# Distributed Atom Space - Requirements

August/2018

This is an early-stage draft of a requirements specification for a Distributed Atom Space (DAS) considered as part of a future OpenCog version. This document is a compilation of many discussions carried out amongst several people in the SingularityNET AI Team.

## Table of contents

# Introduction

## Project Overview

In the next few years it is desirable that AtomSpace and the associated tools transition from being an R&D tool to a scalable, robust framework that can be used within a variety of demanding practical applications (as well as for research projects that involve intensive real-time interaction and/or huge amounts of data). This document gathers some requirements and ideas aimed at this transition.

## Business requirements

The high-level long-term goals for this project are making AtomSpace roughly as fast and as scalable as competing no-SQL knowledge-stores (preferably more so), while retaining its basic flexibility and suitability as an AGI platform. And preferably we want to do this without messing up nice stuff like the OpenCog NLP pipeline and the PLN chainers, which utilize the AtomSpace API. So the following business requirements must be met.

1. Provide effective management of AtomSpaces that are too big to fit in RAM of any one machine that is available.
2. Decrease the overall processing time required to carry out AI operations to reduce cost per AI operation.
3. Decrease memory footprint providing better overall throughput in comparing with current implementation to reduce cost per AI operation.
4. Provide ability to use AtomSpace in the manner of hierarchical cache structure. In other words, provide a way to look for a specific Atom locally before start searching it among other components and fetching remotely.
5. Provide ability to request for Atom(s) based on a given Atom's property.
6. Provide ability to request for subgraphs based on patterns.
7. To isolate application layer from source code modification keeping the AtomSpace API as-is or with minor changes.

## Assumptions and constraints

1. We want to build an abstraction layer on top of the current AtomSpace API.  So we don't want to change AtomSpace API in any severe/radical way. Of course some minor adjustments or API changes may be required to accommodate the new layer.
2. Changing the implementation underneath the Atomspace API may be OK, as nearly all existing OpenCog code accesses the Atomspace only via the API and is independent of the underlying data structures.

3. We consider a distributed system composed of multiple components. In a simple case the components may be servers in a network, but we may also want to consider other cases. Each component will contain one or more AtomSpaces which are considered to be local to each other.
4. Atoms have an Universally Unique ID (UUID) based in a hash function of its constituents (collisions are allowed but the hash function is such that collisions are rare). Because UUID is computed based in atom's constituents it is unique among all components
5. If useful, it is OK for Atoms to also have a "handle" which is a locally unique id (unique in the scope of a single component). It's possible to use only UUID but in practice using local handles may be better because (a) they are smaller objects and their use causes a significative gain in memory usage and (b) they can be implemented efficiently to get atom data faster than via UUID (e.g. they can be implemented as C++ pointers).
6. In addition to the information that defines an atom (type, name and outgoing set) atoms may carry additional information (e.g. attention values and truth values which are lists of numbers)
7. Atoms tend not to be that large; a node may be a few dozen to a few hundred bytes.
8. Atom creation and deletion and modification are common events, and occur according to complex patterns that may vary a lot over time, even for a particular OpenCog instance.
9. Some atoms will remain around for a really long time, others will be ephemeral and get removed shortly after they're created.
10. For most of OpenCog's cognitive tasks, the Atoms involved in thinking need to live in RAM. But the saving/retrieving of Atoms to/from disk also has a role to play.

## Glossary

For sake of simplicity, the reader of this document is supposed to be familiar with all the theoretical concepts, data structures and algorithms related to OpenCog and its components. A minimal glossary is provided below to avoid ambiguity but it isn't meant to be a complete explanation of the listed terms.

- **Atom** - the theoretical abstract definition of the most elementary entity in an AtomSpace, i.e. a node (uniquely identified by its type and name) or link (uniquely identified by its type and outgoing set).
- **Clone** - atom sitting in RAM. Thus, given a single, pure "abstract/theoretical" atom, there might be two different realized atoms (i.e. two different clones) on two different servers, having the same name/outgoing-set.
- **Serialized atom** - analogously, we may call atoms living on disk, or flying on a wire, "serialized atoms".
- **KBB (Knowledge Building Block)** - a subgraph of nodes and links with a self-contained semantic meaning. The actual form and size of a KBB vary from application to application (or even in the same application) but typically it's a link and the whole subgraph defined by it and its constituents. E.g.:

```
(EvaluationLink
    (PredicateNode "predicate1")
    (ListLink
        (ConceptNode "concept1")
        (ConceptNode "concept2")
    )
)
```

In degenerate cases, a KBB could be solely a node or a single link between two nodes.

- **Handle** - Locally (in the scope of a single server) unique ID of an atom
- **UUID** - Universally (in the scope of all servers) unique ID of an atom.
- **STI** - Short-term importance.
- **LTI** - Long-term importance.
- **DAS** - Distributed Atom Space. A set of atoms stored (RAM and/or disk) across several servers.

