Join Ordering for Constraint Handling Rules

Leslie De Koninck* and Jon Sneyers*

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract. Join ordering is the problem of finding cost optimal execution plans for matching multi-headed rules. In the context of Constraint Handling Rules, this topic has received limited attention so far, even though it is of great importance for efficient CHR execution. We present a formal cost model for joins and investigate the possibility of join optimization at runtime. We propose some heuristic approximations of the parameters of this cost model, for both the static and dynamic case. We discuss an $\mathcal{O}(n \log n)$ optimization algorithm for the special case of acyclic join graphs. However, in general, join order optimization is an NP-complete problem. Finally, we identify some classes of cyclic join graphs that can be reduced to acyclic ones.

1 Introduction

Constraint Handling Rules (CHR) [4] is a high-level language extension based on multi-headed guarded committed-choice rewrite rules. While originally designed for the implementation of constraint solvers, CHR is increasingly used as a general purpose programming language. Much work has been devoted to the optimized compilation of CHR [1, 5, 10]. A crucial aspect of CHR compilation is finding matching rules efficiently. Given an active constraint, searching for matching partner constraints corresponds to joining relations — a well-studied topic in the context of databases [8, 9, 12–14]. The performance of join methods depends on indexing and join ordering.

In the context of CHR, join ordering has been discussed in [1, 5]. In that work, only static (compile-time) information is used in determining the optimal join order. Moreover, little attention is given to the complexity of the optimization algorithm, since join ordering is done at compile time and because the input sizes (the number of heads of a rule) are expected to be very small ("usually at most 3 in practice" [1]). In this paper we consider dynamic (run-time) join ordering based on constraint store statistics such as selectivities and cardinalities. We also take into account that CHR programs increasingly contain rules with more than three heads. (Some examples of such programs are given in Section 6.2.)

The main contributions of this paper are the following. We formulate a generic cost model for join ordering, which provides a solid foundation to develop

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

and evaluate heuristics. We introduce dynamic join ordering. Finally we discuss an efficient $\mathcal{O}(n \log n)$ join order optimization algorithm for acyclic join graphs.

Example 1. Consider the following rule from a CHR implementation[1]of an algorithm by Hopcroft [6] for minimizing states in a finite automaton:

```
partition(A,I), delta(T,A,X), a(A,I,X) \ b(J,T) <=> b_prime(J,T).
```

In this rule, when `partition(A,I)` is activated, in order to achieve the correct time complexity, `A` and `I` should be used to find matching `a(A,I,X)` constraints; then `A` and `X` can be used to find matching `delta(T,A,X)` constraints; the resulting `T` can finally be used to find a matching `b(J,T)` constraint. However, the K.U.Leuven CHR system currently uses `A` to find `delta(T,A,X)`, then `T` to find `b(J,T)`, and finally `A`, `I` and `X` to find `a(A,I,X)`. As a result, the optimal time complexity is not obtained. The heuristic introduced in [5] also results in a suboptimal join order if the functional dependencies $\texttt{delta}(T, A, X) :: \{T, A\} \rightsquigarrow \{X\}$ and $\texttt{b}(J, T) :: \{T\} \rightsquigarrow \{J\}$ are taken into account. This rule and many other rules for which current systems use a suboptimal join order motivate a closer investigation of the join ordering problem for CHR. □

This paper assumes a strong familiarity with CHR [4], the refined operational semantics ω_r [3], and optimizing CHR compilation [1, 10].

The rest of this paper is organized as follows. Section 2 gives a high level discussion of our approach. In Section 3, we give a generic cost model for computing a join, whose parameters are estimated in Section 4. Next, in Section 5, alternative methods for computing the optimal join order are investigated. Section 6 discusses some remaining issues and we conclude in Section 7.

2 Approach

2.1 Processing Trees

A join execution plan can be represented by a binary tree, called the processing tree. The leaves of a processing tree represent the different heads to be joined and its nodes represent the (partial) join of two or more heads. One particularly interesting type of processing trees are the so-called *left-deep* trees which are used by most database systems, as well as current CHR implementations. A left-deep processing tree is inductively defined as follows: a single head is a left-deep tree, and a tree in which the root node has a left-deep tree as its left child and a head as its right child is also a left-deep processing tree.

A join proceeds by traversing all tuples of the left child of the join node and looking up appropriate tuples of the right child for each of them. This is often called a *nested loop* join. It supports pipelining which means that intermediate results do not need to be stored. In this paper, we assume that partial joins are not indexed, i.e., if they are temporarily stored, we can only traverse them by

[1] All example programs can be found at `http://www.cs.kuleuven.be/~leslie/join/`.

sequential search. This implies that a *right-deep* processing tree[2] is problematic, since we have no indexing support to perform the nested loop joins.

The most general type of processing trees are called *bushy trees*. Both the left and right child of a bushy tree node can be a (non-trivial) processing tree itself. In general, bushy trees do not support pipelining and also using indexes is often a problem. However, there are cases in which these disadvantages are outweighed by reduced intermediate result sizes.

