IT 10 030

Examensarbete 30 hp
Juni 2010

Querying JSON Streams

Yang Bo

Institutionen för informationsteknologi
Department of Information Technology

Abstract

Querying JSON Streams

Yang Bo

A data stream management system (DSMS) is similar to a database management system (DBMS) but can search data directly in on-line streams.

Using its mediator-wrapper approach, the extensible database system, Amos II, allows different kinds of distributed data resource to be queried. It has been extended with a stream datatype to query possibly infinite streams, which provides DSMS functionality.

Nowadays, more and more web applications start to offer their services in JSON format which is a text-based, human comprehendible format for representing simple data structures and associative arrays. For example, one of the most popular websites worldwide, Twitter, has developed a stream interface that one can register to receive large amounts of Twitter messages in JSON format. Another popular website Facebook and some weather services also provide stream interfaces through JSON

The objective of the project is to develop a general JSON stream reader as an Amos II wrapper, allowing such streams to be easily queried and managed by a DSMS. To implement such a system, three primitive foreign functions are defined to allow queries to JSON streams using the query language AmosQL of Amos II. The usefulness of the developed wrapper is demonstrated on the popular Twitter social network data stream.

Handledare: Györö Gidófalvi
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 030
Tryckt av: Reprocentralen ITC

Table of contents

1. Introduction

JSON (JavaScript Object Notation) [12] format is a lightweight computer data interchange format which is text-based, human comprehendible, and language independent for representing simple data structures and associative arrays. In recent years, more and more web applications like Yahoo and Google have started to offer some of their web services in JSON.

A growing number of applications such as network monitoring, telecommunications data management, manufacturing and sensor networks, are dealing with a new type of data which is in the form of continuous data streams rather than stored tables as in conventional database. Their clients require long-running continuous queries as opposed to one-time queries. Data stream management systems (DSMSs) [10] process on-line stream queries as well as queries to regular stored data.

In the extensible main-memory DBMS [17] Amos II [21] database queries are expressed in AmosQL [4], which is a query language based on functions. AmosQL has been extended with a special *stream* datatype to support queries over possibly infinite streams. This extension as well as some stream primitives provide Amos II with DSMS functionality.

Amos II also provides a generalized foreign function mechanism and has native APIs between Amos II and the programming languages C, Lisp, and Java. By using the foreign function interface, external data sources can be queried by AmosQL.

Twitter is one of the most popular websites worldwide nowadays. It’s a free social network that enables its users to share and discover what is happening right now. Twitter has developed a general API to access its internal database as well as a stream interface that one can register with to receive large amounts of Twitter messages. Of particular interest in this project is the Twitter streaming API [5] that returns data as a stream of JSON Objects.

The purpose of this project is to develop a general AMOS II wrapper for JSON streams data using foreign functions and libraries. As a result, a JSON stream query system (JSQ) is defined. Instead of developing a new query language, the existing AmosQL is extended to support querying JSON streams. The JSQ system can be mainly divided into two functional components: authentication and query. The first one sends the URL, username, and password as parameters to a web application for authentication and returns a connection identity if it is passed. The second function uses the connection identity to query streaming data and processes them as Amos II recognizable data types.

As proof-of-concept the system is tested for the messages emitted by Twitter’s stream interface. The Twitter stream interface is an HTTP-based interface that given a URL, a user’s credential, and keywords as parameters, returns a continuous stream of Twitter messages in JSON format.

To implement the wrapper, the public interface to Twitter streams was investigated along with some other concepts like HttpClient [8], which is used to generate HTTP request and return response data incrementally.

The remainder of this report is organized as follows. Section 2 introduces the technology that is related to this project. Section 3 presents the implementation details of the wrapper. Section 4 describes related work. The report is completed with a conclusion and future work.

2. Background

2.1Database Management Systems (DBMSs)

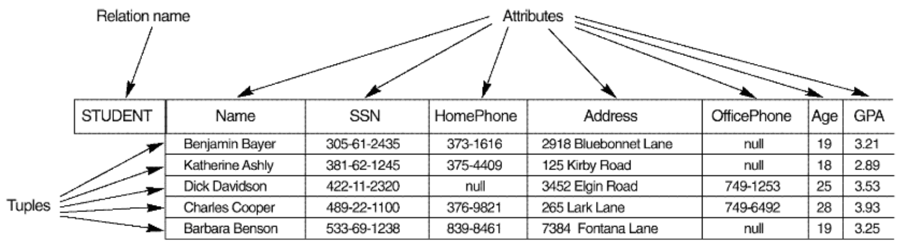
A Database Management System (DBMS) is a collection of programs that enables users to create and maintain a database [17]. It allows users and other software to store and retrieve data in a structured way by using some data model, e.g. the relational data model where data is represented as tables. It also provides appropriate languages and interface for the users who need to manipulate database.

