Efficiency Guide

version 5.1

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

Chapter 1

Efficiency Guide

1.1 Introduction

1.1.1 Purpose

In the most perfect of all worlds this document would not be needed. The compiler would be able to make all necessary optimizations. Alas the world is not perfect!

All considerations for efficiency are more or less implementation dependent. Efficient code is not always good code from the perspective of generality, ease of understanding and maintaining. Therefor programming becomes a balance act between generality and efficiency. So how do we manage to walk on the lim and not fall off? Well, on a structural level there are things that you can keep in mind while you design your code. This guide will try to help you use data structures and mechanisms of Erlang/OTP in the intended way, this will help you avoid many unnecessary bottlenecks. Apart from those structal considerations, you should never optimize before you profiled your code and found the bottlenecks. Also remember not all code is time critical, if it does not matter if takes a few seconds more or less there is no point in trying to optimize it. Profiling erlang code is easy using tools such as `eprof`, `fprof` (from release 8 and forward) and `cover`. Using these tools are just a matter of calling a few functions in the respective library modules. Taking the appropriate measures to speed the code up once you found the bottlenecks can be a bit harder. You may have to invent new algorithms or in other ways restructure your program. This guide will give you some background knowledge to be able to come up with solutions.

> Note:
> For the sake of readability, the example code has been kept as simple as possible. It does not include functionality such as error handling, which might be vital in a real-life system. Inspiration for the examples is taken from code that has existed in real projects.

1.1.2 Pre-requisites

It is assumed that the reader is familiar with the Erlang programming language and concepts of OTP.

1.2 List handling

1.2.1 Creating a list

Calling `lists:append(List1, List2)` will result in that the whole List1 has to be traversed. When recursively building a list always attach the element to the begining of the list and not to the end. In the case that the order of the elements in the list is important you can always reverse the list when it has been built! It is cheaper to reverse the list in the final step, than to append the element to the end of the list in each recursion. Note that ++ is equivalent to lists:append/2. As an example consider the tail-recursive function fib that creates a list of Fibonacci numbers. (The Fibonacci series is formed by adding the latest two numbers to get the next one, starting from 0 and 1.)

```
> fib(4).
    [0, 1, 1, 2, 3]
```

DO

```
fib(N) ->
    fib(N, 0, 1, [0]).

fib(0, Current, Next, Fibs) ->
    lists:reverse(Fibs); % Reverse the list as the order is important

fib(N, Current, Next, Fibs) ->
    fib(N - 1, Next, Current + Next, [Next | Fibs]).
```

DO NOT

```
fib(N) ->
    fib(N, 0, 1, [0]).

fib(0, Current, Next, Fibs) ->
    Fibs;

fib(N, Current, Next, Fibs) ->
    fib(N - 1, Next, Current + Next, Fibs ++ [Next]).
```

1.2.2 Deleting a list element

When using the function delete in the lists module there is no reason to check that the element is actually part of the list. If the element is not part of the list the delete operation will be considered successful.

DO

```
...
NewList = lists:delete(Element, List),
...
```

DO NOT

```
...
NewList =
case lists:member(Element, List) of
    true ->
        lists:delete(Element, List);
    false ->
        List
end,
...
```

1.2.3 Unnecessary list traversal

Use functions like lists:foldl/3 and lists:mapfoldl/3 to avoid traversing lists more times than necessary. The function mapfold combines the operations of map and foldl into one pass. For example, we could sum the elements in a list and double them at the same time:

```
> lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end,
0, [1,2,3,4,5]).
{[2,4,6,8,10],15}
```

Also consider the function evenMultipleSum below where we make one list traversal in the first version and three in the second version. (This may not be a very useful function, but it serves as a simple example of the principle.)

DO

```
evenMultipleSum(List, Multiple) ->
   lists:foldl(fun(X, Sum) when (X rem 2) == 0 ->
                       X * Multiple + Sum;
                  (X, Sum) ->
                       Sum
               end, 0, List).
```

DO NOT

```
evenMultipleSum(List, Multiple) ->
    FilteredList = lists:filter(fun(X) -> (X rem 2) == 0 end, List),
    MultipleList = lists:map(fun(X) -> X * Multiple end, FilteredList),
    lists:sum(MultipleList).
```

1.2.4 Deep and flat lists

`lists:flatten/1` is a very general function and because of this it can be quite expensive to use. So do not use it if you do not have to. There are especially two senarios where you do not need to use flatten

- When sending data to a port. Ports understands deep lists so there is no reason to flatten the list before sending it to the port.
- When you have a deep list of depth 1 you can flatten it using `append/1`

Port example

DO

```
...
port_command(Port, DeepList)
...
```

DO NOT

```
...
port_command(Port, lists:flatten(DeepList))
...
```

A common way that people construct a flat list in vain is when they use append to send a 0-terminated string to a port. In the example below please note that “foo” is equivalent to [102, 111, 111].