Example 2. Consider the rule

```
a(X), b(X,Y), c(Y,Z), d(Z,A), e(A) <=> ...
```

Assume the first head is the active constraint, there are many combinations of the second and third head resulting in the same value for `Z` and there are many `d/2` constraints for any value of its first argument, though few for which a matching `e/1` constraint exists. Clearly, we can waste a lot of effort recomputing the join of `d(Z,A)` and `e(A)` for each combination of `b(X,Y)` and `c(Y,Z)`. In this case it might be better to compute the join of `d(Z,A)` and `e(A)` once and store it for later retrieval. □

In the remainder of this paper, we only consider left-deep processing trees.

2.2 Indexing

When joining a partial match with a head, we can use indexes on constraint arguments for faster retrieval. When no indexes are available, we are forced to consider each instance of a constraint in the store. Modern CHR implementations employ various indexing schemes. We describe the indexing structures offered by the K.U.Leuven CHR system [11] which covers most of the indexing in other implementations as well. The K.U.Leuven CHR system offers the following indexing structures:

- Hash table lookup for ground argument positions
- Array lookup for dense integer arguments (the same complexity as hash tables, but lower constant factors)
- Attributed variables indexing: allows quick lookup of all constraints containing a particular variable

In order to make indexing more uniform and of optimal complexity, we have extended the CHR system with hash table support for non-ground terms. It supports constraint insertion and deletion in $\mathcal{O}(1)$ amortized average-case time, $\mathcal{O}(1)$ lookup of all constraints with a given key, and $\mathcal{O}(n)$ update cost after unification where n is the number of affected constraints. The indexing structure allows for constant time lookup, regardless of the instantiation of the variables. However, we cannot use this indexing structures for lookups via partially instantiated *arguments*, e.g.

[2] A right-deep processing tree is similar to a left-deep one, but with the left and right child switched.

```
a(X,Y), b(f(_,X,_),Y) ==> ...
```

Other types of indexes could be proposed, for example search trees for inequality guards ($<$, $>$, $\leq$, $\geq$). Also other often used (explicit) equality guards could be supported, for example for patterns like

```
a(X,Y), b(X,Y1) <=> Y1 =:= Y - 1 | ...
```

We do not consider such indexing structures as they are currently not supported by CHR implementations. Therefore, for guards other than implicit equality guards, we filter out non-matching heads *after* retrieval.

2.3 Static and Dynamic Join Ordering

Current CHR implementations attempt to determine an optimal join order (if they optimize at all) at compile time, after analyses such as functional dependency analysis [2]. The join order therefore remains fixed during execution. We call this *static* join ordering. Next to static join ordering, we also consider join ordering at runtime: *dynamic* join ordering. In the dynamic case, we can use runtime information such as constraint cardinalities and selectivities. However, at runtime we cannot afford to spend as much time on optimization as at compile time.

Example 3. To show the advantages of dynamic join ordering, consider the rule:

```
supports(A,chelsea), supports(B,liverpool) \ friend(A,B) <=> true.
```

Let there be n `supports`/2 constraints, $\sqrt{n}$ of which have `liverpool` as their second argument. Let there be a functional dependency from the first argument of `supports`/2 to the second. We now consider two cases. In the first case, there are n `friend`/2 constraints for every value of the first argument. In the second case, there are only a small constant number (c) of `friend`/2 constraints for each such value. Each person initially has a representative number of friends supporting Liverpool, i.e., $\sqrt{n}$ in the first case and $\sqrt{c}$ in the second case.

Let the first head be active. We try two join orders: in Θ_1, the second head is retrieved before the third, and in Θ_2, the third head is retrieved before the second. Using the cost formula of Section 3, we find the following costs:

Case	C_{Θ_1}	C_{Θ_2}
1	$\sqrt{n} + \sqrt{n}$	$n + \sqrt{n}$
2	$\sqrt{n} + \sqrt{c}$	$c + \sqrt{c}$

Clearly, the wrong join order leads to a suboptimal complexity: $\Theta(n)$ versus $\Theta(\sqrt{n})$ in the first case and $\Theta(\sqrt{n})$ versus $\Theta(1)$ in the second. Moreover, the optimal join order depends on runtime statistics that could change during execution, so no static order can be optimal. □

Dynamic join ordering still allows for a broad range of design choices. A join order should be available at the moment an active occurrence starts looking for partner

constraints. However, after firing a first rule instance, runtime statistics may change and a different join order may become optimal. If we change the join order in between rule firings for the same active occurrence, we may no longer be able to use the partial joins that have already been computed, which probably outweighs the advantage of a better join order. Another issue is whether an optimal join order should be recomputed each time an occurrence becomes active. Clearly, a previously optimal join order remains optimal as long as runtime statistics do not considerably change, and so we can store it for later reuse.

3 Cost Model

We now introduce a cost model to estimate the join cost of a particular join order. Our cost model assumes that rules are propagation rules with bodies that do not fundamentally change the relevant statistics, similar to what is assumed in the context of join order optimization for database systems. In the case of simplification or simpagation rules, it seems advantageous to optimize for finding the first full matches. In [8], this topic is handled by taking into account the probability that a given partial match cannot be extended into a full match. We return to this issue in Section 6.1.