DBMSs are used to store persistent data and support complex queries for applications. While this model adequately represents commercial catalogues or repositories of personal information, many current and emerging applications require support for online analysis of rapidly changing data streams. Therefore, a new kind of system to manage streaming data will be introduced in chapter 2.2, so called *Data Stream Management Systems (DSMSs)*.

2.1.1 The relational data model

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values [17]. Each row in the table is a collection of related data values which represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to represent the meaning of the values in each row. In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation.*

An example of a table is shown in Figure 2.1:


Relation name
Attributes
Tuples

STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

Figure 2.1 Data in relational format

2.1.2 Structured Query Language (SQL)

SQL is a standard language for definiting, querying, and updating commercial relational DBMSs. It uses the terms *table*, *row*, and *column* for *relation*, *tuple*, and *attribute*, respectively [17]. The general syntax to create a new table is:

```
   create table <table name>
```

One basic statement for searching information from a database is the SELECT statement. The basic form of the SELECT statement, sometimes called a *select-from-where* block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form [17]:

```
SELECT <attribute list>
FROM  <table list>
WHERE <condition>
```

Where *attribute list* is a comma separated list of attribute names whose values are to be retrieved by the query; *table list* is a comma separated list of the relation names required to process the query; *condition* is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

2.2 Data Stream Management System (DSMS)

Traditional database management systems are best equipped to run one-time queries over finite stored data sets. In many applications nowadays such as network monitoring, telecommunications data management, click stream monitoring, and so on, data takes the form of continuous data streams rather than finite stored data sets, and clients require long-running continuous queries as opposed to one-time queries [7].

Limitations of traditional DBMSs in supporting streaming applications have been recognized, prompting research to augment existing technologies and build a new kind of systems, called *Data Stream Management Systems (DSMSs)*, to manage streaming data.

2.2.1 DSMS Architecture

An abstract architecture of DSMS is shown in Figure 2.2 [16].

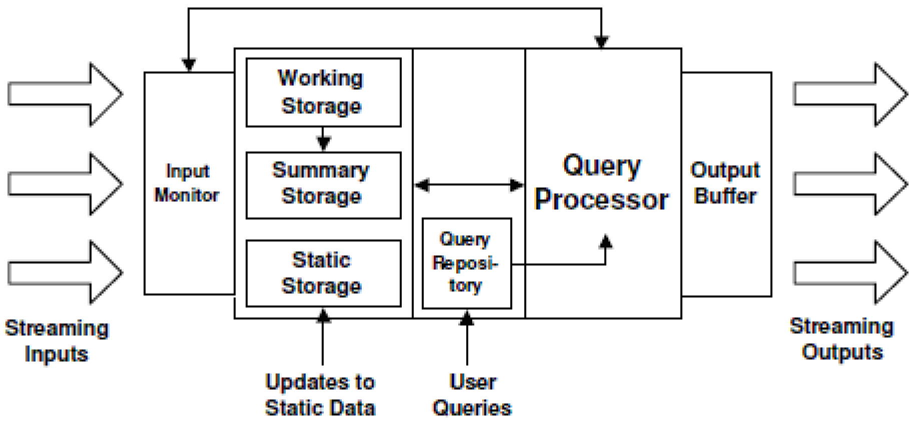


Figure 2.2 Abstract architecture for a data stream management system

In Figure 2.2, an input monitor controls the input rates, perhaps by dropping packets. The input data are typically stored in three partitions: temporary working storage (e.g. for window queries), summary storage for stream synopsis, and static storage for meta-data (e.g. physical location of each source). Long-running queries are registered in the *Query Repository* and are placed into groups for shared processing. Snapshot queries over the current state of the stream may also be posed. When a continuous query is registered with DSMS system, a query plan is compiled from it. Query plans are composed of *operators*, which perform the actual processing, *queues*, which buffer tuples (or references to tuples) as they move between operators, and *synopses*, which store operator state. The query processor communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered [16].

2.2.2 Streaming query language

For simple continuous queries over streams, it is possible to use the relational query languages such as SQL by replacing references to relations with references to streams, and create new tuples in the result.

However, as continuous queries grow more complex, e.g., with the addition of aggregation, subqueries, windowing constructs, and joins of streams and relations, the semantics of a conventional relational language applied to these queries quickly becomes unclear [16]. To address this problem, some query languages for stream data are designed. For example, the Stanford University developed a new query language called Continuous Query Language (CQL) [1].

The abstract Semantics of CQL is based on two data types--*streams* and *relations*--and three classes of operators over these types: operators that produce a relation from a stream (*stream-to-relation*), operators that produce a relation from other relations (*relation-to-relation*), and operators that produce a stream from a relation (*stream-to-relation*).

