# Introduction to Unified Tabling

Mostowski Collapse[1]

[1] XLOG Technologies AG, Switzerland

May 15, 2026

**Abstract.** The problem of tabling in logic programming radiates beyond pure memoization into areas such as subsumption checking, fixpoint iteration and suspension methods. Nevertheless, in this report we are mainly occupied with the many shades of doing memoization. One of the popular data structures to perform Prolog queries and answers memoization, is to use tries. Tries are found in both SWI-Prolog and XSB Prolog and we compare against these Prolog systems. But the route of Dogelog Player is a little bit different. Inspired by the hashCode() contract as found in Java, we have realized a series of measures to arrive at a hash based memoization for Prolog queries and answers.

**Keywords:** Prolog, Tabled Logic Programming, SWI, XSB

## 1 Introduction

Any indexing data structure will face the follow-up problem of copying or non-copying of Prolog terms. While execution term sharing made Prolog a viable competitor to LISP, the situation for memoization is frustrating. In the first 3 chapters we introduce sharing of ground terms, that can be explored with multi-argument clause indexing.

The last chapters shows how term expansion can be used to define a table/1 directive. The road taken here is the name giver of this report, since queries and answers are cached uniformly via the dynamic database. We also give an overview how our functor based indexing was extended by hash based indexing.

# 2 Stored Hashes for Dogelog Player

## 2.1 Overview

Dogelog Player is a Prolog system for the JavaScript, Python and Java target with libraries mostly written in 100% Prolog. Among the libraries we find tree maps library(util/tree) and hash maps library(util/hash). To speed up hash maps for frozen compounds we introduced precomputed hashes.

## 2.2 Function Pointers

The library adopts a unique design to work around a problem, that our Prolog system is not object oriented. If our Prolog system were object oriented, we could adopt the Java contract consisting of .hashCode() and .equals() methods and would have Logtalk style definition:

```
:- protocol(hashable).

% hash predicate: returns an integer hash for a term
 :- public(hashCode/2).

% equality predicate: succeeds if two terms are equal
 :- public(equals/2).

:- end_protocol.
```

Our design is such that we drop the Grady Booch classic OO mantra object = state + behavior. And use the fact that ISO core standard Prolog offers call/n, i.e. higher order logic programming via term closures. So, the default constructor hash_new/1 of our hash table reads as follows:

```
% hash_new(-Hash)
hash_new(R) :-
   hash_new(R, term_hash, ==).
```

The secondary constructor hash_new/3 accepts two closures, one closure for .hashCode() and one closure for .equals(). The hash table routines are then 100% implemented in Prolog, using the Dogelog Player extension change_arg/3 to provide a destructive version as well.

### 2.3 Stored Hashes

Precomputed hashes can be realized by changing the implementation of the frozen compound data structure. Like the new Dogelog Player specific built-in change_arg/3 for non-frozen compounds, this is a native intervention, for the internal frozen compounds data type:

```
class Frozen(Structure):
    def __init__(self, functor, args):
        super().__init__(functor, args)
        self.hash = object_hash_code(functor)
        i = 0
        while i < len(args):
            obj = args[i]
            if is_frozen(obj):
                self.hash = int32(self.hash*31 + obj.hash)
            else:
                self.hash = int32(self.hash*31 +
                              object_hash_code(obj))
            i += 1
```

One might ask whether the additional field .hash is justified or not. From the viewpoint of sharing, it is expected that the field .hash will not occupy much space, since frozen compounds are not copied. Their number is limited by the dynamic database size where frozen compounds live.

```
def equals(first, second):

    if first is second:
        return True
    if (is_frozen(first) and is_frozen(second) and
          first.hash != second.hash):
        return False
```

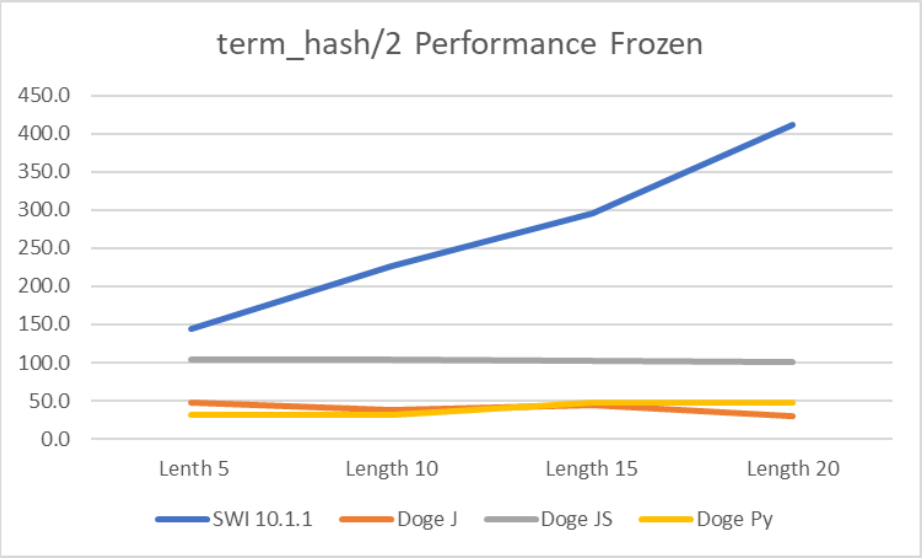
The above code shows a main benefit of the .hash field. One might be familiar with the positive pointer equivalence test, but the new .hash can now be used as a negative test for equality before testing the functor of a compound and entering the arguments of a compound.

### 2.4 term_hash/2 Perf

The challenge here is to make no mistakes. In the end the statically pre-computed hashes have to agree with the dynamically computed hashes by the built-in term_hash/2, since a frozen compound might still be syntactically equivalent to a non-frozen compound. We benchmarked:

```
/* retrieval+hash */
test3(N) :-
   between(1,1000,_),
   data(N, X),
   term_hash(X, _),
   fail.
test3(_).
```

Interestingly Python beats JavaScript, we have no explanation. Also, SWI-Prolog seems to be extremely slow, again we have no explanation, but it throws light we only tested lists X of size 5, 10, 15 and 20. We find that SWI-Prolog is linear whereas Dogelog Player is constant time.




The constant time for a frozen compound can be explained since .hash is precomputed. Further for a small Prolog non-frozen compounds that are composed from a constant number of frozen compounds the computation of term_hash/2 is designed so that it will be constant time again.

### 2.5 Distinct Problem

Both Dogelog Player and SWI-Prolog offer a sequence library that provides a predicate distinct/1. This is a meta predicate that uses a state across Prolog calls to filter solutions that are equal modulo (=@=)/2. While Dogelog Player opted to use hash table, SWI Prolog uses its trie data structure.

