^n \prod_{k=1}^j (\sigma_{\star}(k-1) \cdot \sigma_{\text{eq}}(k) \cdot \mu(k))$$

- ▶ $\sigma_{\text{eq}}(k)$ is the chance that the k -th a priori lookup succeeds (finds at least one matching partner)
- ▶ $\mu(k)$ is the average number of results for succeeding a priori lookups of the k -th partner (“multiplicity”)
- ▶ $\sigma_{\star}(k)$ is the chance that the a posteriori guard is satisfied for the k -th partner (given that the a priori guard holds)

Cost approximation

12/22

- ▶ To compute the exact values of $\sigma_{\text{eq}}(k)$, $\mu(k)$, $\sigma_{\star}(k)$, you have to do the join
 \implies much too expensive dynamically, impossible statically
- ▶ Hence: heuristics!
- ▶ Trivial bounds:

$$0 \leq \sigma_{\text{eq}}(k) \leq 1$$

$$0 \leq \sigma_{\star}(k) \leq 1$$

$$1 \leq \mu(k) \leq |\text{store of } k\text{-th head}|$$

Static approximations

13/22

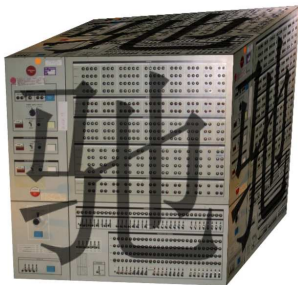
- ▶ $\sigma_{\text{eq}}(k)$: use upper bound 1
- ▶ $\mu(k)$:
 - ▶ if there are functional dependencies [Duck and Schrijvers 2005], we can statically derive $\mu(k) = 1$
 - ▶ otherwise: heuristic based on degrees of freedom
- ▶ $\sigma_*(k)$: \pm arbitrary heuristic, e.g.:
 - ▶ 1 if G_*^k is empty (true)
 - ▶ 0.5 if G_*^k is a comparison ($<$, \leq , \geq , $>$)
 - ▶ 0.25 if G_*^k is an arithmetic equality ($=:=$)
 - ▶ 0.95 if G_*^k is an inequality ($=\backslash=$, $\backslash==$)
 - ▶ 0.75 otherwise
 - ▶ if G_*^k is a conjunction, multiply these numbers

Dynamic approximations

14/22

- ▶ Worst-case bounds:
 - ▶ $\sigma_{\text{eq}}(k)$, $\sigma_*(k)$: use upper bound 1
 - ▶ $\mu(k)$: maintain the maximal number of results per lookup key (e.g. for hash tables: the maximal bucket size)
- ▶ Approximations:
 - ▶ $\mu(k)$: maintain the average number of results per lookup key
 - ▶ $\sigma_*(k)$: as in static case, or maintain success rate of a posteriori guards
 - ▶ $\sigma_{\text{eq}}(k)$: maintain number of distinct keys and size of key domain; the ratio of these numbers is a reasonable estimate assuming keys are randomly sampled
- ▶ Hybrid approach: optimize weighted sum, e.g.

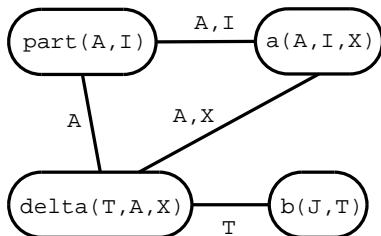
$$[\text{approximation}] + 0.05[\text{worst-case bound}]$$



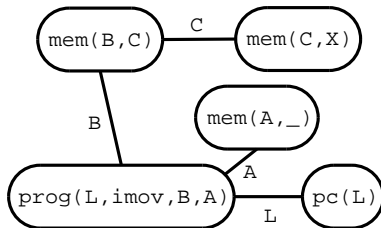
- 1 Introduction
 - CHR
 - The problem
 - Significance
 - Approach
- 2 Cost model
 - Static approximations
 - Dynamic approximations
- 3 **Optimization**
 - Join graphs
 - Acyclic join graphs
 - Cyclic join graphs
- 4 Conclusion

Join graphs: examples

`part(A,I), delta(T,A,X),
 a(A,I,X) \ b(J,T)
 <=> b'(J,T).`



`prog(L,imov,B,A),
 mem(B,C), mem(C,X)
 \ mem(A,_), pc(L)
 <=> mem(A,X), pc(L+1).`



Acyclic join graphs

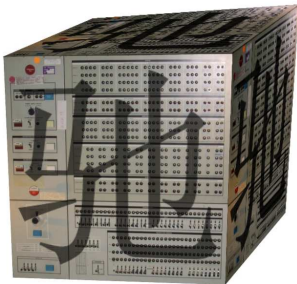
17/22

- ▶ There is an $O(n \log n)$ algorithm [Krishnamurthy et al, 1986] to find the optimal join order, under these assumptions:
 - ▶ The join graph is acyclic, so we can consider it to be a tree rooted in the active constraint
 - ▶ The optimal order respects the tree order (lookup parent before child)
 - ▶ Representative selection

Cyclic join graphs

18/22

- ▶ For general join graphs, the optimization problem is NP-complete [Ibaraki and Kameda, 1984]
- ▶ Still, n is usually small (< 10), so exponential dynamic programming algorithms may be OK
- ▶ Could also construct (multiple) spanning tree(s) and run the acyclic algorithm
- ▶ Some special cases of cyclicity (e.g. some cliques) can be eliminated
 - ▶ See paper
 - ▶ 39 out of 49 cyclic join graphs could be made acyclic



- 1 Introduction
 - CHR
 - The problem
 - Significance
 - Approach
- 2 Cost model
 - Static approximations
 - Dynamic approximations
- 3 Optimization
 - Join graphs
 - Acyclic join graphs
 - Cyclic join graphs
- 4 Conclusion

- ▶ More realistic cost model
- ▶ Also consider runtime information; dynamic join ordering
- ▶ Static and dynamic cost approximations
- ▶ Efficient join order optimization algorithm for acyclic graphs (ported from the database literature to CHR)
- ▶ Elimination of some types of cycles in join graphs
- ▶ First-few answers (simplification rules) vs all answers (propagation rules) (see paper)

- ▶ Theorems & proofs
- ▶ Implementation, experimental evaluation
- ▶ Join ordering in parallel
- ▶ Hybrid between static and dynamic
- ▶ Trade-off between optimization cost and join cost
- ▶ Profiling for static ordering
- ▶ Other ways to eliminate cycles in join graphs

▶ *Questions?*