Some example CQL queries like following [1]:

```
1.Select Istream(*) From S [Rows Unbounded] Where S.A > 10
```

Stream S is converted into a relation by applying [Rows Unbounded] . The relation-to-relation filter "S.A > 10" acts over this relation, and the Istream() function streams the filtered relation as result.

```
2.Select * From S1 [Rows 1000], S2 [Range 2 Minutes] Where S1.A = S2.A And
S1.A > 1
```

The above query is a windowed join of the last 1000 tuples of streams S1 and the tuples of stream S2 that have arrived in previous 2 minutes.

```
3.Select Rstream(S.A, R.B) From S [Now], R Where S.A = R.A
```

This continuous query probes a stored table R based on each tuple in stream S and streams the result.

The extensible main-memory DBMS Amos II has extended its query language AmosQL with a special *stream* datatype to allow the querying of possibly infinite streams, thereby enable DSMS functionality. These extensions are used in the present work to provide queries over JSON streams provided by the Twitter Streaming API [5].

2.3 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format which is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 [12]. Like XML[1], it is a text-based, human readable format for representing simple data structures and associative arrays. XML is not as well-suited to data-interchange as JSON because it carries a lot of excess information and does not match the data model of most programming languages. By contrary, JSON is completely language independent and uses conventions that are familiar to programmers of the C-family of languages, like C, C++, C#, Java, JavaScript, Perl, Python, and so on.

Examples of XML and JSON are in Figure 2.3 and 2.4 [13]:

[1] XML (Extensible Markup Language) is a set of rules for encoding documents electronically.

```
{"widget": {
    "debug": "on",
    "window": {
        "title": "Sample Konfabulator Widget",
        "name": "main_window",
        "width": 500,
        "height": 500
    },
    "image": {
        "src": "Images/Sun.png",
        "name": "sun1",
        "hOffset": 250,
        "vOffset": 250,
        "alignment": "center"
    },
    "text": {
        "data": "Click Here",
        "size": 36,
        "style": "bold",
        "name": "text1",
        "hOffset": 250,
        "vOffset": 100,
        "alignment": "center",
        "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
}}
```

Figure 2.3 Example of JSON

```
<widget>
    <debug>on</debug>
    <window title="Sample Konfabulator Widget">
        <name>main_window</name>
        <width>500</width>
        <height>500</height>
    </window>
    <image src="Images/Sun.png" name="sun1">
        <hOffset>250</hOffset>
        <vOffset>250</vOffset>
        <alignment>center</alignment>
    </image>
    <text data="Click Here" size="36" style="bold">
        <name>text1</name>
        <hOffset>250</hOffset>
        <vOffset>100</vOffset>
        <alignment>center</alignment>
        <onMouseUp>
            sun1.opacity = (sun1.opacity / 100) * 90;
        </onMouseUp>
    </text>
</widget>
```

Figure 2.4 Example of XML

The above examples express the same data in JSON and XML format, respectively. Without the complex layers of tags used in XML, JSON is easier for both humans and computers to read and write.

2.3.1 JSON data model

JSON has two basic (possibly nested) structures: *JSONObject* and *JSONArray*.

A *JSONObject* is a collection of *name/value* pairs. It begins with { (left brace) and ends with } (right brace). Each *name* is followed by : (colon) and each of the *name/value* pairs is separated by comma (,) [12]. The structure of a JSONObject is presented in Figure 2.5.

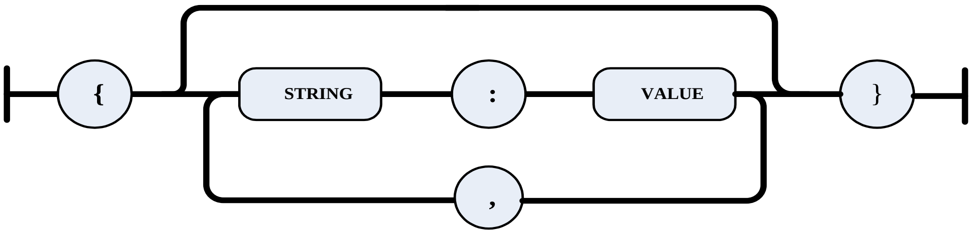


Figure 2.5 Structure of *JSONObject*

In various languages, this is realized as an *object, record, struct, dictionary, hash table, keyed list,* or *associative array*. In AmosQL, it corresponds to a datatype called *Record*.

A *JSONArray* is an ordered collection of values. It begins with a left bracket ([) and ends with a right bracket (]). Values are separated by comma (,) [12]. The structure of a JSONArray is presented in figure 2.6.

