An Application-Oriented Context Pre-fetch Method for Improving Inference Performance in Ontology-based Context Management

Jaeho Lee, Insuk Park, Dongman Lee, and Soon J. Hyun

School of Engineering
Information and Communications University
103-6 Munji, Yuseong, Daejeon, Korea
{leejaeho, ispark, dlee, shyun}@icu.ac.kr

Abstract

Ontology-based context models are widely used in a ubiquitous computing environment. Among many benefits such as acquisition of conceptual context through inference, context sharing, and context reusing, the ontology-based context model enables context-aware applications to use conceptual contexts which cannot be acquired by sensors. However, inferencing causes processing delay and it becomes a major obstacle to context-aware applications. The delay becomes longer as the size of the contexts managed by the context management system increases. In this paper, we propose a method for reducing the size of context database to speed up the inferencing. We extend the query-tree method to determine relevant contexts required to answer specific queries from applications in static time. By introducing context types into a query-tree, the proposed scheme filters more relevant contexts out of a query-tree and inference is performed faster without loss of the benefits of ontology.

1. Introduction

An application is considered context-aware if it adapts to a user's or its own context such as location, state, and so on. There are several examples of context-aware applications. Among them is Teleport System (Dey et al. 2001) in which a user's desktop environment follows the user as he or she moves from one workstation to another. Another example is the Navigation system (Baus et al. 2002) which dynamically displays the navigation information according to the speed of traveling in order to help a user's attention. To facilitate the development of context-aware applications, a context model is required to manage such functions as storing, searching, and sharing contexts that change dynamically.

There have been many research efforts on context modeling such as application-oriented model (Dey et al. 2001; Kindberg et al. 2000), graphical model (Henricksen et al. 2002), and ontology-based model (Gu et al. 2004; Chen et al. 2004). The ontology-based context model is widely used because of its advantages of sharing and reusing knowledge, and especially, of inferring a conceptual context which cannot be acquired by the data gathered from sensors.

Several context management systems have proposed ontology-based context models for the rapid and reliable development of context-aware applications (Wang et al. 2004; Lee et al. 2004; Ranganathan and Campbell 2003; Khedr and Karmouch 2004). The main roles of a context management system are to collect contexts from sensors, to infer conceptual contexts, and to deliver an appropriate context to applications. Inference, however, is time- and resource- consuming. The processing time for inference increases proportionally to the size of contexts as well as the number of applied rules. As a ubiquitous computing environment becomes more intelligent, the size of contexts grows accordingly. This makes it difficult to achieve the inference result in a timely manner. Although most context management systems with the ontology-based context model recognize the problem, none of them provides solutions to it.

In this paper, we propose a context pre-fetch method to reduce the size of context database to be loaded in a working memory[1] to speed up inference. Our experiences in developing context-aware applications show that only a subset of the whole contexts is used in inferencing, e.g., less than 40 contexts out of 2,000 contexts. Therefore, we argue that irrelevant contexts can be filtered out from the whole contexts in a working memory. We extend the query-tree method (Levy et al. 1997) to identify the contexts required to answer specific queries of applications in static time. We adopt constraint predicates of the query-tree method to restrict the contexts and pre-fetch the contexts relevant to applications' queries. We classify contexts into three type categories; sensed, deduced, and defined context. With constraint predicates, the context types are also used as irrelevance claims to determine whether a certain context is required for them or not. Experimental results show that the proposed scheme allows the size of contexts in a working memory and the processing time for inference to be maintained smaller without loss of the benefits of

[1] Working memory is a memory space used for inference.

reasoning than the query-tree method. It certainly helps ontology-based context management to be scalable in a resource-limited ubiquitous computing environment.

The rest of this paper is organized as follows. In chapter 2, we show related work. In chapter 3, our approach is explained in detail. In chapter 4, we describe an implementation, followed by the evaluation in chapter 5. Finally in chapter 6, conclusion is provided and future work is discussed.

2. Related Work

2.1. Ontology-Based Context Model

The Context Broker Architecture (CoBrA) has developed an ontology-oriented context model to make easy knowledge-sharing across distributed systems (Chen et al. 2004). They have used the F-OWL inference engine to get a conceptual context from raw contexts (Zou et al. 2004). The F-OWL inference engine has features to enhance reasoning performance. The engine introduces the Tabling method to reduce the processing delay required for rule-based inference. The Tabling is used in various inference engines such as the Jena2 (A Semantic Web Framework for Java) (Carroll et al. 2004). According to it, once it is proved that a subject-property-object triple is in the target ontology model, the triple is added to an appropriate table. Even though a few queries at first may take time to get results, the next queries get quick responses. However, when update of the model such as data assertion or retraction takes place, the table gets invalidated. Therefore, it is impossible for the context models to take advantage of Tabling as context changes frequently at runtime.

