IT 09 035
Examensarbete 30 hp
November 2009

# Searching Web Feeds from a Functional Database Management System

Niklas Gåfvels

Institutionen för informationsteknologi
*Department of Information Technology*

UPPSALA UNIVERSITET

**Teknisk- naturvetenskaplig fakultet**
**UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

## Abstract

# Searching Web Feeds from a Functional Database Management System

*Niklas Gåfvels*

Web feeds are a popular technique to distribute information about contents of web pages. RSS and Atom are two standards used to syndicate web contents as web feeds. This project investigates how to make different kinds of Internet web feeds searchable by implementing a general wrapper for web feeds in an extensible and functional DBMS, Amos II. The system, RSS-Amos, makes it possible to search the contents of any RSS or Atom based web feed using the query language AmosQL. New web feeds simply have to be declared to the system in order to make them searchable. The system guarantees that added feeds always are up to date when queries are made. The wrapper is implemented in Java using the ROME API from java.net. The project includes an evaluation of the performance of the system. Due to the fact that the actual data sources are located on the Internet, a cache of read feeds has been implemented to improve performance. The cache makes queries over 150 times faster.

Handledare: Tore Risch
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 09 035
Tryckt av: ITC

# 1. Introduction

The Internet consists of numerous web pages presenting news articles. Two common goals of web pages are to maximize the amount of information that can be presented on the display and to reach as large public as possible. *Web feeds* provide a popular technology to represents and distribute web pages in a compact format. RSS [1] and Atom [4] are two standards used when web contents are distributed to reach a wider audience using web feeds. The web feed format makes it suitable for incorporation in other web pages, computer software and devices. The distribution of web contents is called *syndication* [6]. By syndication of web content it will reach a larger public than just using the web page alone. An *RSS web feed* consists of a list of triples of title, summary and a link to the article. If the reader finds the information interesting the whole story can be accessed with the provided link. It is common to use software called *aggregators* [27] that keep track of multiple feeds. Aggregators automatically inform the reader when there are updates made on a site. There exist aggregators for all kinds of devices, e.g. mobile phones and PDAs.

The RSS-Amos system implements a general query facility to search different kinds of web feeds. It is based upon the Amos II functional database system [18], which can be extended to query new data sources. A *wrapper* is an interface between Amos II and a data source. A wrapper makes it transparent to query the new data source using a query language. The RSS-Amos implementation includes a wrapper for web feeds. The wrapper is implemented in Java using available public Java-based libraries for web feed access. A *foreign function* in Amos II is a function written in some external language that can be used in queries. The wrapper mechanism uses foreign functions written in Java and the *ROME* [15] library to download and parse the feeds and articles.

Having the web feeds as data sources makes it possible to query them with Amos II using AmosQL [1] [4] [6][21] or SQL [8]. Queries can be specified to search and join web feeds, searching for, e.g. syndicated articles.

RSS-Amos stores in an Amos II database meta-data about known web feeds. The address of each feed stored in the meta-database is used when articles belonging to the feed are downloaded.

To increase the performance and limit the need to access the Internet, a cache for web feeds is implemented in RSS-Amos using main memory tables in Amos II. In an improved parallel feed caching implementation, Java threads are used to increase the performance by downloading multiple web feeds in parallel.

# 1 Background

## 1.1 Web feeds

Web feeds is a technique to represent the contents of a web page as a "stream" of information. In Swedish the translation for web feed is *ström* or *flöde*. Most larger web sites use web feeds to inform the human readers about the latest news on their site e.g. BBC, CNN, Apple, or Google. A web feed contains *syndicated* web contents meaning that the web content is going to be spread/distributed outside the original web page. A web feed consists of a title, a summary of the news, and a link to the web page containing the full article [1][4][6]. Usually a user subscribes for a feed in order to get updates automatically. A *web feed reader* (or just *reader*) is a program that shows the feed in some kind of GUI (Graphical User Interface). A web feed makes it possible for a web page to reach a larger public resulting in a higher hit rate for the web page. The web page showing the web feed may also get a higher hit rate when readers can get more information and usage from the page. The feeds can be shown in many formats. You can have a web feed as a screen saver (the news are rolling over the screen), show the web feed in your web page, get a pop up in the taskbar when there are new news, read the web feed in your mobile phone, or use a web feed reader where you can have numerous feeds showing in a Internet Explorer called *aggregators* [27].

There exists numerous free RSS search engines on the Internet. Many of them have focus on searching in blogs but also news feeds, e.g. www.*search4rss.com*, *www.plazoo.com*, *www.google.com/reader* and *www.yourfeeds.com*. Many of these search engines have the same search layout and search capabilities: a textbox, a search button, and the possibility to filter with a given category.

Web feeds are not suitable for representing all kinds of web pages. A suitable web page is a page where the contents changes dynamically. The best example is news papers on the Internet. News papers on the Internet usually post information about new articles as they arrive to a news paper. A news article usually consists of a title, a summary and a link to the whole story, which is also the normal way to format feeds [1][4][6].

RSS [1] and Atom [4] are the two different standards used to syndicate web contents as a web feed.

I have found one example of program importing RSS feeds [1] into relational databases. The program is called *UltimateNews - RSS to database fetch 2.0* and it periodical reads RSS feeds [1] and stores the information in one of the DBMSs *MS SQL*, *MySQL*, *Oracle*, or *MS Access* [28].

In this project all versions of RSS and Atom feeds [1][4] can be imported into Amos II making it possible to query them using AmosQL [18]. The system automatically makes sure that feeds used are up to date when they are used in a query.

### 1.1.1 RSS

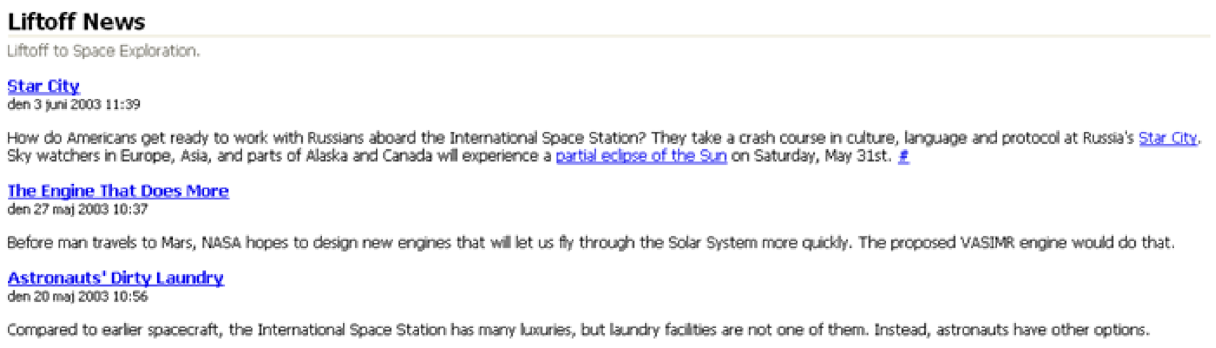
RSS is a general format used for representing web feeds. RSS web feeds are called *RSS channels*. The following terms are used as synonyms for RSS channel: *RSS, RSS feed, RSS/XML, or RSS/RDF*. RSS (Real Simple Syndication, Rich Site Summary, or RDF Site Summary) has a multicoloured history. The different names are a good example of this. RSS started with Netscape in 1999 with version 0.90 [1][13][16]. Netscape released version 0.91 before they decided to stop their development of RSS. Another company named *UserLand Software* made their own version of RSS version 0.91 [1][13][16]. There are some differences between the two versions but the structure is the same, e.g. the XML element *textinput* in Netscape's version is named *textInput* in the version from UserLand Software and the way to represent hour of day in Netscape's version is 0-23 while UserLand Software's version uses 1-24 [12]. UserLand Software has released version 0.92, 0.93 and 0.94 before the release of their final version, version 2.0 [1][13]. There exists a version 1.0 of RSS developed by *RSS-DEV Working Group* [17]. This group based their version on the original version from Netscape, version 0.90. However, RSS Version 1.0 uses RDF (Resource Description Framework) making this version incompatible with all the versions from UserLand Software. RDF is a standard used to describe web meta-data [24]. UserLand Software released their final version of RSS as version 2.0. However, there actually exists two versions of RSS version 2.0 [1][13][16]. The first is the version from UserLand and the second version is from *Berkman Center for Internet & Society at Harvard Law School* [1]. In June 2003 *Berkman Center* [1] got to be the owner of the RSS specifications. There have been some small changes to the UserLand Software specifications but the new releases is still called version 2.0.

**Table 1: RSS version history**

| Version | Date |
|---|---|
| 0.90 | 1999-03-15 |
| 0.91 Netscape | 1999-07-10 |
| 0.91 UserLand | 2000-06-04 |
| 0.92 | 2000-12-25 |
| 0.93 | 2001-04-20 |
| 0.94 | 2002-08 |
| 1.0 | 2000-08-14 |
| 2.0 UserLand | 2002-09-18 |
| 2.0 Harvard | 2003-07-15 |

RSS-Amos uses the specification of RSS version 2.0 from the Berkman Center at Harvard [1] as template when representing feeds and in the creation of data structures. The format of Atom [4] is handled by mapping into RSS version 2.0 [1].

Figure 1 shows an example of how an RSS version 2.0 web feed looks in a browser. The textbox shows the XML code representing the web feed.



**Liftoff News**

Liftoff to Space Exploration.

Star City
den 3 juni 2003 11:39

How do Americans get ready to work with Russians aboard the International Space Station? They take a crash course in culture, language and protocol at Russia's Star City.
Sky watchers in Europe, Asia, and parts of Alaska and Canada will experience a partial eclipse of the Sun on Saturday, May 31st. #

The Engine That Does More
den 27 maj 2003 10:37

Before man travels to Mars, NASA hopes to design new engines that will let us fly through the Solar System more quickly. The proposed VASIMR engine would do that.

Astronauts' Dirty Laundry
den 20 maj 2003 10:56

Compared to earlier spacecraft, the International Space Station has many luxuries, but laundry facilities are not one of them. Instead, astronauts have other options.

**Figure 1: Example of an RSS version 2.0 document**

```
<?xml version="1.0"?>
<rss version="2.0">
   <channel>
      <title>Liftoff News</title>
      <link>http://liftoff.msfc.nasa.gov/</link>
      <description>Liftoff to Space Exploration.</description>
      <language>en-us</language>
      <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>
      <lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
      <docs>http://blogs.law.harvard.edu/tech/rss</docs>
      <generator>Weblog Editor 2.0</generator>
      <managingEditor>editor@example.com</managingEditor>
      <webMaster>webmaster@example.com</webMaster>
      <item>
         <title>Star City</title>
         <link>http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp</link>
         <description>How do Americans get ready to work with Russians aboard the International
Space Station? They take a crash course in culture, language and protocol at Russia's &lt;a
href="http://howe.iki.rssi.ru/GCTC/gctc_e.htm"&gt;Star City&lt;/a&gt;.</description>
         <pubDate>Tue, 03 Jun 2003 09:39:21 GMT</pubDate>
         <guid>http://liftoff.msfc.nasa.gov/2003/06/03.html#item573</guid>
      </item>
      <item>
         <description>Sky watchers in Europe, Asia, and parts of Alaska and Canada will
experience a &lt;a
href="http://science.nasa.gov/headlines/y2003/30may_solareclipse.htm"&gt;partial eclipse of the
Sun&lt;/a&gt; on Saturday, May 31st.</description>
         <pubDate>Fri, 30 May 2003 11:06:42 GMT</pubDate>
         <guid>http://liftoff.msfc.nasa.gov/2003/05/30.html#item572</guid>
      </item>
      <item>
         <title>The Engine That Does More</title>
         <link>http://liftoff.msfc.nasa.gov/news/2003/news-VASIMR.asp</link>
         <description>Before man travels to Mars, NASA hopes to design new engines that will let
us fly through the Solar System more quickly.  The proposed VASIMR engine would do
that.</description>
         <pubDate>Tue, 27 May 2003 08:37:32 GMT</pubDate>
         <guid>http://liftoff.msfc.nasa.gov/2003/05/27.html#item571</guid>
      </item>
      <item>
         <title>Astronauts' Dirty Laundry</title>
         <link>http://liftoff.msfc.nasa.gov/news/2003/news-laundry.asp</link>
         <description>Compared to earlier spacecraft, the International Space Station has many
luxuries, but laundry facilities are not one of them.  Instead, astronauts have other
options.</description>
         <pubDate>Tue, 20 May 2003 08:56:02 GMT</pubDate>
         <guid>http://liftoff.msfc.nasa.gov/2003/05/20.html#item570</guid>
      </item>
   </channel>
</rss>
```

RSS version 2.0 is a dialect of XML, which means that it has some special XML-tags following the XML 1.0 specification [25]. Usually a dialect contains a *namespace* defining the elements of the dialect. However, the elements of RSS version 2.0 do not belong to a namespace. The motivation for this is that the use of a namespace would make version 2.0 incompatible with earlier versions of RSS. A valid RSS version 2.0 document must follow the specifications on the Berkman Center site [1]. It is valid to extend the dialect but then a namespace has to be defined for the new elements and attributes and the name must be changed.

A *channel* contains the meta-data about a web feed. In Table 2 you can see all existing meta-data elements belonging to an RSS channel version 2.0. Required elements are marked with green/dark colour [1].

**Table 2: Meta-data elements of an RSS channel**

🟩 Required element

| Element | Description |
|---|---|
| Rss | The attribute version representing the Channel version. |
| Title | The title of the feed e.g. *BBC News* |
| Description | A text describing the feed e.g. *Visit BBC News for up-to-the-minute news, breaking news …* |
| Link | The address to this feed **or** a web page e.g. *http://news.bbc.co.uk/go/rss/-/2/hi/europe/default.stm* |
| Image | A picture/icon showing on the top of the feed e.g. |
| Language | The natural language of the article, e.g. *en-gb* |
| Cloud | Indicates that it is possible to be notified when a feed is updated. |
| Copyright | Copyright notice, e.g. *Copyright: (C) British Broadcasting Corporation* |
| Docs | A link to a document describing the RSS structure e.g. *http://www.bbc.co.uk/syndication/* |
| lastBuildDate | The date and time when the channels was updated last e.g. *Mon, 02 Mar 2009 18:18:24 GMT* |
| managingEditor | The e-mail address to the person responsible for the the channel |
| pubDate | The date when this channel was published e.g. *Mon, 02 Mar 2009 08:11:12 GMT* |
| Rating | PICS rating as an integer |
| skipDays | The days of the week when there will be no updates to the channel, e.g. *Saturday, Sunday* |
| skipHours | The hours of the day when there will be no updates to the channel, e.g. *0-23* |
| webMater | E-mail address to the system administrator hosting the channel |
| Category | One or several categories explaining the type of contents of the channel, e.g. *Business, Europe* |
| Generator | The name of the program that created the channel |
| Ttl | An integer telling how often the channel should be updated by the browser or reader. E.g. *ttl=15* means that the channel should be read every 15 minutes |
| textInput | A textbox that can be used for adding comments from readers |

The three elements *rss*, *title*, and *description* are the only required ones in an RSS Channel version 2.0. Elements contained in other elements are called *sub-elements*. A sub-element represents additional properties belonging to an element, e.g. the *image* element have six properties, *title*, *url*, *link*, *description*, *width*, and *hight*, which is stored in sub-elements.

**Table 3: Sub-elements to the meta-data of an RSS channel**

| Element | Sub-elements/attributes | | | | | |
|---|---|---|---|---|---|---|
| Image | title | url | link | description | width | hight |
| Cloud | domain | port | path | registerProcedure | protocol | |
| textInput | title | description | name | link | | |

News/stories/articles are called *items* in RSS. There is actually no requirement to have any news in the channel. The real content about each article is stored in item elements. A channel can have any number of items. Table 4 shows all sub-elements allowed for items [1]. At least one of the sub-elements *title* or *description* needs a value. It is possible that the whole news/article/story is present in the *description* element and it is only the *description* element that is allowed to contain HTML encoding.

**Table 4: Sub-elements belonging to an item of an RSS channel**

| Element | Description |
|---|---|
| title | The title of the article/news |
| link | The address to the site with the full article/news |
| description | A summary of the article/news |
| author | An email address to the person that wrote the article/news or the person responsible for the channel |
| category | One or more elements describing the category of the article/news, e.g. *Sweden* or *Economy* |
| comments | A URL to a webpage with comments to the article/news |
| enlcosure | Indicates whether there is some media associated to the item, e.g. a picture or audio file |
| guid | A string representing a globally unique identifier. It can be used to identify if an article is new. This is often the same as the link. |
| pubDate | The date when the article/news was created. The format of the date is specified in *rfc 822*.**Fel! Hittar inte referenskälla.**. |
| source | Indicates whether the information came from another feed, in which case the address to that is specified as XML |

The following four sub-elements of an item have their own sub-elements:

**Table 5: Sub-elements to an RSS item**

| Element | Sub-elements/attributes | | | | |
|---|---|---|---|---|---|
| category | domain (optional) | | | | |
| comments | url | | | | |
| enlcosure | url | length | type | hight | width |
| source | url | | | | |

### 1.1.2 Atom

As RSS, Atom is a standard used to syndicate web contents. A web feed created by Atom is called a *feed* corresponding to a *channel* for RSS. The first version of Atom, version 0.3, was created in December 2003. The motivation for this new syndication format was the fact that the specifications of RSS version 2.0 were frozen to preserve backward compatibility and modifications to RSS version 2.0 had to be done under another name [1][6]. Due to these facts a new syndication format named Atom was created. Atom is implemented without the need of backward compatibility to the multicoloured history of RSS. Atom has an XML namespace (*http://www.w3.org/2005/Atom*). With Atom it is possible to store different kinds of human readable text content in an element, e.g. one element may contain pure text and another one html and both elements can be parsed correctly by a reader. Text elements may have a *type* attribute specifying the type of contents. In RSS only the description could contain HTML encoding. In 2004, Atom version 1.0 was released and the specifications of Atom moved into the Internet Engineering Task Force (IETF) under *rfc 4287* [6]. Moving to IETF was a tactical move to make Atom more attractive than RSS. However, RSS version 2.0 is still the most popular feed syndication format. Big sites like BBC, CNN, and Apple use RSS version 2.0 [1].

