The External Storage Facility in SICStus Prolog

by

Hans Nilsson
The External Storage Facility in SICStus Prolog

Hans Nilsson

Swedish Institute of Computer Science, SICS
Box 1263, S-164 28 Kista, Sweden

and

ELLEMTEL AB
Box 1505, S-125 25 Älvsjö, Sweden

23 August 1991

R91:13

Abstract

The SICStus Prolog External Database implements an efficient way of storing possibly non-ground Prolog terms on disk with indexing on user-specified parts of the terms. The model and the implementation are described and some performance data are given.
Contents

1 Introduction 2
2 Background 2
3 Notation in this Report 3
4 The Design 3
   4.1 Superimposed Codewords 3
   4.2 Dynamic Hashing 4
   4.3 The algorithm in SICStus 5
5 The Implementation 9
   5.1 Memory Administration 9
   5.2 Language Details 10
6 Performance 10
   6.1 Method of Measurement 10
   6.2 The Hardware 11
   6.3 Results 11
7 Conclusions 14
8 Future work 14
9 Acknowledgements 15
References 15
A Key Generation 16
B Files and How They Interact 18
   B.1 The basic files 18
   B.2 The Cache 18
   B.3 "h - files" 19
   B.4 make and the config.c file 19
C Debugging 21
D Diagrams 27
1 Introduction

This work is a part of the project Industrialization of SICStus Prolog [1] which is a project aiming at making an industrial quality prolog system.

The part External Database has as goal to make a "continuation of the Prolog database out onto the disk". That is, to be able to handle disk storage of Prolog terms efficiently.

The goals of this report is twofold, namely to:

1. describe the method and to give some performance results
2. describe the implementation

The user interface to the database is described in [1]

2 Background

There are two common ways of enhancing a Prolog with an external database:

- Attaching an existing DBMS (Data Base Management System) to Prolog.
- Making a special handler, integrated with the Prolog system.

The first method — attaching a DBMS — gives functions like transactions, rollback, multiuser support etc with a minimal effort. Another advantage is that Prolog can be used for investigating already existing databases. A (sometimes big) disadvantage is that it is very difficult to use the flexible data structures of Prolog.

In a relational database, the Prolog structure \( f(a,b,c) \) is usually interpreted as a tuple with the values \((a,b,c)\) in the relation \( f \). If an argument is a Prolog variable or a structure, it is very difficult to represent that in the database. In practice, one is limited to using structures with atoms as arguments.

An overview of Prologs attached to DBMSs can be found in [4].

The other method — making a special handler — gives the possibility of using any Prolog term giving the power of a general language. However, we miss the great development efforts already put into DBMS and we can't process already existing data.

We can however write programs that uses full Prolog terms containing variables, lists structures etc. The indexing mechanism could be made more flexible and take care of the special Prolog situation.

This approach is taken in NU-Prolog [3].

Other work include Aditi [11] were they build a deductive DBMS with a Prolog-like query language.

SICStus already has an interface to the DBMS Oracle [9]. This report describes the special database handler that was developed for SICStus.

Note that the SICStus External Database is optimized for retrieving one tuple a time, while relational DBMS retrieves a whole relation. This makes the SICStus External Database suited for "Prolog-like" applications.
3 Notation in this Report

The following notational conventions are followed in this report:

- The SICStus Prolog syntax is used. Terms are written with this style: \( f(a, b, c) \).

- Functions involved in key generation are written with the function symbol in calligraphic style (CHV) and the argument part in the style used by the argument: \( C(f(a, b, c)) \) (The function \( C \) with a Prolog term as argument) or \( C(a_i) \).

- The symbol \( \oplus \) is a binary operator combining two keys into one.

4 The Design

The SICStus external database uses a modified version of the superimposed codewords in NU-Prolog [10, 2] and dynamic hashing (survey in [5]).

4.1 Superimposed Codewords

Superimposed Codewords is a way resembling hash coding of making integer keys to compound terms [7].