The Semantic Space has also developed an ontology-based formal context model to address critical issues including formal context representation, knowledge sharing, and logic-based context reasoning (Wang et al. 2004). The system uses Jena2 (Carroll et al. 2004) to inference the ontology-based contexts. However, similar to CoBrA, performance degrading is a problem when the size of the context increases. They suggest three solutions. First, the time-critical context-aware applications execute without reasoning process. Second, context reasoning is independently performed by resource-rich devices such as the residential gateway. Among them, most active solution is two-level structure, i.e., high-level and domain-specific. By dividing the context into two levels, the size of context used in inference can be reduced because each domain-specific context is managed separately and loaded dynamically when context-aware applications move into the domain. However, in this case, not the entire context is used by context-aware applications in a domain, and the aforementioned performance degrading problem of the logic-based reasoning still remains when the size of a domain becomes large. We also take the two-level structure to design our context ontology. Furthermore, the proposed method in this paper focuses on reducing the domain-specific contexts since all domain-specific contexts are not relevant to a specific query, or a context-aware application. In this way, our method makes the system scales well even though the size of a domain becomes large.

2.2. Methods to Speed up Inference

We have examined efforts to speed up reasoning inside inference engines. Some of the ontology inference engines nowadays are Jena2 (Carroll et al. 2004), Racer (Racer), FaCT (FaCT), Hoolet (Hoolet), and Triple (Triple). Jena2 is a Java framework for writing Semantic Web applications. Jena2 provides inference for both the RDF and OWL semantics. However, the response time of reasoning increases along with the number of triples because Jena2's memory-based Graph model simply has been implemented as the triple pattern (S, P, O) matches by iterating over the Graph. Although the reasoning engines such as Racer, FaCT, Hoolet, and Triple are well-developed, none of them scales well when dealing with the OWL test case *wine ontology* (http://www.w3.org/TR/owl-guide/wine.owl) (Zou et al. 2004).

Alon Y. Levy, et al. proposed a method for search space pruning to speed up inference using the query-tree is shown (Levy et al. 1997). The query-tree is a powerful static-analyzing tool for determining knowledge base containing rules and finite structure that encodes all derivations of a given set of queries. Using the method, we can select rules and ground facts used in deriving answers to the queries. We adopt and extend the query-tree method to determine relevant contexts required to answer specific queries given from applications at the time of initializing the application.

3. Context Pre-fetch Method

3.1. Design Considerations

In the ontology-based context model, the speed of inference can be improved by reducing two factors; (a) the number of rules and (b) the size of context database (Gu et al. 2004). Rules are divided into two types. One is rules for ontology reasoning such as `subClassOf`, `Symmetric`, and `Transitive` semantics as shown in Figure 1. Ontology reasoning is responsible for checking logical requirements which include class consistency, class subsumption, and instance checking (Gu et al. 2004). Since it does not make sense losing any semantics of ontology for speeding up the inference, the rules for ontology reasoning are kept in a working memory at runtime. The other is user-defined rules for generating conceptual contexts. These rules are needed only when context-aware applications ask for those conceptual contexts. We assume that user-defined rules required for an application's operation are given by the application. Thus, we cannot reduce the number of both ontology and user-defined rules in order for an application to work correctly even though the small number of rules makes the inference time shorter. Therefore, we focus on reducing the size of context database for speeding up inference.

Reducing the size of context database is a process in which the system reasons only relevant contexts to an application's query and fetches and places them in a separate context database in a working memory. This process is the same as Irrelevance reasoning in AI. The relevant contexts are what the context-aware application's queries need to access to get their results at runtime. We can guess relevant contexts from the context-aware application's queries by decomposing them into primitive contexts, or ground facts. One of techniques to guess relevant contexts is the query-tree method (Levy et al. 1997).

In the query-tree method, a query-tree built from a given query is traversed and irrelevant facts are pruned using *constraint predicates*. Constraint predicates are used to specify restrictions on ground facts. For example, suppose that a query, `AgeOf(x, y)`, is restricted by a constraint predicate, `y<=150`. It means that the ground facts larger than 150 are irrelevant to the query. They are found statically using constraint predicates before the query is requested and excluded when evaluating the query at runtime. In other words, only the ground facts satisfied with `y<=150` are placed in a working memory for the query at runtime. However, in the case of a context-aware application's queries, there are many queries about contexts acquired from sensors. The contexts acquired from sensors do not need to be placed in a working memory until the value of context comes in from sensor since they are meaningless before real values are sensed. It motivates us to devise a method for filtering further contexts which are relevant but useless to applications out of the query-tree. It helps to scale up the ontology-based context management in such a resource-limited ubiquitous computing environment. We introduce context types, which indicate when and how context can get a meaningful value, for further filtering besides constraints predicates. They are described in the next section.

Transitive-Property	`(?P rdf:type owl:TransitiveProperty)` `(?A ?P ?B) (?B ?P ?C) -> (?A ?P ?C)`
Symmetric Property	`(?P rdf:type owl:SymmetricProperty)` `(?X ?P ?Y) -> (?Y ?P ?X)`
inverseOf Property	`(?P owl:inverseOf ?Q) (?X ?P ?Y)` `-> (?Y ?Q ?X)`
Equivalent Property	`(?P owl:equivalentClass ?Q) ->` `(?P rdfs:subClassOf ?Q),` `(?Q rdfs:subClassOf ?P)`
subClassOf	`(?A rdfs:subClassOf ?B)` `(?B rdfs:subClassOf ?C)` `-> (?A rdfs:subClassOf ?C)`

Figure 1. Part of OWL Property Rules

3.2. Context Representation and Categorization

We design context ontology for a home environment using Web Ontology Language (OWL). Context ontology is divided into upper-level and lower-level context ontology similar to the CONON (Gu et al. 2004). An upper-level context defines each class and property, and expresses the relationships and constraints between properties using ontology semantic rules. A lower-level context defines instantiations of domain-specific facts using general concepts and properties of an upper-level one. For example, lower-level contexts both `'Bedroom'` for a home domain and `'Office'` for a business domain are described by an upper-level context, `'Room'` as shown in Figure 2.