3.1 Notation

Consider given an active identified constraint a, trying occurrence H_0 of a rule with n remaining heads: $H_1, \ldots, H_n$. A join order Θ is a permutation of $\{0, 1, \ldots, n\}$ such that $\Theta(0) = 0$. The matching is done by nested partner lookups in the order $\Theta(1), \Theta(2), \ldots, \Theta(n)$. We use the notation $\mathsf{st}(H_i)$ to denote all constraints (with identifier) in the CHR store which have the same constraint predicate as H_i. We use $\bar{h}$ to denote a tuple and h_i to denote its i^{th} element (starting from 0). We use $\bar{\exists} F$ to denote the existential closure of a formula F. Suppose the rule has a guard G, which determines which elements of $\mathsf{st}(H_1) \times \ldots \times \mathsf{st}(H_n)$ are part of the join (we consider both the implicit and the explicit guard as part of G). Without loss of generality, this guard can be considered as a conjunction $g_1, \ldots, g_m$, where each conjunct g_j is a host-language constraint on a subset of the head variables. We define the guard scheduling index $\mathsf{gs\_index}(g_j)$ to be the earliest possible position in the join computation after which g_j can be evaluated:

$$\mathsf{gs\_index}(g_j) = \min\{k \in \{0, \ldots, n\} \mid \mathsf{vars}(g_j) \subseteq \bigcup_{i=0}^{k} \mathsf{vars}(H_{\Theta(i)})\}$$

If we have mode declarations for g_j, we can further refine this definition to consider only its input variables (cf. [5]). We use G^k to denote the part of G that can be evaluated after the k^{th} lookup:

$$G^k = \bigwedge\{g_j \mid 1 \leq j \leq m \wedge \mathsf{gs\_index}(g_j) = k\}$$

We distinguish between two types of host-language constraints in the guard: the first type are equality guards, the second type are other guards. The difference between both types of guards is that the first type can be evaluated in an *a priori* way: the equality can be used to do a hash-table lookup which means that partner constraints that do not satisfy the equality are not even considered. Constraints of the second type can only be evaluated in an *a posteriori* way: we consider all candidate partner constraints and check whether the guard holds. We denote the type-1 guards of G^k with G^k_{eq} and the type-2 guards with $G^k_\star$. We denote the built-in store with $\mathbb{B}$ and the built-in constraint theory with $\mathcal{D}$.

3.2 Partial Joins

Definition 1. *Given a built-in store $\mathbb{B}$, a CHR store described by $\mathsf{st}(H_i)$, a join order Θ, and an active constraint a, we define $\mathcal{J}^k_\Theta$, the partial join up to the k^{th} partner, as follows: if $\mathcal{D} \not\models \mathbb{B} \rightarrow \bar{\exists}\big(G^0 \wedge H_0 = a\big)$, then $\mathcal{J}^k_\Theta = \emptyset$ for all k. Otherwise, $\mathcal{J}^k_\Theta$ is defined inductively: $\mathcal{J}^0_\Theta = \{a\}$ and $\mathcal{J}^k_\Theta = \mathcal{J}^{k-1}_\Theta \bowtie H_{\Theta(k)}$, where*

$$\mathcal{J}^{k-1}_\Theta \bowtie H_{\Theta(k)} = \left\{ \bar{h} \in \mathcal{J}^{k-1}_\Theta \times \mathsf{st}(H_{\Theta(k)}) \;\middle|\; \mathcal{D} \models \mathbb{B} \rightarrow \bar{\exists}\left(G^k \wedge \bigwedge_{i=0}^{k} h_i = H_{\Theta(i)} \right) \right\}$$

The full join $\mathcal{J}^n_\Theta$ corresponds to the set of all tuples of matching partner constraints, and does not depend on the join order Θ. We use $\mathcal{E}^k_\Theta \supseteq \mathcal{J}^k_\Theta$ to denote the partial join where in the last lookup, the condition is weakened to only the type-1 part of the guard:

$$\mathcal{E}^k_\Theta = \left\{ \bar{h} \in \mathcal{J}^{k-1}_\Theta \times \mathsf{st}(H_{\Theta(k)}) \;\middle|\; \mathcal{D} \models \mathbb{B} \rightarrow \bar{\exists}\left(G^k_{\text{eq}} \wedge \bigwedge_{i=0}^{k} h_i = H_{\Theta(i)} \right) \right\}$$