The structure of the meta-data of an Atom feed looks like this.

**Table 6: Meta-data elements of an Atom feed**

🟩 Required element

| Element | Description |
|---|---|
| 🟩 author | The author of the feed. |
| contributor | Optional co-worker of the author |
| category | Defines the category of the element, which is the sub-element label presented for humans |
| generator | The tool used to create the feed |
| icon | The image representing an icon to the feed |
| 🟩 id | A unique id for the feed |
| link | A reference to a web resource with information about the feed or the address to the feed itself . |
| logo | A picture that is larger then the icon |
| rights | Copyright information about the feed |
| subtitle | The description/subtitle of the feed |
| 🟩 title | The title of the feed |
| 🟩 updated | A date construct indicating the last change of the feed |

The following elements of an Atom feed have sub-elements.

**Table 7: Sub-elements to an Atom feed**

🟩 Required element

| Element | Sub-elements/attributes | | | | | |
|---|---|---|---|---|---|---|
| category | 🟩 term | scheme | Label | | | |
| generator | uri | version | | | | |
| icon | uri | | | | | |
| id | uri | | | | | |
| link | 🟩 href | rel | Type | hreflang | title | length |
| logo | uri | | | | | |

The news/story/article of an Atom feed is called an *entry* corresponding to an RSS item. The structure of an Atom entry looks like this.

**Table 8 elements of an entry**

Required element

| Element | Description |
|---|---|
| Author | The author of the entry |
| category | The category of the entry, i.e. the label presented for humans |
| content | A link to the content or the content itself. If the *src* attribute is present it means that the contents is provided as a link |
| contributor | Optional co-worked of the author |
| Id | A unique id for the entry |
| Link | A reference to a web resource other than the contents |
| published | The first time the entry was created |
| Rights | Copyright information about the entry |
| source | If the entry is taken from another feed the metadata about the original feed is stored here |
| summary | The summary of the contents of the entry |
| Title | The title of the entry |
| updated | The last change of the entry |

The following elements of an Atom entry have sub-elements.

**Table 9 sub-elements of an entry**

Requried element

| Element | Sub-elements/attributes | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Category | term | scheme | label | | | | | | | | |
| Content | type | text | src | | | | | | | | |
| Id | uri | | | | | | | | | | |
| Link | href | rel | type | hreflang | title | length | | | | | |
| Source | author | contributor | generator | Icon | id | link | logo | Rights | subtitle | title | updated |

An Atom feed can contain Atom entry elements but this is not required. The full specification for Atom can be found at http://www.w3.org/2005/Atom.

### 1.1.3 Mappings between RSS and Atom in RSS-Amos

RSS-Amos has a mapping between corresponding elements in RSS and Atom in order to be able to handle both formats. RSS version 2.0 from Berkman Center [1] is the basic template/model for the representation of web feeds in the system. The elements in RSS 2.0 are mapped to corresponding functions in RSS-Amos. Table 10 shows the mappings of the meta-data of the two web feed standards.

**Table 10: Meta-data mappings**

| RSS | Atom |
|---|---|
| title | Title |
| description | Subtitle |
| link | Link |
| image | logo element if present else the icon element |
| language | Taken from the xml:lang element of the XML document |
| cloud | - |
| copyright | Rights |
| docs | from the XML namespace |
| lastBuildDate | Updated |
| managingEditor | email from author |
| pubDate | Published |
| rating | - |
| skipDays | - |
| skipHours | - |
| webMaster | email element if present else the name element if present, otherwise the uri attribute from the author element |
| category | label element if present, otherwise the term attribute from the category element |
| generator | Generator |
| ttl | - |
| textInput | - |

The *version* element is not used in RSS-Amos. The motivation for this is that all versions are treated as RSS version 2.0. There is no mapping for the elements *cloud, rating, skipDays, skipHours, ttl,* or *textInput*.

The mapping of *item* and *entry* looks like this:

**Table 11 mapping of an *item* and *entry***

| RSS | Atom |
|---|---|
| Title | Title |
| Link | The href attribute from the link element |
| description | The summary if present otherwise content element |
| author | The sub-element name from author element if present otherwise from the contributor element |
| category | The term attribute from the category element |
| comments | - |
| enlcosure | All links with where sub-elements *rel* is not equal to alternate |
| Guid | Id |
| pubDate | published element if present otherwise the updated element |
| source | From author names if present otherwise either from contributor names if present or from the rights element |

All sub-elements except *comments* in an *entry* got a mapping in a *item*.

## 1.2 Amos II

Amos II (Active Mediator Object System) [21]is a DBMS with a functional database model. Amos II is designed to be stored in main memory (MM). Amos II has a functional query language called AmosQL. Amos II can be used as a standalone DBMS or a server. It is furthermore possible to search external data sources using the *wrapper* facilities of Amos II. The system can be used on Windows and Linux.

The functional database model used in Amos II consists of *objects*, *types*, and *functions*. The RSS-Amos wrapper represents a web feed as a user defined type named *Feed*.

### 1.2.1 Types

It is possible to create user defined *types* in Amos II. A user defined type consists of a name of the type and attributes represented be *functions* described in the chapter 1.2.2. Instances of *stored types* are objects stored in the local database. The command `create type` is used when creating stored types. For example, creating a stored type called *Person* with the attributes *firstname* and *secondname* is done with the command

```
create type Person properties (firstname Charstring, secondname
Charstring);
```

An object is represented by a *literal* or a *surrogate*. A surrogate is similar to an instance of a class in C++ or Java, which has to be explicitly created and deleted. A surrogate has an OID (Object Identifier). A literal is built-in type, e.g. *Charstring* and *Integer*.

Amos II has two types of collections, *bag* and *vector*. A bag is an un-ordered set of result tuples or objects. The result from a query in Amos II is represented by a bag of result tuples [18]. A vector represents a sequence of any object that can be indexed like an array [18]. A vector can be created using curly brackets e.g. `set :myvector = {8,9,10};`. The example created a vector with three elements. The second value (9) can be accessed using the index 1 i.e. `:myvector[1];`.

Amos II has another kind of type called *mapped type*. A mapped type differs from the user defined type in that instances of a mapped type are not stored in the database, but are defined through a query. A mapped type provides an object-oriented database view of data. In RSS-Amos mapped types represent views of objects retrieved from web feeds. Instances of mapped types must be identified with a unique key, which is given by the query specifying the mapped type. The specifying query is called a *core cluster function* that retrieves the instance of the mapped type. The syntax for creating a mapped type looks like this [18].

```
create_mapped_type(Charstring name, Vector keys, Vector attrs,
                   Charstring ccfn);
```

*name* is the name of the mapped type
*keys* specifies the unique key for each instance of the mapped type. The parameter *keys* is a vector containing the name or names of attributes that constitutes the unique key.
*attrs* is the names of all the properties of the mapped type.
*ccfn* is the name of the core cluster function.

RSS-Amos uses a mapped type called *Rssitem* to represent articles in feeds.

*Rssitem* will be further explained in Chapter 2.1.

### 1.2.2 Functions

Functions provide properties and attributes of objects. Functions are instances of the meta-type named *Function*. Defining an attribute *name* for the type *Person* is done by this function definition [18].

```
create function name(Person) -> Charstring as stored;
```

There are five different kinds of functions: *stored*, *derived*, *foreign*, *procedure* and *overloaded*. In the example above the function kind was *stored*; it defines attributes stored on instances of types. Some examples of signatures of stored functions for the type *Feed* are:

```
create function title(Feed) -> Charstring as stored;
create function description(Feed) -> Charstring as stored;
create function link(Feed) -> Charstring as stored;
```

Queries in AmosQL are expressed in terms of functions using an SQL-like selec-from-where syntax, for example:

```
select title(theFeed)
from Feed theFeed
where language(theFeed) = ”en-us”
```

A *stored function* is analogous to a table in a relational database or an attribute of a Java object. In this example the table would be named *name* containing data of the literal type *Charstring* and the table name is related to the type *Person*.

```
create function  name(Person)->Charstring as stored;
```

A *derived function* is a function that is defined in terms of other functions as a query. A derived function cannot update the database. An example of a derived function used in RSS-Amos is a function named *rss_TimeForUpdate(Charstring src)->Boolean* that computes the time span since the feed was updated. It uses the Amos II built in functions *timespan* and *now* combined with the property *lastupdate* of *Feed*.

```
timespan(Timeval, Timeval) -> <Time, Integer usec>
```
Compute difference in *Time* and microseconds between two time values [18].

```
now() -> Timeval
```
The current absolute time [18].

A *foreign function* is a function implemented in an external programming language. Amos II supports the external programming languages Java, C, C++, and Lisp. Java is the only external programming language used in this project. The declaration of a foreign function looks much like the declaration of a stored function. The following example is a foreign function that depends on a precompiled Java class named *StreamDirector*. In the Java class there has to exist a *public* method called *getStream* that has two arguments, one of the type *CallContext*, and one of the type *Tuple*. The Java method implementing the function *getStream* throws the exception *AmosException*. The directory containing the class *StreamDirector* has to be stored in the *CLASSPATH*. Here is an example of the Java method matching this description [19].

```
public void getStream(CallContext ctx, Tuple tpl) throws AmosException
```

The foreign function using *getStream* is declared in Amos II as:

```
create function rss_GetStream(Charstring)->Bag of
<Charstring,Charstring, Charstring, Charstring, Vector, Charstring,
Charstring, Charstring, Charstring, Vector, Charstring, Vector> as
foreign "JAVA:StreamDirector/getStream";
```

A *stored procedure* is a function that can change the state of the database. The body of the stored procedure can consist of multiple AmosQL statements. In RSS-Amos the id of a *Feed* is managed by the code below.

```
//Create a stored function for storing the next id
create function rss_rssstream_id()->Integer as stored;
set rss_rssstream_id() = 1;

//The stored procedure will change the value of the stored function
//rss_rssstream_id and return
create function rss_get_next_rssstream_id()->Integer as
begin
        declare integer id;
        set id = rss_rssstream_id();
        set rss_rssstream_id() = id + 1;
        result id;
end;
```

The stored procedure *rss_get_next_rssstream_id()* is called whenever a new id is needed.

*Overloaded functions* are functions that have different implementations depending on the arguments given. Different *resolvents* of an overloaded function have the same name but different signatures. A signature consists of the function name and the type of the arguments. This is an example of two overloaded procedures used in RSS-Amos:

```
create function rss_AddAndGetStream(Charstring src)->Boolean
create function rss_AddAndGetStream(Charstring src,
                                     Charstring short_name)->Boolean
```

A function can be *multidirectional*. This means that depending on what arguments are known (bound) different implementations can be called. This is a simple example from the user´s manual and it shows the usage of binding patterns [18][21][23]:

```
create function sqroots(Number x)-> Number r
       as multidirectional
          ("bf" foreign 'sqrts' cost {2,2})
          ("fb" foreign 'square' cost {1.2,1});
```

The example function has one argument and returns a literal. If the argument *x* is known (meaning that an argument value is passed when the call is made) the foreign function *sqrts* is called. If the *r* is known, but not *x*, the inverse foreign function *square* is called. If both *x* and *r* are known the query optimizer will call the cheapest of *sqrts* or *square*. To decide this, the optimizer is given *cost estimates*. The query optimizer can calculate costs for functions that do not use foreign functions, while for foreign functions the user can specify the estimated cost as in the example. The cost is specified as a *vector* with two values. The first value indicates how expensive the call is and the second value is the *fanout*. The fanout is the estimated size of the result.

RSS-Amos uses a multidirectional core cluster function where the cost and fanout may differ depending on the parameters given. For example, one of the binding patterns in the core cluster function representing the mapped type *Rssitem* requires the address of the feed to be bound and the RSS items are computed (i.e. unbound). This binding pattern has a fanout of 20. The fanout is set to 20 because the average number of articles of a feed is 20 (this is an average value that I have calculated based on 148 different feeds) [21][23].

# 2 The RSS-Amos system

Web feeds are treated as an external data source in RSS-Amos and data extracted from web feeds can be used in queries as any other data source. Figure 2 illustrates how RSS-Amos provides query facilities over different web feeds.

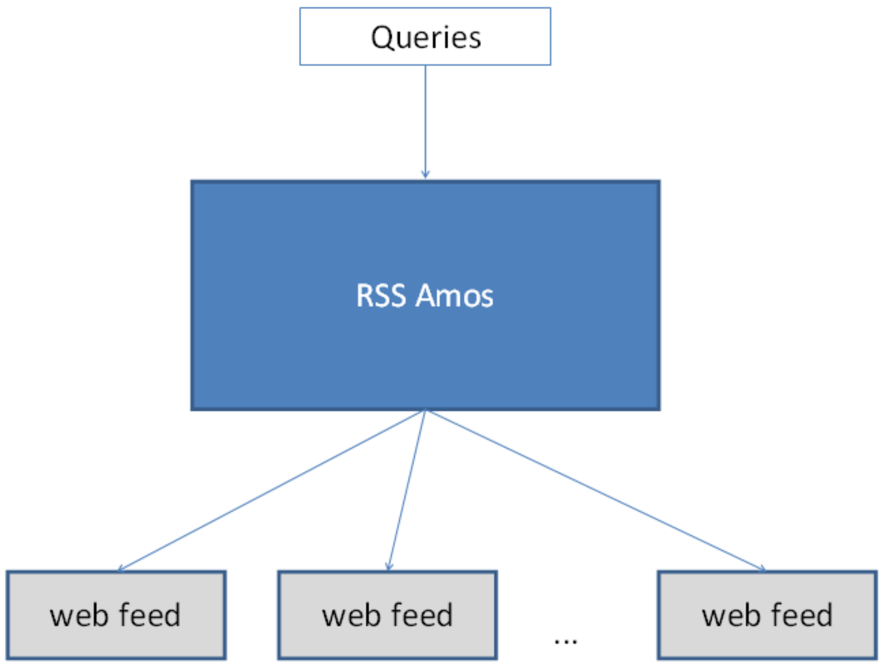


**Figure 2: High level view of RSS-Amos**

This is an example of a query that lists all titles from the articles in the web feed named *bbc*:

```
select title(article)
from Rssitem article
where short_name(feedof(article))="bbc";
```

RSS-Amos stores meta-data about web feeds. This meta-data is crucial for the system because it makes web feeds accessible from RSS-Amos queries. The user must explicitly register each new web feed with RSS-Amos. The meta-data is then automatically created when a user adds a web feed to the database. For example:

```
rss_AddAndGetStream(
'http://newsrss.bbc.co.uk/rss/newsonline_world_edition/europe/rss.xml',
'bbc');
```

RSS-Amos wraps articles from the RSS channels and Atom feeds as a mapped type called *Rssitem*. Meta-data about RSS channels and Atom feeds are stored as a type called *Feed*. These types can be used in queries.

Figure 3 shows the subsystems in RSS-Amos. The implementation of RSS-Amos consists of three layers. The top layer is the representations of articles from a web feed as instances of a mapped type *Rssitem*. Instances of this type are called *RSS items*.