A context is encoded as a triple which has a form of (subject, predicate, object) in OWL. While the subject and object are merely physical and logical entities or sensed values, the predicate makes a semantic relation between two entities. For example, the `'hasDevice'` property of `'Bedroom'` in Figure 2 is represented as a form of `'<Bedroom, hasDevice, Bed>'`. In addition, the context of a triple form can be extended to represent a complex context by combining the predicate.

We classify a context into a sensed context, a deduced context, and a defined context like (Gu et al. 2004; Henricksen et al. 2002). Every predicate of a context has a property, `'owl:classifiedAs'` to specify its type. (referred to as the `'Room'` ontology in Figure 2). We use the types of contexts to determine whether a query should be used to build a query-tree for selecting relevant contexts. Other researchers also proposed context categorization but they used it for other purposes, i.e., expressing the quality of a given context, compensating for context imperfection (Gu et al. 2004; Henricksen et al. 2002).

First, a sensed context such as `'person;locatedAt'` is acquired from a sensor at runtime. A sensed context is dependent on a sensor running in an environment. Therefore, a sensed context is meaningless until a sensor corresponding to a given context actually works at runtime. Accordingly, a context query for a sensed context such as `'<?p person;locatedAt Bedroom>'` cannot get an answer before runtime. Second, a deduced context such as `'person;hasStatus'` is acquired only by inference after the sensed context, `'person;locatedAt'` is sensed. To get a deduced context, we define user-defined rules which consist of context queries relevant to other types of contexts. For example, a deduced context such as 'a user's current status is `sleeping`' is obtained only when the current contexts satisfy the sleep-rule[2]. It contains context queries for the current state of a bed, and a person's current location. Among these queries, `'<?p person;locatedAt Bedroom>'` and `'<?d device;hasState ON>'` are the queries for a sensed context obtained at runtime. Accordingly, the sleep-rule cannot generate in static time the deduced context such that a person's current status is 'sleeping'. Finally, a defined context such as `'<TV device;locatedIn Bedroom>'` is defined by a user and

[2] sleep-rule : (?p rdf:type person:Person) (?d rdf:type device:Device) (?p person:locatedAt room:Bedroom) (?d device:locatedIn room:Bedroom) (?d device:hasState ON) -> (?p person:hasStatus status:sleeping).
The rule consists of five queries. Each query has a literal as variable. Among them, (?d device;hasState ON) means what is devices whose hasState value is ON>.

is rarely updated over its lifetime once after it is determined. Therefore, the result of a context query for defined context is always the same whenever the query is evaluated. Consequently, a defined context is only a context that can be acquired before runtime. Thus, we consider context queries for a defined context in a pre-fetch process.

```
<owl:Class rdf:ID="Room">
<rdfs:subClassOf>
   <owl:Restriction>
   <owl:onProperty rdf:resource="#doorState"/>...
<owl:ObjectProperty rdf:about="&room;Brightness">
 <rdfs:domain rdf:resource="#Room"/>
 <rdfs:range rdf:resource="&state;Brightness"/>
 <owl:classifiedAs rdf:resource="&icu;Sensed"/>
</owl:ObjectProperty>  ......
```

(a) Upper-level Context Ontology

```
<Room rdf:ID="Bedroom">
 <hasDevice rdf:resource="&device;Bed"/>  ......
 <Brightness rdf:resource="&state;BedroomBrightness"/>
 </Room>
```

(b) Home Domain Context Ontology

```
<Room rdf:ID="Office">
 <hasDevice rdf:resource="&device;Desk"/>  ......
 <hasDevice rdf:resource="&device;Fax"/>
</Room>
```

(c) Business Domain Context Ontology

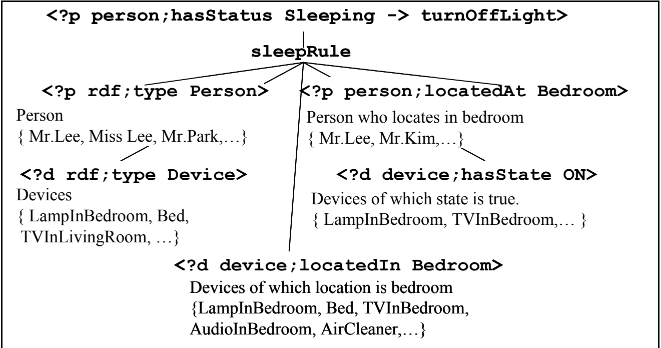
Figure 2. 'Room' Upper-level Context Ontology and 'Bedroom' and 'Office' Lower-level Context Ontology

3.3. Context Pre-fetch

The proposed method uses a pair of memory spaces; (a) a working memory and (b) a pre-processing memory. The working memory is used to perform inference over the contexts that support a context-aware application's operation at runtime. The pre-processing memory is used for the selection of relevant contexts in static time before use. We define 'pre-fetch' as a series of processes for building a query-tree, selecting relevant context in the pre-processing memory, and delivering the selected context into the working memory.