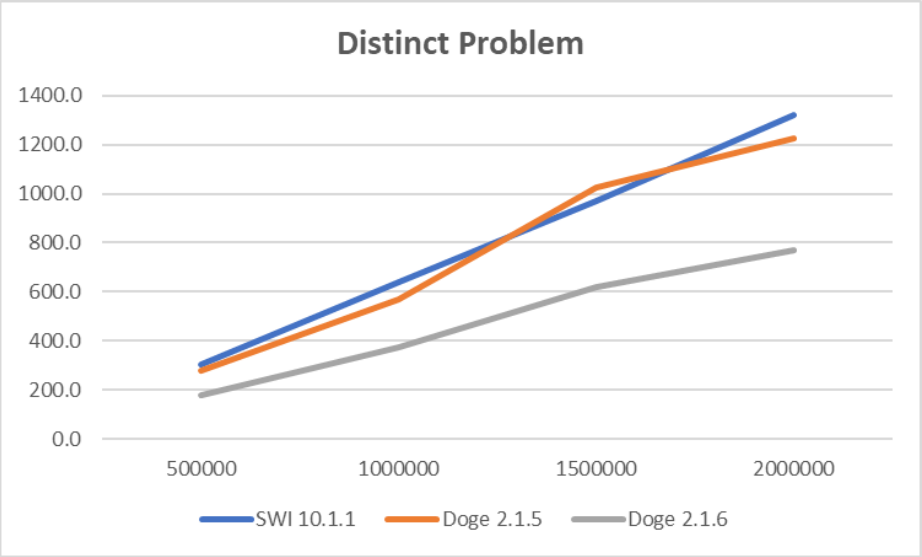
To perform the testing, we created random dumps with basic arithmetic expressions. The data generation can be coded in Prolog itself with a little help of random/1 built-in. In total we created 100000 rows for a dynamic predicate dump/1. The data and test look as follows.

```
?- random_dump.
true.
?- dump(X).
X = 4+(2+((6-0)*(6-(2+2*(8*4))-4)-9))+5*((2-(0+6))*9*7);
X = (0+(0*(1-4)-7)+3*3)*1;
X = 3-2+2+8
Etc..
```

The test is then to cycle through all the data and an run all solutions through distinct/1. Fetching the data itself might incur some cost, which is mitigated in Dogelog Player since it provides program sharing (PS). Like in our previous testing, we subtracted retrieval timing:

```
/* retrieval+distinct */
test2 :-
   distinct(dump(_)),
   fail.
test2.
```

The retrieval itself is only a small time component, we find for the conducted test that program sharing (PS) in Dogelog Player occupies more or less the same time as SWI-Prolog. But the program sharing (PS) has repercussions for copy_term/2 in our distinct/1 realization:




In the above we see the additional benefit of .hash as it will be part of the upcoming release of version 2.1.6 of Dogelog Player. We tested against version 2.1.5 of Dogelog for Java and against SWI-Prolog 10.1.11. It shows that the new stored .hash give us a 2x times speed-up.

### 2.6 Summary

Stored .hash values are popular through the Java .hashCode() contract. Prolog systems shy away because of additional memory or in favor of other data structures. We show that they can be also beneficial for frozen compounds and demonstrate a 2x times speed-up for distinct/1.

# 3 Assertion Sharing in Dogelog Player

## 3.1 Overview

Dogelog Player is a Prolog system for the JavaScript, Python and Java target that borrows much of its design from formerly Jekejeke Prolog. With the upcoming release we have complemented the existing program sharing (PS). We added a further type, namely assertion sharing (AS).

## 3.2 Assertion Sharing

Up to release 2.1.5 of Dogelog Player we provided only program sharing (PS). This is a notoriously hard problem for Prolog systems, especially those that are based on a WAM. While WAMs do provide term sharing, during execution, they can usually not share clause data.

Dogelog Player could profit that it is not WAM cells based and that it keeps everything in the host language heap. In as far it emits code, that creates so called frozen compounds. Being subject to the host language garbage collection, these can then be easily shared during clause instantiation:

```
/* Compound Compilation up to Dogelog 2.1.5 */
sys_encode_term(C, D) --> {compound(C)}, !,
   {C =.. [F|L]},
   sys_encode_list(L, R),
   {sys_const_only(R) ->
         D = frozen(F,R);
         D = functor(F,R)}.
```

In the upcoming release 2.1.6 of Dogelog Player we will provide assertion sharing (AS) as well. Again, a hard problem for WAM machines, but nevertheless do able by deriving it from program sharing (PS). Instead of emitting a constructor, we just emit a value holder:

```
/* Additional Compound Compilation from Dogelog 2.1.6 on */
sys_encode_term(C, D) --> {ir_is_frozen(C)}, !,
   {D = share(C)}.
```

What makes it impure, is the fact that frozen compounds are invisible to ordinary Prolog code. So, we had to introduce a new built-in ir_is_frozen/1 that tests a given Prolog

term, whether it's a frozen compound or not. A lot of work is spared, since a frozen compound is not further analyzed.

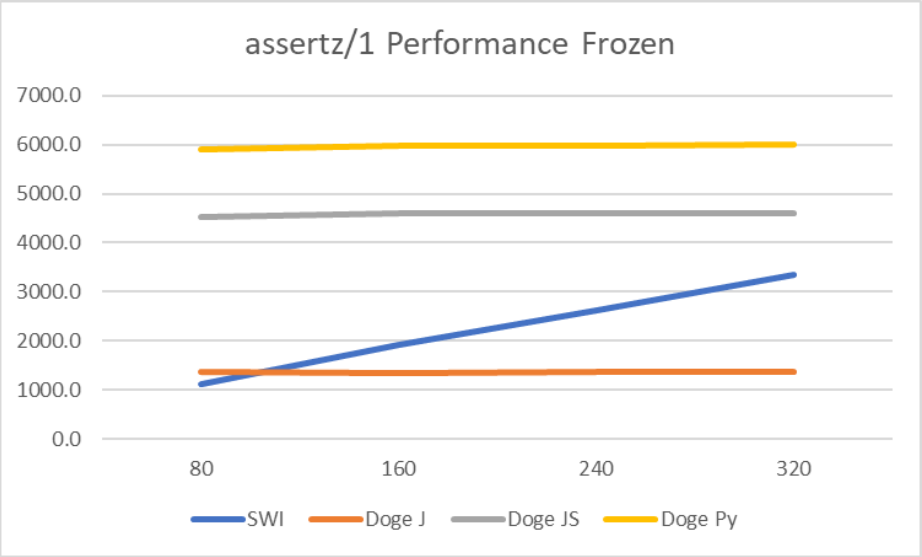
### 3.3 assertz/1 Perf

While program sharing (PS) has a positive effect on clause/2, assertion sharing has a positive effect on assertz/2. We use a silly test case that just retrieves data from a predicate and then stores it in another predicate. The test data itself is only frozen lists, a special case of frozen compounds:

```
data(5, [0, 1, 2, 3, 4]).
data(10, [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]).
data(15, [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]).
data(20, [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3,
4]).

/* retrieval+store */
test3(N) :-
   retractall(dump(_)),
   between(1,1000,_),
   data(N, X),
   assertz(dump(X)),
   fail.
test3(_).
```

To see something, we use much larger lists. The problem is having a backend such as sys_encode_term/2 written in 100% Prolog still causes quite some overhead even with assertion sharing (AS). The irony is, as one will later see, this does not matter for certain applications:




In the above we compared against SWI-Prolog 10.1.11. It seems SWI-Prolog, although not strictly WAM, has neither program sharing (PS) nor assertion sharing (AS), since the time needed to execute the test is linear with the size of the list. On the other hand, Dogelog Player can do it in constant time.