3.3 Cost Formula

The total cost for performing a join according to a join order Θ depends on the sizes of the intermediate partial joins $\mathcal{J}^k_\Theta$. The size $\left|\mathcal{J}^k_\Theta\right|$ of the partial join $\mathcal{J}^k_\Theta$ can be written as

$$\left|\mathcal{J}^k_\Theta\right| = \left|\mathcal{J}^{k-1}_\Theta\right| \cdot \sigma_{\text{eq}}(k) \cdot \mu(k) \cdot \sigma_\star(k)$$

where

$$\sigma_{\text{eq}}(k) = \left|\mathcal{A}^k_\Theta\right| \;/\; \left|\mathcal{J}^{k-1}_\Theta\right|$$
$$\mu(k) = \left|\mathcal{E}^k_\Theta\right| \;/\; \left|\mathcal{A}^k_\Theta\right|$$
$$\sigma_\star(k) = \left|\mathcal{J}^k_\Theta\right| \;/\; \left|\mathcal{E}^k_\Theta\right|$$

and

$$\mathcal{A}^k_\Theta = \left\{ \bar{h} \in \mathcal{J}^{k-1}_\Theta \mid \exists h_k \in \mathsf{st}(H_{\Theta(k)}) : (\bar{h}, h_k) \in \mathcal{E}^k_\Theta \right\}$$

Intuitively, $\sigma_{\text{eq}}(k)$ represents the percentage of tuples of the partial join $\mathcal{J}^{k-1}_\Theta$ for which the *a priori* lookup of the next partner constraint is successful, that

is, at least one constraint exists in $\mathsf{st}(H_{\Theta(k)})$ that satisfies the equality guard. The intuitive meaning of $\mu(k)$ is the average multiplicity of *a priori* lookups of the next partner constraint, that is, the number of constraints from $\mathsf{st}(H_{\Theta(k)})$ that correspond to a given tuple from $\mathcal{J}_\Theta^{k-1}$ with respect to G^k_{eq}, averaged over all tuples $j \in \mathcal{J}_\Theta^{k-1}$ for which at least one such constraint exists. Finally, $\sigma_\star(k)$ intuitively corresponds to the percentage of tuples from $\mathcal{J}_\Theta^{k-1} \times H_k$ that satisfy the remaining guard $G^k_\star$ given that the equality guard G^k_{eq} is satisfied.

Assuming that finding the set of partners that satisfy the *a priori* guard can be done in constant time, and assuming that evaluating the *a posteriori* guard also takes only constant time per tuple, the total cost of joining n heads using the sequence of joins $(((H_{\Theta(1)} \bowtie H_{\Theta(2)}) \bowtie H_{\Theta(3)}) \dots \bowtie H_{\Theta(n)})$ is proportional to $C_\Theta^{[1..n]}$, the sum of the *a priori* sizes of all partial joins:

$$C_\Theta^{[1..n]} = \sum_{j=1}^{n} \frac{|\mathcal{J}_\Theta^j|}{\sigma_\star(j)} = \sum_{j=1}^{n} \prod_{k=1}^{j} \big(\sigma_\star(k-1) \cdot \sigma_{\text{eq}}(k) \cdot \mu(k)\big)$$

Clearly, different join orders have a different cost. It is our objective to find the join order with minimal cost.

4 Approximating Costs

In this section we propose static and dynamic approximations of the cost model that was defined in the previous section.

4.1 Static Cost Approximations

Note that the following trivial inequalities always hold:

$$0 \leq \frac{\sigma_{\text{eq}}(k)}{\sigma_\star(k)} \leq 1 \leq \mu(k) \leq \big|\mathsf{st}(H_{\Theta(k)})\big|$$

In the static approach we use the upper bound 1 as an estimate for $\sigma_{\text{eq}}(k)$.

Estimating $\mu(k)$. CHR constraints have multiset semantics in general, but in practice constraints have set semantics or at least multiplicities that are bounded by a small constant. If we can derive a functional dependency for $H_{\Theta(k)}$ given the variables of the previous heads $\bigcup_{i=0}^{k-1} \mathsf{vars}(H_{\Theta(i)})$ and the equality guard G^k_{eq}, then we know that there can be at most one $H_{\Theta(k)}$ constraint for a given tuple of $\mathcal{J}_\Theta^{k-1}$. So in that case, $\mu(k) = 1$. Similarly, if we can derive a generalized functional dependency with multiplicity m, we get $1 \leq \mu(k) \leq m$ so we can estimate $\mu(k)$ to be in the middle of that interval: $\frac{m+1}{2}$. In general we can use a heuristic based on degrees of freedom, but we have no space to explain it here.

Estimating $\sigma_\star(k)$. Let $G^k_\star = g_1, \ldots, g_m$. As an estimate for $\sigma_\star(k)$, we use the following product: $\sigma_\star(g_1) \cdot \ldots \cdot \sigma_\star(g_m)$, where $\sigma_\star(g_i) = 0.5$ if g_i is of the form `_<_`, `_>_`, `_=<_` or `_>=_`; $\sigma_\star(g_i) = 0.25$ if g_i is of the form `_=:=_` or `_ is _`; $\sigma_\star(g_i) = 0.95$ if g_i is of the form `_=\=_` or `_\==_`; and $\sigma_\star(g_i) = 0.75$ otherwise. Of course this heuristic is very arbitrary and ad-hoc, and it can be extended or modified. The idea is that disequalities usually hold, (arithmetic) equalities usually do not, and less-than(-or-equal) usually holds in half of the cases. Unknown guards are assumed to hold in 75% of the cases, and guards in conjunction are treated as if they independently filter the valid partner constraints.