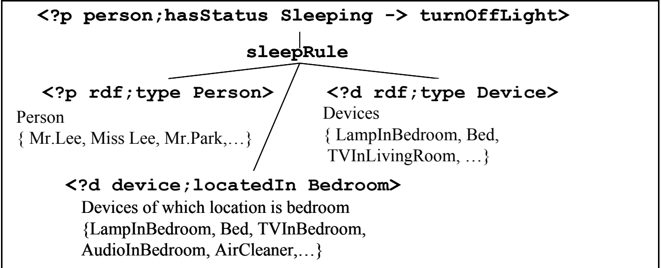
Before the pre-fetch process, we load the upper-level context ontology into the working memory at the initialization time to use for ontology reasoning. Relationships and constraints defined in the upper-level ontology are used to check the consistency of asserted contexts or infer contexts at runtime. For examples, the `'locatedAt'` property of the`'Person'` upper-level context ontology and the `'hasPerson'` property of the `'Room'` upper-level context ontology are in the `'owl:inverseOf'` relationship to each other. In such a case, an assertion of a sensed context in a working memory such as `'<Mr. Lee locatedAt Bedroom>'` makes the value of the `'hasPerson'` property of the `'Bedroom'` context ontology `'Mr. Lee'`. Thus, the upper-level context ontology has to remain in the working memory during runtime. All context ontology is loaded into the pre-processing memory to examine a query over the whole contexts set.

We prune the contexts irrelevant to an application's operation from the whole contexts in the pre-fetch process. Pruning is done in two steps; (a) pruning by a constraint predicates adopted from the query-tree method (Levy et al. 1997), (b) pruning by predicate types implied by the context category.



<?p person;hasStatus Sleeping -> turnOffLight>
sleepRule
<?p rdf;type Person>
Person
{ Mr.Lee, Miss Lee, Mr.Park,…}
<?p person;locatedAt Bedroom>
Person who locates in bedroom
{ Mr.Lee, Mr.Kim,…}
<?d rdf;type Device>
Devices
{ LampInBedroom, Bed, TVInLivingRoom, …}
<?d device;hasState ON>
Devices of which state is true.
{ LampInBedroom, TVInBedroom,… }
<?d device;locatedIn Bedroom>
Devices of which location is bedroom
{LampInBedroom, Bed, TVInBedroom, AudioInBedroom, AirCleaner,…}

(a) Query-tree based on Lighting application's query before filter out queries for sensed context



<?p person;hasStatus Sleeping -> turnOffLight>
sleepRule
<?p rdf;type Person>
Person
{ Mr.Lee, Miss Lee, Mr.Park,…}
<?d rdf;type Device>
Devices
{ LampInBedroom, Bed, TVInLivingRoom, …}
<?d device;locatedIn Bedroom>
Devices of which location is bedroom
{LampInBedroom, Bed, TVInBedroom, AudioInBedroom, AirCleaner,…}

(b) Final state of query-tree based on Lighting application's query

Figure 3. An Example of Query-Tree

First, we use a constraint predicate to prune the contexts irrelevant to an application's operation. Context queries in a context-aware application are described by the notation of *triple match* which is one of ontology query languages. It returns all statements that match with a template in a form of (subject, predicate, object), where each term is either a constant or a don't-care (Wilkinson et al. 2003). Our context ontology described by using OWL is also encoded in a triple form and allows a property to specify the restrictions of its domain and range. Thus, for a given context query, we can limit the search space of the query from the restrictions described in the context ontology. For example, in a context query, `'<?p, locatedAt, ?r>'`, only a `'Person'` type of a context is allowed to be `'?p'` and only a `'Room'` type of a context is `'?r'` by the `'Person'` ontology. We define constraints specified by the restrictions of a predicate described in the upper-level ontology as constraint predicates. We build a query-tree based on the context queries of context-aware applications. Figure 3 (a) shows a query-tree based on a lighting application's query. In Figure 3 (a), `'<?p person;hasStatus Sleeping>'` query is for deduced context. A query for deduced context can be derived into sub-queries for other types of a context. Therefore, a given query is divided into sub-queries which are '<?p rdf;type Person>','<?d rdf;type Device>','<?p person;locatedAt Bedroom>','<?d device;locatedIn Bedroom>', and '<?d device;hasState ON>'. The label of each node in Figure 3 (a) represents the constraints specified by the predicate of each triple. Context queries for a context-aware application's operation are derived from building a query-tree. And then, the query-tree helps generate the relevant contexts satisfying the constraint predicates of each node on it. In the query of a lighting application, the number of relevant contexts pruned by the constraint predicates is about 3,000 out of almost 6,000 contexts.