DO

```
...
TerminatedStr = [String, 0], % String="foo" => [[102, 111, 111], 0]
port_command(Port, TerminatedStr)
...
```

DO NOT

```
...
TerminatedStr = String ++ [0], % String="foo" => [102, 111, 111, 0]
port_command(Port, TerminatedStr)
...
```

Append example

DO

```
> lists:append([[1], [2], [3]]).
[1,2,3]
>
```

DO NOT

```
> lists:flatten([[1], [2], [3]]).
[1,2,3]
>
```

Note:
If your deep list is a list of strings you will not get the wanted result using flatten. Remember that strings are lists. See example below:

```
> lists:append([["foo"], ["bar"]]).
["foo","bar"]
> lists:flatten([["foo"], ["bar"]]).
"foobar"
```

1.3 Functions

1.3.1 Pattern matching

Pattern matching in function, case and receive-clauses are optimized by the compiler. In most cases, there is nothing to gain by rearranging clauses.

1.3.2 Function Calls

A function can be called in a number of ways and the cost differs a lot. Which kind of call to use depends on the situation. Below follows a table with the available alternatives and their relative cost.

Note:
The figures shown as relative cost is highly dependent on the implementation and will vary between versions and platform. The order from lowest to highest cost will however be stable and is very useful to be aware of.

Type of call	Example	Relative cost (5.1)
Local call	`foo()`	1.00
External call	`m:foo()`	1.07
Fun call	`Fun = fun(X) -> X + 1 end, Fun(2)`	2.52
Apply fun	`Fun = fun(X) -> X + 1 end, apply(Fun,[2])`	3.32
Apply MFA/3	`apply(M, Foo, [])` or `M:Foo()`	7.09

Table 1.1: Different ways of calling a function

Apply is the most expensive way to call a function and should be avoided in time critical code. A well motivated use of apply is in conjunction with generic interfaces where several modules provide the same set of functions. The use of `apply/3` for just calling different functions within the same module (i.e `apply(mymodule,Func,Args)`) is not recommended. The use of Funs can often be a more efficient way to accomplish calls which are variable in runtime.

The last entry in the table above shows the syntax `M:Foo(A1,A2,An)` (where M and Foo are bound variables) which is equivalent with `apply(M,Foo,[A1,A2,An])`. From an efficiency point of view it is recommended to use the `M:Foo(A1,A2,An)` form since this gives the compiler an opportunity to optimize the call in future versions.

1.3.3 Memory usage in recursion

When writing recursive functions it is preferable to make them tail-recursive so that they can execute in a constant memory space.

DO

```
list_length(List) ->
    list_length(List, 0).

list_length([], AccLen) ->
    AccLen; % Base case

list_length([_|Tail], AccLen) ->
    list_length(Tail, AccLen + 1). % Tail-recursive
```

DO NOT

```
list_length([]) ->
    0. % Base case
list_length([_ | Tail]) ->
    list_length(Tail) + 1. % Not tail-recursive
```

1.3.4 Unnecessary evaluation in each recursive step

Do not evaluate the same expression in each recursive step, rather pass the result around as a parameter. For example imagine that you have the function in_range/3 below and want to write a function in_range/2 that takes a list of integers and atom as argument. The atom specifies a key to the named table range_table, so you can lookup the max and min values for a particular type of range.

```
in_range(Value, Min, Max) ->
    (Value >= Min) and (Value =< Max).
```

DO

```
in_range(ValuList, Type) ->
    %% Will be evaluated only one time ...
    [{Min, Max}] = ets:lookup(range_table, Type),
    %% ... send result as parameter to recursive help-function
    lists_in_range(ValuList, Min, Max).


lists_in_range([Value | Tail], Min, Max) ->
    case in_range(Value, Min, Max) of
        true ->
            lists_in_range(Tail, Min, Max);
        false ->
            false
    end;

lists_in_range([], _, _) ->
    true.
```

DO NOT

```
in_range([Value | Tail], Type) ->
    %% Will be evaluated in each recursive step
    [{Min, Max}] = ets:lookup(range_table, Type),
    case in_range(Value, Min, Max) of
        true ->
            lists_in_range(Tail, Type);
        false ->
            false
    end;

in_range([], _, _) ->
    true.
```

1.4 Tables and databases

1.4.1 Ets, Dets and Mnesia

All examples using Ets has an corresponding example in Mnesia. In general all Ets examples also applies to Dets tabels.