It works briefly in the following way in NU-Prolog [10] (which is modified in the SICStus External Database): the key is formed out of many components in the stored item in such a way that one can later check if some component was present. It is therefore possible to store say \( f(a, b) \) and later search for a term such that it contains for example \( f \) and a and get a match.

To each stored term is an integer key assigned. The key is built by combining hash values from each atomic component in the term. There is also a part describing where there are variables in the term.

For each atomic component \( a_0, \ldots, a_n \) of the term \( T \) to be indexed a hash value \( H(a_i) \) is calculated. For each variable \( v_0, \ldots, v_m \) in \( T \) a variable mask \( V(v_j) \) is calculated.

The codeword \( C(T) \) will be:

\[
C(T) = \bigoplus_{i=0}^{n} H(a_i) \oplus \bigoplus_{j=0}^{m} V(v_j)
\]  

(1)

where \( \oplus \) combines two keys into one. A binary representation of \( H(a_i) \) and \( V(v_j) \) should have only a few bits set so that \( C(T) \) differs for different terms \( T \).

The operator \( \oplus \) must preserve information about the arguments that are combined, that is:

For \( \oplus \) in (equation 1) there exist a logical function \( f_B \) such that

\[
f_B(C(T), H(a_i)) = \text{true}, \quad 0 \leq i \leq n
\]  

(2)
This codeword and the term are stored. When asking a query, an expression is generated which tells what bits must be set in the codeword to make the term a potential match. The stored codewords are investigated to find out the set of potential matches and they are read on backtracking to find the terms which actually unifies the query.

In [10] some optimizations are described which improves this linear search strategy significantly.

Example 1: Codewords.
Assume the term is \( f(a(1)) \) with indexing on all elements, that is, of \( f, a \) and \( 1 \).
The codeword of the term will be:

\[
C(f(a(1))) = \mathcal{H}(f) \oplus \mathcal{H}(a) \oplus \mathcal{H}(1)
\]

Now assume the term to be indexed is \( f(a(X)) \). The codeword will be:

\[
C(f(a(X))) = \mathcal{H}(f) \oplus \mathcal{H}(a) \oplus \mathcal{V}(X)
\]

Example 2: Operator.
The operators \( @ = \text{bitwise OR} \) and \( f_{\oplus} = \text{bitwise AND} \) fulfills the requirements (equations 1 and 2) above because \( f_{\text{OR}}(a \text{ OR } b, a) = (a \text{ OR } b) \text{ AND } a = \text{true} \).

4.2 Dynamic Hashing

*Dynamic Hashing* [6, 5] is a collection of methods for dynamically changing hash coding schemes when the table grows. This is done without rehashing of the old contents.

In figure 1 is the principle of a directory based scheme shown. This is the basic idea for the version in SICStus. The left part shows a directory with pointers to buckets. The buckets stores the key and the data. When searching for a key, the directory is indexed with the last bits in the key and the the addressed bucket is searched for the key. In the figure, all even keys are stored in the upper bucket and all odd keys in the lower one.

When a bucket is full, the directory could expand (see figure 1 again). That means that the directory doubles it size and all pointers are copied from the upper half to the lower half. The bucket which was full is split, that is it is divided into the old bucket and a new one. In the figure the entry at 1 is split into the entries 01 and 11. Now we use twice as many bits at the end of the key, but some entries point to the same bucket (00 and 10 in figure 1).

After some expansions, there will be many buckets having multiple pointers to them. If such a bucket is full, it can simply be split without the need of first expanding the directory. An example of this is shown in figure 2.

In the directory an integer is stored called *global depth* whose value is the number of bits used for indexing the directory. Every bucket has a *local depth* telling the number of bits
used for access of it. When a bucket is full, if the local depth is less than the global depth the bucket will be split, otherwise the directory will expand.
In [5] more details are presented as well as some more efficient directory-less methods.

4.3 The algorithm in SICStus

4.3.1 Modifications to the Superimposed Codewords Algorithm

In SICStus, the algorithm is modified in the following way:

- When storing a term, all possible query keys are generated by replacing non-variables with variables so that all possible (matching) queries are found. In appendix A is some Prolog code describing the key generation.