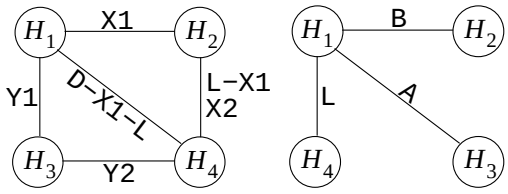
4.2 Dynamic Cost Approximations

In the dynamic approach, we maintain statistics about the constraint store and use them to find upper bounds and approximations of the cost of join orders.

Worst-case bounds. In the worst case, the trivial upper bound of 1 is reached for both $\sigma_{\text{eq}}(k)$ and $\sigma_\star(k)$. For $\mu(k)$ we can obtain a tighter upper bound by maintaining, for every data structure which is used for constraint lookup, the maximal number of constraints per lookup key — e.g., for hash-tables this corresponds to the maximal bucket size. The equality guard G^k_{eq} determines the data structure that will be used for the lookup. The maximal number of constraints per key of that data structure is hence an upper bound for $\mu(k)$.

Approximations. Instead of maintaining the maximal number of constraints per key, we can also easily maintain the average number of constraints per key. This average is a good approximation of $\mu(k)$ if we assume that the effectively used lookup keys are representative. As an approximation of $\sigma_\star(k)$ we can either use the same heuristic as in the static case, or estimate the value by dynamically maintaining the success rate of the type-2 guard. Finally, we approximate $\sigma_{\text{eq}}(k)$ as follows. We maintain the total number of distinct keys in every constraint data structure, and we compute the size of the key domain, as follows: for every type used in constraint arguments, we maintain the type domain size as the number of distinct (ground or nonground) values that were effectively encountered in arguments of that type. The size of a single-argument key domain is simply the domain size of its type. For multi-argument keys we use the product of the corresponding type domain sizes. In a sense, $\sigma_{\text{eq}}(k)$ represents the chance that a key exists (i.e., lookup on a key is non-empty). We approximate it by dividing the number of keys by the size of the key domain. This is reasonable assuming that the keys tried are randomly sampled from the key domain.

Hybrid heuristic. In practice, it makes sense to use a weighted sum of the above approximation and the worst-case bound. If the approximation results in a cost C_a and the worst-case bound results in a cost C_w, we may want to optimize $(1 - \alpha) \cdot C_a + \alpha \cdot C_w$, where α is some small constant, e.g., $\alpha = 0.05$.



(a) Cyclic (b) Acyclic

Fig. 1. Cyclic and Acyclic Join Graphs

5 Optimization

This section discusses algorithms that can be used to determine the optimal join order given our cost model and approximations of its parameters. Section 5.1 introduces the notion of join graphs. In Section 5.2, an $\mathcal{O}(n \log n)$ algorithm for the special case of acyclic join graphs is presented (where n is the number of heads). Section 5.3 deals with the general case of cyclic join graphs.

5.1 Join Graphs

The join graph for a given rule is constructed as follows: the vertices correspond to the heads of the rule and the edges link heads whose variables are connected via guards.[3] Join graphs are cyclic or acyclic.

Example 4. Figure 1 shows an example of both a cyclic and an acyclic join graph. The cyclic join graph in Figure 1(a) corresponds to the following rule (from the timed automaton program):

```
dist(X1,Y1,D), fincl(X2,X1), fincl(Y2,Y1) \ fdist_init(X2,Y2,L)
    <=> \+ memberchk_eq(D-X1,L) | ...
```

Figure 1(b) shows the acyclic join graph corresponding to the following rule (from the RAM simulator program):

```
prog(L,move,B,A), mem(B,X) \ mem(A,_), prog_counter(L) <=> ...
```

The head indices refer to the textual order of the heads in the rules. □

5.2 Acyclic Join Graphs

In this section, we show how the optimal join ordering can be found in $\mathcal{O}(n \log n)$ time for n-headed rules with an acyclic join graph. This result is based on results from database query optimization . An acyclic join graph can be represented as a tree with the active constraint as its root node. The algorithm proposed in this section requires that the following assumptions hold:

[3] This includes implicit equality guards.

1. The optimal join order respects the tree order, i.e., no head is looked up before its parent in the join tree.
2. All selections are representative: given a join and a head H extending the join, then adding or removing other heads to/from the join does not influence the selectivity and multiplicity for H.