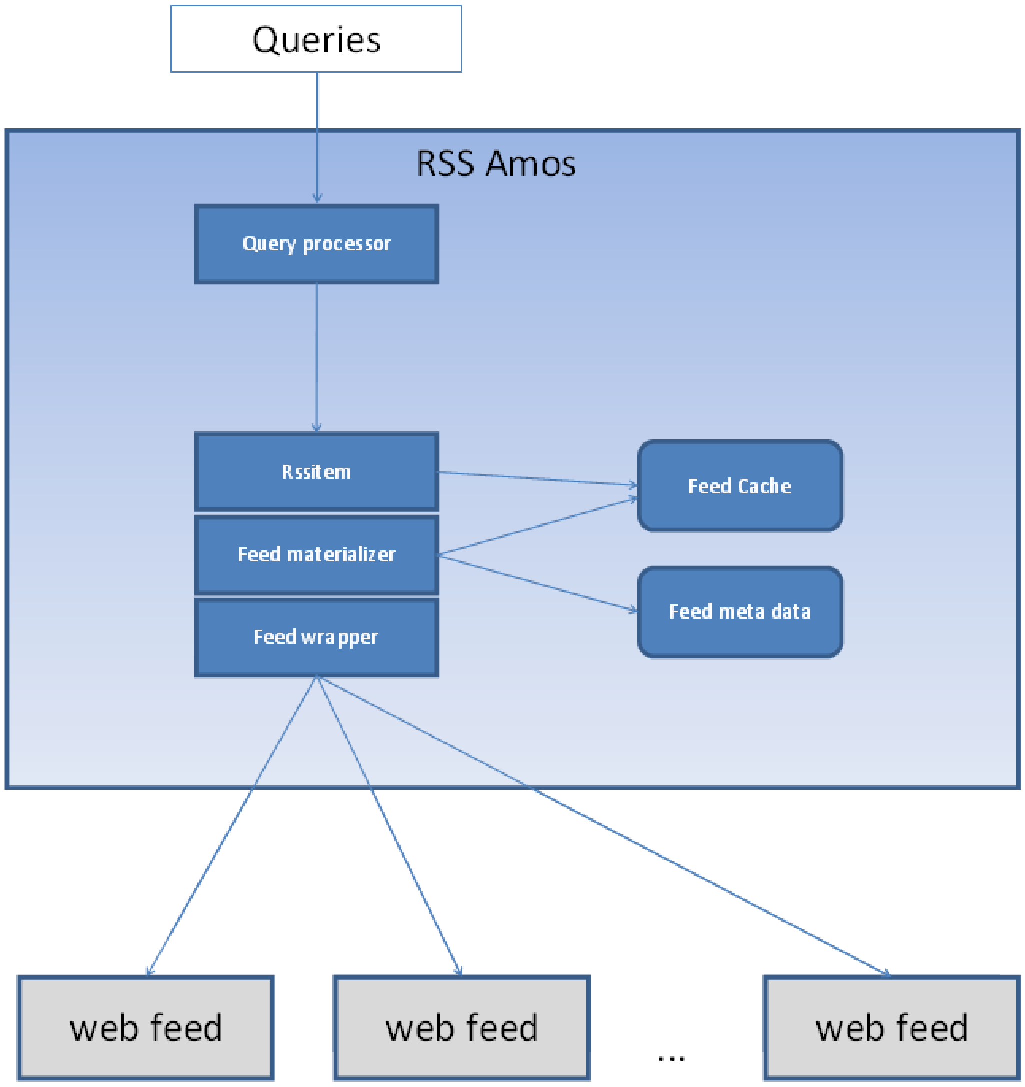


**Figure 3: RSS-Amos components**

The *query processor* is the general query processor of Amos II [21]. The *feed wrapper* is responsible for accessing the Internet and retrieving articles. The articles are downloaded from the Internet using foreign functions in Java emitting (streaming) tuples back to RSS-Amos for further query processing. The *feed materializer* is responsible for managing retrieved RSS items in the feed cache. The feed cache is used to increase the performance of querying *Rssitems*. The feed materializer uses the *feed meta-data* stored in the database when RSS items are retrieved. All meta-data is stored in a type called *Feed*. The feed materializer passes an address to a feed as an argument to foreign functions in the wrapper to retrieve the articles of the feed. The address of a retrieved feed is stored in the feed meta-data. Which feed to use depends on the query. The feed materializer assigns to each downloaded article a unique identifier, *uid*. The system checks if the same article is downloaded twice, in which case the old article is retaitned in the cache. The *uid* of the last cached article is stored in the stored function *rss_lastid()*.

The type *Rssitem* is a mapped type representing articles retrieved from web feeds. The declaration of the mapped type *Rssitem* looks like this:

```
create_mapped_type("Rssitem", {"uid"},
   {"uid", "title", "description", "description_type", "streamsrc",
    "link", "categories", "author", "pubdate", "source", "comments",
    "enclosures", "guid", "foreign_markup"}, "RSSItem_cc");
```

Here *create_mapped_type* creates a mapped type named *Rssitem* that use the core cluster function *RSSItem_cc* when retrieving an instance of the type *Rssitem*. The mapped type *Rssitem* includes the same properties as an item in a RSS channel version 2.0. Additional properties not found in RSS version 2.0 are marked with a star in Table 12.

The system function *create_mapped_type* will do some useful refactoring. The refactoring creates functions for every attribute of the mapped type e.g. *title(Rssitem)->Charstring* and *description(Rssitem)->Charstring*. The implementation of the core cluster function has varied through the project in order to investigate different implementation alternatives, which will be explained later.

The core cluster function is a multi-directional function that searches feeds. It will update the feed cache if the feed has not been updated within a *time to live* (*TTL*) , specific for each feed. The core cluster function maps retrieved tuples into objects of the mapped type *Rssitem*. The definition of the core cluster function looks like this:

```
create function RSSItem_cc()->Bag of
    <Integer uid key, Charstring title, Charstring description,
    Charstring description_type, Charstring streamsrc, Charstring link,
    Vector categories, Charstring author, Charstring pubdate,
    Charstring source, Charstring comments, Vector enclosures,
    Charstring guid,Vector foreign_markup> as multidirectional
    ("bfffffffffffff" select rss_Materialize(uid) cost{1,1})
    ("ffffbfffffffff" select rss_Materialize(streamsrc) cost{1,20})
    ("ffffffffffffff" select rss_MaterializeThread() cost {500,100000});
```

The core cluster function *rssItem_cc* is a multidirectional function that calls different stored procedures to retrieve RSS items for different binding patterns. The stored procedures update the feed cache when needed.

Table 12 lists the functions defined for type *Rssitem*.

**Table 12: Functions over the mapped type *Rssitem***

| Stored function | Type | Description |
|---|---|---|
| uid* | Integer | The unique id of an object of type *Rssitem*. |
| Title | Charstring | |
| description | Charstring | |
| description_type* | Charstring | The sub-element of a description |
| Streamsrc* | Charstring | The address to the feed |

| | | |
|---|---|---|
| Link | Charstring | |
| categories | Vector | Specifies one or several multiple categories in pairs of <name,domain> |
| Author | Charstring | |
| Pubdate | Charstring | |
| Source | Charstring | |
| comments | Charstring | |
| enclosures | Vector | Specifies one or several enclosures in pairs of <type, url, length, and optional fields...> |
| Guid | Charstring | |
| feedof* | Feed | Returns the *Feed* that the *Rssitem* belongs to |
| foreign_markup* | Vector | Specifies one or several foreign_markups in pairs of <optional fields...> |

Stored functions marked with * differ from the elements of the RSS v. 2.0 specification and they are explained below

* The stored function *uid* uniquely identifies objects of type *Rssitem*. These identifiers are maintained by the system when web feeds are imported.

* The stored function *description_type* is extracted as an own element from description to simplify usage.

* The stored function *streamsrc* is added to keep a link to the feed and it is used when articles are emitted from the feed wrapper.

* The stored function *feedof* defines a relationship to the feed that the *Rssitem* belongs to.

* The stored function *foreign_markup* contains additional elements found in RSS items and Atom entries that do not belong to the original specification, e.g. elements from a namespace.

The stored type *Feed* represents the meta-data about web feeds based on the elements in RSS channel version 2.0. The meta-data is shown in Table 2. Unlike *Rssitem* the type *Feed* is a regular stored type whose extent is stored in the Amos II database. Some additional properties that are not part of RSS 2.0 but used by the system are added to the *Feed* type.

The relationship between the type *Feed* and the mapped type *Rssitem* is shown in Figure 4. Every object of type *Rssitem* has a corresponding object of type *Feed* and the function *feedof(Rssitem)->Feed* stores the mapping. On the other hand, an object of type *Feed* may have several objects of type *Rssitem* since one feed usually consists of multiple articles.

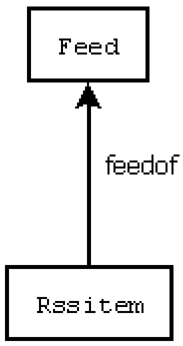


**Figure 4 Relationship between *Feed* and *Rssitem***

A more detailed description of the implementation will be described in the following chapters.

## 2.1 Design decisions

Three implementations were made during the development of RSS-Amos: the *naive implementation*, *feed caching*, and *parallel feed caching*. The different implementations represent the development cycle. The naive implementation had only the focus to make it possible to query an RSS channel from Amos II without any performance considerations. The feed caching implementation had focus on limiting the number of calls to the Internet by adding to the system a cache of articles. The parallel feed caching implementation increased the performance further by parallelizing the foreign function responsible of downloading articles from the Internet to the article cache. Parts of every implementation are reused in the other implementations.

### 2.1.1 Naive implementation

This was the first stage of the implementation of RSS-Amos. The focus was to retrieve articles from a feed located on the Internet without any caching and represent the articles as instances of the mapped type *Rssitem*.

This implementation consisted of the type *Feed*, the mapped type *Rssitem*, one core cluster function, one stored procedure, and two foreign functions implemented in Java. As mentioned objects of type *Rssitem* represent items from an RSS channel version 2.0 and objects of type *Feed* represent the meta-data of an RSS channel version 2.0. Below is the definition of functions over type *Feed* used in the naive implementation:

```
create function title(Feed)->Charstring as stored;
create function description(Feed)->Charstring as stored;
create function link(Feed)->Charstring as stored;
create function language(Feed)->Charstring as stored;
create function categories(Feed)->vector of Charstring as Stored;
create function copyright(Feed)->Charstring as stored;
create function managingEditor(Feed)->Charstring as stored;
create function webmaster(Feed)->Charstring as stored;
create function pubdate(Feed)->Charstring as stored;
create function lastbuilddate(Feed)->Charstring as stored;
create function generator(Feed)->Charstring as stored;
create function docs(Feed)->Charstring as stored;
create function cloud(Feed)->Vector of Charstring as stored;
create function image(Feed)->Vector of Charstring as stored;
create function rating(Feed)->Charstring as stored;
create function skipdays(Feed)->Vector of Charstring as stored;
create function skiphours(Feed)->Vector of Charstring as stored;
create function textinput(Feed)->Vector of Charstring as stored;
create function ttl(Feed)->Integer as stored;
create function rss_GetStream (charstring)->bag of <Charstring,
    Charstring, Charstring, Charstring, Vector, Charstring, Charstring,
    Charstring, Charstring, Vector, Charstring, Vector> as foreign
    "JAVA:StreamDirector/getStream";
create function rss_AddStream(charstring)->boolean as foreign
    "JAVA:StreamDirector/addStream";
```

The two foreign functions are named *rss_GetStream* and *rss_AddStream*. The foreign function *rss_GetStream* takes an address to a feed as argument, downloads all articles and return them as a stream. The foreign function *rss_AddStream* adds meta-data about a feed in the database. There is no support for Atom feeds in the naive implementation. The type *Feed* implements all elements of RSS version 2.0 except the element *version*, which is not represented because only one kind of feed is represented. The core cluster function consists of a call to a single stored procedure that downloads all articles for every instance of *Feed* using the *for each* statement in AmosQL [18]. For every *Feed* instance accessed by the *for each* loop, the stored procedure calls the foreign function *rss_GetStream* responsible for the retrieval of all articles for a given feed [18]. The foreign function *rss_AddStream* is responsible for retrieving the meta-data of a feed when a new feed is stored as a new instance of *Feed* in the RSS-Amos database. There is no logic in the native implementation to add new RSS channels; everything is handled by the Java implementation of the foreign function *rss_AddStream*.

The naive implementation has one large bottleneck. The Internet is accessed each time a query includes a reference to an RSS item. Accessing the Internet involves steps that degrade the performance severely. A call to the Internet usually involves a DNS-lookup, accessing the external network through a number of routers, communicating with a web server using HTTP, and the parsing of the returned data representing the feed. The current state of the networks used and the load on the accessed web server will vary on every call and becomes the bottleneck of the system.

The same definition of type *Rssitem* in the naive implementation is also used in the two other implementations. The signature of the core cluster function given in Chapter 2 is the same in all implementations, while the function bodies are different. Figure 5 illustrates the structure of the type *Rssitem*. Every attribute is represented as a stored function with *Rssitem* as argument type. The result types of the functions can be found in Table 12. Figure 5 shows stored functions as circles, e.g.:

```
create title(Rssitem)->Charstring as stored;
```

Multi-valued attributes are shown as a circle with two lines. They are implemented using vectors, e.g.:

```
create function foreign_markup(Rssitem)->Vector as stored.
```

The definition of type *Feed*, the body of the core cluster function *rssItem_cc*, and the Java implementation of the foreign function *rss_GetStream* are different in the other implementations and the foreign function *rss_AddStream* is removed and replaced by another foreign function.

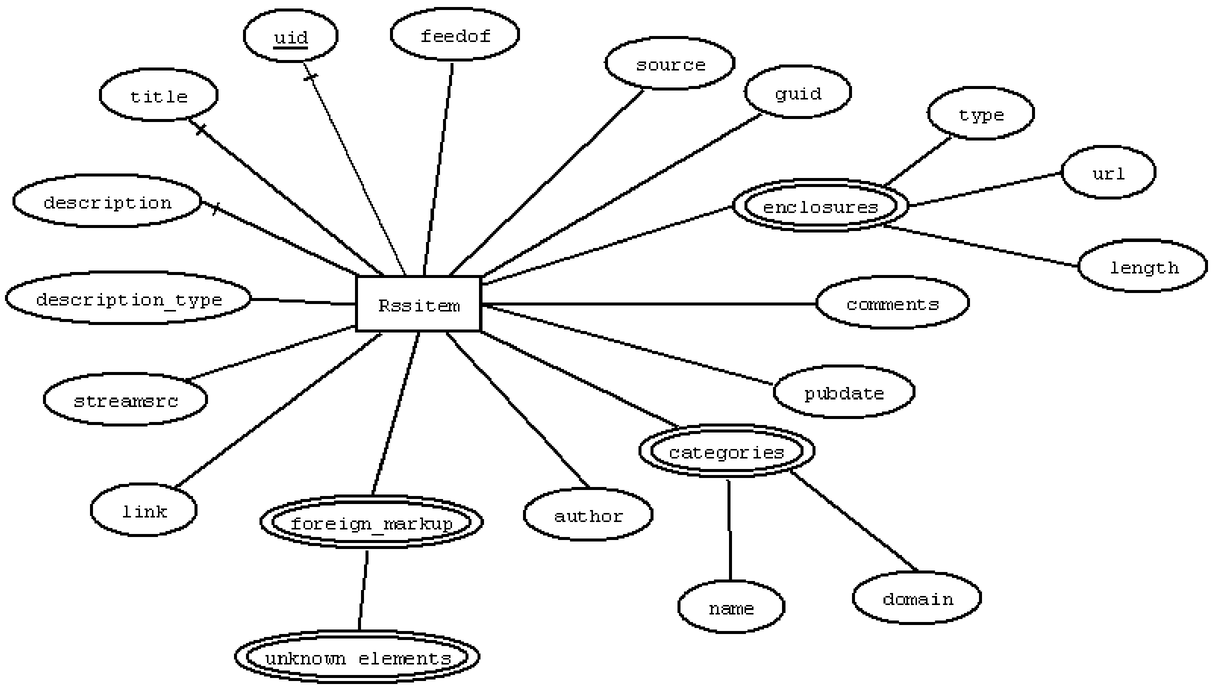


**Figure 5: The type *Rssitem* used in all implementations**

### 2.1.2 Feed caching

The feed caching implementation of RSS-Amos uses a cache of downloaded articles. The motivation for the cache was to limit the number of times the Internet was accessed. The cache consists of a stored function called *rss_cache* implementing the *feed cache* in Figure 3. The logic of managing the cache is implemented as a number of stored procedures in Amos II.

The cache stores all downloaded articles in the system. The cache consists of all the properties of an *Rssitem* in Figure 5, except *feedof*. The cache is represented by the following stored function:

```
create function rss_cache(Charstring src) ->
  Bag of <Integer id key, Charstring title,
          Charstring description,
          Charstring description_type, Charstring link,
          Vector of Vector categories, Charstring author,
          Charstring pubdate, Charstring source,
          Charstring comments, Vector of Vector enclosures,
          Charstring guid, Vector of Vector foreign_markup>
  as stored;

create_index("rss_cache", "description", "hash",
             "multiple");
```

The stored function *source* in the cache is the address to the feed and computed by the property *stream_src(Rssitem)*. The stored function *description* is indexed with a non-unique hash index. Using an index increases the performance of the cache logic and queries where the whole description is given in the query [23].

The core cluster function is multi-directional in the feed caching implementation. Depending on which variable is known (bound) a specific stored procedure is called to do the actual processing and materialization. Each stored procedure has costs and fanouts specified [23]. This is the definition of the core cluster function in the feed caching implementation:

```
create function rssItem_cc()-> Bag of
   <Integer uid key, Charstring title, Charstring description,
   Charstring description_type, Charstring streamsrc, Charstring link,
   Vector categories, Charstring author, Charstring pubdate,
   Charstring source, Charstring comments, Vector enclosures,
   Charstring guid, Vector foreign_markup>
 as multidirectional
    ("bfffffffffffff" select rss_Materialize(uid) cost{1,1})
    ("ffffbfffffffff" select rss_Materialize(streamsrc) cost{2,20})
    ("ffffffffffffff" select rss_Materialize() cost {500,100000});
```

The core cluster function used in the feed cache implementation is multi-directional. The multi directional core cluster function makes it possible to call different functions depending on the binding pattern, e.g. if the address of the feed is known only one feed is processed but if no feed address is known all feeds in the system are processed by the cache logic. Three different resolvents of overloaded function *rss_Materialize* is used in the core cluster function for retrieving RSS items. How the retrieval works is illustrated by Figure 6.