### 3.4 Counting Problem

Some Prolog systems offer an aggregate library. Under the hood, they typically either take the route of some sort algorithm or of some map data structure. In the following we consider another approach in that we utilize the dynamic database of the Prolog system. The naive solution reads:

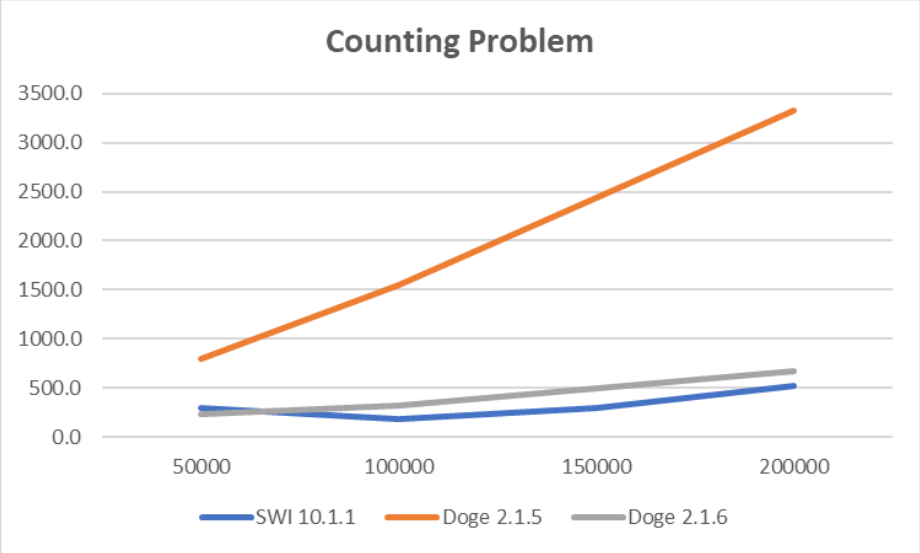
```
/* Naive Counting */
test2 :-
   retractall(count(_,_)),
   dump(X),
   (retract(count(X, N)) ->
       M is N+1,
       assertz(count(X, M));
       assertz(count(X, 1))),
   fail.
test2.
```

Due to the availability of deep indexing, the naive solution will perform relatively well in SWI-Prolog. Since the data is arithmetic expressions such as 3*2–5*4 with very few

functors and numbers, it is not suitable for Dogelog Player, which does not yet have deep indexing.

```
/* Hash Index Counting */
test2 :-
   retractall(count(_,_,_)),
   dump(X),
   term_hash(X, H),
   (retract(count(H, X, N)) ->
       M is N+1,
       assertz(count(H, X, M));
       assertz(count(H, X, 1))),
   fail.
test2.
```

The above code shows a cludge, in that we use the built-in function term_hash/2 to generate an extra key. This function has recently been improved by stored .hash field for frozen components. Assertion sharing (AS) enables access, showing a benefit for the counting problem:




Testing different dump/1 input sizes 50000, 100000, 150000 and 200000 we see a speed-up of a factor 4x from version 2.1.5 to version 2.1.6. This puts us close to the performance of SWI-Prolog, despite that our assertz/1 predicate calls 100% Prolog routines such as sys_encode_term/4.

### 3.5 Summary

We had to introduce a new built-in ir_is_frozen/1 to test whether a Prolog term is a frozen compound or not. The 100% Prolog programmed assertz/1 then takes a short cut and generates a value holder. When combined with term_hash/2 this speeds up a dynamic database counting example.

# 4 Just-in-Time Indexing in Dogelog Player

## 4.1 Overview

Dogelog Player is a Prolog system for the JavaScript, Python and Java target that borrows much of its design from formerly Jekejeke Prolog. While Dogelog Player shipped only with first argument indexing for almost 4 years, we now added multi-argument just-in-time indexing.

## 4.2 Just-In-Time Indexing

The transition from first argument indexing to multi-argument just-in-time indexing came naturally. The first argument indexing of Dogelog Player was a stripped-down version of single index from formerly Jekejeke Prolog, in that we don't provide var/1 indexing. A predicate object holds:

```
public static final class Provable {
   public Logical rope;
   public Logical guard;
   public Map map;
   public Object func;
   Etc...
}
```

The class Logical encapsulates an input order clause list. Responsible for indexing are then the field guard and map, whereas guard holds all clauses that have no index value, while map points to further Logicals depending on the index value of the clause.

The field func was for built-in function pointers. In our current take we simply removed the 3 fields rope, guard and map from the predicate object and gave them their own encapsulation in the form of a class Stick. Instead of nulling func for defined predicates, we store a logical or stick there:

```
public static final class Provable {
   public Object func;
   Etc...
}

public static final class Stick {
    public Logical rope;
    public Object[] guards;
    public Map[] maps;
```

```
}
```

The constant MAX_INDEX controls the width of the two arrays guards and maps, so that a stick can have multiple maps and guards. This allows for multiple indexes, and the indexes parallel the arguments from a clause head. Just-in-time indexes are a matter of promoting Logical to Stick.

### 4.3 Programs Impact

In a classical Warren Abstract Machine (WAM) set-up, programs are seen as doing their own clause routing. The instructions are switch_on_XXX, which insinuates jump tables. Which quickly breaks down already for atoms, even with the cludge of atom tables that internalize atoms.

Formerly Jekejeke Prolog saw clause routing for programs and the dynamic database simply as both relying on clause indexing. On the non-functional side, one sees a gain in time performance with the cost of more storage. On the functional side it is difficult to observe clause indexing.

```
maplist(C, L, R) :-
   maplist_(L, R, C).

maplist_([], [], _).
maplist_([X|L], [Y|R], C) :-
   call(C, X, Y),
   maplist_(L, R, C).
```

In the above, the helper maplist_/3 is usually seen to cater for first argument indexing. Indexing reduces candidate clause sets, leading to fewer choice points. The top-level will not prompt to display more solution since the Prolog system checks the presence of choice points:

```
?- maplist(plus(3), [4,2,5], X).
X = [7, 5, 8].
```

Interestingly the same helper maplist_/3 can also cater for non-first argument indexing. Without any means for improved choice point elimination in the second argument, we will see the phenomenon of a so called spurious choice point, which upon backtracking simply fails:

```
/* Dogelog Player 2.1.6 */
?- maplist(plus(3), X, [7,5,8]).
X = [4, 2, 5];
fail.

/* Scryer Prolog 0.10 */
?- maplist(plus(3), X, [7,5,8]).
   X = [4,2,5]
;  false.
```

On the other hand, our upcoming release 2.2.1 it will give:

```
/* Dogelog Player 2.2.1 */
?- maplist(plus(3), X, [7,5,8]).
X = [4, 2, 5].

/* SWI-Prolog 10.1.5 */
?- maplist(plus(3), X, [7,5,8]).
X = [4, 2, 5].
```

### 4.4 Queries Impact

Well one might get already exited by getting rid of certain spurious choice points. The excitement might increase when we get a better picture how indexes impact the dynamic database. If there is no index a Prolog system falls back to backtracking over clauses, which amounts to a scan.