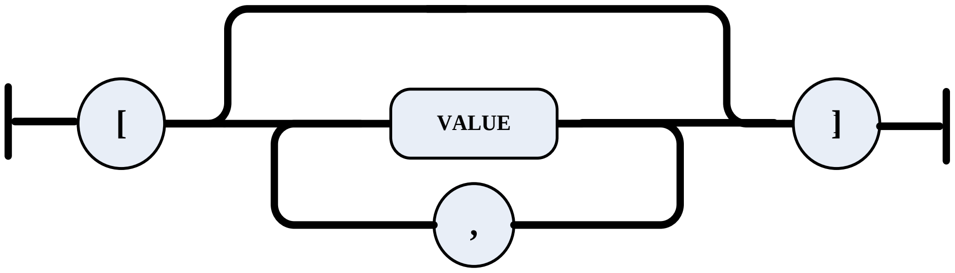


Figure 2.6 Structure of JSONArray

In most languages, this is realized as an *array, vector, list* or *sequence*. In AmosQL, it is mapped to a datatype called *Vector*.

A *value* can be a string in double quotes, a number, true, false, null, a JSONObject, or a JSONArray. These structures can be nested. The description of *value* is presented in figure 2.7.

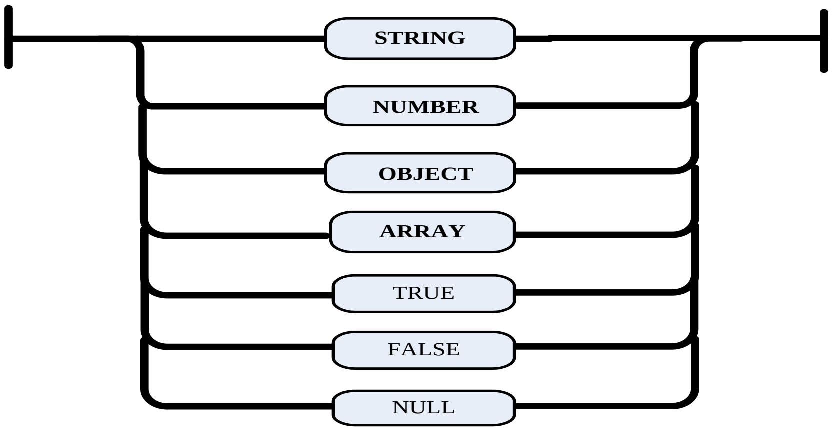


Figure 2.7 Possible data types of value

The above structures are universal data structures. Almost all the modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures [12].

In the present work JSON objects represented using corresponding Amos II data types are used for representing the contents in streams of JSON objects delivered over the internet.

2.4 Twitter Streaming API

Twitter was first created by Jack Dorsey in 2006 and now has gained notability and popularity worldwide. It is a free social networking and micro-blogging service that enables its users to send and read messages known as *tweets* [7]. *Tweets* must be under 140 characters in length and can be sent via the Twitter website, Short Message Service (SMS) or external applications. Senders can restrict delivery to those in their circle of friends or, by default, allow open access [7].

The Twitter Streaming API allows near-realtime access to various subsets of Twitter public statuses[1]. However, access to restricted resources is extremely limited by this API and is only granted on a case-by-case basis after acceptance of an additional terms of service document [5].

2.4.1 Protected and public accounts

There are two kinds of Twitter accounts: protected and public. The protected accounts are accounts that belong to users who have explicitly checked the 'protect my tweets' option in their profile. Only the statuses created by non-protected accounts are the candidates in the streaming API.

2.4.2 Authentication and connection

When connecting to the Streaming API, the HTTP Basic Authentication [8] is required which means a client must provide the credentials - in the form of a user name and password - of a valid Twitter account.

Because the Streaming API is HTTP-based, it is necessary to form a HTTP request using an HTTP client. The selected HTTP client should be able to return the response data incrementally [5]. Most robust HTTP clients satisfy this requirement. In this project Apache HttpClient [8] is chosen to handle this use case.

To access the Streaming API, each account can create only one streaming connection. Subsequent connections from the same account may cause previously established connections to be disconnected [5]. Excessive connection attempts, regardless of success, will result in an automatic and temporary ban of the client's IP address [5].

2.4.3 Streaming access methods

Currently, there are four methods provided in Twitter's Streaming API:

statuses/firehose
returns all public statuses.

statuses/sample
returns a random sample of all public statuses. The default access level provides a small proportion of the result returned from the *statues/firehose* method.

statuses/filter
returns public statuses that match one or more filter predicates[2].

statuses/retweet
returns all retweets[3].

At default access levels, all accounts may access the methods *statuses/sample* and *statuses/filter*. Access to other methods requires a special arrangement with Twitter and few applications require this level of access. Since Twitter does not recommend applications to access the *statuses/firehose* and *statues/retweet*

[1] For the example of public statuses, please refer to appendix A.
[2] The filter predicates is the criteria that Twitter Streaming API is using to filter public statuses.
[3] A Retweet in Twitter is when you see a tweet you like written by someone else, and you want to share that tweet with your Followers. [11]

methods, we will only consider the first two methods in this project. Furthermore, as the *statuses/filter* method can use predicates to filter the streamed data which makes its result more meaningful, the JSQ system is developed and tested mainly based on the *statuses/filter* method. The detailed description of *statuses/filter* will be in chapter 2.4.4.