After the first step, we use predicate types to filter out context queries for sensed contexts in a query-tree. As explained in the previous section 3.2, the queries for a sensed context cannot be any answered at the query-tree build time, so they can be filtered out from the query-tree. By filtering out the queries in advance, the number of queries needed to be pre-fetched is further decreased. Accordingly, the pre-fetch processing time is reduced and irrelevant contexts are pruned. Figure 3 (b) shows a query-tree after the first and second steps.

Finally, remaining context queries in a query-tree are evaluated at the pre-processing memory and then, the result of evaluating contexts are delivered into the working memory for further inference.

Figure 4 shows whole procedures of context pre-fetch.

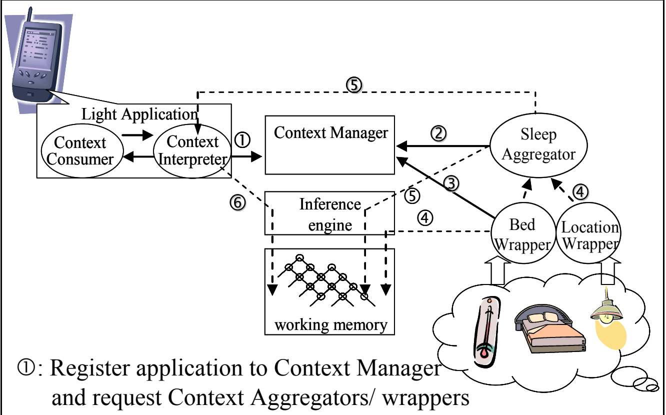
1. Initialize the pre-processing memory and the working memory.
   1.1. Load upper-level and lower-level context ontology into the pre-processing memory.
   1.2. Load upper-level context ontology into the working memory.
2. Build query-trees based on queries of a context-aware application at the time of initializing it.
3. Resolve queries for deduced context into queries for other types of context.
4. Filter out queries for sensed context in the query-tree.
5. Evaluate the remaining queries in the query-tree on the pre-processing memory loaded upper-level and lower-level context ontology.
6. Assert the results into the working memory for the inference engine.

Figure 4. Procedures of Context pre-fetch

4. Implementation

We have implemented the proposed method as part of our ubiquitous computing middleware, Active Surroundings (Lee et al. 2004). We designed a context ontology for a home environment in OWL and used the Jena2 Semantic Web Toolkit for evaluating rules and queries over the context ontology. First, we show a running process of a lighting application in the Active Surroundings without pre-fetch. Describing how a context-aware application runs using the middleware is to provide better understanding on how the pre-fetch method works for the middleware using the ontology-based context model.

A lighting application is depicted conceptually in Figure 5. The operation of the lighting application is very simple such that a light turns off when the user's status is 'sleeping'. In Figure 5, for the lighting application to be performed, it needs to be subscribed to the context management system in advance. The Context Wrapper, of which a concept is introduced in (Dey et al. 2001), transforms a signal from sensors into a form of a context and updates it to the working memory. The Context Aggregator possesses a rule to produce a conceptual context and generates it when the present values of a context satisfy the conditions of the rule. Sleep Aggregator checks whether a user is 'sleeping' or not. Therefore, the aggregator is automatically registered also. Likewise, required by the Sleep Aggregator, the Location Wrapper and the Bed Wrapper are registered automatically as well. Each of the registered aggregators and wrappers keeps a list of the applications and aggregators that use its context. When context changes occur to an aggregator or a wrapper, it notifies to the applications or aggregators in the list. At runtime, the Location Wrapper and the Bed Wrapper obtain contexts about current user location and bed status from the sensors and reflect them to the working memory. Then, the Sleep Aggregator examines the value of location and bed context. It concludes the current status of the user to be 'sleeping' and reflects it to the working memory. Finally, the lighting application examines whether the user's current status is 'sleeping', and turns off the light if conditions are satisfied.

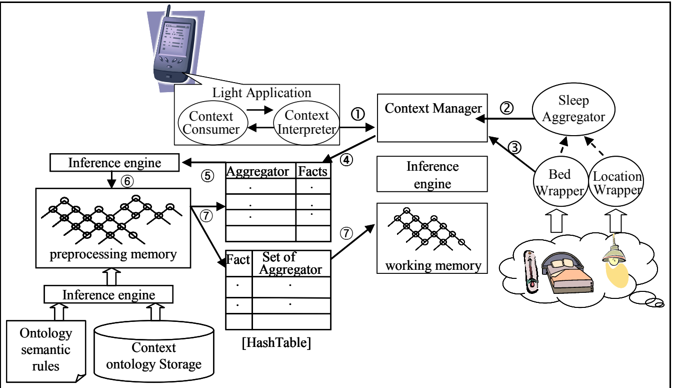


①: Register application to Context Manager and request Context Aggregators/ wrappers
②: Register Context Aggregator to Context Manager and request Context Wrappers
③: Register Context Wrappers to Context Manager
④: Notify context change from wrapper to aggregator / reflect the change into a working memory
⑤: Notify context change from aggregator to application / reflect the change into a working memory
⑥: Query to check the condition context to run application

Figure 5. Operation of Context-aware Application without pre-fetch method

As shown in Figure 6, the context management system delivers queries in the registered Context Aggregators and