We can posit that scanning a clause set of size N takes O(N) time. An index with map size M reduces the clause set to size N/M in the average. So that for a hash map which has O(1) access, the scanning of the same clause set now only takes O(1)+O(N/M) = O(N/M) time. The speed-up is then:

$$S = \text{speed-up} = \frac{\text{scan}}{\text{hash+scan}} = \frac{O(N)}{O(N/M)} = O(M)$$

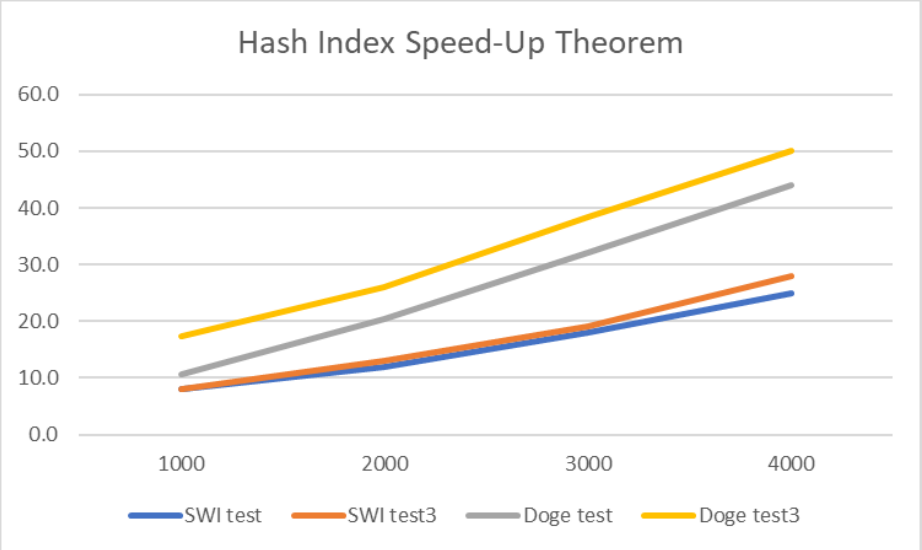
We first didn't believe our eyes, but could demonstrate the speed-up for a secondary index by a simple testing set-up. The binary predicate edge/2 is uniformly populated

with random pairs (X,Y) where X and Y are in the range 0..299. Both test/0 and test3/0 should take the same time:

```
test :-
   between(0, 299, B),
   edge(_, X),
   X = B,
   fail.
test.

test3 :-
   between(0, 299, _),
   between(0, 299, B),
   edge(_, B),
   fail.
test3.
```

Testing Dogelog 2.2.1 and SWI-Prolog 10.1.5 confirms the formula:




So, with a map of size M, we can do O(M) more work in the same time, this is what the loop in test3 does, it does the lookup of some B exactly M times more often than the loop in test. SWI-Prolog displays a speed-up in its jiti_list/1 command. Dogelog Players new command shows M:

```
```

```
/* SWI-Prolog 10.1.5 */
?- jiti_list(edge).
Predicate   #Clauses Index Buckets Speedup Coll Flags
————————————————————————————————————————————————
user:edge/2 4,000    2     512     236.4   0
true.

/* Dogelog Player 2.2.1 */
?- jiti_list(edge).
% Prädikat      edge/2
% Argumente     2
% Kardinalität  300
true.
```

In our formula we over simplified by setting O(1)+O(N/M) = O(N/M). Practically doing O(M) times the query, gives a time complexity of O(M)*(O(1)+O(N/M)) = O(M) + O(N). This is seen for Dogelog Player in the graph, in that the line for test3 is just the line for test vertically displaced.

### 4.5 Updates Impact

The impact on assertz/1 and retract/1 for dynamic database updates is less pronounced. There are at least two factors at work, that mask any index efforts. First of all, we currently don't have a Logical ropes implementation, that would allow O(1) operations for insert and delete.

This is less severe since indexes reduce the clause set, and therefore the Logical ropes require less effort. Secondly, we must keep in mind that Dogelog Player is a 100% Prolog written Prolog system. Predicates such as assertz/1 still execute considerable amounts of Prolog code:

```
assertz(T) :-
   sys_frost_horn(T, 0, C),
   functor(T, F, A),
   ir_clause_add(F, A, C, 3).

sys_frost_horn(Horn, Flags, Native) :-
   sys_trans_horn(Horn, Trans),
   sys_encode_horn(Trans, Encode, Flags),
   sys_host_make(Encode, Native).
```

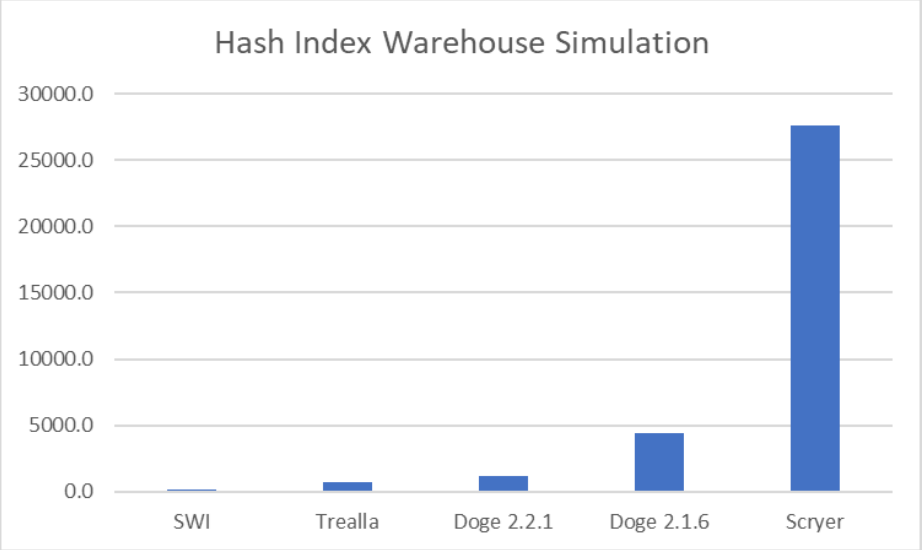
As a benchmark for updates, we use a random warehouse simulation. In each step a random Yn is asserted and a random Xn is retracted. We take Yn and Xn from the integer random number generator found in the Sinclair ZX81. It generates random numbers in the range $0..2^{16}-1$:

```
zx81(X, Y) :- Y is (X*75+74) mod 65537.

warehouse(0, _, _) :- !.
warehouse(N, X, Y) :-
   assertz(foo(bar, Y)),
   (retract(foo(bar, X)) -> true; true),
   zx81(X, Z), zx81(Y, T),
   M is N-1,
   warehouse(M, Z, T).

warez :-
   retractall(foo(_,_)),
   warehouse(10000, 1, 24992).
```

Testing Trealla Prolog 2.91.5 and Scryer Prolog 0.10 as well we find:




On the positive side we find indeed a speed up from Dogelog Player 2.1.6 to Dogelog Player 2.2.6. While Dogelog Player 2.1.6 deploys a first argument index, which is useless in the present example. Dogelog Player 2.2.1 will deploy a compound index as seen by the jiti_list/1 command:

```
?- jiti_list(foo).
Predicate  #Clauses Index Buckets Speedup Coll Flags
________________________________________________________
user:foo/2 2,999    2     4,096   3000.0  1432
true.

?- jiti_list(foo).
% Prädikat      foo/2
% Argumente     1
% Kardinalität  1
% Argumente     1+2
% Kardinalität  2999
true.
```

SWI-Prolog is more defensive, it only creates an index on the second argument, but displays a similar hypothetical speed-up. The speed-up materializes less for Dogelog Player, since the 100% Prolog written assertz/1 add quite an overhead to the insert operations.