2.4.4 Statuses/filter

The *statuses/filter* method returns public statuses that match one or more filter predicates.

At least one predicate parameter, *track* or *follow*, must be specified [6]. With the *follow* parameter, the method returns public statuses of the given set of users. With the *track* parameter, the method returns data that include the specified keywords in their text field. *Track* is case-insensitive. Terms are exact-matched, and also exact-matched ignoring punctuation. Phrases, keywords with spaces, are not supported [6]. Keywords containing punctuation will only exact match tokens [6].

Here are the examples provided by the Twitter Streaming API documentation about how *track* and *follow* predicates are specified in curl[1] :

```
Example: Create a file called 'tracking' that contains, exactly and excluding the
quotation marks: "track=basketball,football,baseball,footy,soccer". Execute: curl -d
@tracking http://stream.twitter.com/1/statuses/filter.json -uAnyTwitterUser:Password.
You will receive JSON updates about various crucial sportsball topics and events.
```

```
Example: Create a file called 'following' that contains, exactly and excluding the
quotation      marks:      "follow=12,13,15,16,20,87".      Execute:      curl      -d
@following http://stream.twitter.com/1/statuses/filter.json                            -
uAnyTwitterUser:Password. You will receive JSON updates from Jack Biz, Crystal, Ev,
Krissy, but not from Jeremy, as he's a private user.
```

Two optional query parameters, *count* and *delimited*, are also valid. *Count* indicates the number of previous statuses to consider for delivery together with the real-time streams. Its range is -150,000 to 150,000. Positive values transition seamlessly to the live stream. Negative values terminate when the historical stream[2] has finished, useful for debugging [6]. *Delimited* indicates that statuses should be delimited in the stream. Statuses are represented by a length, in bytes, a newline, and the status text that is exactly length bytes [6].

In summary, the statuses/filter method can be described like this [6]:

URL: http://stream.twitter.com/1/statuses/filter.format
Formats: xml, json
Method(s): POST
Parameters: count, delimited, follow, track
Returns: stream of status elements

2.4.5 Parsing response

The Streaming API can return data in XML or JSON formats. Twitter encourages the clients to use the more compact JSON representation.

Parsing JSON responses from the Streaming API is simple: every object is returned on its own line, and ends with a carriage return [6]. Newline characters (\n) may occur in object elements (the text element of a status object, for example), but carriage returns (\r) should not [5]. Parsing XML is slightly more challenging, as objects include newlines and carriage returns.

[1] Curl is a command line tool for transferring data with URL syntax, supporting various protocol.
[2] The historical stream is a number of previous public statuses before the live stream delivery.

In this project, the received stream data is in JSON format.

2.5 Amos II

Amos II is an object-relational DBMS. It allows different kinds of distributed data resource to be queried using its mediator-wrapper approach. Amos II expresses its database queries in AmosQL which is a query language based on functions [21].

2.5.1 Data Model

All entities in the database are represented as objects and managed by the system. They are divided into mainly two types: *literals* and *surrogates* [14]. Literals are system maintained objects such as numbers and strings. Surrogate objects are defined by the users of system and characterized as having explicit object identifiers (OID's) [14]. Examples of surrogates could be objects that representing real world entities like a car or a person.

Objects in Amos II are created as instances of data types. The general
syntax of creating a new type is [19]:

```
create type <typename>
```

Given the type a new object is defined as:

```
create <typename> instances <variable>
```

Functions in Amos II model the semantics of objects e.g properties of objects, computations over objects, and relationships between objects [20]. A function definition consists of signature and implementation. The signature defines the types and optional names of arguments and result parameters. The implementation computes the result using the given argument values. The general syntax of creating a function signature is [19]:

```
create function <functionname(type)>
                           -> <return type>
```

This function can then be called by the users or other functions.

2.5.2 External interface

Amos II contains number of primitives for accessing different external data sources by defining *wrappers* for each kind of external source [19]. A wrapper makes it possible to query an external data source using AmosQL.

The basis for accessing external systems from Amos II is to define *foreign functions* [19]. A *foreign function* is defined in external programming language and Amos II provides interfaces to Java, C and Lisp currently. In this project, the JSON stream wrapper is implemented in Java.

To implement a foreign function in Java usually needs three steps [4]:

1. Implement the function in Java code. The foreign functions implemented in Java are similar to any other Java functions with a signature and a body but they always have two arguments: *context* and *tuple*. The *context* is a data structure managed by the system to pass information between the foreign functions and the rest of the system [4]. The parameter *tuple* is a *Tuple* object holding both the argument(s) and the result(s) of the function [4]. The syntax is:

```
public void <functionname>(CallContext cxt, Tuple tpl) throws
AmosException;
```

2. Hook the foreign function with Amos II using AmosQL. The foreign function must be assigned a function resolvent by executing the AmosQL statement[4]:

```
create function <fn>(<argument declaration>)
                      -><result declaration>
  as foreign 'JAVA:<class file>/<method>';
```

   where *fn* is the name of the foreign function in AmosQL, *argument declaration* is the signature of the argument(s), *result declaration* is the signature of the results, *class file* is the name of the class (file) which is implemented in Java, *method* is the name of the method implementing it. This definition is done from Java by calling the method *execute* or by an AmosQL command in the top loop [4].