Select/Match operations

Select/Match operations on Ets and Mnesia tables can become very expensive operations. They will have to scan the whole table that may be very large. You should try to structure your data so that you minimize the need for select/match operations. However if you need a select/match operation this will be more efficient than traversing the whole table using other means such as `tab2list` and `mnemosyne`. Examples of this and also of ways to avoid select/match will be provided in some of the following sections. From R8 the functions `ets:select/2` and `mnesia:select/3` should be preferred over `ets:match/2`,`ets:match_object/2` and `mnesia:match_object/3`.

> **Note:**
> There are exceptions when the whole table is not scanned. This is when the key part is bound, the key part is partially bound in an `orded_set` table, or if it is a mnesia table and there is a secondary index on the field that is selected/matched. Of course if the key is fully bound there will be no point in doing a select/match, unless you have a bag table and you are only interested in a sub-set of the elements with the specific key.

When creating a record to be used in a select/match operation you want most of the fields to have the value '_'. To avoid having to explicitly set all of these fields people have created functions that takes advantage of the fact that records are implemented as tuples. This is not such a good idea, that is why you from R8 can do the following

```
#person{age = 42, _ = '_'}.
```

This will set the age attribute to 42 and all other attributes to '_'. This is more efficient then creating a tuple with '_' values, that then is used as a record. It is also much better code as it does not violate the record abstraction.

Deleting an element

As in the case of lists, the delete operation is considered successful if the element was not present in the table. Hence all attempts to check that the element is present in the Ets/Mnesia table before deletion are unnecessary. Here follows an example for Ets tables.

DO

```
...
ets:delete(Tab, Key),
...
```

DO NOT

```
...
case ets:lookup(Tab, Key) of
    [] ->
        ok;
    [_|_] ->
        ets:delete(Tab, Key)
end,
...
```

Data fetching

Do not fetch data that you already have! Consider that you have a module that handles the abstract data type Person. You export the interface function `print_person/1` that uses the internal functions `print_name/1`, `print_age/1`, `print_occupation/1`.

> **Note:**
> If the functions `print_name/1` etc. had been interface functions the matter comes in to a whole new light. As you do not want the user of the interface to know about the internal data representation.

DO

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(Person),
            print_age(Person),
            print_occupation(Person);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Interanal functions
print_name(Person) ->
    io:format("No person ~p~n", [Person#person.name]).

print_age(Person) ->
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(Person) ->
    io:format("No person ~p~n", [Person#person.occupation]).
```

DO NOT

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(PersonID),
            print_age(PersonID),
            print_occupation(PersonID);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Interanal functions
print_name(PersonID) ->
```

```
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.name]).

print_age(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.occupation]).
```

Non persistent data storage

For non persistent database storage, prefer Ets tables before Mnesia local_content tables. Even the cheapest Mnesia operations, `dirty_write` operations, carry a fixed overhead compared to Ets writes. Mnesia must check if the table is replicated or has indices, this involves at least one Ets lookup for each `dirty_write`. Thus, Ets writes will always be faster than Mnesia writes.

tab2list

Assume we have an Ets-table, which uses `idno` as key, and contains:

```
[#person{idno = 1, name = "Adam",  age = 31, occupation = "mailman"},
 #person{idno = 2, name = "Bryan", age = 31, occupation = "cashier"},
 #person{idno = 3, name = "Bryan", age = 35, occupation = "banker"},
 #person{idno = 4, name = "Carl",  age = 25, occupation = "mailman"}]
```