### 4.6 Summary

For programs one sees fewer spurious choice points. For the dynamic database, in queries one sees a speed-up of almost a factor M, where M is the size of the index map. For updates the speed-up materializes less, due to the overhead of our 100% Prolog written insert operations.

# 5 Tabling Directive for Dogelog Player

## 5.1 Overview

Our new library silkworm for predicate tabling tries to do the same, as the library railgun for finite domain constraints. Namely provide a portable library that is only based on the ISO core standard. Due to a series of recent improvements of Dogelog Player the tabling directive is now competitive.

## 5.2 Term Expansion

The directive is conceived as a source-to-source transformation, done via term expansion. To define an according term expansion, we had to first enhance the term expansion so that it can return multiple clauses which we made understand by the core. The old term expansion was simply:

```
expand_term(Clause, Clause2) :-
   term_expansion(Clause, Clause3), !,
   expand_term(Clause3, Clause2).
expand_term(Clause, Clause).
```

The new term expansion handles list of clauses as well:

```
expand_term(T, [Clause2|Rest2]) :- nonvar(T), T = [Clause|Rest], !,
   expand_term(Clause, Clause2),
   expand_term(Rest, Rest2).
expand_term(T, []) :- nonvar(T), T = [], !.
```

Armed with term expansion we defined a directive table/1. As a side effect the given predicate is registered for tabling, preparing a query cache, preparing an answer cache and creating a wrapper. Subsequent clauses for the predicate will be called from the wrapper. Let's look at fib/2:

```
:- table(fib/2).
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, X) :- M is N-1, L is M-1, fib(M, Y), fib(L, Z), X is Y+Z.
```

The resulting wrapper is fib/2 itself. The anatomy of the wrapper is such that it does first an assertz/1 if absent for the query cache fib_k/2, and then an assertz/1 if absent for all the answer from the fib_i/2 to the answer cache fib_r/2. Finally, there is a clause that reads the answer cache:

```
?- listing(fib).
fib(A, B) :-
   \+ fib_k(A, B),
   assertz(fib_k(A, B)),
   fib_i(A, B),
   \+ fib_r(A, B),
   assertz(fib_r(A, B)),
   fail.
fib(A, B) :-
   !,
   fib_r(A, B).
true.
```

The original fib/2 clauses are now at the predicate fib_i/2:

```
?- listing(fib_i).
fib_i(0, 1) :-
   !.
fib_i(1, 1) :-
   !.
fib_i(A, B) :-
   C is A-1,
   D is C-1,
   fib(C, E),
   fib(D, F),
   B is E+F.
```

### 5.3 Egg Dropping

We were first curious whether our directive works for a Datalog-ish example, means an example without function symbols. The Egg drop problem asks for the minimal number of trials, that suffice to determine the exact floor level of a building. Our code reads:

```
:- table(drop/3).
```

```
drop(0, _, R) :- !, R = 0.
drop(_, 0, R) :- !, R = 0.
drop(F, 1, R) :- !, R = F.
drop(F, E, M) :- E > 1, F > 0,
   aggregate_all(min(N), (between(1,F,K),
       E2 is E-1,
       F1 is K-1,
       drop(F1, E2, N1),
       F2 is F-K,
       drop(F2, E, N2),
       N is 1+max(N1,N2)), M).
```

The code is challenging for example for SLG-Tabling, because we use the aggregate_all/3 meta predicate with a min aggregator. Even when modelled via findall/3 and min_list/3, Scryer-Prolog could not solve it. While SWI-Prolog, XSB Prolog and our own Dogelog Player agree on the result:

```
/* SWI, XSB and Dogelog */
?- drop(100, 10, X).
X = 7.

/* Scryer */
?- drop(100,10,X).
   false.

/* Trealla */
N/A
```

To test tabling with an according Trealla Prolog feature was not possible, since at the moment of writing this article, Trealla Prolog did not provide a tabling directive. The correctness of our tabling was quite assuring. In the following section we investigate its performance as well.

### 5.4 Native Assert

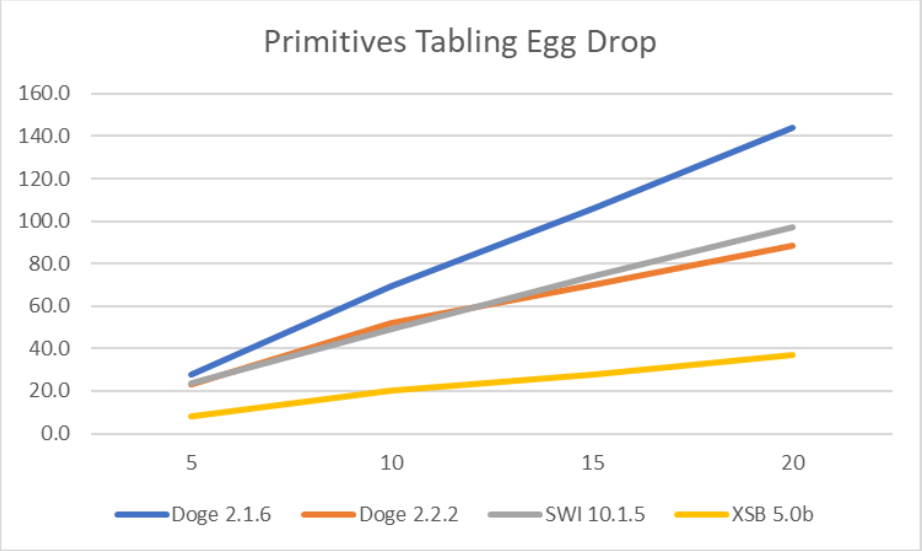
Our work on improving the dynamic database to allow improved tabling started with release 2.1.6 of Dogelog Player. In that release we introduced assertion sharing. The subsequent release 2.2.1 of Dogelog Player offered Just-in-Time Indexing. But we noticed a 100% pure Prolog bottleneck.

To remove it we introduced a built-in ir_clause_ossify/6:

```
ir_clause_ossify(H, B, R, F, W, C): internal only
   The built-in succeeds in C with a Java object representing
   a clause with head instructions H, body instructions B,
   cut option R, indexing flag F and functor W.
```

In release 2.2.1 we used the new built-in already to encode dynamic clauses for the JavaScript, Python and Java targets. In release 2.2.2 we used it to encode static clauses as well, speeding up the Prolog text consult time. It was also paralleled by a new ir_goal_ossify/4 for directives.