# Functional Requirements

1. Two or more clones of the same atom may be present in different components of DAS.
2. Each component of DAS may have only one clone of a given atom i.e., within a single component, two processes acting on a certain atom can be confident they are acting on exactly the same thing.
3. Two or more components may create or update clones of the same atom at the same time. Potentially this can cause inconsistency in atom's state. DAS is supposed to solve this inconsistency accordingly.
    a. Inconsistencies should be solved by heuristics appropriate to the semantics of the atoms involved, e.g., by PLN's Rule of Choice (which invokes the PLN revision rule much of the time).
    b. There is a trade-off between the overall consistency of the atoms in the whole DAS and total processor time spent to keep this consistency. DAS must provide ability to a user to adjust a trade-off between the overall consistency of the atoms in the whole DAS and total processor time spent to keep this consistency.
        i. DAS must provide ability to tune system parameters to allow a high level of inconsistency.
        ii. DAS must provide ability to tune system parameters to enforce a high level of consistency.
        iii. Regardless the trade-off parameters setup, DAS must assure that any inconsistency don't remain forever i.e., DAS should make an effort to resolve inconsistencies, although this effort will be balanced against the need for efficiency.
    c. DAS must use STI and LTI to guide the resolution of inconsistencies, so that the more important Atoms have their inconsistencies resolved faster.
4. There should not be local collision of UUID, i.e., if an atom being inserted in DAS has an UUID collision with another atom in the same server, this collision must be solved immediately.
5. UUID collisions between atoms in different components must be solved the first time one of the atoms is retrieved as answer for a request made in the server where the other atom resides.
6. DAS is supposed to be capable of storing an "as large as necessary" number of atoms. So it can not rely on the number of servers to provide this feature since this number may be restricted by deployment constraints. So DAS must use disk or DB persistence as additional/optional storage resources.
7. KBBs involving highly important atoms should be kept in RAM.
8. User must be able to explicitly persist an atom or a set of atoms to save room in RAM for more important information.
9. User must be able to explicitly restrict the scope of a newly created atom to the local component. Atoms with restrict scope can't be returned as answer on requests from other components.

10. User must be able to move sets of atoms to a specific component, allowing local access for AI algorithms in that component.
11. User must be able to pin sets of atoms in a given component, preventing than from being moved by any underlying balance strategy of DAS.
12. User must be able to clone atoms in different components in order to minimize the amount of time spent by AI algorithms running in one component when access to atoms living in other components are required.
13. DAS must be able to return  the handle of an atom given an UUID.
14. DAS must be able to return the handle of an atom given its definition (type + name for nodes or type + outgoing set for links).
15. DAS must be able to completely erase an atom, removing it from whatever storage entity used by DAS (yes, this is just a simple DELETE. The emphasis is to enforce that the erased atom is not supposed to be kept in disk, DB etc).
16. DAS must be able to return the handle of all links with a specific (UUID, handle or type/name/outgoing set) atom in its outgoing set.
17. DAS must be able to return the handle of all links with a specific (UUID, handle or type/name/outgoing set) atom in its incoming set.
18. DAS must be able to return handles of all atoms of a given type.
19. DAS must be able to return only the number of atoms of a given type.
20. DAS must be able to return handles of the M atoms with highest/lowest LTI or STI. Optionally constrained to a given atom type.
21. DAS must be able to return handles of all atoms with time-stamp within a certain time-interval.
22. DAS must be able to return handles of all atoms with spatial location within a certain region, during a certain time-interval.
23. DAS must be able to consider only non-persisted atoms for FR 16 to 22 above.
24. DAS must be able to consider only atoms in local component  for FR 16 to 22 above.
25. DAS must be able to return all variable assignments that satisfy a given pattern.
26. DAS must be able to return all KBBs that satisfy a given pattern.
27. DAS must be able to consider KBBs of atoms, rather than just individual atoms, as elementary units to be distributed amongst components. The actual form/size of a KBB is application-dependent, and should thus be user-configurable.
28. DAS must be able to consider KBBs of atoms, rather than just individual atoms, as elementary units to persist amongst cache stages. The actual form/size of a KBB is application-dependent, and should thus be user-configurable.