If we *must* return all data stored in the Ets-table we can use `ets:tab2list/1`. However, usually we are only interested in a subset of the information in which case `ets:tab2list/1` is expensive. If we only want to extract one field from each record, e.g., the age of every person, we should use:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                      name='_',
                      age='$1',
                      occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:map(fun(X) -> X#person.age end, TabList),
...
```

If we are only interested in the age of all persons named Bryan, we should:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                      name="Bryan",
                      age='$1',
                      occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:foldl(fun(X, Acc) -> case X#person.name of
                               "Bryan" ->
                                   [X#person.age|Acc];
                                _ ->
                                    Acc
                           end
            end, [], TabList),
...
```

REALLY DO NOT

```
...
TabList = ets:tab2list(Tab),
BryanList = lists:filter(fun(X) -> X#person.name == "Bryan" end,
                         TabList),
lists:map(fun(X) -> X#person.age end, BryanList),
...
```

If we need all information stored in the ets table about persons named Bryan we should:

DO

```
...
ets:select(Tab, [{#person{idno='_',
                        name="Bryan",
                        age='_',
                        occupation = '_'}, [], ['$_']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:filter(fun(X) -> X#person.name == "Bryan" end, TabList),
...
```

Ordered_set tables

If the data in the table should be accessed so that the order of the keys in the table is significant, the table type `ordered_set` could be used instead of the more usual `set` table type. An `ordered_set` is always traversed in Erlang term order with regards to the key field so that return values from functions such as `select` , `match_object` and `foldl` are ordered by the key values. Traversing an `ordered_set` with the first and next operations also returns the keys ordered.

> **Note:**
> An `ordered_set` only guarantees that objects are processed in *key* order. Results from functions as `ets:select/2` appear in the *key* order even if the key is not included in the result.

1.4.2 Ets specific

Utilizing the keys of the Ets table

An Ets table is a singel key table (either a hash table or a tree orded by the key) and should be used as one. In other words, always use the key to look up things when possible. A lookup by a known key in a set Ets table is constant and for a orded_set Ets table it is O(logN). A key lookup is always preferable to a call where the whole table has to be scanned. In the examples above, the field `idno` is the key of the table and all lookups where only the name is known will result in a complete scan of the (possibly large) table for a matching result.

A simple solution would be to use the `name` field as the key instead of the `idno` field, but that would cause problems if the names were not unique. A more general solution would be create a second table with name as key and idno as data, i.e. to index (invert) the table with regards to the `name` field. The second table would of course have to be kept consistent with the master table. Mnesia could do this for you, but a home brew index table could be very efficient compared to the overhead involved in using mnesia.

An index table for the table in the previous examples would have to be a bag (as keys would appear more than once) and could have the following contents:

```
[#index_entry{name="Adam", idno=1},
 #index_entry{name="Bryan", idno=2},
 #index_entry{name="Bryan", idno=3},
 #index_entry{name="Carl", idno=4}]
```

Given this index table a lookup of the age fields for all persons named “Bryan” could be done like this:

```
...
MatchingIDs = ets:lookup(IndexTable,"Bryan"),
lists:map(fun(#index_entry{idno = ID}) ->
                  [#person{age = Age}] = ets:lookup(PersonTable, ID),
                  Age
          end,
          MatchingIDs),
...
```

Note that the code above never uses `ets:match/2` but instead utilizes the `ets:lookup/2` call. The `lists:map` call is only used to traverse the `idno`'s matching the name "Bryan" in the table, why the number of lookups in the master table is minimized.

Keeping an index table of course introduces some overhead when inserting records in the table, why the number of operations gaining from the table has to be weighted against the number of operations inserting objects in the table. However that the gain when the key can be used to lookup elements is significant.

1.4.3 Mnesia specific

Secondary index

If you frequently do a lookup on a field that is not the key of the table, you will lose performance using "mnesia:select/match_object" as this function will traverse the whole table. You may create a secondary index instead and use "mnesia:index_read" to get faster access, however this will require more memory. Example:

```
-record(person, {idno, name, age, occupation}).
        ...
{atomic, ok} =
mnesia:create_table(person, [{index,[#person.age]},
                             {attributes,
                                   record_info(fields, person)}]),
{atomic, ok} = mnesia:add_table_index(person, age),
...

PersonsAge42 =
     mnesia:dirty_index_read(person, 42, #person.age),
...
```