Given a join order Θ and assume that for given j, join order Θ' is cheaper, where Θ' equals Θ except that the j^{th} and $(j+1)^{th}$ elements are switched, i.e., $\Theta' = \Theta \circ (j, j+1)$. Since under our assumptions, $\sigma_\star(i)$, $\sigma_{\text{eq}}(i)$ and $\mu(i)$ only depend on the parent of the i^{th} head in the join tree, we have that

$$\begin{aligned} C_{\Theta'} - C_\Theta = C_\Theta^{[1..j-1]} \cdot (&\sigma_\star(j-1) \cdot \rho(j+1) \cdot (1 + \sigma_\star(j+1) \cdot \rho(j)) \\ &- \sigma_\star(j-1) \cdot \rho(j) \cdot (1 + \sigma_\star(j) \cdot \rho(j+1))) \leq 0 \end{aligned}$$

where $\rho(i) = \sigma_{\text{eq}}(i) \cdot \mu(i)$ and $C_\Theta = C_\Theta^{[1..n]}$. We further simplify this as follows:

$$\rho(j+1) + \rho(j) \cdot \rho(j+1) \cdot \sigma_\star(j+1) \leq \rho(j) + \rho(j+1) \cdot \rho(j) \cdot \sigma_\star(j)$$

$$\rho(j+1) \cdot (1 - \rho(j) \cdot \sigma_\star(j)) \leq \rho(j) \cdot (1 - \rho(j+1) \cdot \sigma_\star(j+1))$$

Because $\rho(i) \geq 0$ for any i:

$$\frac{\rho(j+1) \cdot \sigma_\star(j+1) - 1}{\rho(j+1)} \leq \frac{\rho(j) \cdot \sigma_\star(j) - 1}{\rho(j)}$$

We expand the comparison measure towards sequences: sequences s and t can be switched if and only if $(\mathcal{J}_\Theta = \mathcal{J}_\Theta^n)$

$$C_s + |\mathcal{J}_s| \cdot C_t \leq C_t + |\mathcal{J}_t| \cdot C_s$$

or, expressed in terms of *ranks* of sequences (as in [7, 9]):

$$\text{rank}(s) = \frac{|\mathcal{J}_s| - 1}{C_s} \leq \frac{|\mathcal{J}_t| - 1}{C_t} = \text{rank}(t)$$

Now, consider a head H_i whose direct parent H has a higher rank. We can show that only descendants of this parent are allowed between H and H_i. Assume that another head H_j appears in between H and H_i in the optimal order. Since H_j is not a descendant of H, it can be placed before H. Assume it has a lower rank than H, then we can move it from in between H and H_i to before H and the resulting cost is smaller. If it has a higher rank than H, then it also has a higher rank than H_i and it can be moved to after H_i, resulting again in a smaller cost. In conclusion, if a head has lower rank than its parent, then only descendants of this parent, can be in between the head and its parent in the optimal join order.

Below is the algorithm as given in [9], which outputs a single chain representing the optimal join order.

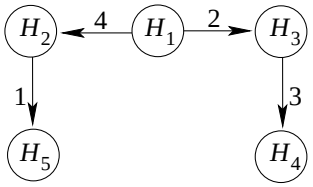
1. If the tree is a single chain then stop.
2. Find a subtree, rooted at r, all of whose children are chains.

3. Merge the chains based on ranks such that the resulting single chain is nondecreasing on the rank.
4. Normalize the root r of the subtree as follows: while the rank of the root r is greater than its immediate child c: replace r and c by a new node representing the subchain r followed by c.
5. Go to 1.

Example 5. Consider the following rule where the first head is active:

```
h1(X1,Y1), h2(X1,X2), h3(Y1,Y2), h4(Y2), h5(X2) ==> ...
```

It leads to the (directed) join graph below, in which an edge from to H_i is annotated with $\sigma_{\text{eq}}(H_i) \cdot \mu(H_i)$:



Head	$\lvert\mathcal{J}\rvert$	C	Rank
H_2	4	4	3/4
H_3	2	2	1/2
H_4	3	3	2/3
H_5	1	1	0

Assume that we start with the textual order as initial join order:

$$C_{[1,2,3,4,5]} = 4 + 4 \cdot 2 + 4 \cdot 2 \cdot 3 + 4 \cdot 2 \cdot 3 \cdot 1 = 60$$

Since $\text{rank}(H_5) < \text{rank}(H_2)$, but H_2 should be before H_5 according to the tree order, we have that no heads should be scheduled in between H_2 and H_5. Indeed, since $\text{rank}(H_3) < \text{rank}(H_2)$ it is cheaper to put H_3 before H_2:

$$C_{[1,3,2,4,5]} = 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 3 + 2 \cdot 4 \cdot 3 \cdot 1 = 58$$

and similarly for H_4 and H_2:

$$C_{[1,3,4,2,5]} = 2 + 2 \cdot 3 + 2 \cdot 3 \cdot 4 + 2 \cdot 3 \cdot 4 \cdot 1 = 56$$

Now H_2 is directly before H_5 and we combine both into a new node with $\text{rank}([H_2, H_5]) = 3/8$. For the new node, we have that $\text{rank}([H_2, H_5]) < \text{rank}(H_4)$ and so we can switch them:

$$C_{[1,3,2,5,4]} = 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 1 + 2 \cdot 4 \cdot 1 \cdot 3 = 42$$

and similarly for $[H_2, H_5]$ and H_3:

$$C_{[1,2,5,3,4]} = 4 + 4 \cdot 1 + 4 \cdot 1 \cdot 2 + 4 \cdot 1 \cdot 2 \cdot 3 = 40$$

Now all heads (sequences) are sorted by rank and the join order is optimal. □

5.3 Cyclic Join Graphs

While we have an efficient algorithm for finding the optimal join order for acyclic join graphs, the same optimization problem for general graphs has been proven to be NP-complete [7]. The standard approach consists of applying an exponential algorithm based on dynamic programming. Alternatives include applying the algorithm for acyclic join graphs to various spanning trees of the join graph [9, 14], and using randomized algorithms like simulated annealing and genetic algorithms [13]. Due to space considerations, we do not go into detail here.