# Requirement Traceability Matrix

In Table 1 we show the correlation between functional requirements and high-level business requirements.

| | B1 | B2 | B3 | B4 | B5 | B6 | B7 |
|---|---|---|---|---|---|---|---|
| F1 | | X | | X | | | |
| F2 | | | | | | | X |
| F3 | X | X | | | | | |
| F4 | | | | | | | X |
| F5 | X | X | | | | | X |
| F6 | X | | X | X | | | |
| F7 | X | | X | X | | | |
| F8 | X | | X | X | | | |
| F9 | X | | | | | | |
| F10 | X | X | X | | | | |
| F11 | X | X | | | | | |
| F12 | X | X | | | | | |
| F13 | | | | | X | | X |
| F14 | | | | | X | | X |
| F15 | | | | X | | | X |
| F16 | | | | | X | | X |
| F17 | | | | | X | | X |
| F18 | | | | | X | | X |
| F19 | | | | | X | | X |
| F20 | | | | | X | | X |
| F21 | | | | | X | | X |
| F22 | | | | | X | | X |
| F23 | | | | X | | | |
| F24 | X | | | | | | |
| F25 | | | | | | X | |
| F26 | | | | | | X | |
| F27 | X | | | | | | |
| F28 | | | | X | | | |

Table 1: Traceability of functional requirements to business requirements

# Non-Functional Requirements

1. Regarding consistency amongst clones in different components, where it is not possible to have good balance between high consistency and high efficiency the priority must be given to performance.
2. DAS must be able to keep 5 to 10 million atoms on each server to allow AI algorithms to run in a fast and sensible way.
3. DAS must be able to run on a cluster of roughly equally powerful machines.
4. Typically we want clusters as large as dozens of servers.
5. In addition to atom storage proper, each server in the cluster may run one or more AI algorithms potentially using local and remote atoms. Such algorithms tend to create/use information locally (in the hosting server) but not necessarily.
6. DAS must add very small overhead for requests that can be answered exclusively with local information. I.e. user's requests to DAS that can be answered without inter-server communication should be answered in the same (or nearly the same) time as requests to a regular AtomSpace.
7. User's requests to DAS that demands inter-server communication should be answered at a rate of at least 5 user requests/second.

# Use Cases

In this section we list concrete use cases for DAS to illustrate the importance of the requirements presented in the previous sections.

## Use case 1: Distributed Incremental Pattern Mining From Social Network Data

Mining patterns from social network data, such as what kinds of people tend to do to make friends. New data can be fed into the pattern miner incrementally.

- **Layout:** One central server + N pattern mining workers run in different machines.
- **Data:** Before start pattern miner, the data should have been converted into Atomese and stored in the distributed corpus Atomspace. The mined patterns are stored in a separate distributed Atomspace from the corpus Atomspace.
- **Central server:** It has two functions (1) works as a task distributor to assign Links to every mining worker on request by giving a list of Link handle uid. (2) run an interestingness evaluation task on user request.
- **Mining worker**: When a mining worker is launched, it connects to the server and fetch the Links from the distributed Atomspace by the handle uids assigned to it by the server, and then start to mine all possible patterns from these Links and add them into the separate distributed pattern Atomspace. If a pattern already exists, simply increase the count by 1. When a worker
- **Hardware requirements:** Common pc ( like 2G RMB) should be enough for both the server and mining workers.

### Input data

The user provides a graph database of social network data, for example in neo4j, contains entity, relations, each entity and relation can have attributes, for example:

- Entities: Person (name, gender, age, occupation), Organization, School ...
- Relations: Friend, work_in, graduate_from …

Each Entity can be mapped into EvaluationLinks with all of its attributes, like:

```
(EvaluationLink
 (PredicateNode "name")
   (ListLink
       (EntityNode "5531")
       (ConceptNode "Adam")
   )
)
```

```
(EvaluationLink
   (PredicateNode "occupation")
   (ListLink
       (EntityNode "5531")
       (ConceptNode "teacher")
   )
)
```

Because relation types are limited and usually predefined in such database,  a relation can be mapped into a relationLink rather than an EvaluationLink:

```
(FriendLink
   (EntityNode "5531")
   (EntityNode "9891")
)
```

E.g. in a MMORPG, if there are around 5M active players, each player have 10 friends in average in the game, then there are 25M FriendLinks.

## Atomspace insert / update operation

Different workers may add the same pattern into the distributed Atomspace or update its count at the same time, which requires basic multithreading security.

## The current implementation of distributed pattern miner in Opencog without Distributed Atom Space

This section is just for reference so as to explain how it works in current implementation without distributed Atomspace.

- **The basic framework**
  One central server + N pattern miner workers run in different machines.
  The huge amount of input data is to divided into small slightly overlapped portions and stored in worker machines.
- **Patten miner worker**
  A pattern miner task with one portion of data, can be launched in a worker; one task application will load this portion of data into its local Atomspace and then starts to mine all possible patterns from its local Atomspace. Every 10 or x patterns mined will be sent to the central server via network. If one task is finished, the process can be shut down and a new task with a new portion of data can be started.
- **Central server**
  The central server is always waiting be connected by workers. It keeps receiving the patterns sent by the workers and adding the patterns into its local Atomspace and . If a pattern already exist, just increase its occurrences. When all the workers have finished mining and disconnected from the server, the user can choose to keep waiting for new workers to connect or start interestingness evaluation.
- **Hardware requirements**
  The central server requires large RAM (at least 64 G).
  The workers should be just common pc ( like 2G RMB)

# Use case 2: Distributed Inference Control Learning

Inference Control Learning in the URE (Unified Rule Engine) is the process of learning how to efficiently guide reasoning to produce new knowledge, validate conjectures, etc.