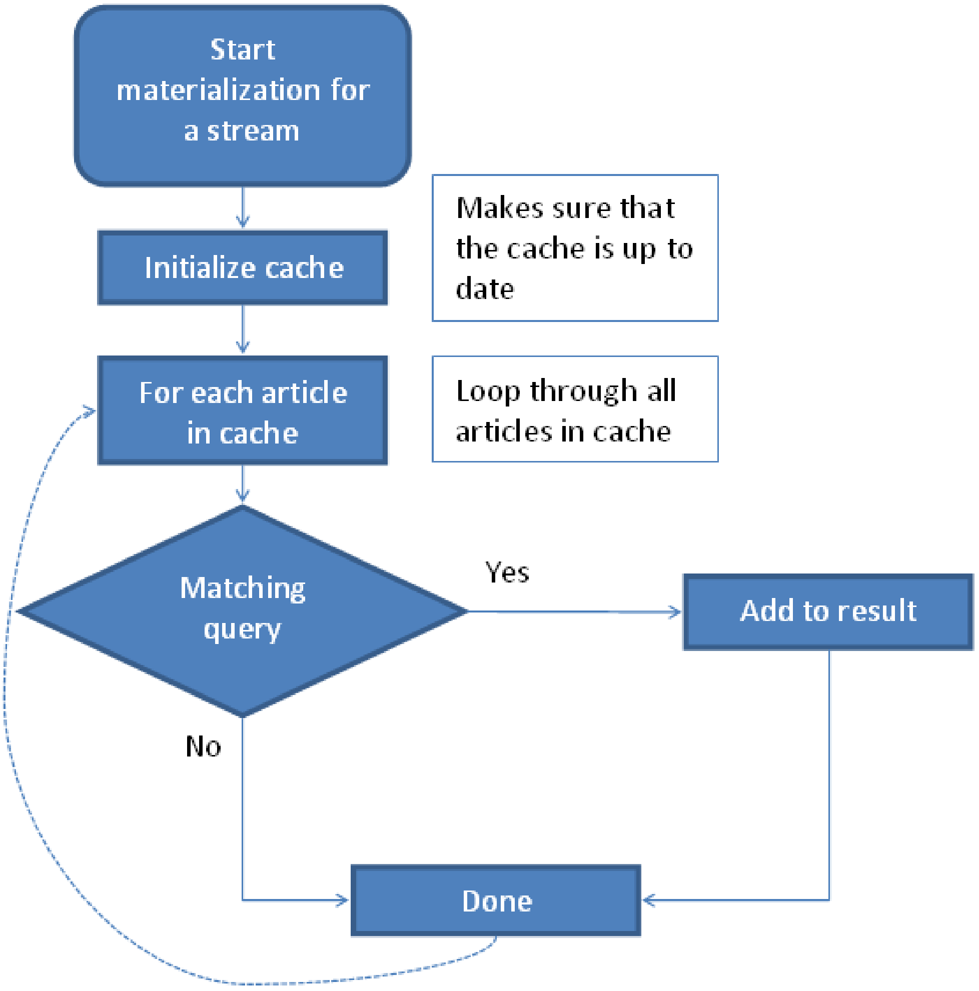

Start materialization for a stream
Initialize cache
Makes sure that the cache is up to date
For each article in cache
Loop through all articles in cache
Matching query
Yes
Add to result
No
Done

**Figure 6: Flow chart for the retrieval of *Rssitem* objects**

When the query optimizer has decided, based on the binding pattern, which procedure to call, one of the resolvents of *rss_Materialize* starts the retrieval by initializing the cache. The initialization of the cache is crucial. The initialization makes sure that the cache contains articles from the specified feed and that the articles' time to live (ttl) has not passed. The stored function *ttl(Feed)->Integer* specifies how long the articles of a feed can be considered valid before there is need for an update. To make the initialization possible two new stored functions was added to the type *Feed*. The new stored functions have no direct correspondence in RSS version 2.0 or Atom. The stored functions are *customttl* and *lastupdate*. The stored function *lastupdate* is updated every time the feed is read from the Internet making it possible for the system to calculate the age of articles stored in the cache. The *ttl* is not a required field in RSS version 2.0 and it is not present in the Atom specification. If the *ttl* of a feed is not valid (equal to 0 or not set) a default value (15 minutes) is stored by the system in *customttl(Feed)->Integer*. It is possible for the user to control the update interval by overriding the default setting.

The feed caching implementation has four new stored functions compared to the naive implementation, named *id, short_name, cache,* and *address*. The stored function *id(Feed)->Integer key* stores a unique numeric id to identify each *Feed* instance. The function *short_name(Feed)->Charstring key* makes it possible for the user to provide nick name for feeds, making querying specific feeds easier. The stored function *cache(Feed)-> Bag of <Integer id, Charstring title, Charstring description, Charstring description_type*, *Charstring link, Vector of Vector categories, Charstring*

*author, Charstring pubdate, Charstring source, Charstring comments, Vector of Vector enclosures, Charstring guid, Vector of Vector foreign_markup* > retrieves the contents of the feed cache for a feed. The stored function *address(Feed)-> Charstring key* stores the URL to the feed. The motivation for the function *address* is that the stored function *link* does not always provide the actual URL address of the feed. For example, the feed BBC Europe has the address *http://newsrss.bbc.co.uk/rss/newsonline_world_edition/europe/rss.xml* while the *link* element has the value *http://news.bbc.co.uk/go/rss/-/2/hi/europe/default.stm*

The graphical definition of *Feed* is shown in Figure 7. In Figure 7 stored functions are illustrated as circles, e.g. *description(Feed)->Charstring*. Stored functions representing multiple values are shown as a circle with two lines. Multiple values are stored in *vectors*, e.g. *categories(Feed)->Vector of Charstring*.

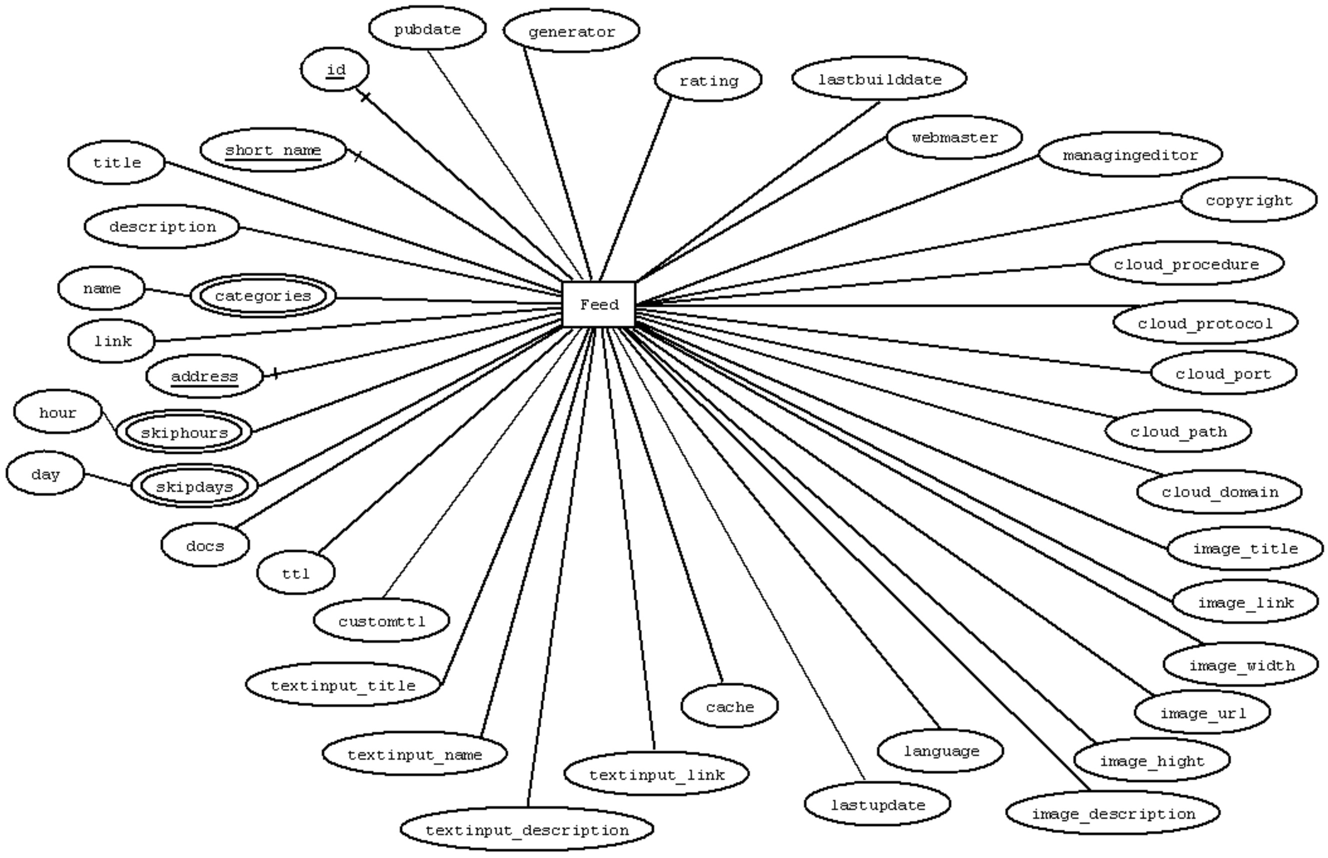


**Figure 7 The definition of *Feed* used in the implementations as a cache**

This is the declaration in Amos II of functions over the type *Feed* in both the feed caching and the parallel feed caching implementations:

```
create function id(Feed)->Integer key  as stored;
create function short_name(Feed)->Charstring key  as stored;
create function title(Feed)->Charstring  as stored;
create function description(Feed)->Charstring  as stored;
create function link(Feed)->Charstring  as stored;
create function address(Feed)->Charstring key  as stored;
create function language(Feed)->Charstring  as stored;
create function categories(Feed)->vector of charstring  as stored;
create function copyright(Feed)->Charstring  as stored;
create function managingEditor(Feed)->Charstring  as stored;
create function webmaster(Feed)->Charstring  as stored;
create function pubdate(Feed)->Charstring  as stored;
create function lastbuilddate(Feed)->Charstring  as stored;
create function generator(Feed)->Charstring  as stored;
create function docs(Feed)->Charstring  as stored;
create function cloud_domain(Feed)->Charstring  as stored;
create function cloud_path(Feed)->Charstring  as stored;
create function cloud_port(Feed)->Charstring  as stored;
create function cloud_protocol(Feed)->Charstring  as stored;
create function cloud_procedure(Feed)->Charstring  as stored;
create function image_description(Feed)->Charstring as stored;
create function image_hight(Feed)->Charstring as stored;
create function image_width(Feed)->Charstring as stored;
create function image_url(Feed)->Charstring as stored;
create function image_link(Feed)->Charstring as stored;
create function image_title(Feed)->Charstring as stored;
create function rating(Feed)->Charstring as stored;
create function skipdays(Feed)->Vector of charstring as stored;
create function skiphours(Feed)->Vector of charstring as stored;
create function textinput_title(Feed)->Charstring as stored;
create function textinput_name(Feed)->Charstring as stored;
create function textinput_description(Feed)->Charstring as stored;
create function textinput_link(Feed)->Charstring as stored;
create function ttl(Feed)->Integer as stored;
create function customttl(Feed)->Integer as stored;
create function lastupdate(Feed)->Timeval as stored;

create function cache(Feed f) -> Bag of
    <Integer id, Charstring title, Charstring description,
     Charstring description_type, Charstring link,
     Vector of Vector categories, Charstring author, Charstring pubdate,
     Charstring source, Charstring comments, Vector of Vector
enclosures,
     Charstring guid, Vector of Vector foreign_markup>
   as select rss_cache(s)
      from charstring s
      where address(f)=s;
```

With the feed cache in *rss_cache*, RSS-Amos will not download a web feed every time an article is used in a query. Connecting and retrieving a feed every time an article is referenced makes the naive implementation very slow. The cache logic will decide if the cached version should be used or if an update is needed. If the cache does not contain any articles for a referenced feed, they will be downloaded from the Internet. If there are articles stored in the cache, the system checks if it is time for an update or if the cached articles are still up to date. To decide if the articles are up to date, the time span between the last update and the current time is compared using the *ttl* or *customttl*. RSS-Amos uses the built in functions *timespan* and *now* [18] to do the actual calculation. The following stored procedure decides if it is time to update a feed. It shows how the built in functions are used (*src* is the *address* of the feed).

```
create function rssTimeForUpdate(Charstring src)->Boolean as
begin
        /*if lastupdate have a value*/

        if count(select lastupdate(stream) from Feed stream where
        address(stream)=src) > 0 then
        begin
            declare Time timediff, Integer ttl, Integer customttl;
            declare Integer minutestimediff;


            select t, ttl_custom, ttl_minute
            into timediff, customttl, ttl
            from Time t, Integer us, Integer ttl_minute,
                 Integer ttl_custom, Feed stream
            where address(stream)=src and
                  <t,us> = timespan(lastupdate(stream),now()) and
                  ttl_minute=ttl(stream) and
                  ttl_custom=customttl(stream);

            /*Calculate the total timespan in minutes*/
            set minutestimediff = hour(timediff)*60 + minute(timediff);
            /*if the custom ttl is set use it*/
            if customttl > 0 then
             begin
                if minutestimediff > (customttl) then
                    result true
                else result nil
             end
            else /*no custom ttl*/
             begin
                if minutestimediff > ttl then result true
                else result nil
             end
       end
       else
/*If the src is not stored in Feed or lastupdate is not set always
update*/
          result true
end;
```

When *rssTimeForUpdate* returns true, a download of all the articles in the feed is made by calling the foreign function *rss_GetStream*. If there already exist articles from the feed in the cache (this is the often the case) the descriptions from the cache is compared with the descriptions of the downloaded articles.

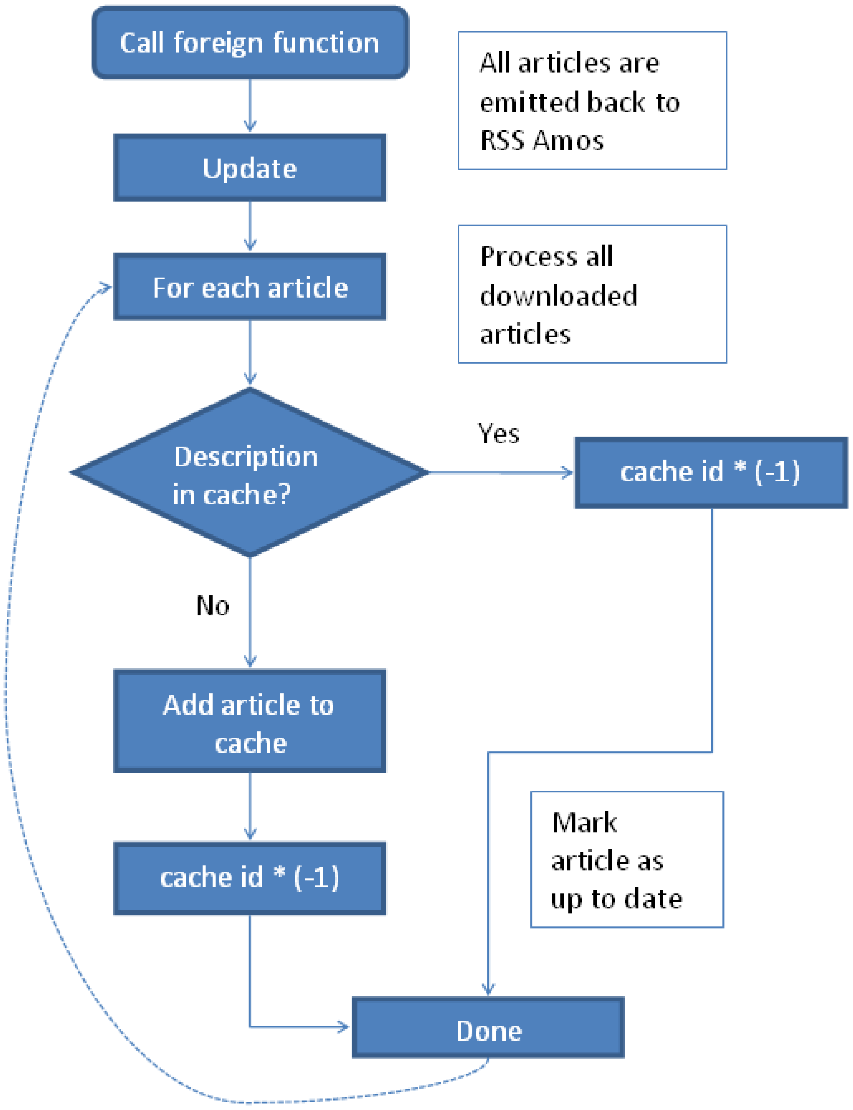


**Figure 8: Management of the cache**

When the update of the cache begins, the stored function *lastupdate* of the specific *Feed* is set to the current time. An article in the cache is considered up to date if the downloaded article for the specific feed has the same description as the one stored in the cache. In this case the system marks the cached article as up to date by negating the *uid* of the *Rssitem* object. For example, an article with the unique id 123 will get the id -123 (-123 is still unique) in the cache. Downloaded articles are added to the cache if the description does not exist. When all articles are processed old articles have to be removed and negative ids are restored to their positive values. Figure 9 shows the process of cleaning the cache after an update.