Context Wrappers to the pre-fetch component at that time. We use two hash-tables to keep the consistency of a context ontology in the working memory. One of the hash-table stores context facts and a Context Aggregator as the hash keys to check whether the Context Aggregator is pre-fetched already. Context facts mean the results after pre-fetching about the Context Aggregator. The other hash-table stores pairs of a set of Context Aggregators and a context fact as the keys to check whether the context fact can be retracted from the working memory at runtime or not. The set of Context Aggregators means a set which consists of Context Aggregators of which pre-fetch result is the context fact. Through the hash-tables, the result set of pre-fetch for a Context Aggregator is stored and managed to support dynamic assertion to and retraction from the working memory. For instance, when a new application is registered at runtime, Context Aggregators needed by the application are checked in the first hash-table if they were pre-fetched already. If they exist in the hash-table, the pre-fetch process is skipped. On the other hand, when a Context Aggregator is unregistered to the context management component, the context facts corresponding to the Context Aggregator are checked using the first hash-table. Then, whether the context facts are still used by other Context Aggregators is checked using the second hash-table. If none of Context Aggregators uses context facts, then they can be retracted safely from the working memory. Figure 6 shows operation process of the lighting application on top of Active Surroundings with the proposed method.



①: Register application to Context Manager and request Context Aggregators/ Wrappers.

②: Register Context Aggregator to Context Manager and request Context Wrappers.

③: Register Context Wrappers to Context Manager

④: Check whether the requested Context Aggregator is pre-fetched already.

⑤: If the Aggregator is not in the hashtable [aggregator-facts], build a query-tree of the query in the ContextAggregator.

⑥: Pre-fetch relevant context using the query-tree.

⑦: Add the result facts to [aggregator-facts] and to [fact-set of aggregator] hashtable, and insert the facts into working memory.

Figure 6. Operation of Context-aware Application with pre-fetch method

5. Evaluation

5.1. Proof of Completeness

We show Theorem 1 that the results of a context-aware application's query both on contexts pruned by the pre-fetch method and on the whole contexts are the same through a proof by contradiction.

Theorem 1: The result of a context-aware application's query on contexts pruned by the pre-fetch method is the same as the one on the whole contexts.

Proof : Let q be a context-aware application's query, and $S_{\text{pre-fetch}}$ be contexts pruned by pre-fetching about q, and $S_{\text{nonpre-fetch}}$ be the whole contexts set, and C_{sensed}, C_{deduced}, C_{defined} be the set of sensed contexts, the set of deduced context, and the set of defined context in $S_{\text{nonpre-fetch}}$ respectively, and C_{sensed}', C_{deduced}', C_{defined}' be the set of sensed context, the set of deduced context, and the set of defined context in $S_{\text{pre-fetch}}$ respectively.

Suppose that the result of an application's query on pruned contexts by the pre-fetch method is different from the one on the whole contexts set. Since the types of context are only three, i.e., sensed context, defined context, and deduced context (referred to as Context categorization in section 3.2.), the types of a context query are also three. Thus, the proof is shown in each case separately.

Case 1: Assume that q is a query for the sensed context,

- Results of q on $S_{\text{pre-fetch}} \in C_{\text{sensed}}'$ and Results of q on $S_{\text{nonpre-fetch}} \in C_{\text{sensed}}$
- $C_{\text{sensed}}' = C_{\text{sensed}}$, because both C_{sensed}' and C_{sensed} are given from sensors at runtime.

Thus, the results of q both on $S_{\text{pre-fetch}}$ and on $S_{\text{nonpre-fetch}}$ are the same, when q is a query for the sensed context.

Case 2: Assume that q is a query for the defined context,

- Results of q on $S_{\text{pre-fetch}} = C_{\text{defined}}'$ and Results of q on $S_{\text{nonpre-fetch}} = C_{\text{defined}}' \subset C_{\text{defined}}$, because the results of q are C_{defined}' that is the result evaluated about q in the pre-fetch time.

Thus, the results of q both on $S_{\text{pre-fetch}}$ and on $S_{\text{nonpre-fetch}}$ are the same, when q is a query for the defined context.

Case 3: Assume that q is a query for the deduced context,

- q is divided into sub-queries for the sensed context and defined context in the pre-fetch time,
- Results of q on $S_{\text{pre-fetch}} \in C_{\text{sensed}}' \cup C_{\text{defined}}'$
- C_{sensed}' and C_{defined}' are proved by Case 1, Case 2.

Thus, the results of q both on $S_{\text{pre-fetch}}$ and on $S_{\text{nonpre-fetch}}$ are the same, when q is a query for the deduced context. In case that q is a query for the deduced context which consists of another deduced context, we can prove it in the same way of the case 3 by decomposing deduced sub-queries recursively until all sub-queries are decomposed into sensed and defined context queries. From the cases 1, 2, and 3, we have a contradiction, which means that the assumption is false. Therefore, it must be true that the result of a context-aware application's query on contexts pruned by the pre-fetch method is the same as the one on the whole contexts.

5.2. Experimental Result

Experiments were run on a 3.0GHz PC with 1GB of RAM running Windows XP. Our context model in use consists of about 2000 RDF Triples. It can be seen as a small size of context. To show the improvement of the time taken for inference on a large scale context, we extend our context model by defining several domain areas. We also prepare the different type of queries from a simple query to a complex one and practical queries in use on a running system as shown in Figure 7.