- When retrieving a term, the key is generated for the query. Since all possible query keys were stored with the wanted term, ordinary index mechanisms can be used for finding it. When it is fetched, it will be unified with the query. If that fails, the next term with the same key is read and so on.

In this way we get rid of the linear search when retrieving terms but we store more keys and we also get a more complicated index structure.

The problem with this method is that we can get very many keys for each term if we have a complicated Specification. That leads to much data stored and much time needed for the key generation during store and fetch. There is also a limit in the length of the variable mask on how many parts of a term that may be indexed.

Example 3: A simple database.
We will store the terms $f(a)$ and $g(b)$ in the database.
For \( f(a) \) the keys are: \( C(f(a)) \), \( C(f(\_)) \) and \( C(\_) \).

For \( g(b) \) the keys are: \( C(g(b)) \), \( C(g(\_)) \) and \( C(\_) \).

The tree first keys are stored with a reference to the first term and the three latter terms are stored with a reference to \( g(b) \). Note that at least two of the keys have equal values.

Assume a query \( f(X) \). The key \( C(f(X)) \) is generated and the retrieve predicate returns the first stored term with that key. In our case it could be \( f(a) \). \( f(X) \) is unified with \( f(a) \) which succeeds and binds \( X \) to \( a \).

On backtracking, the predicate returns the next term with the key \( C(f(X)) \) which might be \( g(b) \). The unification fails, the predicate looks for more terms with this key but fails and the whole predicate fails.

\( \Box \)

4.3.2 The Indexing Structure

The dynamic hashing described above assumes that each stored item has only one key each. In the SICStus approach the situation is a bit more complicated since we have many keys for each term.

The SICStus structure is best described by following a fetch and a store operation. See also figure 3.
Fetch  We want to fetch a term $T$ which unify with the term (the query) $Q$:

1. The key $C(Q)$ is generated for $Q$.
2. The last bits of the key is used to index the directory. The number of bits to use is given by the global depth.
3. Each entry in the directory points to a c-bucket. There could be more than one pointer to each c-bucket because of the dynamic hashing scheme.
4. The c-bucket is a collection of pairs of keys and references to p-buckets. All keys in one c-bucket ends with the same bitpattern. A binary search will find the right key (if it is present).
5. The p-bucket is a collection of references to terms. There is only one pointer to each p-bucket. Therefore one p-bucket corresponds to exactly one key.
6. The term is a representation of the Prolog term. It has many pointers to it since there are many keys for each term.
7. The term is read and unified with $Q$. If they unify we have of course a match. If not, we read the term pointed to by the next entry in the p-bucket and try to unify that one and so on.
8. All unifying terms can be found by trying all terms pointed to by the p-bucket (backtracking on the Prolog level).

Store  We want to store a term $T$. All keys for $T$ are generated and for each key we do:

1. Find the c-bucket as for a fetch.
2. Binary search the c-bucket for the key. If it is present, store the pointer to the term in the referenced p-bucket. If the key is not present, insert it and allocate a p-bucket in which the term pointer is stored. For overflow handling, see section 4.3.3.

4.3.3 Overflow Handling

C-buckets  C-buckets are fix-sized tables which can’t be expanded. When they are full and we try to insert something in them we have an overflow. It is important that this is handled efficiently.

When we want to insert a key in the middle of a c-bucket something called pushing starts. When the place where to insert the key have found everything below it is pushed one step down. This will cause some items to be pushed “over the border” to the next c-bucket. If this happens more than a certain number of times in the same insert operation, we give up and try to split this c-bucket. If it can be split, it is likely that we get at least one free position to insert the item in. If it can’t be split, we try to expand the directory. This will fail only if it has reached its maximum size.

If neither the split nor the expansion succeeds, we insert a spill c-bucket in the chain. This is to prevent lengthy pushes of maybe several thousands of items.
**P-buckets** Since we don't need to maintain any internal ordering, new p-buckets are simply linked to the previous ones when there is need for more space.