Transactions

Transactions is a way to guarantee that the distributed mnesia database remains consistent, even when many different processes updates it in parallel. However if you have real time requirements it is recommended to use dirty operations instead of transactions. When using the dirty operations you lose the consistency guarantee, this is usually solved by only letting one process update the table. Other processes have to send update requests to that process.

```
...
% Using transaction

Fun = fun() ->
          [mnesia:read({Table, Key}),
           mnesia:read({Table2, Key2})]
          end,

{atomic, [Result1, Result2]}  = mnesia:transaction(Fun),
...

% Same thing using dirty operations
...

Result1 = mnesia:dirty_read({Table, Key}),
Result2 = mnesia:dirty_read({Table2, Key2}),
...
```

Mnemosyne

Mnesia supports complex queries through the query language Mnemosyne. This makes it possible to perform queries of any complexity on Mnesia tables. However for simple queries Mnemosyne is usually much more expensive than sensible handwritten functions doing the same thing.

> **Warning:**
> The use of mnemosyne queries in embedded real time systems is strongly discouraged.

Assume we have an mnesia-table, which uses `idno` as key, and contains:

```
[#person{idno=1, name="Adam",  age=31, occupation="mailman"},
 #person{idno=2, name="Bryan", age=31, occupation="cashier"},
 #person{idno=3, name="Bryan", age=35, occupation="banker"},
 #person{idno=4, name="Carl",  age=25, occupation="mailman"}]
```

If we need to find all persons named Bryan we should:

DO

```
...
Select = fun() ->
                 mnesia:select(person,
                               [{#person{name ="Bryan", _ = '_'}, [], ['$_']}],
                               read)
         end,

{atomic, Result} = mnesia:transaction(Select),
...
```

DO NOT

```
...
Handle = query
         [ Person || Person <- table(person),
                     Person.name = "Bryan"]
        end,
{atomic, Result} = mnesia:transaction(fun() -> mnemosyne:eval(Handle) end),
...
```

1.4.4 Older versions of Erlang/OTP

If you have a an older version than R8 of Erlang/OTP you would have to use match and match_object instead of select. The select call is introduced in R8 and is not present in earlier releases. Then the code would look as follows.

Selecting the age field:

```
...
lists:append(ets:match(Ets, #person{idno='_',
                                    name='_',
                                    age='$1',
                                    occupation = '_'})),
...
```

The `lists:append/1` call above transforms the list of lists returned by `ets:match/2` into a flat list containing the values of the field age in the table.

Selecting people called Bryan:

```
...
ets:match_object(Ets, #person{idno='_',
                              name="Bryan",
                              age='_',
                              occupation = '_'}),
...
```

```
...

Match = fun() ->
           % Create record instance with '_' as values of the fileds
           Person = mnesia_table_info(person, wild_pattern),
           mnesia:match_object(person,
                               Person#person{name ="Bryan"},
                               read)
           end,

{atomic, Result} = mnesia:transaction(Match),
...
```

1.5 Processes

1.5.1 Creation of an Erlang process

An Erlang process is very lightweight compared to most operating system threads and processes but it is always important to be aware of its characteristics. Each Erlang process takes a minimum of 318 words of memory for heap, stack etc. The heap is increased in Fibonacci steps depending on data created by the program. The stack is increased by means of nested function calls and a non terminating recursive call will increase the stack until all memory resources are exhausted and the Erlang node will be terminated. The latter means that you need to write tail-recursive process loops.

DO

```
loop() ->
 receive
   {sys, Msg} ->
         handle_sys_msg(Msg),
         loop();
   {From, Msg} ->
         Reply = handle_msg(Msg),
         From ! Reply,
         loop()
 end.
```

DO NOT

```
loop() ->
 receive
   {sys, Msg} ->
         handle_sys_msg(Msg),
         loop();
   {From, Msg} ->
         Reply = handle_msg(Msg),
         From ! Reply,
         loop()
 end,

 io:format("Message is processed ~n", []).

 %% The last line in the example above will never be executed and
 %% will eventually eat up all memory.
```

Figure 1.1: ”Don’t buy too many spades!”

A good principle when deciding which processes you need is to have one process for each truly parallel activity in the system. Consider the analogy where you have three diggers digging a ditch, and to speed things up you buy a fourth spade. Alas that will not help at all as a digger can only use one spade at a time.

1.5.2 Process messages

All data in messages between Erlang processes is copied, with binaries between processes at the same node as the only exception. Binaries are shared between Erlang processes and only the reference to a binary is copied.