Query	Description
Q1 (Simple query)	<?p rdf:type Person> ∧ <?p locatedAt Bedroom>
Q2 (Complex query)	<?p rdf:type Person>∧<?p gender ?pg>∧ <?p birthDate ?pbr>∧<?p name ?pn>∧ <?p locatedAt ?pr>∧<?p hasStatus ?ps>∧ <?u rdf:type UserPreference>∧ <?u onPerson ?p>∧<?u hasWeight ?uw>∧ <?u hasService ?us>∧<?d rdf:type Device>∧ <?d used ?du>∧<?d hasService ?ds>∧ <?d hasState ?dst>∧<?d hasDimLevel ?ddl>∧ <?d locatedIn ?r>∧<?r rdf:type Room>∧ <?r hasPerson ?p>∧<?r hasDevice ?d>∧ <?r SoundLevel ?rs>∧<?r DoorState ?rd>∧ <?r Brightness ?rb>
Q3 (WatchTV rule)	<?p rdf:type Person>∧<?d rdf:type Device>∧ <?p locatedAt ?d>∧<?d hasState xsd:true>
Q4 (Sleep rule)	<?p rdf:type Person>∧<?d rdf:type Device> <?p locatedAt Bedroom>∧ <?d locatedIn Bedroom>∧ <?d hasState xsd:true>∧ <?l hasDimLevel xsd:0>
Q5 (EnterBedroom rule)	<?p rdf:type Person>∧ <?p locatedAt BedroomDoor>

Figure 7. Sample query for evaluation

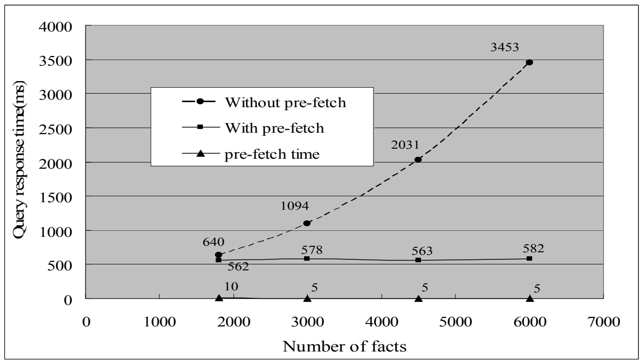


Figure 8. Query response time in simple query

Figure 8 shows the result of the experiment for Q1 (simple query) on context database in the working memory. As described in Figure 7, Q1 query is very plain. Thus, the response time of query is affected strongly by number of context facts in a working memory. Therefore, the query response time also increase as a consequence when the number of facts increases in the case without applying pre-fetch method. While, the graph of the result using the pre-fetch method shows the fixed response time for a query regardless of the increase of the number of facts. The processing time for pre-fetching the relevant context facts is negligible.

Figure 9 shows the result of the experiment for Q2 (complex query) on context database in the working memory. Q2 is a complex query that needs most of context facts stored in the working memory. For the complex query, the response time of the case applying the pre-fetch method increases according to the number of context facts. It is because there are many relevant context facts pre-fetched for the Q2 query. In the case of a very complicated query, many context facts are pre-fetched. However, even the worst case, the response time becomes no longer than the case without pre-fetch because the size of pre-fetched results is not bigger than that of the whole context set in any case.

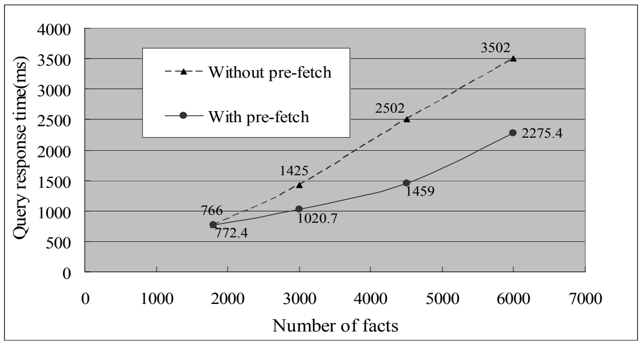


Figure 9. Query response time in complex query

Finally, we test an environment where several realistic applications supporting people's daily life run actually. As shown in Figure 7, three queries, Q3, Q4, and Q5, are used to activate our sample applications. In Figure 10, each graph shows the increase of the response time according to the number of context facts at runtime.

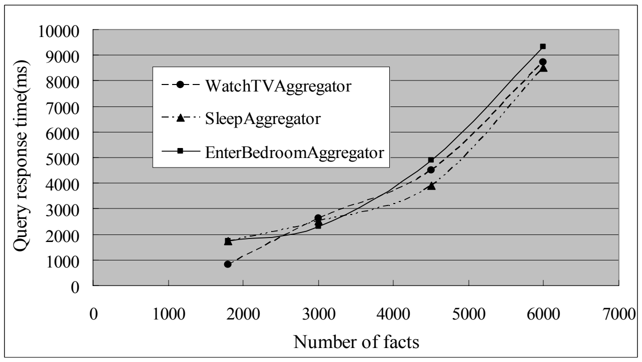


Figure 10. Query response time in Active Surroundings which doesn't apply the pre-fetch method

The results of the experiment with applying the pre-fetch method are shown in Figure 11. The value depicted in graph is the time which adds the processing time for pre-fetch and the response time of the three queries. As shown in Figure 11, the response time remains constant.