4.3.4 **Deletions**

It is complicated to delete a term. That is because there might be choice points that could cause the term to be read on backtracking. Therefore deleted terms are just marked as unavailable. They may be physically removed by some kind of compaction routine.

4.3.5 **The Specification**

A *Specification* defines which parts to index on in a term. The syntax of a *Specification* is:

- the atom on is a *Specification*
- the atom off is a *Specification*
- a structure with the function symbol on or off and every argument being a *Specification* is a *Specification*

The following table defines the key $C(S, T)$ as a function of the *Specification* $S$ and the *Indexed term* $T$.  

8
<table>
<thead>
<tr>
<th>$S$</th>
<th>$T$</th>
<th>$C(S,T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>atomic</td>
<td>Hash value $C(T)$</td>
</tr>
<tr>
<td></td>
<td>variable</td>
<td>Variable mask $V(T)$</td>
</tr>
<tr>
<td></td>
<td>$F(...)$</td>
<td>$C(on,F)$</td>
</tr>
<tr>
<td>off</td>
<td>any</td>
<td>0</td>
</tr>
<tr>
<td>$S_0(S_1,S_2,...,S_n)$</td>
<td>$A_0(A_1,A_2,...,A_m)$</td>
<td>$\bigoplus_{i=0}^{\min(n,m)} C(S_i,A_i)$</td>
</tr>
</tbody>
</table>

**Example 4: Specifications.**

The following example shows some specifications, terms and what parts that are indexed in them.

Remember that the Prolog list notation $[a,b,c]$ is shorthand for $(a, (b, (c, [])))$.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Term</th>
<th>Indexed parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>on(on)</td>
<td>$a(b)$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$a(b,c)$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$[a,b(c),d]$</td>
<td>$a$</td>
</tr>
<tr>
<td>off(on)</td>
<td>$a(b)$</td>
<td>$b$</td>
</tr>
<tr>
<td></td>
<td>$a(b,c)$</td>
<td>$b$</td>
</tr>
<tr>
<td></td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$[a,b(c),d]$</td>
<td>$a$</td>
</tr>
<tr>
<td>on(off,on(on(on)))</td>
<td>$a(b)$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$a(b,c)$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$a(b,c,d)$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$a(b,c(d))$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$[a,b,c]$</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td>$[a,b(c),d]$</td>
<td>$b$</td>
</tr>
</tbody>
</table>

Note that the Specification in the first example, on(on), is the one used in the internal SICStus database.

□

5 The Implementation

5.1 Memory Administration

All of the indexing structure is stored in one virtual address space with special accessing and allocation routines. This virtual space is stored on a number of UNIX files.

A special "software cache" is implemented (see figure 4). All accesses to the virtual space is passed through this cache.

This software cache is handled in the following way: Every time a function starts using a structure, it calls a macro which translates the virtual address to a physical address. If
the virtual address is not in the cache, it is read from a file into the cache. The physical address is always an address into the cache. Care must be taken to call that macro again when the function has called another function (which may have changed the cache). When an update is completed changed pages of the cache is flushed to the disk.

5.2 Language Details

The external database is implemented as a module in a library. It is written both in Prolog and in C with the foreign function interface in SICStus.

Key generation, indeterminism and checking of user supplied parameters are done in Prolog. The rest (dynamic hashing and file accesses) are done in C.

The internal organization is explained in more detail in appendix B.

6 Performance

6.1 Method of Measurement

The performance is measured with a Prolog program that accesses a database. The program stores and reads terms in 16 procedures. The terms are foo0(.,0), foo1(.,1),..., foo15(.,15), foo0(.,16).... The Specification is on(off,on).
The 1000 first terms are stored and the time is measured. 1000 terms randomly selected among the stored ones are read and the time is measured again. This repeats with the next 1000 terms and so on. Note that the read terms are always selected among all terms, not just the 1000 last ones.

The times are reduced by the time the prolog program ("the user program") uses so only the time in the database predicates are measured.