Efficient reasoning is amongst the hardest problems in computer science. One way to tackle it is to learn how to bias reasoning to be efficient on problems that matter to us. It will always be generally inefficient, but as long as these inefficiencies hover around problem classes of no interest, that is OK.

To do that we

1. Run reasoning over a collection of problems.
2. Record the traces of every decision the URE makes while reasoning.
3. Mine hidden patterns in these traces.
4. Turn these patterns into control rules so that the URE would be more efficient at solving these problems the next time.

If the problems the URE is exposed to are sufficiently general, and the control rules learned sufficiently abstract, then the efficiency should be transferable to new problems as well.

The catch is that it requires phenomenal computational power. First, in order to extract valuable information from problems, some must be solved. If the problems are hard, this alone can be challenging. Second, in order to attain decent confidences on the control rules, a large number of problems must be attempted. Third, learning the control rules can itself be costly.

Fortunately, the first two steps, reasoning over a collection problems and recording their traces, is embarrassingly parallel. Each reasoning instance can live in its own process/machine, and record as well post-process traces on its own local atomspace. Things start getting a more tricky in the third step because unarguably many patterns will only surface when mining data across AtomSpaces. Then the fourth step can probably take place in a centralized manner, gathering only knowledge obtained from the previous step that seem relevant to produce control rules.

The remaining of the document mostly focuses on step 3, mining hidden patterns from traces distributed across atomspaces.

## Input Data

We only give the representations for backward chaining, which only minorly differs from forward chaining. Backward chaining traces are represented by two types of knowledge.

1. Connecting inferences to other inferences:

```
ExecutionLink (stv 1 1)
   SchemaNode "URE:BC:expand-inference"
   List
     <inference>
     <premise>
     <rule>
   <new-inference>
```

   Meaning that `<inference>` is expanded from `<premise>` with `<rule>` to produce `<new-inference>`.

2. Relating inferences and success:

```
EvaluationLink <TV>
   PredicateNode "URE:BC:preproof-of"
   List
     <inference>
     <target>
```

   Here the notion of success is determined by whether a certain inference is a preproof of some target. That is whether subsequent expansions may lead to an inference proving the target or not, where `<TV>` reflects how much we know it (that knowledge is not always certain unless the target gets proven).

Both inferences and rules are represented by `BindLink` (a rule can be seen as an atomic inference):

```
BindLink
   <variables>
   <clauses>
   <rewrite>
```

## Preprocessing

Let us first explain what a control rule is. Its general form is:

```
ImplicationScope <TV>
  <vardecl>
  And
    Execution
      GroundedSchema "URE:BC:expand-inference"
      List
        <inference>
        <premise>
        <rule>
      <new-inference>
    Evaluation
      Predicate "URE:BC:preproof-of"
      List
        <inference>
        <target>
    <pattern>
  Evaluation
    Predicate "preproof-of"
    List
      <new-inference>
      <target>
```

expressing that, given that `<inference>` expending from `<premise>` with `<rule>` produces `<new-inference>`, and `<inference>` is a preproof of `<target>`, possibly following some extra `<pattern>`, the odds that `<new-inference>` is a preproof of `<target>` is `<TV>`.

Said simply, this represents the probability that applying some rule in a given inference gets us hopefully closer to proving the target.

To do well the distance from the proof should be used instead of the binary notion of preproof, because some paths are shorter than others, and we want the inference control to choose the shortest path. But the notion of preproof is simpler to start with.

In principle no preprocessing would be required, one would only need to run this query on the URE to infer control rules. However in practice any preprocessing that may reduce the computation is worth doing. In particular one can pre-compute all conjunctions of the form

```
And
   Execution
     GroundedSchema "URE:BC:expand-inference"
     List
       <inference>
       <premise>
       <rule>
     <new-inference>
   Evaluation
     Predicate "URE:BC:preproof-of"
     List
       <inference>
       <target>
```

Which is done by merely applying the backward chainer to the target above with the fuzzy conjunction introduction rule. This can be done on each atomspace separately, thus is embarrassingly parallel, as even if some inferences are shared across problems, applying that query on the entire distributed atomspace is not expected to produce any extra knowledge.

## Rule-Engine Queries

The pattern miner (which is already a URE process) combined with other forms of reasonings will be used for uncovering patterns to then be turned into control rules. For that a rule base (usually small, thus requiring no distribution) need to be run the entire distributed atomspace containing all traces over all problems.