3. Define an optional *cost hint* to estimate the cost of executing the function. Amos II functions can be *multi-directional* which means that also their inverses can be executed when some result are known and some corresponding argument values are not [19]. As a consequence, one multi-directional foreign function can have different execution costs. *Cost hint* is used to tell the query optimizer which implementation should be chosen in complex queries as the most efficient one [19].

3. The JSON Stream Query System (JSQ)

JSQ is an extension to Amos II to enable wraping JSON streaming data. From the user's point of view, it is a system that can receive JSON streaming data by calling Amos II foreign functions and query the streaming data with AmosQL. The JSON stream is generated by web applications such as Twitter Streaming API. The structure of the system is presented in Figure 3.1.

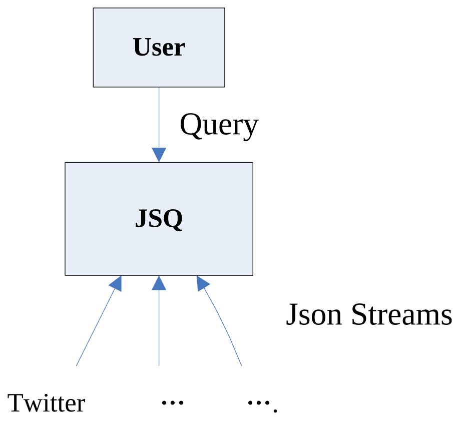


Figure 3.1 User level architecture of JSQ

3.1 Example queries and results

1. Find five most recently tweets talking about 'Obama' on Twitter:

```
JavaAMOS 1>set :s=jsonstream(0,{"track":"Obama"});
0.026 s
JavaAMOS 2>select r["text"] from Record r where r in section(:s,0,5);
```

The query result is:

```
"RT  @washingtonpost:  President  Obama  will  promote  more  education
spending   in   his   State   of   the   Union   speech   Wednesday.   -
http://wpo.st/PLR ..."
"An article  about President Obama support for a new agency to protect
consumers against lending abuses misstated... http://bit.ly/81VFXC"
"US troop buildup is big; Afghan buildup is key (AP): AP - President
Barack Obama has his troop surge. Afghan 7FEyBA !"
"Shepard  Fairey  Faces  Criminal  Investigation  In  AP/Obama  Poster  Case
(source: Arts Journal): ''In October, the L.A.... http://bit.ly/dgsjr8"
"Loaded: Jobs and Obama on the same day: Two important dudes are giving
big keynote addresses today: Steve Jobs and... http://bit.ly/aU0Nqu"
```

The execution time is about 12 sec.

2. Find the locations of five people who are talking about 'Obama' on Twitter currently:

```
JavaAMOS 1>set :s=jsonstream(0,{"track":"Obama"});
0.016 s
```

```
JavaAMOS 2>select r["user"]["location"] from Record r
where r in section(:s,0,5);
```

The query result is:

```
"Miami"
"null"
"iPhone: 40.769150,-73.865921"
"California"
"Canada"
```

The execution time is 6.688 sec.

3. Get the tweets from the people you are following with and their names in real-time:

```
JavaAMOS 1>set :s=jsonstream(0,{"follow":"92778994,70900570,26480012"}
);
JavaAmos  2>select  r["text"],r["user"]["name"]  from  Record  r  where  r
in :s;
```

The execution result is:

```
<"This is a testing tweet. ","Bo Yang">
<"This is the second testing tweet.","Bo Yang">
…
```

3.2 System Architecture

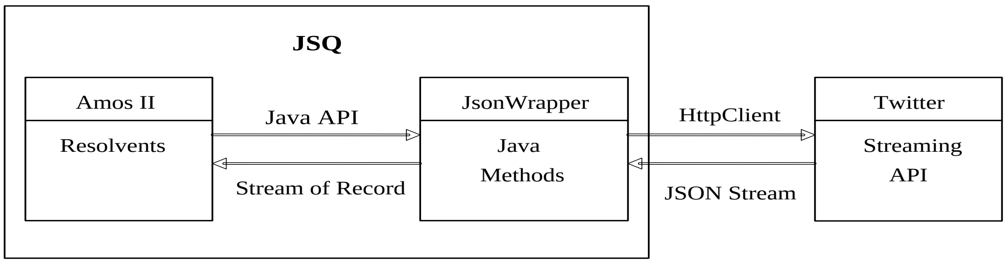


Figure 3.2 System Architecture of JSQ

Figure 3.2 shows the architecture of the JSON Stream Query System based on the Twitter Streaming API. The left part is the core of system where Amos II foreign functions are defined in Java and called by Amos II through its Java API. The Java class is called *JsonWrapper*. It is implemented to access web applications through Jakarta Commons HttpClient and to process the received JSON stream data into Amos II recognizable type of stream data such as *record*. Hence the users of Amos II can query JSON streams directly by using this system. Notably, the component to the right of the *JsonWrapper* can be other web applications that return data as JSON streams.