Three types of time could be measured:

1. the "user" time in the UNIX process
2. the "system" time in the UNIX process
3. the real elapsed time (called the "runtime" in this report)

Times measured with 1 and 2 don't differ much. Number 1 measures exactly the part of time the indexing structure imposes. Number 3, however, includes the file access time in UNIX. Unfortunately also other processes contribute to this time. I have however tried to keep the load down on the machine running the test.

In diagram 1 (appendix D) we can see that the indexing time is nearly constant regardless of the database size. This shows that the dynamic hashing performs well. Since the time is almost constant, there is probably no need for a directory-less [5] approach of the dynamic hashing.

6.2 The Hardware

All tests except the Oracle timings are done on the same machine. The machine is a SUN-3/60 with 8 Mbyte memory and a local disk. The local disk have a mean positioning time of 16 ms, latency time 8.3 ms and a data transfer rate of 3.0 Mbyte/s.

6.3 Results

All diagrams referred to in this section appear in appendix D.

6.3.1 SICStus External Database and Oracle

A fair comparison is difficult to do. Oracle is a commercial multiuser relational DBMS running as a server in a network while the SICStus External Database is (by now) single-user and running locally on one machine. Also, the test run is very bad for the relational model but well suited for the Prolog-like model so the comparison is a bit unfair to Oracle. A test well suited for the relational database model couldn't be done because of lack of time.

In this test, the same test program was run as in the other tests but stores and fetches were routed to Oracle. The machine was a sun4. The runtime for the Oracle run and for a calibration run with the SICStus External Database are shown in diagram 2. The unit of the x-axis is number of terms, since this is a comparable measure of the size.
We see that the runtime is much better for the SICStus External Database in this interval. But what happens above this size? Oracle is made for handling larger databases than just a few thousands terms ($\approx 1$ Mbyte).

Assume that the access time of Oracle is constant, and that the quote of the times for sun3 and sun4 is equal for both Oracle and SICStus in the whole interval. We scale the Oracle times and get the result in diagram 3. We see that the SICStus External Database still is faster than Oracle under those assumptions (and limitations).

### 6.3.2 External vs Internal Database

For databases larger than a few thousand terms, the external database is faster than the internal database if the indexing is NOT used in the internal one (diagram 4).

If indexing is used in the internal database, that one is much faster than the external one (as it should be).

The external database uses more space than the internal one (diagram 5). The used space is however a linear function of the number of terms in the database (if the terms are of equal length).

### 6.3.3 Dynamic Hashing

The dynamic hashing has a great impact on the performance compared with a large static directory (diagram 6). Those data was measured with:

1. The dynamic hashing described in this paper.

2. The same index structure, but the directory was fixed to $2^7$ entries during the test. No c-buckets were allocated until they were actually needed, so no splitting occurred.

Both the insert and retrieve times were almost constant when dynamic hashing was used. The insert was slower for small databases with dynamic hashing. This is because of the expand and split mechanism which takes more time than the gain for small databases. The retrieve time was nearly always better with dynamic hashing than without because the index structure had adopted to the stored keys.

There is also a small space gain for small databases with dynamic hashing (diagram 7). The reason for this is that initially all keys are entered in the same c-bucket. Without dynamic hashing there will initially be many c-buckets with few keys in each. There were no space differences if there were more than about 20,000 terms stored.
6.3.4 The Cache Organization

The cache size is in both cases 512 kByte but organized as 512 1 kByte pages in one case and 2 x 256 1 kByte pages in the other.

Test runs with the different cache organizations show no differences on the speed (not shown in the diagrams). The hit ratio doesn’t differ much either (diagram 8). This might be an effect of the test method that doesn’t impose much locality. Normally a 2-way cache should give better performance.

However, the hitratio is much better with dynamic hashing than without (diagram 8). This is probably an effect of that without dynamic hashing there are lots of c-buckets scattered all over the address space. With dynamic hashing, they tend to be fewer and more filled.