Now let's break it down for the simplest possible rules we can learn, context free control rules expressing the probability of producing a preproof by expanding any inference from any premise with a given inference rule, regardless of how that inference and premise look like. This is equivalent to setting the weight of the inference rule (or the "second order weight" as confidence is taken into account).

The result is gonna be like:

```
ImplicationScope <TV>
  And
    Execution
      GroundedSchema "URE:BC:expand-inference"
      List
        Variable "$inference"
        Variable "$premise"
        <rule>
      Variable "$new-inference"
    Evaluation
      Predicate "URE:BC:preproof-of"
      List
        Variable "$inference"
        Variable "$target"
  Evaluation
    Predicate "preproof-of"
    List
      Variable "$new-inference"
      Variable "$target"
```

where everything is a variable except `<rule>` which hold constant.

Given the previous preprocessing done described in the section above, such control rule only require a call a single PLN rule, conditional direct evaluation. What that rule is doing is, given a certain implication, fetch all instances of its antecedent that are true, let's call that set `A`, then for each element of `A`, take its valuation (mapping from variables to values), apply it to the consequent of the implication to obtain a consequent instance, and if it is true, add it to `C`, then calculate the resulting TV on the implication as follows:

$$TV.strength = |C|/|A|$$

$$TV.count = |A|$$

(this is not the best possible estimate which may vary depending on the prior, but we let aside for sake of simplicity).

It is important to realize that the pattern matchings required to obtain `A` and `C` need to take place over the entire Distributed Atom Space of traces.

Of course interesting control rules will require more sophisticated reasoning schemes such as pattern mining but in the end it will probably end up requiring the same sort of distributed pattern matching as in this simple case. So starting with that should already be quite meaningful.

## Knowledge Base Properties

The size of an atomspace of trace seems to grow almost linearly in our toy problem (the one in that same directory).

| Number of iterations | Atoms of trace |
|---|---|
| 50 | 1K |
| 100 | 2.1K |
| 200 | 4.3K |

This makes sense. First, each successful iteration produces an inference. Second, although inferences grow with the depth of the search, much of their content is being shared due to being stored on the atomspace. However, it is not expected that the grow would be linear in the worst case, because due to unification/substitution a new inference may share little with the inference it has been expanded from. Based on that we conclude that the growth is quadratically bounded, and probably in average as follows

```
|T| = a*N^b
```

Where `N` is the number of iterations, and `a` and `b` are parameters ultimately depending on the problem, the knowledge and rule bases. `b` is likely within 1 and 2, linear in the best case, quadratic in the worst. If the rule base is sufficiently restricted it may even go sub-linear, as in this case not all inferences can be successfully expanded.

Let us estimate the size of an atomspace of traces for a real world problem. Let say that `N=10K`, `a=10` and `b=1.1`, thus `|T|=251K`, which to our experience requires about 250MB of RAM. According to our early experiments about a hundred of problem instances must be run to begin to collect enough traces to learn more sophisticated (beyond context-free) control rules, though that seems to be really a minimum.

So it seems a few workstations, each with a dozen cores and dozen GB of RAM would be able run a moderate scale (beyond toyish) inference control learning experiment in about a day. A couple of hours for solving 100+ problem instances, and perhaps the rest of the day for mining and producing control rules, the effort depending essentially on how sophisticated we want it to be, which is hard to determine in advance. Learning context-free rules might just be a question of minutes, while mining complex patterns for producing context-sensitive rules might easily take hours or days.

### Actual Databases and Test Scripts

A toy single-thread experiment can be found in that same folder and described in length in the README.md.

## Use case 3: MOSES

MOSES is a program learner. It works by evolving islands of programs called demes, each representing a subregion of the program space. Optimizing a deme consists of searching that subregion for programs that maximize a given fitness function. Optimizing multiple demes is embarrassingly parallel, i.e. each deme can be created and optimized almost in isolation w.r.t. the other demes (assuming they do explore relatively disjoint subregions).

Once a deme has been optimized (searched till some criteria are met) its most promising candidates are sent to the meta-population. The meta-population is a population of programs that can be used to spawn more demes. For that a selected program is turned into a template, called an exemplar, where some parts of it can be mutated. The set of all possible mutations of that exemplar defines the program region of the deme.

With MOSES being ported to the AtomSpace, programs will soon be represented as atoms. Each deme thus far explored will be stored in its own AtomSpace and the meta-population will be an AtomSpace too.

On top of the program candidates, the fitness function (including for instance associated data, in the case of data fitting) will have to be copied to each deme so MOSES can evaluate each candidate.

So we would start with a centralized architecture with a master AtomSpace containing the meta-population, and peripheral slave AtomSpaces containing demes being optimized. Then eventually move towards a more hierarchical architecture to avoid a bottleneck on the master.