3.3 Implementation

Because the Twitter Streaming API returns data as JSON streams, it is a good instance to demonstrate the JSQ system. JSQ only needs the Twitter URL, account information, and query keywords as the parameters of HttpClient and pass them to the server side. The implementation of JSQ is general and contains no Twitter related library so that it can be applied to other JSON stream web services easily.

The implementation of JSQ can be divided into two stages:

Stage 1: Initialize an HttpClient connection with given URL, username and password. This step authenticates whether the URL, username, and password are valid. If they are, the connection will be stored with an identifier in the JSQ system and could be reused later by other HttpClient methods. If they are not, an exception will be thrown.

Stage 2: Use the connection defined above to send query parameters to the server side and get response data as a JSON stream. Then process the JSON stream as a stream of Amos II recognizable data types, e.g. *Record* or *Vector*. When the stream query is ended (e.g. by a stop condition or explicit interrupt) a *finally{…}* statement aborts the execution of the HTTP method and releases the connection being used by the HTTPClient method. This is a crucial step to keep things flowing because we must tell HttpClient that we are done with the connection and that the resource can now be reused [9].

The above stages are implemented in Java since Java APIs are provided by all of the HttpClient, JSON and Amos II. Each stage is implemented as a separate foreign function. According to the wrapper mechanism of Amos II, before these foreign functions can be used, they need to be hooked up to Amos II. This is done by creating function signatures and assigning them to the corresponding foreign functions. Therefore, the system consists of three foreign function definitions as shown in Figure 3.3:

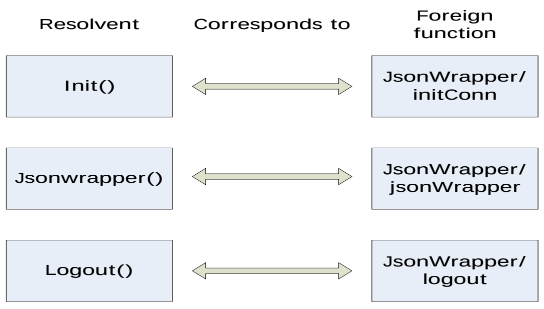


Figure 3.3 Foreign functions and their corresponding resolvents

In Amos II the functions are defined as:

```
create function init(charstring url,charstring name,
                      charstring pass)-> Integer
  as foreign "JAVA:JsonWrapper/initConn";

create function jsonwrapper(Integer id,Record r)
                             ->Bag of Record
  as foreign "JAVA:JsonWrapper/jsonWrapper";
```

where the *id* is the connection identifier returned from the *init()* function and *r* is the query parameters, e.g. {“track”: “a”}

The *Stream* data type in Amos II supports queries to possibly infinite streams, the function *jsonstream()* is defined to stream the indefinite result generated by *jsonwrapper()*. The system function streamof indicates the a stream rather than a bag is returned:

```
create function jsonstream(Integer id,Record r)
                     ->Stream of Record
  as streamof(jsonwrapper(id,r));
```

The stream data emitted by Twitter is a series of *JSONObjects*. Those *JSONObjects* will be processed into Amos II objects of type *Record* where a record in Amos II represents name/value pairs, the same as the structure of *JSONObject*. If the stream data is in *JSONArray* format, it will be processed into object of type *Vector* before returned back to Amos II. In other words, AmosQL is extended to support JSON stream query in JSQ system. The following table lists the type mappings from JSON data types to Amos

II data types:

JSON	**AmosQL**
Object	Record
Array	Vector
Integer	Integer
Real	Real
String	Charstring
True/False	Boolean

According to this table, besides *JSONObject* and *JSONArray*, the other primitive data types in JSON are mapped to the corresponding *literal* objects in Amos II. These extensions allow users to search a JSON stream for complex conditions that are expressed simply using the high level declarative AmosQL syntax. While the general filter method allows only to search for disjunction of predicates in the text field of the JSONObjects, this extension allows both to search for disjunctions and conjunctions of any predicate over any field of the JSONObjects. For example, it is able to query the tweets including both keywords “a” and “is” as:

```
JavaAMOS 6> set :m=jsonstream(0,{"track":"a"});
0.016 s
JavaAMOS 7> set :n=jsonstream(0,{"track":"is"});
0.015 s
JavaAMOS 8> select r1["text"] from Record r1,Record r2 where r1 in
section(:m,0,2) and r2 in  section(:n,0,2) and r1["id"]=r2["id"];
4.89 s
```

3.4 Evaluation

The execution time of a query may be impacted by many factors such as the contents of query parameters in HTTP requests, the time of the querying and so on. For example, using “Obama” and “a” will get quite different number of results per second.