The hitratio is decreasing with increasing database size because of the measurement method. Remember that the database is expanding and that the accesses are randomly distributed all over it. Therefore the working set will grow with the size of the database.

6.3.5 The Hashfunction

In this test, different hashfunctions were tested. The terms were different from the other tests: test(abc0), test(abc1) … test(abc6000). The Specification was on(on). The tested algorithms were:

- Just the sum of ascii values. This is expected to be poor [8].

- The product of the ascii values. This should be better since the values are uniformly distributed ([8]).

- The sum used as seed for a pseudo random number generator. This was proposed in [10]. The reason was to get a codeword with just a few bits set.

- The product used as seed.

- As a “best value” the keys were leap numbered so they all got a unique key. This is of course not useful in practice.

The best was of course the leap numbering but the product was just a bit slower (diagram 9).
7 Conclusions

The important parts for the performance are:

- Indexing on relevant parts of the terms.
- The choice of hash function.
- The dynamic hashing scheme.

The method of generating all possible query keys performs well for a "small" Specification.

The software cache organisation (1- or 2-way) does not influence the time.

8 Future work

There are some areas were no or little work is done:

- Security. If something happens during the short time of a store or a delete the database could become inconsistent and maybe impossible to read.

- Converting a real application to this database handler. Natural language systems may be good examples since they have big databases and do not compute relations. Porting an application which does much computation of whole relations is not a good idea, since the model presented in this paper aims towards tuple-a-time retrieval. Relational operations will include bagof and will be costly in time.

- Multi user support. Today there are no locks on reading nor writing which limits the usefulness.

- A compaction routine is necessary to reclaim space occupied by deleted terms.

- A database created on one machine is portable to another machine if they both have the same "endianity"\(^1\). The "endianity" could be made unimportant by encapsulating pointer and integer reading from the cache with a macro that will be differently defined on machines of different "endianity".

Higher levels is an interesting subject. Here one could add for example transactions and metainterpreters with various search strategies. Integrity constraints and query optimization also belongs to this part.

\(^1\)By "endianity" we mean that different CPUs treats the bytes in words in different order
9 Acknowledgements

I am grateful to Seif Haridi for the valuable ideas about the cache mechanism and to Mats Carlsson for many good comments on the user predicates.

The Industrial SICStus Prolog team (Stefan Andersson, Kent Boortz, Nils Hagner and Thomas Sjöland) have contributed with many ideas and comments during the work.

Finally, Roger Skagervall at ELLEMTEL has given me the database which I have modified to make the test programs.

References


A  Key Generation

The set $S$ of keys for the term $T$ are all the keys $K$ obtained by

$$\text{setof}(K, X^\ast\text{term}(T,X)\text{,key}(X,K)), S).$$

term/2 and key/2 are defined by:

\%-------------------
\% term(+Term, -MoreGeneralTerm)
\% Gives all terms more general than Term by backing
\%

```
term(T,T) :-
    atomic(T).
term(T0,T) :-
    structure(T0),
    functor(T0,F,A),
    functor(T,F,A),
    args_term(A,T0,T).
```

```
args_term(0,_,_).
args_term(N,T0,T) :-
    N>0,
    arg(N,T0,Arg0),
    arg(N,T,Arg),
    term(Arg0,Arg),
    N1 is N-1,
    args_term(N1,T0,T).
```

\%-------------------
\% key(+Term, -KeyForTerm)
\%

```
key(T,K) :-
    atomic(T),
    !,
    hash_function(T,K).
key(T,K) :-
    structure(T),
    !,
    functor(T,F,A),
    key(F,K0),
    args_key(A,T,K0,K).
key(_,0).
```

```
args_key(0,_,K,K).
args_key(N,T,K0,K) :-
    N>0,
    arg(N,T,Arg),
```
key(Arg, KArg),
  K1 is KArg \ K0, % \ is inclusive-or
  N1 is N-1,
  args_key(N1, T, K1, K).

%---------------------
% structure(+Term)
%  Succeeds only if Term is a structure

structure(T) :- functor(T, F, A), atom(F), integer(A), A>0.