MOSES actually already supports such centralized distributed architecture built around MPI. However no AtomSpace is currently used, not yet. Program candidates are coded in MOSES' home-brewed language called Combo, and are being exchanged back and forth between master and slaves as strings. These exchanges though only happen at the creation and destruction of a deme. At the creation the exemplar is sent to a slave, and at destruction promising candidates are sent back to the master. Communications happening during deme optimization could be useful in principle but can probably be ignored for the time being.

### Input data

Both demes and the meta-population contain Atomese programs. Here are some examples of programs.

```
;; f1 + f2
(Plus
  (Schema "f1")
  (Schema "f2"))

;; (p1 and p2) or p3
(Or
  (Predicate "p3")
  (And
    (Predicate "p1")
    (Predicate "p2")))

;; if p1 then f1 else f2
(IfThenElse
  (Predicate "p1")
  (Schema "f1")
  (Schema "f2"))
```

where `p1`, `p2` and `p3` are boolean features and `f1` and `f2` are numerical features of some dataset, and usual operators are high level overloads (for instance `f1 + f2` means the sum of function `f1 and f2`). Obviously the average program size will be much larger than what is presented here. In addition, though the exact form remains to be determined, programs will be explicitly marked as members of a deme, such that

```
;; [f1 + f2] is a member of deme1
(Member
  (Plus
    (Schema "f1")
    (Schema "f2"))
  (Concept "deme1"))

;; [(p1 and p2) or p3] is a member of deme1
(Member
  (Or
    (Predicate "p3")
    (And
      (Predicate "p1")
      (Predicate "p2")))
  (Concept "deme1"))

;; [if p1 then f1 else f2] is a member of deme1
(Member
  (IfThenElse
    (Predicate "p1")
    (Schema "f1")
```

```
    (Schema "f2"))
  (Concept "deme1")))
```

Finally, the whole fitness function will have to be duplicated in each deme AtomSpace. The format of the fitness function is to be determined, but it might look like

```
Lambda
  Variable "$P"
  <fitness>
```

where `<fitness>` could be a least squared error between `$P` and some dataset.

Once a deme is done being optimized it will send its most promising candidates to the meta-population, which will look like

```
;; [if p1 then f1 else f2] is a member of deme1
(Member
  (IfThenElse
    (Predicate "p1")
    (Schema "f1")
    (Schema "f2"))
  (Concept "meta-population"))
```

## Atomspace Queries/Searches

I guess it's premature to tell how the data will be queried and exchanged. One can imagine queries such as the following to fetch promising candidates

```
(Get
  (And
    (Member
      (Variable "$P")
      (Concept "deme1"))
    (Evaluation
      (GroundedPredicate "is-promising")
      (Variable "$P"))))
```

## Knowledge base properties

As a rough estimation, the number of nodes should generally be limited to a dozen of thousands corresponding to the number of features in the data to fit (the most common case for a fitness function). So nodes of that types

```
(Schema "f1")
...
```

or

```
(Predicate "p1")
...
```

if the features are boolean.

The number of links however is only limited by the complexity of the models being evolved. It is not unusual to evolve models with dozens of operators, and since the upper limit of the number of links grows exponentially with the number of operators, that would be going into the billions. In practice though, because evaluating the fitness function on each candidate is so costly, deme optimization would rarely goes above millions of candidates, meaning likely millions of atoms, because candidates share most of their atoms. Given that an atom takes in average about 1.5K of RAM a million atoms would take about 1.5GB of RAM.

The requirements for the meta-population are probably similar.

There can be extra RAM needed if fitness evaluation memoization is used, but that's another problem. It is expected that the fitness function (including data) will be negligible compared to the deme population and meta-population.

## Actual databases and test scripts

The AtomSpace MOSES port being in its infancy there is no existing code to test that. MOSES which is itself however quite mature can be found here https://github.com/opencog/as-moses (if that helps).

# Use case 4: NLP question answering about the system's internal state

## Definitions

- opencog-apps : Any app that a developer might develop using various opencog subsystems as a service
- nl-apis : Natural-language apis or conversational apis are natural-language based interfaces to opencog-apps or opencog. This are similar to ‘Ok Google’, ’Siri’, questions, or any other predefined natural-language pattern that could be used as a standard means of triggering a conversational response.