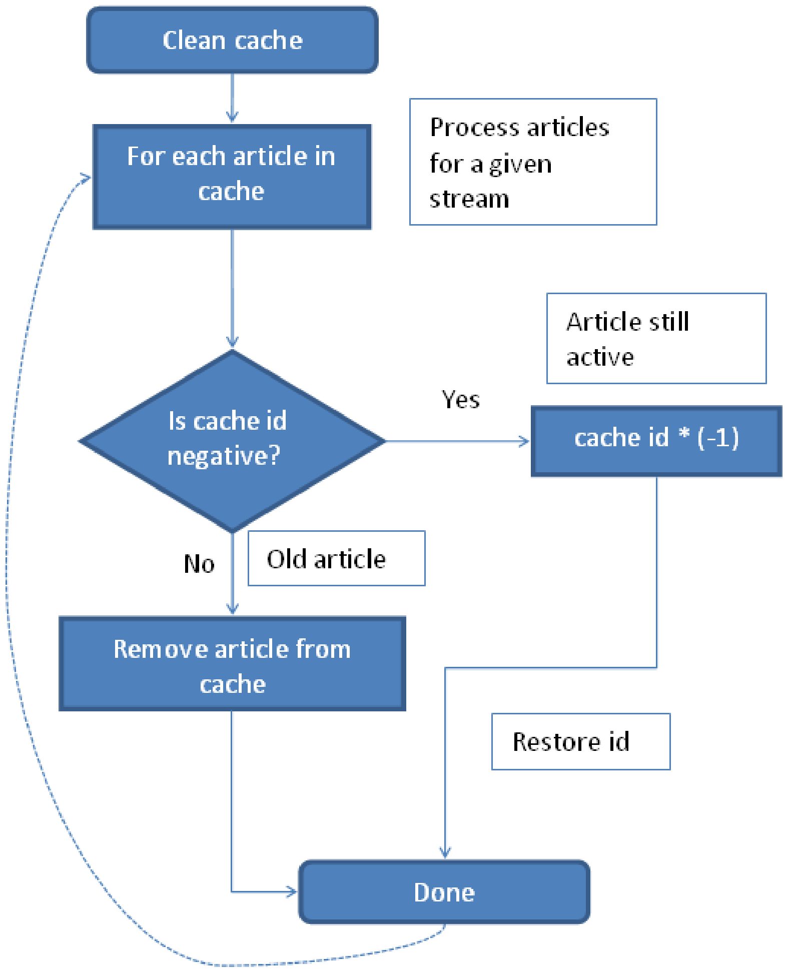


**Figure 9 Cleaning of the cache after an update**

It is possible that more than one feed have the same article and probably the same description. This is supported because the update logic will only process articles with positive ids. After the described processing the cache is up to date and the queried articles are returned from the cache.

The feed caching implementation limits the call to the Internet by using the stored functions *ttl* and *customttl*.

Only the feed sources mentioned in the query are cached. When there are no source address given in the query all feeds stored in the meta-database are accessed, e.g. for the query:

```
count(select from Rssitem r);
```

Accessing every feed in a query can result in many calls to the foreign function *rss_GetStream* to download articles from the Internet. The number of calls to *rss_GetStream* depends on the need for updating the feed cache. The update interval depends on the time since the last update and the values of *ttl* and *customttl*. If the system has not been used for half an hour it is probably the case that all feed caches need an update. If all known feeds are accessed in a query, the feed caching implementation makes separate calls to the foreign function *rss_GetStream* for each feed in sequence. With the basic cached implementation there are no parallel calls to the foreign function making the system wait for one call to complete before another call is made.

The cached implementation uses the following implementation of *rss_Materialize()*, which downloads all feeds that are in need of an update, updates the cache, and then returns all articles in the cache.

```
create function rss_Materialize()->
Bag of <Integer uid, Charstring title, Charstring description,
        Charstring description_type, Charstring streamsrc,
        Charstring link, Vector categories, Charstring author,
        Charstring pubdate, Charstring source, Charstring comments,
        Vector enclosures, Charstring guid, Vector foreign_markup>
 as
begin

/*Make sure that the cache is up to date and filled with missing
streams*/
        for each Feed s
                   rss_Initialize_Cache(address(s));


   for each Integer cache_id, Charstring title, Charstring description,
        Charstring description_type, Charstring link, Vector categories,
        Charstring author, Charstring pubdate, Charstring source,
        Charstring comments, Vector  enclosures, Charstring guid,
        Vector foreign_markup, Charstring src
        where
        <cache_id, title, description, description_type, link,
        categories, author, pubdate, source, comments, enclosures, guid,
        foreign_markup> = rss_Cache(src)
        begin
        /*If the cache is updating make the id positive*/
        if cache_id < 0 then
            set cache_id = cache_id * (-1);

        set uid = cache_id;

        result <uid, title, description, description_type, src, link,
        categories, author, pubdate, source, comments, enclosures, guid,
        foreign_markup>;

        end;
end;
```

The definition of *Feed is* shown in Figure 7. The foreign function *rss_AddStream* is in the naive implementation is not used in the cached implementation. It is replaced by the foreign function *addAndGetStream(Vector of charstring)->bag of <Charstring, Charstring, Charstring, Charstring, Vector, Charstring, Charstring, Charstring, Charstring, Vector, Charstring, Vector> as foreign "JAVA:StreamDirector/addAndGetStream";* The foreign function *addAndGetStream* adds a new feed to system, exactly as *rss_AddStream* did. But with *addAndGetStream* it is possible to specify the *short_name* of the feed being added. The address of the feed and the short name of the feed are passed as a *Charstring Vector.* The address is stored in the first position of the *Vector* that is passed as argument to the foreign function *addAndGetStream*. The foreign function *addAndGetStream* returns all the articles of the added feed making it not suitable for direct calls by a user. Instead the two new procedures *rss_AddAndGetStream(Charstring src, Charstring short_name)->Boolean* and *rss_AddAndGetStream(Charstring src, Charstring short_name)->Boolean* should be used when adding a new feed to the system. The two procedures will call *addAndGetStream* and store the returned articles in the cache. The functions *addAndGetStream* and the two over loaded procedures named *rss_AddAndGetStream* are used also with the parallel feed caching implementation.

### 2.1.3 Parallel feed caching

As described in the feed caching implementation there is a high probability of the need to update all the articles in the system when it has not been used for half an hour. The default *cusomttl* is set to 15 minutes making all Atom feeds [4] (there exist no *ttl* in Atom) and all RSS channels [1] with no *ttl* specified in need of an update after 15 minutes. The calls to the foreign function handling the download are made in a sequential fashion in the previous two implementations. This parallel feed caching implementation uses the cache implemented in the chapter *2.1.2*, the *Feed* shown in Figure 7 and the *Rssitem* shown in Figure 5. The parallel feed caching implementation will try to minimize the number of foreign functions calls being made and limit the time waiting when accessing the Internet by replacing the foreign function *rss_GetStream (charstring)* with *rss_GetStreamsThread (Vector of charstring, Integer)* and the procedure *rss_Materialize()* with *rss_MaterializeThread()*.

The solution:

- Make only one call to a foreign function *rss_GetStreamsThread (Vector of charstring, Integer)* when multiple feeds are referenced.
- Use threads to do downloads of several feeds in parallel.
- Base the number of threads on the number of feeds to download to regulate the parallelism.

The core cluster function calls the function *rss_MaterializeThread()*when the binding pattern does not specify the *uid* or *stream_src* of an *Rssitem* (shown in chapter 2.1.2). This will cause the cache to be initialized for every feed in the system as shown in Figure 6. Two procedures, *rss_Update(Vector of Charstring src)->Integer* and *rss_Initialize_CacheThread(Charstring src)->Boolean*, are added to be used by the implementation of *rss_MaterializeThread()*. The procedure *rss_Update(Vector of Charstring src)* has the same logic as the resolvent *rss_Update(Charstring src)* used in the feed caching implementation. The difference is that the new procedure calls a new foreign function *rss_GetStreamsThread* , which uses threads and takes a vector of feed addresses to download in parallel instead of just one feed address. *rss_Update* will retrieve all articles from feeds in need of an update from the foreign function *rss_GetStreamsThread* and update the cache in the same manner as in the feed caching implementation. The procedure *rss_Initialize_CacheThread* works the same as the procedure *rss_Initialize_Cache* used in the feed caching implementation except that it does not call the *rss_update* procedure. *rss_Initialize_CacheThread* will only determine if the feed given as parameter should be updated or not.

```
create function rss_Initialize_CacheThread(Charstring src)->Boolean as
begin

if rss_online() = true  then
begin
/*if the cache is empty load from Internet*/
    if notany(rss_Cache(src)) = true then
        result true;
    else if (notany(update_rss_cache(src)) = true) and
             (rss_TimeForUpdate(src) = true) then
/*if there is something in the cache and the cache isn't being updated
and it's time for an update*/
        result true;
    else
        result nil
```

```
end
else /*The cache is fine.*/
    result nil
end;
```

When the update is complete the cache is read and returned. The code below is the implementation of *rss_MaterializeThread()*.

```
     create function rss_MaterializeThread()-> Bag of <
     Integer uid,
     Charstring title,
     Charstring description,
     Charstring description_type,
     Charstring streamsrc,
     Charstring link,
     Vector categories,
     Charstring author,
     Charstring pubdate,
     Charstring source,
     Charstring comments,
     Vector enclosures,
     Charstring guid,
     Vector foreign_markup> as
     begin
        declare Vector of charstring updatesources;
        if rss_online() = true  then
        begin
        /*Make sure that the cache is up to date and filled with missing
        streams*/
            select vectorof(b) into updatesources
            from bag of charstring b
            where b= (select address(s) from Feed s
            where rss_Initialize_CacheThread(address(s))=TRUE);

            if dim(updatesources) != 0 then
                rss_Update(updatesources);
        end;

        for each
            Integer cache_id,
            Charstring title,
            Charstring description,
            Charstring description_type,
            Charstring link,
            Vector categories,
            Charstring author,
            Charstring pubdate,
            Charstring source,
            Charstring comments,
            Vector  enclosures,
            Charstring guid,
            Vector foreign_markup,
            charstring src
        where <cache_id,
               title,
               description,
               description_type,
               link,
               categories,
               author,
               pubdate,
               source,
               comments,
               enclosures,
               guid,
               foreign_markup> = rss_Cache(src)
        begin
        /*If the cache is updating make the id positive*/
        if cache_id < 0 then
```

```
            set cache_id = cache_id * (-1);

        set uid = cache_id;

        result <uid, title, description, description_type, src, link,
        categories, author, pubdate, source, comments, enclosures, guid,
        foreign_markup>;

        end;
    end;
```

The foreign function *rss_GetStreamsThread(Vector of Charstring src, Integer feeds)* uses threads to do the actual download. The parameter *src* contains the addresses of the feeds that are in need of an update. The parameter *feeds* contains the number of feeds per thread that the foreign function should use. The call of *rss_GetStreamsThread* is done by *rss_Update* and the value of *feeds* is read from the stored function called *rss_defaultnrofstreams*.

The parallel feed caching implementation changes how the system handles updates of multiple feeds. The cache logic and the definition of *Rssitem* and *Feed* is the same as in the feed caching implementation described in the chapter *2.1.2*.

## 2.2 Java implementation of the RSS-Amos wrapper

This chapter describes the design patterns used in the Java implementation and how the Java interface of Amos II [19] and the Rome library [11] is used in RSS-Amos. The foreign functions *rss_GetStreams*, *rss_GetStreamsThread*, and *addAndGetStream* are implemented in Java. They are responsible for accessing the Internet retrieving information about feeds and their articles.

### 2.2.1 Motivating choice of interfaces

There are a couple of Java libraries concerning feeds. *RSSLib4J [20]* supports all RSS versions but not Atom. *RSSLib4* is not a living project. The last update of *RSSLib4J* was in September 3, 2004 *[20]*.

*Informa* [14] can read RSS 0.9x, RSS 1.0, RSS 2.0, Atom 0.3 and Atom 1.0. However, it is still in beta state. *Informa*'s last release came out in January 2007 as an alpha 2. I will also consider this a dead project [14].

I consider *ROME* (RSS and Atom Utilities) [15] to be a living project. Version 1.0RC1 was the current version when I started working on this project. There have been two releases since then, version 1.0RC2 in January 2009 and version 1.0 in Mars 2009. Before version 1.0RC1 there had been nine beta releases 0.1-0.9. *ROME* supports RSS 0.90, RSS 0.91 Netscape, RSS 0.91 Userland, RSS 0.92, RSS 0.93, RSS 0.94, RSS 1.0, RSS 2.0, Atom 0.3, and Atom 1.0. There is also living subproject of *ROME* e.g. *ROME Fetcher* [10] and *OPML for ROME [11]*. These subprojects are interesting for the future work of this project.

### 2.2.2 Design

The implementation in Java uses the *ROME* library to manage the parsing of RSS channels and Atom feeds. *ROME* depends on *J2SE1.4*- and *JDOM* version 1.0. *JDOM* [1] [4][7][15] is used by *ROME* for reading and writing *XML*. *ROME* consists of six packages. The package ***com.sun.syndication.feed*** in the *ROME* library contains the class *WireFeed*. The *WireFeed* class is the parent of the classes *Channel* and *Feed*. The *Channel* class represents a RSS channel and the *Feed* class represents Atom feed. *ROME* has a general representation of all feed types called *SyndFeedImpl*. *SyndFeedImpl* is easier and the most popular way to work with *ROME* because the programmer does not need to handle different feed types. However, creating an instance of a *SyndFeedImpl* is not efficient. First a subclass of *WireFeed* is created, and then the *SyndFeedImpl* is created using the instance of *Channel* or *Feed*. The performance differences are presented in chapter *2.3.1.2*. The test confirmed that the *SyndFeedImpl* is slower than the *Channel* class resulting in that the *SyndFeedImpl* is not used in the project.

The *Channel* and *Feed* classes can represent all versions of RSS or Atom. The implementation in *ROME* uses inheritance to manage different versions, e.g. the class handling RSS version 0.9 is the parent of the class handling RSS version 0.91. The *ROME* library uses modules to handle different elements in feeds. Modules make it possible to extend *ROME* to handle new elements. I have not seen a need of implementing new modules for this project; thus RSS version 2.0 was the template for the project which is fully supported by *ROME*. There exists a good example of how to extend *ROME* in [9].

Design patterns are a common term when talking about object-oriented development. This part of the report will mention some of these patterns but describing these patterns in depth is outside this project. I have used the classic book *Design Patterns* [3].

I have implemented a class called *StreamDirector,* which manages the creation of classes depending on if it is a RSS channel or an Atom feed that is used. All foreign functions in RSS-Amos point to a function defined in the *StreamDirector* class. The *StreamDirector* acts as the *director* in the design pattern *builder*. The *director* is responsible for creating objects that can create articles. An article is returned to RSS-Amos as a *Vector*. RSS-Amos will parse the returned *Vector* and represent it as the mapped type *Rssitem* in RSS-Amos. Classes that create articles are called *builders* in the design patterns *builder*.

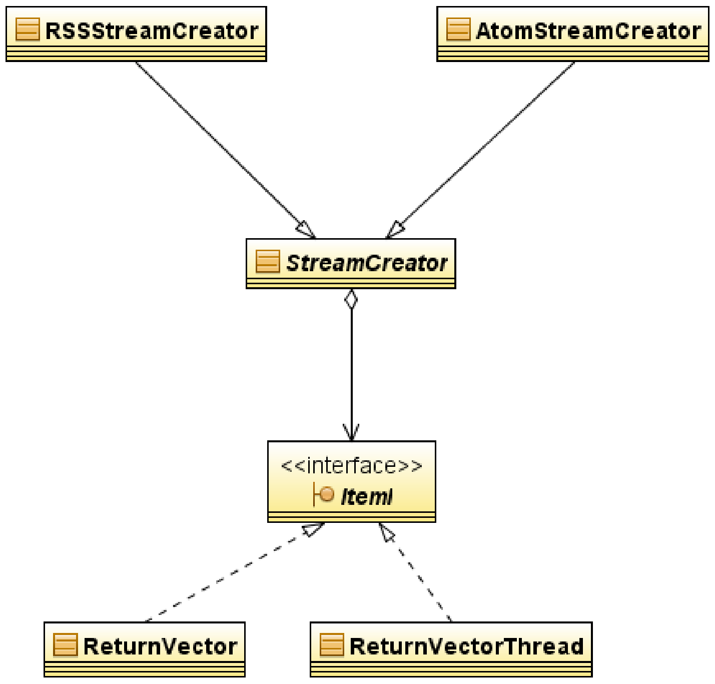


**Figure 10 class diagram of the article creation**

*StreamCreator* is an *abstract* class that acts like a *builder* in the design pattern *builder*. A *builder* defines the steps needed to build the resulting product. In this case the products are articles from a web feed and they are represented as the class *ReturnVector* or *ReturnVectorThread*. The class *ReturnVectorThread* is used by the parallel feed caching implementation, described in chapter 2.1.3, because there is a need to emit one extra field back to RSS-Amos identifying the article to a feed. *StreamCreator* is also acting as the design pattern *Template Method*. In this case the *StreamCreator* defines the order of extracting the information. The function *ExecuteItem* is implemented in *StreamCreator*. *ExecuteItem* is responsible of implementing the algorithm of creating articles. In this case the design patterns *concrete builder* is represented by *RSSStreamCreator* and *AtomStreamCreator*. Both these classes are responsible for extracting data from a feed when they are called from the parent class. It is the *director* that starts the creation of articles by calling the abstract method *ExecuteStream* in *StreamCreator*.

The process of creating an Amos II object of type *Feed* (the meta-data representation of a web feed) uses the same design patterns as used when articles are processed. The same *StreamDirector* is responsible for creating *Feed* types as it is responsible for creating articles. The *builder* is called *StreamTypeCreator* and the *concrete builders* are called *RSSStreamTypeCreator* and *AtomStreamTypeCreator*.