6 Discussion

6.1 First-Few Answers

As already pointed out in Section 3, our cost model is based on the assumption that all fireable rule instances eventually fire. In many cases, this is obviously not true. Consider for example a simplification or simpagation rule in which the active constraint is removed. Clearly, only the first rule instance found will fire. Therefore, it can be advantageous to optimize for finding the first answer. In this section, we give an alternative cost model, which represents the cost for finding the first full match. This model is based on [8].

Let $p(i)$ denote the probability that a tuple from the partial join up to the i^{th} head (in some order Θ) can be extended to a full join tuple. We recursively define $p(i)$ as follows:

$$p(i) = \sigma_{\text{eq}}(i) \cdot (1 - (1 - p(i+1))^{\mu(i) \cdot \sigma_\star(i)})$$

It is the probability that the tuple can be extended by at least one head ($\sigma_{\text{eq}}(i)$), times the probability that at least one of the tuples of $i+1$ heads survives (i.e., one minus the probability that all fail). The total cost of finding the first answer can then be written as follows:

$$C = \sum_{i=1}^{n} \frac{1}{p(i)}$$

Here we assume that $1/p(i)$ never exceeds $|J_i|$. The reasoning is as follows: $p(i) \cdot |J_i|$ tuples out of $|J_i|$ will participate in a full match. Under the assumption that these tuples are uniformly distributed amongst all tuples, we will need to skip $1/p(i)$ tuples before finding the first tuple that takes part in a full match.

For more general types of rules, we can make a weighted sum of the cost model introduced here, and the one from Section 3.

6.2 Cyclic and Acyclic Join Graphs

The table below shows for some example programs the number of n-headed rules ($n \in [1..6]$) and in between brackets the number of rules with cyclic join graphs.

Name	1	2	3	4	5	6	Total
EU Car Rental	5	4	2 (0)	5 (4)	2 (2)		18 (6)
Hopcroft	10	9	4 (1)	1 (1)			24 (2)
Monkey & Bananas	1	7	15 (7)	2 (0)			25 (7)
RAM Simulator	1	2	3 (0)	5 (0)	2 (0)		13 (0)
Timed Automaton		11	10 (9)	3 (3)			24 (12)
Type Inference	26	48	13 (7)	6 (5)	4 (4)		97 (16)
Well-founded Semantics	3	25	8 (1)	4 (2)	1 (1)	2 (2)	43 (6)
Total	46	106	55 (25)	26 (15)	9 (7)	2 (2)	244 (49)

Not every edge in a join graph has the same importance as illustrated below.

Example 6. Consider the rule (from the type inference program)

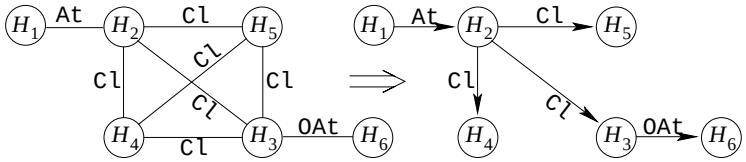
```
st(S,T1), st(S,T2) \ rt(T1,R1), rt(T2,R2) <=> R1 \== R2 | ...
```

The disequality guard creates a cycle in the join graph. However, its influence on the optimal join order is expected to be minimal, i.e., we can assume that it is better to lookup `rt(T2,R2)` via `T2` than using `R1 \== R2` given `R1`. □

Example 7. Consider the rule (from the well-founded semantics program)

```
true(At) \ neg(At,Cl), aclause(Cl,OAt),
    nbulit(Cl,_), nbplit(Cl,_), nbucl(OAt,N) <=> ...
```

The join graph is cyclic because variable `Cl` appears in multiple heads. However, once we have `Cl`, looking up another head containing it does not give us any new information. Under the assumption of representative selection that we made for the approximations of the cost model parameters, we can turn the join graph into a rooted join tree (one for every possible active constraint). The figure below depicts the original join graph (heads are numbered in textual order) and its transformation into a rooted join tree.



The idea is as follows: if the join graph contains a *clique* such that every two nodes in the clique share exactly the same variables, and moreover it holds that from any node outside of the clique, there exists at most one path to any node of the clique, then given a root node, there exists a unique entry point into the clique (possibly the root node belongs to the clique itself) and we can remove all edges in the clique except those connected to this entry point. □

However, if some nodes of a clique share more variables than others, we cannot make the join graph acyclic.