Benchmarking release 2.1.6 against release 2.2.2 gives:




We were benchmarking the egg dropping example for 5, 10, 15 and 20 eggs. We also included SWI-Prolog 10.1.5 and XSB Prolog 5.0b. Thanks to the speed-up by native assert, Dogelog Player is now close, even slightly below SWI Prolog. XSB Prolog is a factor 2–3x faster nevertheless.

### 5.5 Even Palindroms

As next we were curious whether our directive works for a non-Datalog-ish example, means an example with function symbols. For non-Datalog problems with large ground terms in their queries or answers, Dogelog Player usually excels. We looked at this DCG example

```
```

```
:- table(epal/3).
epal([]) → [].
epal([X|L]) → [X], epal(L), [X].
```

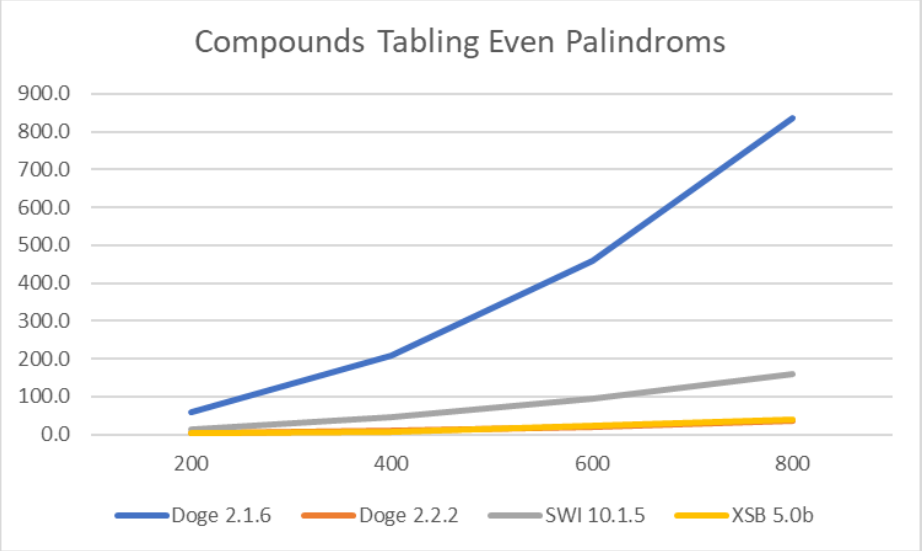
For our testing we created many longer lists:

```
many_lists(N) :-
   K is N*200,
   findall(I, between(1,K,I), L),
   abolish_all_tables,
   retractall(data(_)),
   between(1,100,_),
   assertz(data(L)),
   fail.
many_lists(_).
```

Only the first data/1 entry will cause tabling computation:

```
test :-
   data(X),
   epal(X, _, []),
   fail.
test.
```

The other data/1 entries will cause no computation anymore, meaning the caches will not anymore be modified. Instead, the benchmark will then measure how well the query cache is accessed and how well the answer cache is accessed. We have the following findings:




Its seems that XSB Prolog performs similar like Dogelog Player, possibly using also program sharing and/or assertion sharing, reducing the time for initial modification and repeated access. Our sharing is limited to frozen terms which is the case for the top answer to the epal/3 example.

### 5.6 Momomorphic Adornment

Although our results are encouraging. The current version does not yet realize tabling of non-well-founded recursion. When predicates are encountered that are non-well-founded, the behavior of the library is currently undefined, it might loop or give incomplete results.

Further we do not access variants, neither in the query nor in the answer. This works for example fine, if all the queries to the same predicate have the same adornment. We can also have the same predicate queried with different adornments, if there is an intervening abolish_all_tables:

```
?- epal([1,2], X, []).
X = [1, 2, 2, 1].

?- abolish_all_tables.
true.

?- epal(X, [3,4,4,3], []).
X = [3, 4];
fail.
```

```
```

What does not yet work is mixing different adornments:

```
?- epal([1,2], X, []).
X = [1, 2, 2, 1].

?- epal(X, [3,4,4,3], []).
fail.
```

To solve certain forms of recursion, we have to first deliberate what method we should use in the design of tabling workers. Existing systems such as SWI and XSB use different worker approaches. Similarly, there are different options around to deal with polymorphic adornments.

## 5.7 Summary

While we reach similar performance like SWI-Prolog for Datalog examples, our system parallels the XSB-Prolog performance for non-Datalog examples. What is missing are non-well-founded forms of recursion and polymorphic adornments. We are currently working on addressing both issues.

# 6 Egg-SAT in Dogelog Player

## 6.1 Overview

Dogelog Player is a Prolog system for the JavaScript, Python and Java target. This is a further iteration of our new tabling directive. In this take we added deep indexing to the dynamic database, facing the question tries or not tries? With amazing results for a rewriting problem.

## 6.2 Equational Formulation

Rewriting systems have a long history not only in computer science, but also in mathematics and philosophy. Already the Greeks distinguished commensurable and incommensurable quantities, by Euclidean algorithm sequences that terminate and do not terminate.

Among notable last century pioneering figures, we find Emmy Noether, who introduced ascending chain conditions into ring theory. We consider Boolean rings, modelling its normal form by ordered binary decision diagrams (BDD). The creation rule of an if-then-else nodes ite/3 reads:

```
/* if-then-else node creation, BDD rules */
(V & T) v ((~V) & T) = T
(V & T1) v ((~V) & T2) = ite(V, T1, T2)
```

We look at a very restricted take of BDDs, and we are only interested in the exclusive or operation (xor)/2. On integers the operation is part of the ISO core standard for Prolog, among the operations on bitwise interpretation of integers. We use the Shannon factors to extend it to BDDs:

```
/* xor by Shannon factors, BDD rules */
((V & T1) v ((~V) & T2)) xor ((V & S1) v ((~V) & S2)) =
    (V & (T1 xor S1)) v ((~V) & (T2 xor S2))
((V & T1) v ((~V) & T2)) xor S =
    (V & (T1 xor S)) v ((~V) & (T2 xor S))
T xor ((V & S1) v ((~V) & S2)) =
    (V & (T xor S1)) v ((~V) & (T xor S2))
```

## 6.3 Relational Reformulation

In practice the xor operation on the Shannon factors will operate on if-then-else nodes ite/3, and the last two equations will be used to keep the BDD ordered, i.e. that the

variables are subject to a total order, for example the (@<)/2 from the ISO core standard for Prolog.