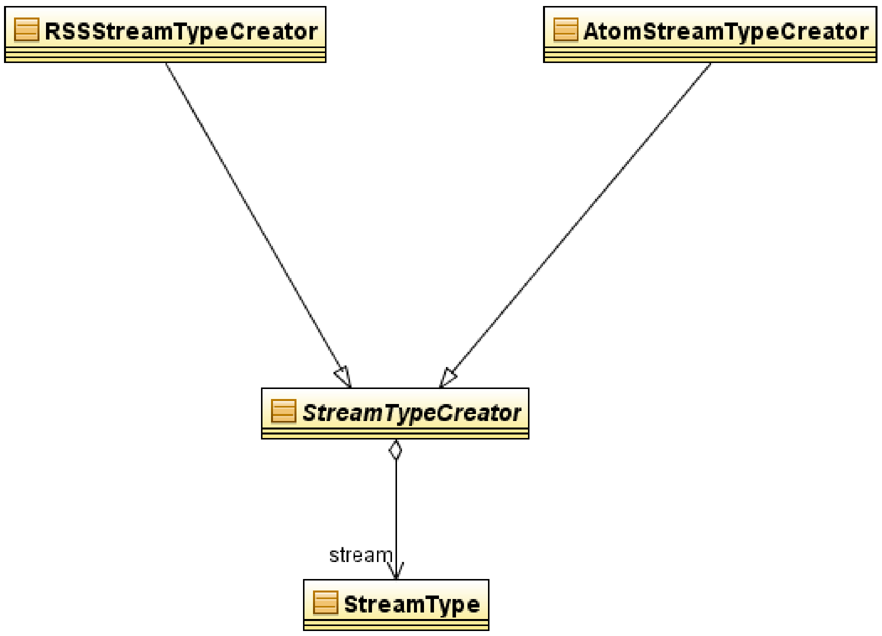


**Figure 11 class diagram of the feed creation**

The implementation in Java uses two techniques communicating with RSS-Amos, the *fast path* interface through the *callin* interface using a *tight connection* (adding feeds), and the *callout* interface when implementing foreign functions that emits articles back to RSS-Amos. This means that the Java implementation controls the creation of *Feed* objects and the Amos II implementation controls the logic when articles are added [18].

The part of the implementation written in Amos II manages articles as *Rssitems*. The download of articles is initiated in the Amos II implementation. When articles are downloaded and processed they are emitted as a stream of *tuples* back to the calling foreign function, e.g. *rss_GetStream*, and processed by the caching mechanism as described in the chapters 2.1.2 and 2.1.3. It is the *StreamDirector* that downloads the feed. When the feed type is determined a suitable *concrete builder* is created. The *concrete builders RSSStreamCreator* and *AtomStreamCreator* are responsible of extracting the information necessary to build an article and fill an instance of *ReturnVector* or *ReturnVectorThread*. The classes *ReturnVector* and *ReturnVectorThread* formats the result in an instance of *Tuple* before it is emitted back to the foreign functions *rss_GetStream(Charstring)* and *rss_GetStreamsThread(Vector, Integer)*. The class *constructor* of *ReturnVector* and *ReturnVectorThread* takes one argument of type *Tuple* representing the vector that is returned. The argument passed to the class *constructor* is the tuple in the result position of the parameter structure given by Amos II when the call was made to the foreign functions *rss_GetStream* and *rss_GetStreamsThread*. The

*ReturnVector* and *ReturnVectorThread* are expecting a specific structure. If the parameters in the call from RSS-Amos changes the *ReturnVector* and *ReturnVectorThread* will not return what is expected or throw an exception. This design will return a stream of tuples to the Amos II implementation. A tuple is emitted as soon it is processed, instead of returning all tuples when all possessing is done.

The web feed is downloaded and processed using the classes in Figure 11. An instance of the class *Feed* is created when all the data and meta-data about the feed its articles are loaded and formatted. The instance of the class *Feed* is created by the call *connection.createObject("Feed")* (*connection* is an instance of *callin.Connection*) that returns the *oid*. The second step is to query the Amos II implementation to get a unique id for the *Feed* instance. This is done with the call

```
connection.callFunction(
"rss_get_next_rssstream_id->Integer").getRow().getIntElem(0)
```

Data is materialized as an object of type *Feed* as in this example:

```
// Oid oidOfInstance and int id already have their values set
Connection  con = new Connection("");
Tuple arg = new Tuple(1);
Tuple value = new Tuple(1);
arg.setElem(0, oidOfInstance);
value.setElem(0, id);
con.addFunction(con.getFunction("RSSStream.id->Integer"), arg, value);
```

The system tries to undo changes if there is an exception during the process of creating the instance of the *Feed*. The first step in the process of adding a *Feed* instance to RSS-Amos is to get an *oid* of the new instance. This *oid* is used in case of the need to undo the creation by the call *connection.deleteObject(oid).* A rollback of the transaction had been another solution but I decided to manage exceptions by deleting the created object. The last step is to propagate the exception back to Amos II as an *AmosException*. All articles of the feed are also emitted back to the *feed cache* of RSS-Amos when a *Feed* is added. All the articles are downloaded when the *Feed* is created so there is no need to access the Internet again until the TTL has expired.

The *feed materializer* of RSS-Amos have two stored procedures that are used when adding a web feed to the meta-database:

*rss_AddAndGetStream(Charstring src, Charstring short_name)->Boolean* and

*rss_AddAndGetStream(Charstring src)->Boolean*.

The parameter *short_name* makes it easier to reference a *Feed* in queries. Instead of using the whole HTTP address of the *Feed* as identification in a query you can assign a *short_name* to the URL. The value of the properties *short_name* and the *id* is the same if the *short_name* is not set in the call. The *short_name* has to be unique. This is enforced in RSS-Amos by setting the *short_name* field to *key* when the *Feed* type is defined in the Amos II implementation. *rss_AddAndGetStream* uses the foreign function *addAndGetStream(Vector of charstring)* to create the *Feeds* instances and emitting the articles back to be handled by logic in the *feed cache* of RSS-Amos. The foreign function *addAndGetStream* uses a method implemented in Java with the same name, *addAndGetStream*. The foreign function *addAndGetStream* will add the articles to the cache as described in chapter 2.1.2.

### 2.2.3 Multi-threaded implementation of parallel feed caching

The parallel feed caching implementation in chapter *2.1.3* described the use of the foreign function *rss_GetStreamsThread* that used threads to improve the performance. *rss_GetStreamsThread* uses the Java method *ExecuteThreadGivenNrOfStreams*. This chapter will describe the details of how the multi threaded implementation works. Classes from the *ROME* library are used. The implementation is located in the class *StreamDirector* and the method is named *ExecuteThreadGivenNrOfStreams*. The method takes as arguments *feedaddresses,* a *Vector* containing all feed address, and *feedsperthread,* the number of feeds per thread. The implementation uses three threads as the minimum number of threads (based on the results from the test in chapter *2.3.1.1*). When three threads are used the feeds are spread evenly among the threads. The number of threads is calculated by using the integer value of dividing *feedaddresses* with *feedsperthread*. If the reminder is not zero the value rounded upwards. The number of feeds per thread will never be larger then *feedsperthread* avoiding slow execution times when one thread will get many feeds to process. Having one thread with *m* feeds to download will cause the system to wait until it has downloaded a feed before the feed can be processed. Table 13 shows an example of the distribution of threads created by the logic of the java implementation of RSS-Amos. Table 13 shows the number of threads used and the distribution of feeds per thread when the number of feeds to download varies. Table 13 shows that the solution is thread centric, e.g. when 41 feeds are used a fifth thread is created having just one feed. The thread centric solution tries to limit the time waiting before a feeds is downloaded by doing more parallel downloads.

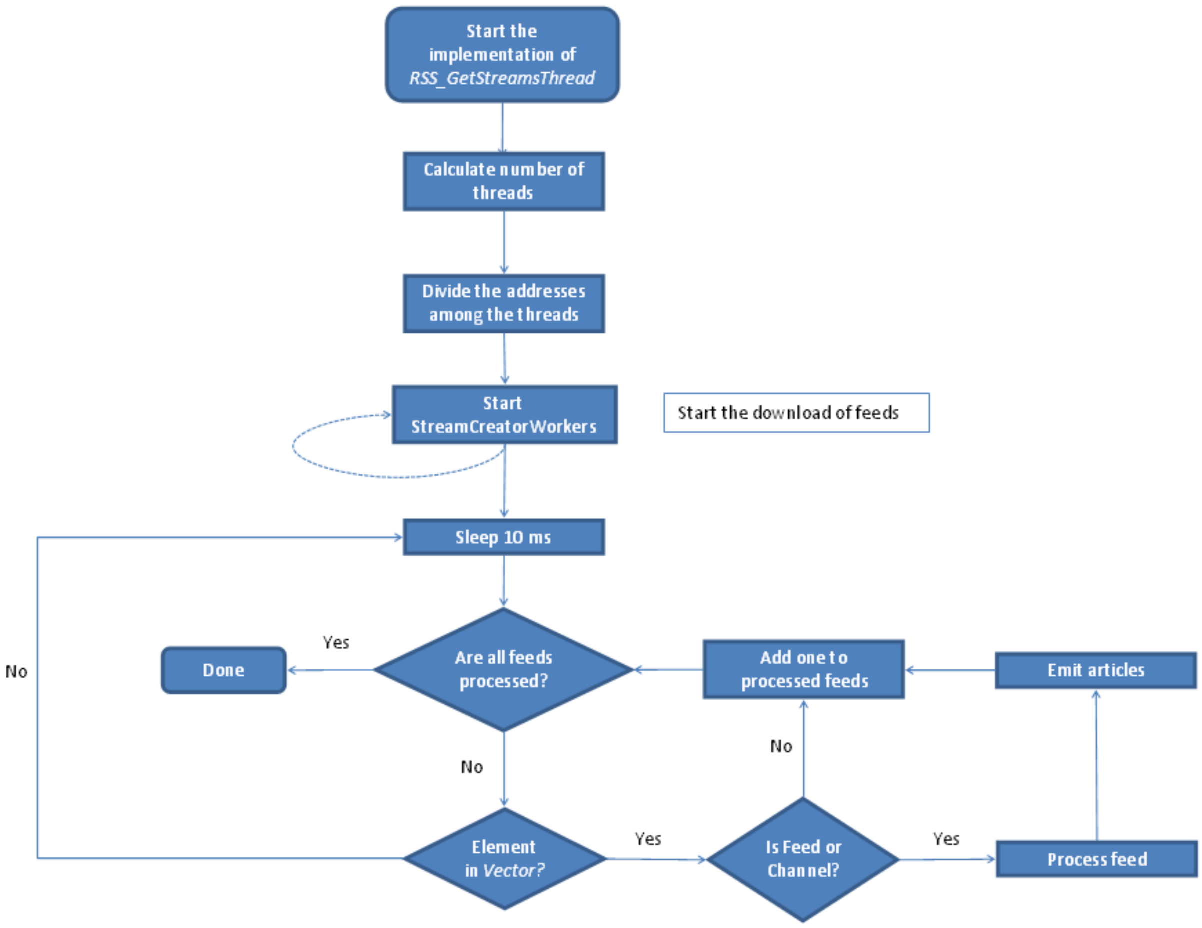
**Table 13 distribution of feeds per thread**

| Feeds | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Thread #1 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Thread #2 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Thread #3 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Thread #4 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Thread #5 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

The method *ExecuteThreadGivenNrOfStreams* uses instances of a class called *StreamCreatorWorker* that extends the Java class *Thread*. This means that *StreamCreatorWorker* has the functionality as a Java thread. *StreamCreatorWorker* is responsible of downloading the feeds. The constructor of *StreamCreatorWorker* takes two arguments, one array of addresses of feeds and one reference to a *Vector*. The *Vector* references an instance of the class *Vector* created in *ExecuteThreadGivenNrOfStreams.* The *Vector* class can be accessed by multiple threads. When a feed is downloaded in an instance of *StreamCreatorWorker* an instance of *WireFeed* is created, *casted* to one of the classes from the *ROME* library, *Feed* or *Channel*, and then added to the referenced *Vector* object. When all feeds are downloaded in a *StreamCreatorWorker* the execution of the thread stops. If there is an error while downloading a feed, an instance of the Java class *Object* is added to the referenced java object *Vector,* the reason will be explained later. The execution process of the method *ExecuteThreadGivenNrOfStreams* will read the contents of the *Vector* instance, looking for feeds to process. If there is no feed to process the execution will wait for 10ms giving the threads time to complete a download. The execution is done when all feeds are processed.

As mentioned, a *StreamCreatorWorker* will add an instance of the class *Object* to the referenced *Vector* when a download has failed. The instance of the class *Object* will indicate that something got wrong, but the execution of the other feeds will not be interrupted. The method *ExecuteThreadGivenNrOfStreams* counts all feeds that have been processed and emitted back to RSS-Amos. When an instance of *Object* is found it is counted as one processed feed. When the number of processed feeds is the same as the number of addresses passed as argument the execution of *ExecuteThreadGivenNrOfStreams* is done.

The execution of the parallel feed caching implementation is shown in Figure 12 and it can be summarized as follows. Create a number of threads, give them some feeds to download and start all threads. When the first download is complete the processing of the feed starts on the main thread and the articles for the feed are emitted back to RSS-Amos. If there is no feed to process the execution sleeps for 10ms before it checks if there exists any feeds to process. When all feeds are downloaded and processed the execution is complete. The optimal case is that there is a new feed to process directly after that the last article is emitted back, that there is not too much context switching between threads, and that the CPU usage is at 100%.



Start the implementation of *RSS_GetStreamsThread*
Calculate number of threads
Divide the addresses among the threads
Start StreamCreatorWorkers
Start the download of feeds
Sleep 10 ms
No
Done
Yes
Are all feeds processed?
Add one to processed feeds
Emit articles
No
No
Element in *Vector?*
Yes
Is Feed or Channel?
Yes
Process feed

**Figure 12: The execution process of the multi threaded implementation**

## *2.3 Performance*

The performance of RSS-Amos is dependent of the state of the networks used when communicating with the servers hosting the feeds used by the system and the use of foreign functions implemented in Java. This chapter will describe the tests made to improve the performance of RSS-Amos.

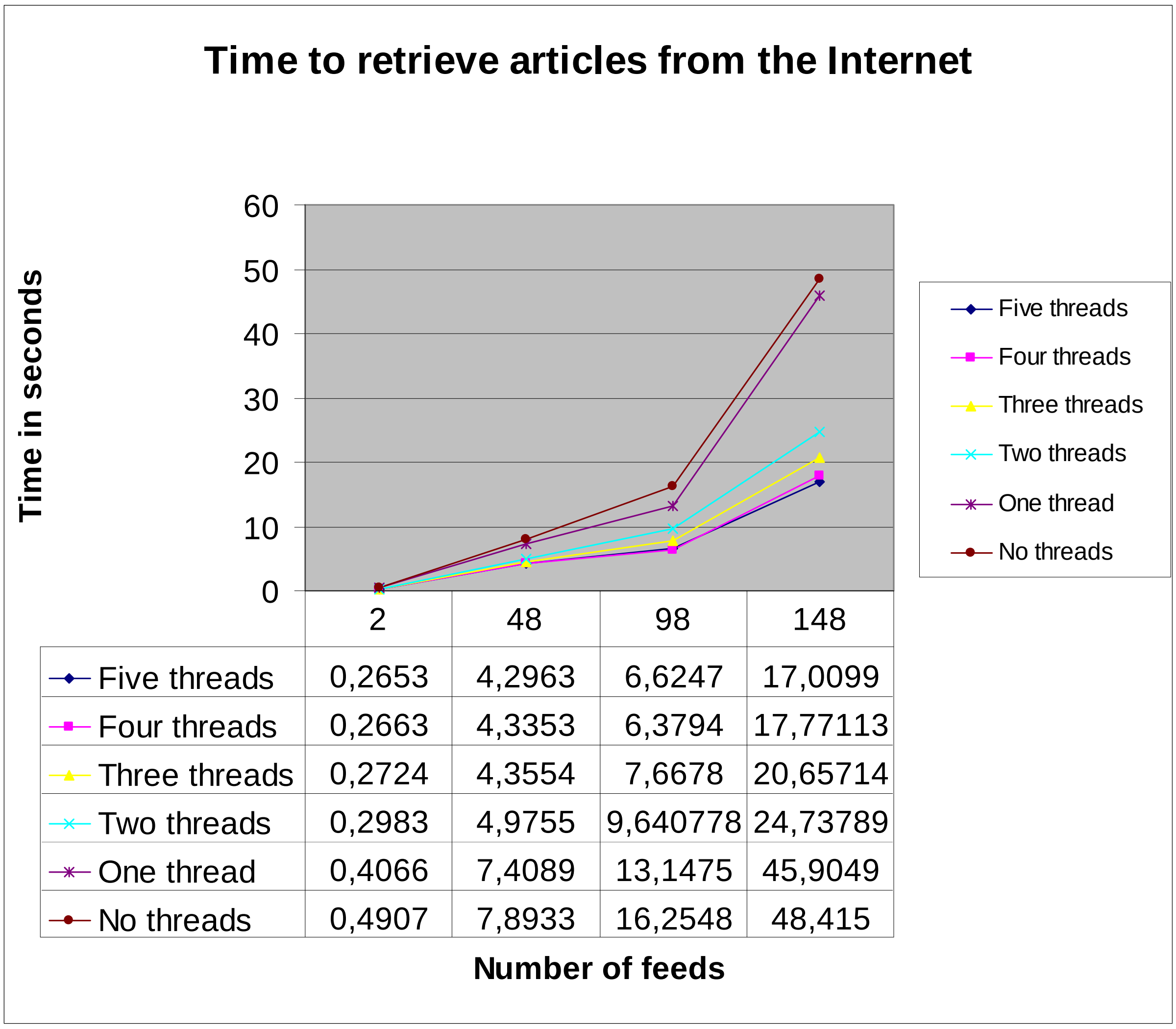
### 2.3.1 Tests

Two tests using the ROME library investigated how to optimize the use of threads that was introduced in the parallel feed caching implementation of RSS-Amos.