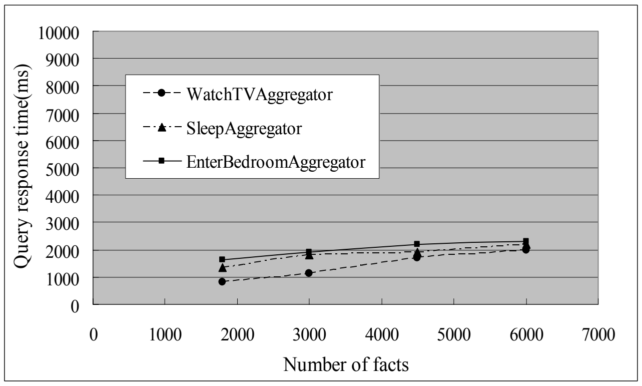


Figure 11. Query response time in Active Surroundings which apply the pre-fetch method

We conclude from the experiments described above that when the number of context facts is less than about 2000, both the result with pre-fetch method and one without pre-fetch method show similar performance. However, if the number of facts is over 2000, the method with the pre-fetch method significantly outperforms the method without the pre-fetch method.

6. Conclusion and Future works

Nowadays, there are number of infrastructures for enabling context-awareness on the basis of ontology-based context model. We consider it will be needed that the module or method which makes an inference process faster over ontology-based context model.

In this paper, we proposed a method to reduce the size of the contexts in working memory by pre-fetching relevant context based on context-aware application's queries. And we apply the method to the existing context management system which uses ontology-based context model, Active Surroundings. By pre-fetching and maintaining relevant context to support context-aware applications into working memory, the inference time on ontology-based context model can be faster than existing method which maintains the whole contexts into working memory. As the result the context-aware applications which use the context generated through inference process can be run quickly comparing with existing method. We currently investigate how to use query optimization techniques together at the pre-fetch time in order to reduce the processing time for pre-fetch.

References

Wang, X.; Dong, J.S.; Chin, C.Y.; Hettiarachchi, S.R.; and Zhang, D. 2004. Semantic Space: an infrastructure for smart spaces. *Pervasive Computing, IEEE. 3(3): 32– 39.*

Levy, A.Y. et al. 1997. Speeding up inferences using relevance reasoning: a formalism and algorithms. *Artificial Intelligence. 97(1-2): 83-136.*

Dey, A.K. et al. 2001. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction Journal. 16(2-4):97-166.*

Gu, T.; Wang, X.H.; Pung, H.K.; and Zhang,D.Q. 2004. An Ontology-based Context Model in Intelligent Environments. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference.*

Henricksen, K. et al. 2002. Modeling Context Information in Pervasive Computing System. *Pervasive 2002, LNCS 2414, 167-180.*

Kindberg, T. et al. 2000. People, places, things: Web presence for the real world. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications.*

Chen, H.; Finin, T.; and Joshi, A. 2004. An Ontology for Context-Aware Pervasive Computing Environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review.*

Lee, D. et al. 2004. A Group-Aware Middleware for Ubiquitous Computing Environments. In *Proceedings of the 14th International Conference on Artificial Reality and Telexis-tence (ICAT).*

Carroll, J.J. et al. 2004. Jena: Implementing the Semantic Web Recommendations. *WWW2004.*

Racer. Available online at http://www.racer-systems.com/.

FaCT. *Description Logic (DL) classifier.* Available online at http://www.cs.man.ac.uk/~horrocks/FaCT/.

Hoolet. *OWL-DL Reasoner.* http://owl.man.ac.uk/hoolet

Triple, http://triple.semanticweb.org/

Zou, Y. et al. 2004. F-OWL: an Inference Engine for the Semantic Web. In *Proceedings of the 3rd International Workshop on Formal Approaches to Agent-Based Systems.*

Baus, J. et al. 2002. A resource-adaptive mobile navigation system. In *Proceedings of Intl. Conf. on Intelligent User Interfaces, San Francisco.*

Fensel, D. et al. 2000. Lessons learned from applying AI to the web. *International Journal of Cooperative Information Systems. 9(4):361-382.*

Freeman-Hargis, J. 2003. Rule-based systems and Identification Trees. Available online at http://ai-depot.com/Tutorial/RuleBased.html.

Ranganathan, A. and Campbell, R.H. 2003. An Infrastructure for Context-Awareness based on First Order Logic. *Journal of Personal and Ubiquitous Computing. 7(6) :353-364.*

Khedr, M. and Karmouch, A. 2004. ACAI: Agent-Based Context-aware Infrastructure for Spontaneous Applications. *Journal of Network & Computer Application.*

Wilkinson, K.; Sayers, C.; Kuno, H.A.; Reynolds, D.; and Ding ,L. 2003. Supporting Scalable, Persistent Semantic Web Applications. *IEEE Data Eng. Bull. 26(4): 33-39.*