## NLP question answering about the system’s internal state

The aim is to query opencog about its internal-states model using natural-language and get the response in natural-language. Here internal-states model means an atomese representation of opencog subsystems such as GHOST, PLN, MOSES, OpenPsi-dynamics, URE, Pattern-matcher, Pattern-miner, and any other system that a developer might develop, and may include such information as

- Time spent processing a task
- Estimated time of completion of a task
- Atomese logs of previous performances
- How much processing power was used to complete a task
- Which subsystems contributed to accomplish a task (subsystem traces)
- Action-selection traces
- Developer defined event logs such as surprising changes in ECAN’s attentional focus or surprising change in truth-value(or any other value) of a set of OpenPsi rules

The possible users are developers of opencog-apps. A developer may use it for debugging as well as part of their opencog-app service. The following are some of the nl-apis that may be used

- Why did you come to this reasoning conclusion?
- Why is it taking too long to complete task x?
- How much time and processing power do you estimate it would take to complete task x?
- What made you do x?
- What percentage of users of opencog-app-123, that have European Citizenship, allow their data to be processed for xyz purposes?
- Is opencog-app-345 GDPR compliant?

The rest of this document demonstrates how to answer “how long it would take to complete task x?”.

### Input data

Input data required for giving such a service probably comes from log files generated by the system. Some of the data we are looking for looks like this, once imported into the atomspace.

```
(AtTime
   (Execution
      (GroundedSchema "scm: time-spent-reasoning")
      (Concept "who-moved-the-cheese")
      (Concept "Ben")
   )
   (TimeInterval
      (Time "1529040789")
      (Time "1529050125")
   )
)
```

The AtomSpace has to be populated with parsed sentences, which will be needed for sentence generation. We can, for instance, parse Wikipedia and books from Project Gutenberg etc, and load the atoms to one or more AtomSpaces. A subset of atoms that will be generated after parsing the sentence “I read books” looks like:

```
(Inheritance
   (Concept "I@123)
   (Concept "I")
)
(Evaluation
   (Predicate "read@456")
   (List
      (Concept "I@123")
      (Concept "books@789")
   )
)
(Implication
   (Predicate "read@456")
   (Predicate "read")
)
(Inheritance
   (Concept "books@789")
   (Concept "book")
)
```

There will also be additional atoms that are used to represent the world, opencog’s subsystems and opencog-apps. The world model could be imported from sources like SUMO, ConcpetNet,

or learnt  by opencog from other sources.  Model of opencog-apps will be a requirement while developing one.

### Atomspace queries/searches

There should be a GHOST rule that will be triggered when someone asks a question. The context of the GHOST rule might be, say if the input matches the pattern *“how long it would take to complete $x”*, and the action may be, say *“check historical record of $x”*, and it would satisfy the goal *“Answer User Questions”*.