#### 2.3.1.1 Optimal feeds per thread

This test was conducted to determine the optimal number of threads to use when multiple feeds are downloaded. The idea is to start multiple threads where each thread is responsible for downloading a number of feeds. There is an overhead using threads because the operating system has to manage context switching. If too many threads are used the overhead will have an impact on the performance. If the number of threads is too few the system has to wait for a download to complete. When deciding the optimal number of threads I used a test that calculates the time it takes to download a varying number of feeds registered in the system in parallel and emitting the information back to Amos II implementation of RSS-Amos. I varied both the number of feeds in the system and the number of threads used in the Java implementation. When there are more threads than feeds the threads are created with one feed each to download; e.g. if the system has two feeds and the test uses five threads, only two threads are created with one feed each. When there are no threads used in the test the foreign function *rss_GetStream* is called for each feed in the system, e.g. if there is 48 feeds in the system 48 calls are made to the foreign function. I also tried to run the emitting of the downloaded feeds in different threads but the context switching between threads was heavy so I decided to run all emitting of the feeds on the main thread.

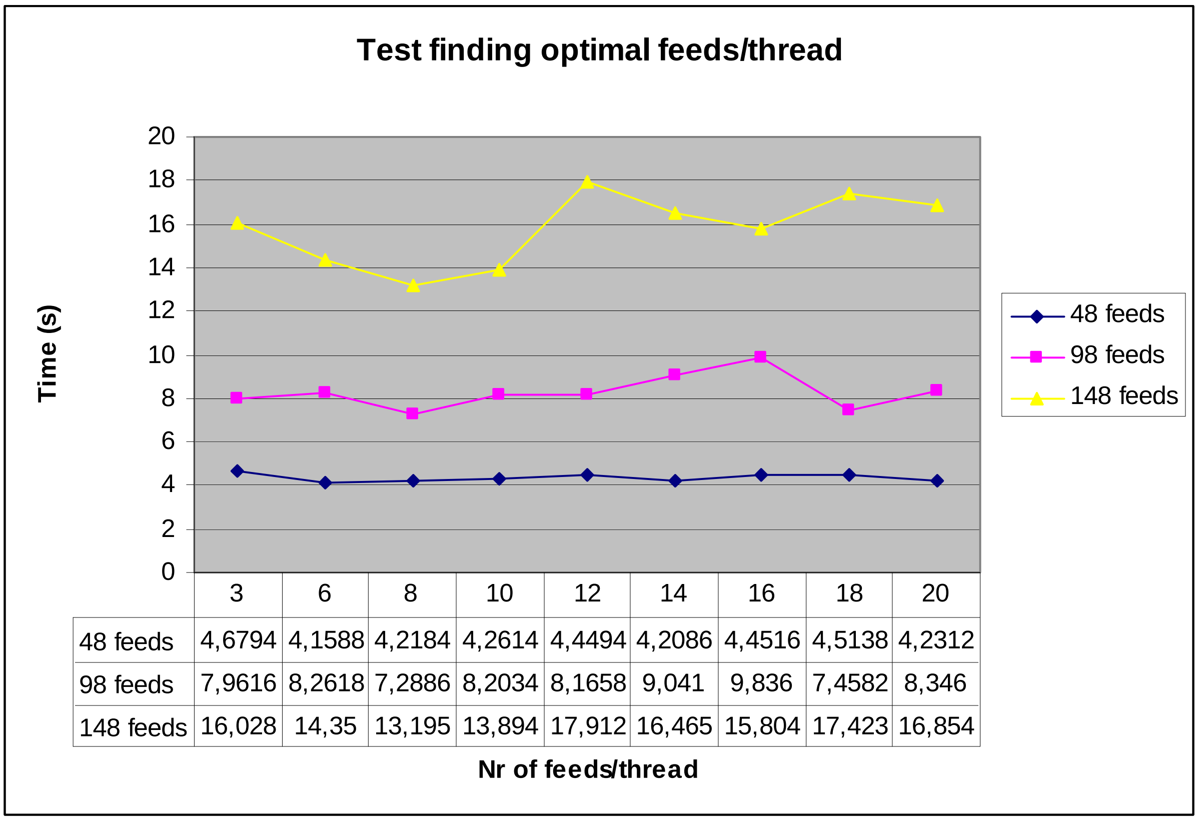
The test was initialized from RSS-Amos. I prepared an *AmosQL* script file with the six test cases no thread, one tread, two threads, tree threads, four threads, and five threads. Each test case downloaded articles for every feed stored in the database and counted the number of articles emitted back. The test had RSS-Amos loaded with 2(54), 48(918), 98(1541) and 148(2871) feeds where the total number of articles is presented within parentheses. Each test case was executed 10 times. The test was conducted on a Sunday between 18.00 and 19.00. I used an IBM T21 with 800MHz CPU and 256MB RAM during the tests.



| | 2 | 48 | 98 | 148 |
|---|---|---|---|---|
| Five threads | 0,2653 | 4,2963 | 6,6247 | 17,0099 |
| Four threads | 0,2663 | 4,3353 | 6,3794 | 17,77113 |
| Three threads | 0,2724 | 4,3554 | 7,6678 | 20,65714 |
| Two threads | 0,2983 | 4,9755 | 9,640778 | 24,73789 |
| One thread | 0,4066 | 7,4089 | 13,1475 | 45,9049 |
| No threads | 0,4907 | 7,8933 | 16,2548 | 48,415 |

**Figure 13: Result from test of download performance**

The result of the test shows that the use of two or more threads improves the performance. The difference in execution time increases with the number of feeds to download. I ran the same test with 10 and 20 threads on the system loaded with 148 feeds resulting in an average of 16,4032s for 10 threads and 16,636s for 20 threads. It is hard to determine the exact number of feeds/thread when the difference in time varies in a couple of milliseconds and the results depend on the performance of the network and the hosting servers. The result hints that having 10-15 feeds per thread gives good execution times. When comparing the execution times of the two tests no thread and one thread, the test with one thread is always faster. The test with one thread was 0,025202s/feed faster then the test with no threads. The reason for this is that there was only one call to the foreign function in the test with one thread instead of one call per feed in the test with no thread. The utilization of the CPU does not show in the diagram but I looked at the task manager during the tests and it showed a utilization between 35-70% when no thread or one thread was used, 45-80% when two threads were used and close to 100% when three and up to 20 threads were used.



| | 3 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
|---|---|---|---|---|---|---|---|---|---|
| 48 feeds | 4,6794 | 4,1588 | 4,2184 | 4,2614 | 4,4494 | 4,2086 | 4,4516 | 4,5138 | 4,2312 |
| 98 feeds | 7,9616 | 8,2618 | 7,2886 | 8,2034 | 8,1658 | 9,041 | 9,836 | 7,4582 | 8,346 |
| 148 feeds | 16,028 | 14,35 | 13,195 | 13,894 | 17,912 | 16,465 | 15,804 | 17,423 | 16,854 |

**Figure 14: Finding optimal feeds/thread**

Figure 13 has the number of threads fixed giving a hint of the optimal number of feeds per thread. Figure 14 shows that it is 8 feeds/thread that is the optimal value. The performance difference is more obvious when more feeds are used. The values shown in Figure 14 are average times of downloading all articles in the system six times. My conclusion is that using three threads is the minimum for a high utilization of the CPU. The results are specific for my test machine. It is a high probability that other machines with faster CPUs and more RAM will process the feeds differently then the test machine and thus has a different optimal number of feeds per thread. The performance of the network and web servers is crucial, the same test can vary 40% between two runs. To make the system suitable for a faster CPU fewer feeds per thread should be used. It is possible to change the number of feeds per thread by changing the value of *rss_defautnrstreams()* in RSS-

Amos. The default value is 8 feeds per thread. The minimum number of threads can not be changed and it is set to three.

The performance of the whole implementation including the cache logic will be discussed in the chapter *2.3.2*.

#### 2.3.1.2 The performance of the ROME library

To investigate the performance differences between the class *SyndFeedImpl* and the class *Channel*, mentioned in the chapter *2.2.2*, a test was made. The test used an XML file stored on the hard drive to avoid accessing the Internet. The XML file was a copy of a real feed from BBC. There were four files in the test, first the original file with 33 articles. The other files had the articles from the original file repeated to show the performance with different number of articles. The same test was conducted twice and the average of the two tests is shown in Figure 15

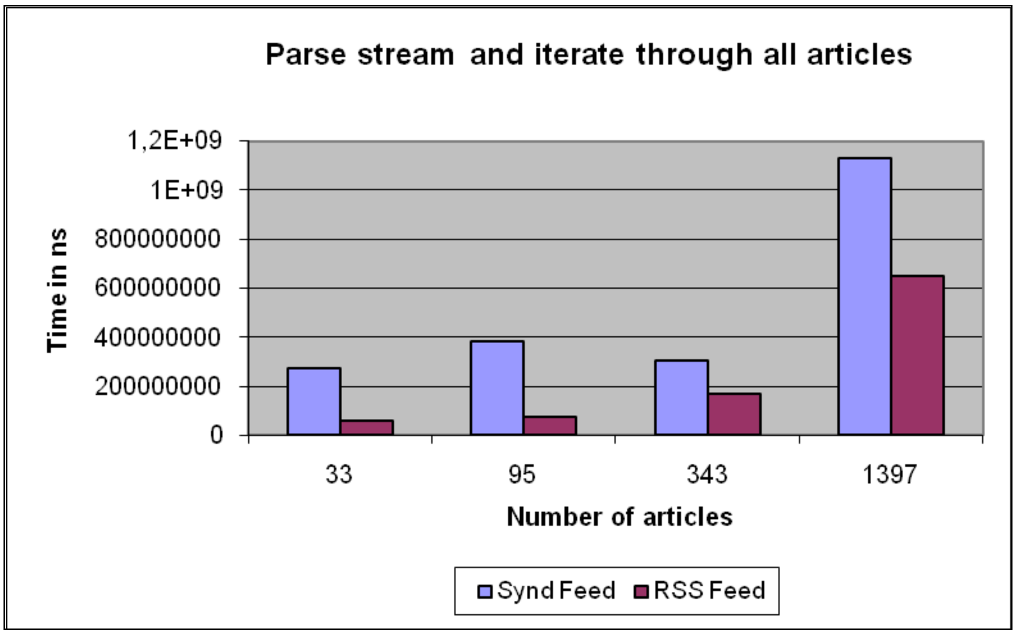


**Figure 15: Test of performance using classes from ROME**

The result shows that the *Channel* class is up to five times faster than *SyndFeedImpl*. There is a larger performance difference between the two classes when there are fewer articles in the feed. This is because there is no performance difference in parsing articles using a *SyndFeedImpl* or *Channel*. The result varies especially in the test with 343 articles. I suspect that this is due to other disc activities. The *SyndFeedImpl* was therefore not used in this project.

### 2.3.2 Evaluation

There have been the different implementations during the project. The implementations are called naive, feed caching and parallel feed caching. Table 14
indicates the most important functionality implemented in the different implementations.

**Table 14 Summary of the different implementations**

| **Functionality/Implementation** | Naive | Feed cache | Parallel feed caching |
|---|---|---|---|
| Uses mapped type | x | x | x |
| Can download feeds | x | x | x |
| Can store feeds | x | x | x |
| Can mutidirectional core cluster | | x | x |

| | | | |
|---|---|---|---|
| function | | | |
| Uses a feed cache | | x | x |
| Uses threads | | | x |

The result in Figure 13 shows a big difference between using one call to the foreign function *rss_GetStream* and the usage of threads and the use of multiple calls to the foreign function in a single thread as in the naive implementation. In the case of 148 feeds the parallel feed caching implementation was more then three times faster than the logic used in the naive and cache implementation.

The naive implementation uses only the mapped type definition and the Java logic when downloading single feeds. The big bottleneck in the naive implementation was that all feeds in the system were downloaded every time a query was made and that only one feed was downloaded in every call to the foreign function *rss_GetStream* responsible of downloading a feed and emitting the articles back to Amos II. The results from the test in Figure 13 shows that in a system with 148 feeds just downloading the articles to RSS-Amos would take 48s.

The largest bottleneck in the feed caching implementation is the call to the foreign function *RSS_GetStream* for each feed that needs an update. The feed caching implementation has some overhead with the management of the cache. The overhead is largest when there is a need for an update of a feed.

It is possible to categorize the types of queries against an *Rssitem* as:

1) Queries with a given feed address (the *streamssrc* property of feed is used).

```
E.g. count(select from Rssitem i where streamsrc(i) =
"http://www.sr.se/xml_news/rss/nyheterrss.xml");
```

2) Queries with no feed address given.

```
E.g. select description(i) from Rssitem i where like(title(i),
                                                     "*Sweden*");
```

Queries with a given feed address will have result size (fanout) as the number of articles belonging to one feed. The average number of articles is close to 20 (148 feeds had a total of 2871 articles this gives an average of 19.4 articles/feed). When the feed address is given and it is time for an update, in an implementation using a cache only one feed has to be downloaded. The overhead in the cache logic will only involve one specific feed making category one queries the fastest category in both the feed caching and parallel feed caching implementations. The naive implementation did not use multi-directional functions resulting in a download of all articles even if the feed source was known. Another problem with the naive implementation is that a query may need to access the data source multiple times during the execution of a single query resulting in multiple calls to the foreign function retrieving the articles as in the query in Table 16.

In the case were a query does not specify an address to a feed all articles from all registered feeds have to be evaluated. In the naive implementation all feeds are downloaded. The feed caching and parallel feed caching implementation checks the last update time and decides if the feed is still up-to-date using the current time, *ttl* and the *customttl* of the feed. If the feed is not up-to-date a download is needed. The feed caching implementation uses one foreign function call for each feed that is in need of an update. The parallel feed caching implementation collects all feeds that are in need of an update in a vector and the makes one call to a foreign function. This foreign function named *RSS_GetStreamsThread* uses threads to minimize the delay of downloading multiple feeds. The *ttl* and *customttl* varies between feeds and the *lastupdate* will probably also vary. These variations can improve the performance of the implementations using the cache because all feeds will not be in a need of an update at the same time. However, if the system has not been used for 30 minutes the probability for an update of all feeds is high.

As an example, assume that two feeds are registered with the system. The execution times below are based on a single query and the execution times includes the time to print the result on the screen.

The following query lists the titles from 52 articles:

```
select title(article) from Rssitem article;
```

The time to execute the query on the different implementations is listed in Table 15:

**Table 15**

|  | Feed caching |  | Parallel feed caching |  |
|---|---|---|---|---|
| Naive | No update needed | Update needed | No update needed | Update needed |
| 0,902s | 0,111s | 1,693s | 0,12s | 0,831s |

The following query lists the tiles from articles belonging to the feed with a short name *bbc:*

```
select title(article) from Rssitem article, rssstream stream
where streamsrc(article)=address(stream) and
      short_name(stream)="bbc";
```

Table 16 shows the time to execute the query on the different implementations.

**Table 16**

|  | Feed caching |  | Parallel feed caching |  |
|---|---|---|---|---|
| Naive | No update needed | Update needed | No update needed | Update needed |
| 21,506s | 0,13s | 0,521s | 0,14s | 0,581s |

The results of this example retrieved only the titles from one specific feed. It gives an idea of how slow the system gets when accessing the data source directly without a cache. The reason for the slow execution time is that the query results in multiple reads of the same data source. Multiple reads to the data source using the naive implementation results in multiple accesses to the Internet, reading and parsing the feed and finally emitting the result back to RSS-Amos.

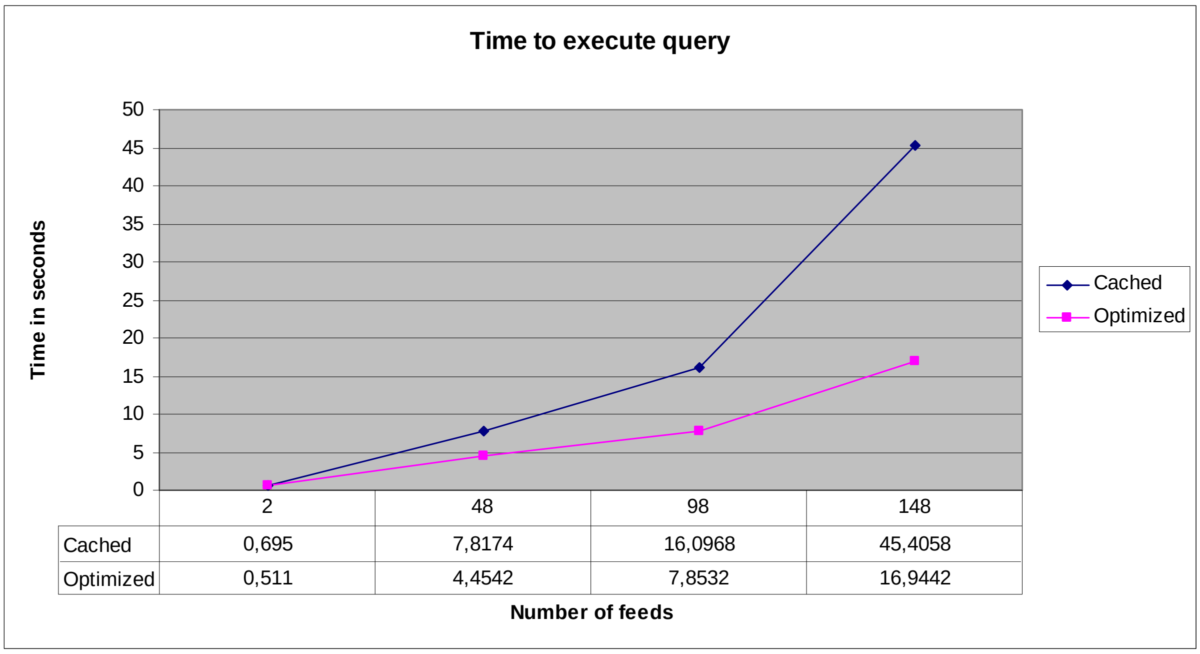
The results from this example show that when there is no need for an update the feed caching (0,111s) and parallel feed caching (0,12s) implementations are over seven times faster then the naive implementation (0,902s) using the simplest query and over 152 times faster using a slightly more complex query. When there is need for a cache refresh using the simple query that references two feeds the naive implementation (0,902s) is faster than the feed caching implementation (1,693s) and the parallel feed caching implementation is two times faster than the feed caching implementation (0,831s). In this case the parallel feed caching implementation uses two threads to download the feeds in parallel.