Please note that this code is not efficient and that it only shows the main principle. In the implementation the term/2 and key/2 are combined into one predicate which also handles the Specification and a variable mask which indicates if and where there are variables in the Term.

The Specification is brought in an extra argument together with the Term and taken apart together with the Term. Arguments not indexed are just skipped.

The generation of variable masks is not included either. The principle is that the predicates brings a mask (an integer) which is updated when a variable is encountered.
B Files and How They Interact

B.1 The basic files

The most important files and how they interact are shown in figure 5.
All user callable predicates are in the file db.pl. These predicates call predicates in db_foreign.pl which is just an interface to the C-functions in the other files.
db_codeword.pl is used for key generation and also for checking the Specification when a database is opened.
The main c-file is db_top.c. It contains the top functions for open, close, store, fetch and delete. It uses mainly two other files: db_insert.c which takes care of the dynamic hashing, and db_file.c which handles the file and directory creation or checking during the opening operation.
db_insert.c uses the help routines in db_alloc.c when it needs a new area in the virtual space. Those routines are on a higher level than for example malloc in C. They allocate a structure and initialize it to empty. All important structures (e.g. c_bucket, p_bucket) have their own routines.

B.2 The Cache

The cache is mainly implemented in db_cache.c. The functions in that file are called only in the following cases:

- A new piece of “raw” memory space is allocated.
- An area is deallocated.
- The cache must read in a new page from the disk. (That is, when an access is done to a part of the memory not used for a while. This is done by calling the macro CACHE_PTR. See below.)
- All “dirty” pages must be flushed to the disk. (After a completed store or delete operation).

A virtual 32 bit address is composed of 3 parts: the most significant bits forms a tag, the middle bits a page number and the least significant bits is an offset in the page. The page selects a part of the cache, the tag is compared with the tag stored in the cache and if they are equal the offset selects the proper part of the page. If the tags differ, a page is read from the disk.

An area can be declared to be single-page or multi-page when allocating it. If it is single page, it must fit on one page. If it is multi-page it may be on several consecutive pages. In the latter case, we must always call the CACHE_PTR macro before an access, even if we called it on the line before. The directory is a multi-page area, the other ones are single-page. This is because the directory can be greater than one page and we normally access only one word in it. The buckets are however accessed many times in the same function call and therefore must have a fast access scheme.
B.3 "h - files"

The two main declaration files are db_decl.h and db_parameters.h. The latter has parameters like the size of the pages in the cache, the number of pages, one or two way cache, parameters for the splitting and expansion etc. The common types and macros are defined in db_decl.h.

The most important macro in db_decl.h is the CACHE_PTR-macro. It converts a virtual address to a physical one (always in the cache). If the page is present in the cache, it just returns a physical pointer to it. If it is not present it calls a function in db_cache.c which loads that page from the disk. That operation might cause another page to be written to the disk. It is therefore necessary to re-call the macro when there has been a call to it in some other place, for instance when another function has been called.

B.4 make and the config.c file

The only strange thing with the Makefile is a configuration file, config.c.

When building the database handler, this file is compiled and run to produce the file db_config.c. During this run, there are checks that the given parameters are not contradictory. A structure may for example not be declared to be single-page and bigger than the page size. If an error is found it is reported and a variable in the generated file is initialized to false. This will force the db_open predicate to fail with an error message if one tries to use the database handler anyway.

The version and date of compilation is also present in this file as a string as well as a string with all the parameter settings.
Figure 5: The main files
C Debugging

There are some aids for finding bugs. The relevant flags are:

CHECK_MAGIC: If it is defined, some structures will have a 32-bit magic number as the first field. This will be checked at some accesses to ensure that the area has not been overwritten. (See db_magic.h).

CHECK_STRUCT: frequently forces a complete traversal of the whole indexing structure with checks of the magic numbers and range checks of the pointers. It is useful for finding the first point were a pointer has been corrupted.

DBG_FREE: forces a check of the free areas structure.