When a message is sent to a process on another Erlang node it is first encoded to the Erlang External Format and then sent on a tcp/ip socket. The receiving Erlang node decodes the message and distributes it to the right process.

Binaries

As there is no copying when sending a binary to a process on the same node, it might be relevant to have your message as a binary. Use binary form in messages if the cost for encoding/decoding to/from binary form can be expected to be less than the gain of transfering in binary form. Cases where binary form could be advantageous are when:

- The message size is very large.
- The message content is to be sent to many receivers.
- The message content will be forwarded unchanged in several messages.

Atom vs Strings

It is more efficient to send atoms than strings. However it is more inefficient to convert all strings with list_to_atom before sending them, then to send the string as it is. The best way is to always use atoms if possible.

DO

```
%% Send message on the following format
{insert, {Name, Location}}
{remove, Name}
{retrieve_location, Name}
```

DO NOT

```
%% Don't send message on the following format
{"insert", {Name, Location}}
{"remove", Name}
{"retrieve_location", Name}
```

1.6 Built in functions

1.6.1 Some notes about BIFs

list_to_atom/1 Since atoms are not garbage collected it is not a good idea to create atoms dynamically in a system that will run continously. Sooner or later the limit 1048576 for number of atoms will be reached with a emulator crash as result. In addition to the bad memory consumption characteristics the function is also quite expensive to execute.

length/1 is an operation on lists and each time the length is tested the entire list must be traversed. Since length is implemented in C it is quite efficient anyway but it still has linear characteristics. The size/1 function which can be applied on tuples and binaries is for example much more efficient since it only reads a size field in the internal data structure.

setelement/3 Compared with element/2 that is very efficient and independent of the tuple size setelement/3 is an expensive operation for large tuples (>50 elements) since it implies that all fields must be copied to a new tuple. Therefore it is not recommended to use setelement in recursions involving large tuples.

split_binary/2 Depending on the situation it is in most cases more efficient to split a binary through matching instead of calling the split_binary/2 function.

DO

```
<<Bin1:Num/binary,Bin2/binary>> = Bin,
```

DON'T

```
{Bin1,Bin2} = split_binary(Bin,Num),
```

1.7 Advanced

1.7.1 Memory

A good start when programming efficient is to have knowledge about how much memory different datatypes and operations require. It is implementation dependent how much memory the Erlang data types and other items consume, but here are some figures for the current erts-5.x beam system. The unit of measurement is memory words and as the current implementation is a 32-bit implementation a word is 32 bits.

Datatype	Memory size
Integer (-16#7FFFFF < i <16#7FFFFF)	1 word
Integer (big numbers)	2..N words
Atom	1 word
Float	3 words
Binary	2..5 + data
List	2 words per element + the size of each element
String (is the same as a List of Integers)	2 words per character
n-Tuple	(n + 1) words + the size of each element - 1
Pid	1 word
Port	1 word

continued ...

... continued

Reference	5 words
Fun	6 + environment
Ets table	initially 768 words + the size of each data record. The table will grow when necessary.
Erlang process	318 words when spawned

Table 1.2: Memory size of different datatypes

1.7.2 System limits

The Erlang language specification puts no limits on number of processes, length of atoms etc. but for performance and memory saving reasons there will always be limits in a practical implementation of the Erlang language and execution environment. The current implementation has a few limitations that is good to know about since some of them can be of great importance for the design of an application.

Processes The maximum number of simultaneously alive Erlang processes is 32768.

Distributed nodes The maximum number of distributed nodes that one Erlang node can connect to during its life time is 255. It can be less than 255 for several mostly platform dependent reasons, for example the maximum number of open file descriptors allowed for a process on the operating system.

Characters in an atom 255

Atoms The maximum number of atoms is 1048576.

Ets-tables default=1400, can be changed with the environment variable `ERL_MAX_ETS_TABLES`.

Elements in a tuple The maximum number of elements in a tuple is 67108863 (26 bit unsigned integer). Other factors such as the available memory can of course make it hard to create a tuple of that size.

Length of binary Unsigned

Total amount of data allocated by an Erlang node The Erlang runtime system can use the whole 32 bit address space, but the operating system often limits one single process to use less than that.

length of a nodename An Erlang node name has the form host@shortname or host@longname. The nodename is used as an atom within the system so the maximum size of 255 holds for the nodename too.

Open files, ports and sockets

Number of arguments to a function or fun

List of Figures

List of Tables