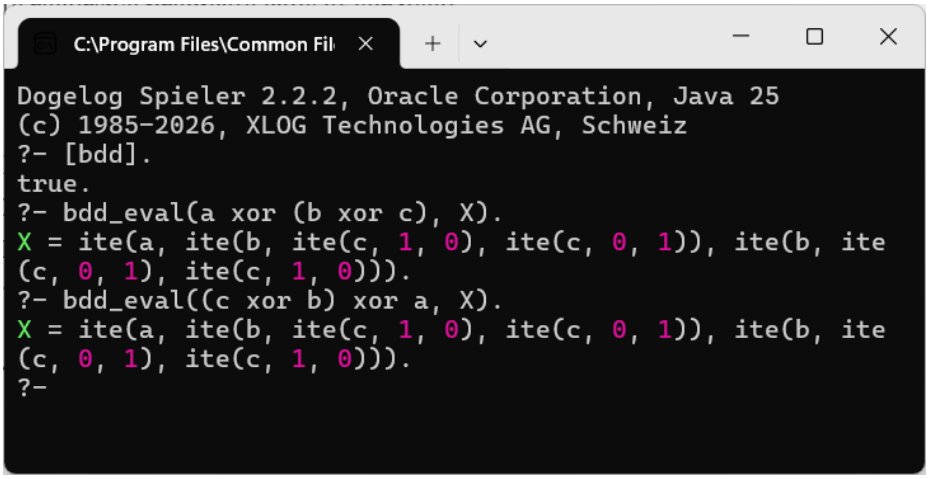
To bring it to Prolog we use a relational formulation by predicates, the equation sign (=)/2 completely disappears. And we happily use the cut (!) so as to priories the clauses of the realized predicates and/or to realize fallback clauses. The node creation itself reads:

```
% bdd_node(+Atom, +ITE, +ITE, -ITE)
bdd_node(_, T, T, T) :- !.
bdd_node(V, T, E, ite(V,T,E)).
```

The (=)/2 equation sign is still present in the above, covertly by using the pattern T twice. But the unification will be used for ground terms only resulting in (==)/2. This will treat the BDDs as purely syntactic without any otherwise defined equality. The binary xor then reads:

```
% bdd_xor(+ITE, +ITE, -ITE)
bdd_xor(ite(V1,T1,E1),  B,  R)  :-  (B  =  ite(V2,_,_),  V1  @<  V2;
integer(B)), !,
   bdd_xor(T1, B, R1),
   bdd_xor(E1, B, R2),
   bdd_node(V1, R1, R2, R).
bdd_xor(ite(V,T1,E1), ite(V,T2,E2), R) :- !,
   bdd_xor(T1, T2, R1),
   bdd_xor(E1, E2, R2),
   bdd_node(V, R1, R2, R).
bdd_xor(B, ite(V2,T2,E2), R) :- !,
   bdd_xor(B, T2, R1),
   bdd_xor(B, E2, R2),
   bdd_node(V2, R1, R2, R).
bdd_xor(B1, B2, R) :- R is xor(B1,B2).
```

Prolog not only does not allow to redefine (=)/2, it has also no function evaluation for user defined functions when we look at the ISO core standard scope. Therefore, in the above code the binary xor realization does a lot of explicit recursive predicate calls to its own definition.



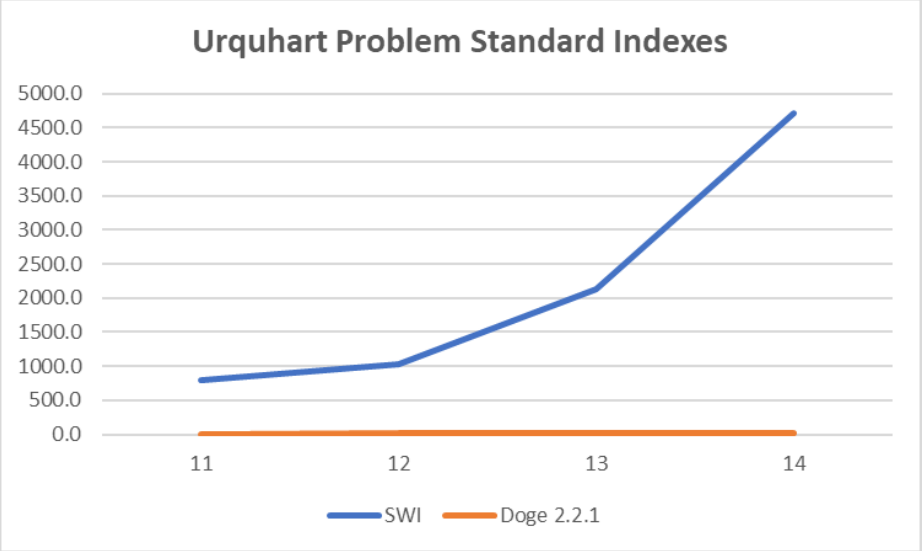

An eval utility is straight forward, and it can turn Boolean ring expressions into BDDs. The language is quite limited, but because xor has associativity (A xor B) xor C = A xor (B xor C) and commutativity A xor B = B xor A, we can already demonstrate in the above what a normal form delivers.

### 6.4 Caching BDDs

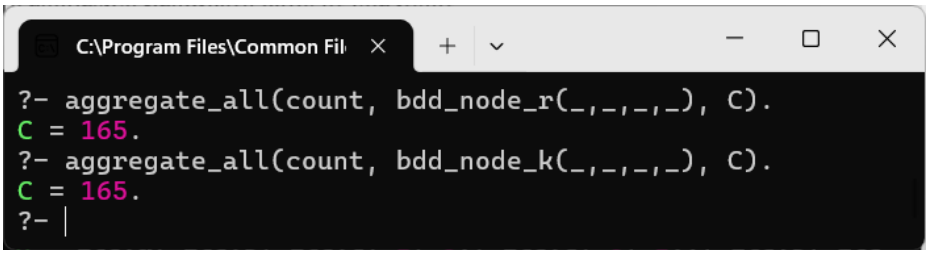
The BDDs trees grow rapidly, but they contain a lot of hot air. For example, in the above example there are common sub expressions ite(c, 1, 0) and ite(c, 0, 1), representing c and ~c. Their elimination is part of the BDD lore, and Minatos ZDD library showed it nicely using a caches. We tried tabling:

```
:- table(bdd_node/4).
:- table(bdd_xor/3).
```

As easy as it looks it doesn't work in most Prolog systems, since the assertion and/or program sharing is weak, or the assertz and retrieval operations are costly. Dogelog Player shows an advantage since it has a series of frozen term optimization:




We could test the Urquart Test case, i.e. X = x1 xor … xor xn, and then evaluate X xor X with default settings against SWI. But XSB run into a table overflow, Scryer Prolog didn't produce a correct answer. Dogelog Player 2.1.6 stays below 20 ms for n=11,12,13,14 while SWI reaches 5 secs.




The above shows how the common expression elimination reduces the memory footprint. In as far it is helpful to think that bdd_node/4 acts as a local interning cache. Its size gives an indicative of the compression factor, in the above showing the query and answer cache of tabling for n=14.