Example 8. Consider the rule (from the monkey & bananas program)

```
goal(a,h,A,B,C), phys_object(A,E,l,n,F,G,H), monkey(E,J,A) ==> ...
```

Variable `A` appears in all three heads, but the second and third head also share variable `E`. Hence we cannot reduce the join graph to a rooted join tree. □

For the example programs of the table above, only 10 of the 49 cyclic graphs remain cyclic after excluding the above two types of cyclicity.

6.3 Join Ordering in Current CHR Implementations

Current CHR implementations generally follow (a variation of) the HAL heuristic [5, 1], if they apply join ordering at all (e.g., Haskell CHR or the ECL^{i}PS^e `ech` library [15] use the textual order of heads). The K.U.Leuven CHR system uses a heuristic that was originally based on the HAL heuristic, but has become more complex over time due to incremental compiler changes.

The cost model for join ordering in the HAL CHR system is based on the assumption that each constraint argument ranges over a domain of the same size s. Given a partial join $\mathcal{J}^i_\Theta$, let w_i be the number of arguments of $H_{\Theta(i+1)}$ that are not fixed by $\mathcal{J}^i_\Theta$ (degree of freedom). The worst case size of $\mathcal{J}^{i+1}_\Theta$ is $\left|\mathcal{J}^i_\Theta\right| \cdot s^{w_i} = s^{w_1+\ldots+w_i}$ In order to eliminate the domain size s, the cost of the total join is approximated by $(n-1)\cdot w_1 + (n-2)\cdot w_2 + \ldots + 1 \cdot w_{n-1}$ which is in fact the logarithm (to base s) of $\left|\mathcal{J}^1_\Theta\right| \cdot \left|\mathcal{J}^2_\Theta\right| \cdot \ldots \cdot |\mathcal{J}^n_\Theta|$ whereas the real cost would be the sum of these partial join sizes. The weights relate to our approach as follows: $\mu(i) \approx s^{w_i}$.

The degree of freedom is reduced by functional dependencies and explicit equality guards. For other guards, the reduction depends on the determinism of the guard predicate. For example, if the predicate is declared as `semidet`, then the degree of freedom is reduced by 0.25 per fixed argument. The motivation is a guard `X>Y` which would remove 0.5 degrees of freedom. Note though that this implies a cost $s^{1.5}$ for a constraint `c(X,Y)` with guard `X>Y` whereas a more accurate estimate would be $s^2/2$.

Finally, we note that although it is mentioned in [1, 5] that we often only need the first full match, the HAL cost model does not reflect this concern.

7 Conclusion

We summarize the contributions of this paper. We have given a new cost model for join orders that is more realistic and flexible compared to earlier work. Our work is the first to consider runtime statistics such as cardinalities and selectivities in the context of CHR. We have shown how these statistics can be used as estimates for the parameters of our cost model. An efficient algorithm from database literature for join order optimization under the restriction of acyclic join graphs, has been ported to the CHR context. Furthermore, it is shown that many cyclic join graphs can in fact be made acyclic under certain assumptions. Finally, we have discussed the issue of optimization for the first-few answers, and have given an alternative cost model for this case.

Future work. In this paper, we have only looked at the theoretical side of the join ordering problem. Although it is already clear that a good join order, and even dynamic join ordering can give rise to a better runtime complexity, it remains an open question whether it is worth the extra effort in typical problems. Therefore, we plan to implement the proposed join ordering schemes. This will also allow us to experiment with ideas such as computing optimal join orders

in parallel using a different thread, and hybrid static and dynamic join ordering (where the dynamic search space is reduced at compile time, e.g., based on the join graph or functional dependency analysis). Other issues we plan to investigate are the trade-off between optimization cost and join cost, using profiling information for static join ordering, and other criteria (apart from those given in Section 6.2) to reduce a cyclic join graph to an acyclic one.

References

1. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Victoria, Australia, Dec 2005.
2. G. J. Duck and T. Schrijvers. Accurate functional dependency analysis for Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, CW Report 421, pages 109–124. Dept. CS, K.U.Leuven, Belgium, 2005.
3. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. on Logic Programming*, LNCS 3132, pages 90–104, 2004.
4. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
5. C. Holzbaur, M. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(4 & 5):503–531, 2005.
6. J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical Report STAN-CS-71-190, Stanford University, CA, USA, 1971.
7. T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.
8. R. J. Bayardo Jr. and D. P. Miranker. Processing queries for first-few answers. In *5th Intl. Conf. on Information and Knowledge Management*, pages 45–52, 1996.
9. R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *12th Intl. Conf. on Very Large Data Bases*, pages 128–137, 1986.
10. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, Jun 2005.
11. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *1st Workshop on CHR, Selected Contributions*, Ulmer Informatik-Berichte 2004-01, pages 1–5. Universität Ulm, Germany, 2004.
12. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *1979 ACM SIGMOD Conf. on Management of Data*, pages 23–34, 1979.
13. M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.
14. A. N. Swami and B. R. Iyer. A polynomial time algorithm for optimizing join queries. In *9th Intl. Conf. on Data Engineering*, pages 345–354. IEEE, 1993.
15. M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.