The GHOST rule would look something like this:

```
(ImplicationLink
   (AndLink
      (TrueLink
         (ExecutionOutputLink
            (GroundedSchemaNode "scm: ghost-execute-action")
            (ListLink
               (ExecutionOutputLink
                  (GroundedSchemaNode "scm: check_historical_record")
                  (ListLink
                     (ListLink
                        (GlobNode "wildcard-$001")
                     )
                  )
               )
            )
         )
         (PutLink
            (StateLink
               (AnchorNode "GHOST Last Executed")
               (VariableNode "$x")
            )
            (ConceptNode "Rule1")
         )
         (ExecutionOutputLink
            (GroundedSchemaNode "scm: ghost-record-executed-rule")
            (ListLink
               (ConceptNode "Rule1")
            )
         )
      )
      (SatisfactionLink
         (VariableList
            (TypedVariableLink
               (GlobNode "wildcard-$002")
               (TypeSetLink
```

```
                  (TypeNode "WordNode")
                  (IntervalLink
                     (NumberNode "0.000000")
                     (NumberNode "-1.000000")
                  )
               )
            )
            (TypedVariableLink
               (VariableNode "how-$003")
               (TypeNode "WordNode")
            )
            (TypedVariableLink
               (VariableNode "how-$004")
               (TypeNode "WordInstanceNode")
            )
            (TypedVariableLink
               (VariableNode "long-$005")
               (TypeNode "WordNode")
            )
            (TypedVariableLink
               (VariableNode "long-$006")
               (TypeNode "WordInstanceNode")
            )
            (TypedVariableLink
               (GlobNode "wildcard-$007")
               (TypeSetLink
                  (TypeNode "WordNode")
                  (IntervalLink
                     (NumberNode "0.000000")
                     (NumberNode "-1.000000")
                  )
               )
            )
            (TypedVariableLink
               (VariableNode "to-$008")
               (TypeNode "WordNode")
            )
            (TypedVariableLink
               (VariableNode "to-$009")
               (TypeNode "WordInstanceNode")
            )
            (TypedVariableLink
               (VariableNode "complete-$010")
               (TypeNode "WordNode")
            )
            (TypedVariableLink
               (VariableNode "complete-$011")
               (TypeNode "WordInstanceNode")
```

```
            )
            (TypedVariableLink
               (GlobNode "wildcard-$001")
               (TypeSetLink
                  (TypeNode "WordNode")
                  (IntervalLink
                     (NumberNode "0.000000")
                     (NumberNode "-1.000000")
                  )
               )
            )
            (TypedVariableLink
               (VariableNode "$S")
               (TypeNode "SentenceNode")
            )
            (TypedVariableLink
               (VariableNode "$P")
               (TypeNode "ParseNode")
            )
         )
         (AndLink
            (ReferenceLink
               (VariableNode "how-$004")
               (VariableNode "how-$003")
            )
            (ParseLink
               (VariableNode "$P")
               (VariableNode "$S")
            )
            (EvaluationLink
               (GroundedPredicateNode "scm: ghost-lemma?")
               (ListLink
                  (VariableNode "long-$005")
                  (WordNode "long")
               )
            )
            (ReferenceLink
               (VariableNode "to-$009")
               (VariableNode "to-$008")
            )
            (WordInstanceLink
               (VariableNode "long-$006")
               (VariableNode "$P")
            )
            (WordInstanceLink
               (VariableNode "to-$009")
               (VariableNode "$P")
            )
```

```
            (EvaluationLink
               (GroundedPredicateNode "scm: ghost-lemma?")
               (ListLink
                  (VariableNode "how-$003")
                  (WordNode "how")
               )
            )
            (ReferenceLink
               (VariableNode "complete-$011")
               (VariableNode "complete-$010")
            )
            (WordInstanceLink
               (VariableNode "how-$004")
               (VariableNode "$P")
            )
            (EvaluationLink
               (GroundedPredicateNode "scm: ghost-lemma?")
               (ListLink
                  (VariableNode "complete-$010")
                  (WordNode "complete")
               )
            )
            (EvaluationLink
               (PredicateNode "GHOST Word Sequence")
               (ListLink
                  (VariableNode "$S")
                  (ListLink
                     (GlobNode "wildcard-$002")
                     (VariableNode "how-$003")
                     (VariableNode "long-$005")
                     (GlobNode "wildcard-$007")
                     (VariableNode "to-$008")
                     (VariableNode "complete-$010")
                     (GlobNode "wildcard-$001")
                  )
               )
            )
            (StateLink
               (AnchorNode "GHOST Currently Processing")
               (VariableNode "$S")
            )
            (ReferenceLink
               (VariableNode "long-$006")
               (VariableNode "long-$005")
            )
            (WordInstanceLink
               (VariableNode "complete-$011")
               (VariableNode "$P")
```

```
            )
            (EvaluationLink
               (GroundedPredicateNode "scm: ghost-lemma?")
               (ListLink
                  (VariableNode "to-$008")
                  (WordNode "to")
               )
            )
         )
      )
   )
   (ConceptNode "GHOST Answer User Questions")
)
```

The question asked by the user will be parsed by `nlp-parse`, which will go through Link Grammar, RelEx, and Relex2Logic. The atoms generated look similar to the ones quoted in the “Input data” section. After that, the above GHOST rule will be selected and triggered by the action selector. The function *“check_historical_record”* will be called and it should return an estimated average time, ideally in a NLG-compatible form, e.g.

```
(QuantityLink
   (ConceptNode "minutes")
   (ConceptNode "three")
)
```

Ideally this will be sent to Microplanner and SuReal to generate the actual reply, e.g. *“It usually takes about three minutes.”*

### Knowledge base properties

Let’s say there are 10 AtTimeLinks generated per second in the log, and it pulls 5 days of records of all services, find and take average to estimate the time for a particular service, which means we are loading:

- AtTimeLink: 4.3M
- ExecutionLink: 4.3M
- GroundedschemaNode: 4.3M
- ConceptNode: 8.6M
- TimeIntervalLink: 4.3M
- TimeNode: 8.6M

The above is a pessimistic estimation, because we can invent some more intelligent way to decide what should be loaded from the logs, instead of just blindly loading everything.

Also, let's say we parse 100K sentences for sentence generation, assuming one parse per sentence and one interpretation of a parse, that gives(very roughly):

- SentenceNode: 100K
- ParseNode: 100K
- ParseLink: 100K
- SentenceSequenceLink: 100K
- InterpretationNode: 100K
- InterpretationLink: 100K
- AtTimeLink: 100K
- TenseLink: 100K
- WordInstanceNode: 1M
- WordInstanceLink: 1M
- WordSequenceLink: 1M
- PartOfSpeechLink: 700K
- LemmaLink: 800K
- WordNode: 800K
- LemmaNode: 800K
- PredicateNode: 150K
- ConceptNode: 500K
- EvaluationLink: 20M
- InheritanceLink: 20M
- ReferenceLink: 10M
- ImplicationLink: 150K
- ExecutionLink: 20M
- SetLink: 200K
- ListLink: 2M
- LgWordCset: 800K
- LgLinkInstanceNode: 800K
- LgLinkInstanceLink: 800K

The rest of the atom types can be neglected.