The naive implementation has another problem than slow execution times. The number of calls made to the Internet and especially the number of calls made to the same web server within a short time period will cause downloads to time out. A time out of one call will throw an *AmosException* in the wrapper causing a stop of the query being executed. Tests using RSS-Amos with 148 feeds have shown that a time out will occur some time between 250s and 550s of execution. Because of the problems with the naive implementation the rest of this chapter will focus on the performance of only the feed caching and parallel feed caching implementations.

The feed caching and parallel feed caching implementations use the same logic when the feed address is known. The parallel feed caching implementation has some overhead with the management of the threads in the wrapper, which can have impact on the performance when there are few feeds to download. Figure 16 shows the results using the feed caching and parallel feed caching implementations including the cache logic with a query involving all feeds in the system when all feeds needs an update.

Test results shown are averages of five executions. The time for the test was at 10.00 on a Friday. The query used in the test was:

```
count(select from Rssitem article));
```



| | 2 | 48 | 98 | 148 |
|---|---|---|---|---|
| Cached | 0,695 | 7,8174 | 16,0968 | 45,4058 |
| Optimized | 0,511 | 4,4542 | 7,8532 | 16,9442 |

**Figure 16: Execution time with the feed caching and parallel feed caching**

Figure 16 shows that the parallel feed caching implementation is the fastest in all the tests. The more feeds the larger differences between the two implementations.

**Table 17: Comparing feed caching- and parallel feed caching implementation**

| Nr of feeds | Times faster |
|---:|---:|
| 148 | 2,69 |
| 98 | 2,05 |
| 48 | 1,75 |
| 2 | 1,36 |

Table 17 shows how many times faster the parallel feed caching implementation is compared with the feed caching implementation.

Table 18 compares the results from the test where just the download times are measured using five threads and the results from the parallel feed caching implementation.

**Table 18: Comparing result from Figure 13 and Figure 16**

| | 2 feeds | 48 feeds | 98 feeds | 148 feeds |
|---|---:|---:|---:|---:|
| Download using 5 threads (s) | 0,2653 | 4,2963 | 6,6247 | 17,0099 |

| | | | | |
|---|---|---|---|---|
| Query using Parallel feed caching impl. (s) | 0,511 | 4,4542 | 7,8532 | 16,9442 |
| Overhead (s) | 0,2457 | 0,1579 | 1,2285 | -0,0657 |
| Overhead (%) | 92,6 | 3,7 | 18,5 | -0,4 |

The overhead varies a lot. The performance of the network and the load of the web servers is probably the reason to the rather large differences. One could think, looking on these results, that the performance difference of the parallel feed caching implementation is low compared with the use of a stored function. But the performance difference can be viewed as comparing querying the cache directly as a stored function and a query using the mapped type to call the parallel implementation, on a system where there is no need of an update. The following example will compare the parallel implementation of the mapped type *Rssitem* with a stored function in Amos II. E.g. in a system with 48 feeds where there is no need for an update, executing the following two queries *Q1* and *Q2*, shows that *Q1* took 0,02s and *Q2* took 0.09s to execute:

```
Q1: count(select from charstring src, vector v
     where v=rss_cache(src));

Q2: count(select from Rssitem article);
```

This example shows that the performance difference of the parallel feed caching implementation is over 400% (0.09s/0.02s) compared to a stored function in Amos II. The overhead performance difference consists of querying all registered *Feeds* in the system, decide if there is a need for an update, reading the cache and then return the result as the mapped type *Rssitem*. The time difference between the two queries is 0,07s. 0,07s is not a long time when comparing it with the download times of feeds from the Internet but it is a long time for a database.

The *ttl* of feeds in the system have effect on the performance of the cache. The distribution of the *ttl* in the test system using 148 feeds looks like Table 19.

**Table 19 the use of *ttl***

| *ttl* | number of *Feeds* |
|---|---|
| 0 | 100 |
| 5 | 1 |
| 10 | 17 |
| 15 | 29 |
| 240 | 1 |

The feeds used consist mostly of RSS channels (the *ttl* is used by RSS but not in Atom). Table 19 indicates that over 65 percent of the feeds do not use the *ttl*. When the *ttl* is not used the *customttl* will be used by the system where the default value is 15 minutes. The use of the *customttl* will result in 129 feeds with an update interval of 15 minutes. This will force the system to update all feeds, except one, if it has not been used in 15 minutes.

My conclusion is that the use of the cache brings an overhead to the system but the cache is needed. The cache makes the system usable. The parallel feed caching implementation is the only implementation that fully utilizes the CPU. Figure 13 shows how much time is spent waiting when there are no threads used compared with the use of threads doing parallel downloads. RSS-Amos tries to optimize the utilization of the CPU by ensuring that *feed materializer* always have a feed to process and at the same time limit the work done on parallel downloads. Figure 14 shows that the process of parallel downloads can have negative affect on the performance of emitting articles to the *feed* *materializer* of RSS-Amos. The parallel feed caching implementation can be tweaked for different systems by changing the number of feeds per thread making the system more flexible and faster. If the performance is crucial the system should be updated once and then put in *offline mode*, which is a configuration in RSS-Amos. Using the system in *offline mode* forces the system to read only articles stored in the cache. How to configure the *offline mode* is explained in Appendix A.

# 3 Summary and Future work and Discussion

RSS-Amos makes it possible to query web feeds using the query language AmosQL. RSS-Amos uses foreign functions to implement the wrapper. The mapped type called *Rssitem* represents a view of articles from web feeds and the type *Feed* represents the meta-data of the web feeds (i.e. feed channel meta-data). RSS version 2.0 is used as template for the properties of an *Rssitem* and *Feed*. RSS-Amos can handle all versions of both RSS and Atom. Future versions of RSS and Atom will be supported if the *ROME* project adds functionality for these and a new jar file is added to the system running RSS-Amos. The use of the Java interface to implement foreign functions makes the system easier to maintain compared with a faster low level language as C. The overhead of the cache was high when comparing queries made involving a stored function on the local system, but negligible compared to the execution time when accessing the Internet. The cache is a key feature that makes RSS-Amos usable. If the cache is not used RSS-Amos performs very poorly resulting in queries timing out or having execution times over one minute or more depending on the number of feeds in the system.

The project has answered these questions:

1) What is the most suitable RSS stream formats to access?
RSS version 2.0 has been used as a template when developing the stream representation in Amos II. All existing stream formats and versions can be represented in this stream type representation.

2) How can one implement a stream query language interface to an RSS stream?
The possibility to query articles is made possible through the mapped type called *Rssitem*. A cache has also been developed in AmosQL making it possible to query a large number of streams within Amos II.

3) What publically available programs should be used to implement such a streamed wrapper?
ROME has been chosen as the most suitable library for this project. ROME supports all stream versions and it is an active project.

4) What are the foreign functions needed for a flexible wrapper?
The wrapper has a multi threaded foreign function named *rss_GetStreamsThread* responsible for downloading articles from multiple feeds. A single threaded foreign function named *rss_GetStream* is used when downloading articles from a single feed. The wrapper uses a foreign function named *rss_AddStream* to make it possible to add feeds to the system.

5) What is the performance of such a streamed RSS wrapper? How can it be improved?
The performance have been investigated and resulted in the development of the cache and the parallel feed caching implementation. The performance can be further improved by the ROME Fetcher [21].

There are two interesting subprojects to *ROME*, *ROME Fetcher* and the *OPML for ROME [10]*[11]. *ROME Fetcher* can improve the performance of updating the articles. The *ROME Fetcher* uses *HTTP* (Hyper Text Transport Protocol) and the *Conditional GET* method [26]. The *Conditional GET* method determines if there have been any updates of the web feed without the need of downloading the feed first. Implementing this logic in RSS-Amos would improve the cache performance in the cases were there have been no updates made to the feed. *OPML* (Outline Processor Markup Language) is a protocol used when information about feeds is communicated between systems [19]. *OPML* makes it possible to import or export meta-data about web feeds. This feature would be very helpful in the process of adding or exporting large numbers of feeds to or from the RSS-Amos. *OPML for ROME* can be found under *document & files* under *ROMEs* project page [15]. UserLand Software has also been involved in *OPML* [19].

## References

[1] Berkman Cente Harvard Law: *RSS 2.0 at Harvard Law*, 2009. (http://cyber.law.harvard.edu/rss/index.html)

[2] D. Elin and T. Risch: *Amos II Java Interfaces*, Uppsala University, 2000.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns : elements of Reusable Object-Oriented Software*, Addison Wesley, ISBN 0-201-63361-2, 1994.

[4] IETF M. Nottingham, R. Sayre: *The Atom Syndication Format*, 2009. (http://www.ietf.org/rfc/rfc4287.txt)

[5] IETF, *STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESAGE*; 2009 (http://www.ietf.org/rfc/rfc0822.txt?number=822)

[6] Internet Content Syndication Council: *Content Creation and Distribution in an Expanding Internet Universe: A White paper, May 2008.*, 2009. (http://www.internetcontentsyndication.org/downloads/whitepapers/content_creation.pdf)

[7] J. Hunter, *JDOM*, 2008. (http://www.jdom.org/index.html)

[8] J. Markus: Translating SQL expressions to Functional Queries in a Mediator Database System Uppsala Master's Theses in Computing Science 293, ISSN 1100-1836, 2005 (http://user.it.uu.se/~udbl/Theses/MarkusJagerskoghMSc.pdf)

[9] java.net, *Rss and atOM utilitiEs (ROME) v0.5 Tutorial, Defining a Custom Module (bean, parser and generator)*, 2008**.** (http://wiki.java.net/bin/view/Javawsxml/Rome05TutorialSampleModule)

[10] java.net, *Rome Fetcher*, 2009 (http://wiki.java.net/bin/view/Javawsxml/RomeFetcher)

[11] java.net, *ROME : Documents & files*, 2009 (https://rome.dev.java.net/servlets/ProjectDocumentList?folderID=5198)

[12] M. Pilgrim, *The myth of RSS compatibility*, 2008. (http://diveintomark.org/archives/2004/02/04/incompatible-rss)

[13] M. Pilgrim, *What Is RSS*, 2008. (http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html)

[14] N. Schmuck, *Informa*, 2008. (http://informa.sourceforge.net/)

[15] Project ROME. (https://rome.dev.java.net/)

[16] R. Dornfest, *RSS: Lightweight Web Syndication*, 2008. (http://www.xml.com/pub/a/2000/07/17/syndication/rss.html)

[17] RSS-DEV Working Group, *RDF Site Summary (RSS) 1.0*, 2009. (http://web.resource.org/rss/1.0/spec)

[18] S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld*: Amos II Release 11 User's Manual*.

[19] Scripting News, Inc, *OPML 2.0 draft spec*, 2009 (http://www.opml.org/)

[20] sourceforge.net, *RSSLib for J*, 2008. (http://sourceforge.net/projects/rsslib4j/)

[21] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, ISBN 3-540-00375-4, 2003.

[22] T. Risch, *Functional Queries to Wrapped Educational Semantic Web Meta-data*, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.

[23] W. Litwin, T. Risch: *Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates*, IEEE Transaction on Knowledge and Data Engineering, Vol. 4, Mo. 6, December 1992.

[24] W3C, *Resource Description Framework (RDF)*, 2008. (www.w3.org/RDF)

[25] W3C, **Extensible** *Markup Language (XML) 1.0 (Fifth Edition): W3C Recommendation 26 November 2008*, 2009. (http://www.w3.org/TR/REC-xml/)

[26] W3.org, *HTTP Method Definitions*, 2009 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html)

[27] Wikipedia, *Aggregator*, 2009-05-05 (http://en.wikipedia.org/wiki/Aggregator)

[28] www.fitsch.ch , *UltimateNews*, 2009 (http://www.softsea.com/review/UltimateNews-RSS-to-database-fetch.html)

# Appendix A

## *Manage RSS-Amos*

### Install

| | |
|---|---|
| *install.cmd* | -compiles the java code and installs RSS-Amos |

### Start

| | |
|---|---|
| *Rssquery.cmd* | -starts the system |
| *Rssquerystartoffline.cmd* | -starts the system offline. To put the system online use the command *set rss_online() = true;* |

### Add a feed

The function *rss_addandgetstream* is used to add a feed and related articles to the system. By adding a feed the meta-data about the feed is stored in the system and the articles of the feed are added to the cache. A feed can have a short name to make it easier to reference it in queries.

```
Rss_addandgetstream(charstring url)

Rss_addandgetstream(charstring url, charstring shortname)
```

E.g.

```
rss_AddAndGetStream(
'http://www.sr.se/xml_news/rss/nyheterrss.xml');

rss_AddAndGetStream('http://newsrss.bbc.co.uk/rss/
newsonline_world_edition/europe/rss.xml', 'bbc');
```

### Online/Offline

If the system is used without a connection to the Internet or if the system should not be updated.

| | |
|---|---|
| `set rss_online() = true;` | -The system will update articles if they are considered old |
| `set rss_online() = false;` | -The system will use the articles that is stored in the cache. |

### Save state

The default name of the database used is *RSSAmos.dmp*. To save the state of the database use the following command.

```
save "RSSAmos.dmp";
```

### Change the number of feeds per thread

The default value is 8 feeds per thread. This is a good value on a Pentium III 800MHz. If a faster CPU is used then try to change the number to something less then 8 to increase the performance when the system updates multiple feeds.

```
set rss_defautnrstreams() = <integer>;
```

### Start logging of the Java implementation

Change the configuration file *log4j.properties* located under the *src* directory and do a re-installation of the system. Enabling the logging will severely degrade the performance.

| | |
|---|---|
| *OFF* | -No logging, the default value. |
| *DEBUG* | -logs debug information in the file *streamwrapper.log* located in the RSS directory |

# Appendix B

## *Types*

### *Rssitem* properties

```
uid(Rssitem)->Integer as stored;

title(Rssitem)->Charstring as stored;

description(Rssitem)->Charstring as stored;

description_type(Rssitem)->Charstring as stored;

streamsrc(Rssitem)->Charstring as stored;

 link(Rssitem)->Charstring as stored;

categories(Rssitem)->Vector as stored;

{{name, domain}, {name, domain}, …}


author(Rssitem)->Charstring as stored;

pubdate(Rssitem)->Charstring as stored;

source(Rssitem)->Charstring as stored;

comments(Rssitem)->Charstring as stored;

enclosures(Rssitem)->Vector as stored;

{{type, url, length}, {type, url, lengt2},  …}
```

```
guid(Rssitem)->Charstring as stored;

foreign_markup(Rssitem)->Vector as stored;
```

*{{unknown elements}, {unknown elements}, …}*

*feedof(Rssitem)->Feed*

## *Feed* properties

```
id(Feed)->Integer key as stored;
short_name(Feed)->Charstring key as stored;
title(Feed)->Charstring as stored;
description(Feed)->Charstring as stored;
link(Feed)->Charstring as stored;
address(Feed)->Charstring key as stored;
language(Feed)->Charstring as stored;
categories(Feed)->Vector of Charstring as stored;

{category, category, category, … }



copyright(Feed)->Charstring as stored;
managingEditor(Feed)->Charstring as stored;
webmaster(Feed)->Charstring as stored;
pubdate(Feed)->Charstring as stored;
lastbuilddate(Feed)->Charstring as stored;
generator(Feed)->Charstring as stored;
docs(Feed)->Charstring as stored;
cloud_domain(Feed)->Charstring  as stored;
cloud_path(Feed)->Charstring  as stored;
cloud_port(Feed)->Charstring  as stored;
cloud_protocol(Feed)->Charstring  as stored;
cloud_procedure(Feed)->Charstring  as stored;
image_description(Feed)->Charstring as stored;
image_hight(Feed)->Charstring as stored;
image_width(Feed)->Charstring as stored;
image_url(Feed)->Charstring as stored;
image_link(Feed)->Charstring as stored;
image_title(Feed)->Charstring as stored;
rating(Feed)->charstring as stored;

skipdays(Feed)->Vector of charstring as stored;

{day1, day2, day3, …}



skiphours(Feed)->Vector of Charstring as stored;
```

*{hour1, hour2, hour3, …}*

```
textinput_title(Feed)->Charstring as stored;
textinput_name(Feed)->Charstring as stored;
textinput_description(Feed)->Charstring as stored;
textinput_link(Feed)->Charstring as stored;
ttl(Feed)->Integer as stored;
customttl(Feed)->Integer as stored;
lastupdate(Feed)->Timeval as stored;

cache(Feed)->bag of Vector;
```

# Appendix C

## *Examples of queries*

```
count(select from Rssitem i where streamsrc(i)=
"http://www.sr.se/xml_news/rss/nyheterrss.xml");



select description(i) from Rssitem i, rssstream s
where like(title(i), "*Sweden*") and streamsrc(i)=address(s) and
short_name(s)=”bbc”;



select title(article) from Rssitem article
where like(description(article),*Obama*);



select distinct ttl(feedof(i)) from Rssitem i;
```