There are three predicates writing out the indexing structure:

\texttt{db:db\_print(?DBref)} prints the whole structure on stdout.

\texttt{db:db\_print(?DBref,+FileName)} as previous but prints on FileName.

\texttt{db:db\_print(?DBref,+FileName,+FileMode)} as previous but opens the file according to FileMode. A printout can be appended to a file by this predicate (useful for repeated printouts).

Example 5: A simple example of \texttt{db\_print}.
Assume the following:

\texttt{| ?- db\_open(xxx,update,D),set\_default\_db(D).}

\texttt{D = '$db'(1759140) ?}

\texttt{yes}

\texttt{| ?- db\_store(f(a),R).}

\texttt{R = '$db\_term'(1759140,292,0) ?}

\texttt{yes}

\texttt{| ?- db:db\_print(_).}

\texttt{----- xxx (update). Address=1759140.}

\texttt{Directory at 4. global\_depth=1, mask=1, num\_entries=2:}

\texttt{0: local\_depth=0, first\_free=4, next=0. Address=1024}

\texttt{ 0: -2147483648 292 292 [ 196]}

\texttt{ 1: 0 132 132 [ 68]}

\texttt{ 2: 3 228 228 [ 196]}

\texttt{ 3: 1073741875 260 260 [ 196]}

\texttt{1: see index 0}

\texttt{yes}
Starting with the line "" we can see: the name is "xxx". The mode is "update". The address of the struct db_node is 1759140 (An address obtained by malloc). Note that this is the integer in the reference returned by db_open.

Next line: The directory starts at virtual address 4. The global depth, mask and number of entries follows.

Next line: The index 0 in the directory points to a c_bucket with local depth 1. 4 positions are occupied (first_free = 4). There is no bucket linked to this one (next=0). The virtual address is 1024.

Next line: The index 0 in the c_bucket has the key -2147483648. The first p_bucket for that key has address 292 and the last one (the tail) has address 292. In brackets is a list of term addresses (virtual).

Next 3 lines: The following three keys (0,3,1073741875) with their respective term references.

Next line: "see index 0" means that this is a pointer to the c_bucket pointed to by index 0.

In this example we can see that the term has keys -2147483648, 3 and 1073741875. It is stored at virtual address 196. There is also another term stored at virtual address 68 with just one key (key=0). This is an information record that db_open uses. The intelligent reader has now drawn the conclusion that the term reference returned by db_store is composed of the database reference, the address to the p_bucket and the index in the latter. Right!

Example 6: A more complicated example of db_print.

After seven more storings of f(a) db:db_print(_) will produce:

----- xxx (update). Address=1759140.
Directory at 4, global_depth=1, mask=1, num_entries=2:
  0: local_depth=0, first_free=4, next=0. Address=1024
    0: -2147483648 292 708 [ 196 356 420 484 548 {708} 612 772
      836]
    1: 0 132 132 [ 68]
    2: 3 228 644 [ 196 356 420 484 548 {644} 612 772
      836]
    3: 1073741875 260 676 [ 196 356 420 484 548 {676} 612 772
      836]
  1: see index 0

The line for the key -2147483648 now tells us: the first p_bucket is still on virtual address 292 but its tail is on 708. The list of term addresses is bigger (seven more). Let's look at that list (inside the brackets): The first five terms are stored on addresses 196, 356, 420, 484, 548. The notation {708} means that a new p_bucket follows on virtual address 708. In this one are the term addresses 612, 772 and 836 stored.
Appendix D  Diagrams

Diagram 1: Total time and indexing time
NOTE: Times are measured on a sun-4!

Diagram 2: SICStus and Oracle

Diagram 3: SICStus and Oracle, extrapolated and normalized
Diagram 4: The External Database Compared with the Internal Database, Time

Diagram 5: The External Database Compared with the Internal Database, Space
Diagram 6: The Effect of Dynamic Hashing on the Time

Diagram 7: The Effect of Dynamic Hashing on the Disk Space
Diagram 8: Hit ratio

Diagram 9: Times for fetching as a function of db-size for different hash functions