### 6.5 Precomputed Hashes

But what about the speed, our just-in-time indexing doesn't go very deep. Other tabling implementations might use deeper indexes. We nevertheless see quite some speed advantage, because even if the indexing isn't deep, scanning becomes cheap, because of our pre-computed hashes:

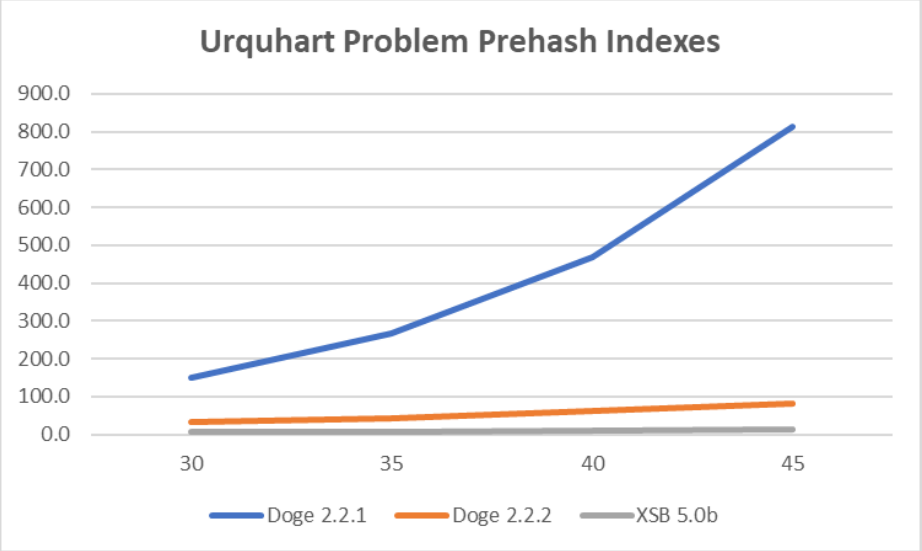
```
/**
 * Determine whether two terms unify.
 */
public static boolean unify(Object first, Object second) {
    [..]
      if (is_frozen(first) && is_frozen(second) &&
            ((Frozen)first).hash != ((Frozen)second).hash)
          return false;
```

The pre-computed hashes have no depth limit; the hash was pre-computed via hash-consing for the full depth of the frozen term. The hash function also knows no left or right, since it treats last and first arguments of frozen terms equally well by the way hash values are scrambled together.

We noticed such an advantage already when we experimented to with library(sequence) and when we compared our hash table based distinct/1 implementation to the trie based implementation by SWI-Prolog. It was 2x times faster. Subsequently we now offer an alternative indexing:

```
:- prehash(bdd_node_r/4).
:- prehash(bdd_node_r/4).
```

To take advantage of the prehash/1 indexes with have to go to n=30, 35, 40, 45. The scanning effort O(N) then because more dominant, while the prehash/1 indexes stay close to O(1). There are also effects concerning the size of the frozen terms, but using MAX_INDEX=2 we already find:




The prehash indexes lower once again drastically the time complexity and do change asymptotic function of the time complexity as well. The memory footprint increases since the maps are now more precise, now carrying hash codes. The method is easier to implement than tries.

We could run the test case also against XSB. We used the option ‘as intern’ in their tabling directive. XSB is again an itch faster ca. a factor 5x, possibly because it doesn’t have the baggage of potential big integers and in the Java target case, that we need to use Integer.valueOf() to access the maps.

### 6.6 Summary

The example is limited to xor, which has associativity (A xor B) xor C = A xor (B xor C) and commutativity A xor B = B xor A. We can demonstrate what a normal form delivers. The prehashes, and not some tries, lower the time complexity, but the XSB intern option is still 5x times ahead.

## 7 Conclusions

The different route taken by Dogelog Player paid out for the last demonstration example of BDD trees. We could show what Minato meticulously wrote down as a caching for its Boolean decision trees, can boil down to a table/1 directive. This is even more astonishing since we do not rely on global interning. We rather do it the Java way, in that the hashCode() contract typically yields a locally computed value. Our approach is also not an afterthought to tries, as it is the case in the interning proposal by XSB. It's rather the primary mechanism of indexing. But the limitations are similar as in XSB, that only ground terms receive the best speed-up.

## References

1. Warren, D.S., *Interning Ground Terms in XSB*, 17 Jul 2013
https://arxiv.org/html/1307.4644v1

2. Minato, S., *Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems*, 1993
https://ieeexplore.ieee.org/document/1600231

## Appendix

Prolog text from library(edge/silkworm):

```
/**
 * table(I):
 * table(I, O):
 * The directive succeeds. As a side effect the predicate I
 * is registered for tabling, preparing a query cache, preparing
 * an answer cache and creating a wrapper. Subsequent clauses
 * for the predicate will be called from the wrapper. The
 * binary predicate allows specifying tabling options.
 */
% term_expansion(+Clause, -Clause)
:- multifile(term_expansion/2).
term_expansion(T, R) :- nonvar(T), T = (:- D),
   nonvar(D), D = table(F/N), !,
   functor(P, F, N),
   sys_make_tabling(P, C, 0),
   R = [(:- multifile('$TABLE'/2)),
        (:- discontiguous('$TABLE'/2)),
        (:- dynamic('$TABLE'/2)),
        '$TABLE'(F,N) | C].
term_expansion(T, R) :- nonvar(T), T = (:- D),
   nonvar(D), D = table(F/N, Opts), !,
   functor(P, F, N),
   sys_table_opts(Opts, 0, Flags),
   sys_make_tabling(P, C, Flags),
   R = [(:- multifile('$TABLE'/2)),
        (:- discontiguous('$TABLE'/2)),
        (:- dynamic('$TABLE'/2)),
        '$TABLE'(F,N) | C].
term_expansion(T, (J :- B)) :- nonvar(T), T = (H :- B),
   functor(H, F, N),
   clause('$TABLE'(F, N),true), !,
   sys_make_pred(H, '_i', J).
term_expansion(H, J) :-
   functor(H, F, N),
   clause('$TABLE'(F, N),true), !,
   sys_make_pred(H, '_i', J).

/*************************************************************/
/* Functor Utility                                           */
/*************************************************************/
```

```
% sys_make_tabling(+Callable, -Clauses, +Integer)
sys_make_tabling(P, [D1, D2, C1, C2], Flags) :-
   sys_make_pred(P, '_i', J),
   sys_make_pred(P, '_k', K),
   sys_make_pred(P, '_r', I),
   sys_make_decl(K, D1, Flags),
   sys_make_decl(I, D2, Flags),
   sys_make_absent(K, G1),
   sys_make_absent(I, G2),
   /* if novel query, compute and store novel answers */
   C1 = (P :- G1, J, G2, fail),
   /* retrieve answers */
   C2 = (P :- !, I).

% sys_make_absent(+Callable, -Goal)
sys_make_absent(K, (G1, G2)) :-
   /* lookup unit */
   G1 = (\+ K),
   /* store unit */
   G2 = assertz(K).

% sys_make_pred(+Callable, +Atom, -Callable)
sys_make_pred(P, A, Q) :-
   P =.. [F|L],
   atom_join(['sys_',F,A], G),
   Q =.. [G|L].

% sys_make_decl(+Callable, -Clauses, +Integer) :-
sys_make_decl(K, [D1, D2], Flags) :- Flags /\ 1 =\= 0, !,
   functor(K, F, N),
   D1 = (:- dynamic(F/N)),
   D2 = (:- prehash(F/N)).
sys_make_decl(K, D1, _) :-
   functor(K, F, N),
   D1 = (:- dynamic(F/N)).
```