Below is the table of execution time that querying tweets with keywords “a” and “Obama” from Twitter Stream API with 5 and 10 results respectively. The unit of time is second.

	a (5)	Obama (5)	a (10)	Obama (10)
1	1.016	20.437	1.312	12.266
2	1.031	17.812	1.234	20.734
3	1.016	10.907	1.219	44.702
4	1.406	14.406	1.203	25.266
5	1.094	11.563	1.265	27.906

The table above should not be seen as an evaluation of the JQS system but rather are an indication of the response intensity of different queries. The execution times are highly depending on the frequency of returning the result from Twitter. According to the table above, it is obvious that querying “Obama” will take much longer time than “a” because “a” appears in tweets more frequently than “Obama”. Even if you request the same keyword at different time in a day or in a period, the execution time will have great difference. In other words, in long-running stream applications, data and arrival characteristics of streams may vary significantly over time [1].

4. Related work

Most of the implemented data stream management systems nowadays, like Aurora [3] and STREAM [1], are used for monitoring applications such as sensors, networking and stock feeds. Another DSMSs called Xstream[15] was developed for supporting high-rate signal processing. The JSQ system is specially developed for querying and management the stream data which are in JSON format.

In all of the systems mentioned above, Aurora [3], STREAM [1] and Xstream [15], new semantics and query language are defined for continuous queries over streams and relations. Instead of developing a new query language in JSQ system, the existing AmosQL query language is extended to allow the users to do queries over JSON streams.

5. Conclusion and future work

The JSON stream querying system (JSQ) is an extension to Amos II. It takes the DSMS functionality of Amos II database and offers the user an easy way to query and manipulate stream data which are in JSON format.

JSQ was implemented using foreign functions written in Java and was tested based on the Twitter Streaming API. Instead of developing a new query language for stream data, AmosQL was extended to allow user to query JSON streams.

Currently, the Java foreign functions are only tested on processing *JSONObject* into Amos II objects of type *Record*, the interface to process *JSONArray* into Amos II objects of type *Vector* is primitively developed and need to be extended in the future.

References

1. Arvind Arasu, Brian Babcock, John ieslewicz, Keith Ito, Mayur Datar, Rajeev Motwani, Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom: STREAM: The Stanford Data Stream Management System, Technical Report. Stanford InfoLab, USA, March, 2004

2. A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, Oct. 2003. http://dbpubs.stanford.edu/pub/2003-67

3. D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun,J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, S. Zdonik Aurora: A Data Stream Management System, Brandeis University, Brown University, M.I.T.

4. Daniel Elin and Tore Risch: Amos II Java Interface, UDBL, Uppsala University, Sweden, Aug. 2000

5. http://apiwiki.twitter.com/Streaming-API-Documentation#ParsingResponses

6. http://apiwiki.twitter.com/Streaming-API-Documentation#statuses/filter

7. http://en.wikipedia.org/wiki/Twitter

8. http://hc.apache.org/httpclient-3.x/

9. http://hc.apache.org/httpclient-3.x/tutorial.html

10. http://infolab.stanford.edu/stream/

11. http://www.everydaytweet.com/2009/04/what-is-retweet-and-how-do-i-retweet.html

12. http://www.json.org/

13. http://www.json.org/example.html

14. Kjell Orsborn: Management of Product Data Using an ExtensibleObject-Oriented Query Language, the Sixth International Conference on Data and Knowledge Systems for Manufacturing and Engineering, DKSME '96, http://user.it.uu.se/~udbl/publ/dksme96.pdf

15. Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, Samuel Madden: XStream: a Signal-Oriented Data Stream Management System, Computer Science and Artificial Intelligence Laboratory, MIT, 32 Vassar St, Cambridge, MA, 02139, USA

16. Lukasz Golab and M. Tamer Ozsu: Issues in Data Stream Management, University of Waterloo, Canada, 2003

17. Ramez Elmasri and Shamkant B. Navathe: Fundamentals of Database Systems, Third Edition, Addison-Wesley, 2003.

18. Risch, T, Josifovski, V: ”Distributed Mediation using a Light-Weight OODBMS”, in 1st ECOOP Workshop on Object-Oriented Databases, Lisbon Portugal, June 1999

19. Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, Martin Sköld, and Erik Zeitler: Amos II Release 12 User's Manual, UDBL, Uppsala University, Sweden, March 29, 2005

20. T Risch, V Josifovski: Distributed Mediation by Object-Oriented Mediator Servers, To be published in “Concurrency – Practice and Experience”, J Wiley & Sons, 2001

21. Tore Risch, Vanja Josifovski, Timour Katchaounov: “AMOS II Concepts”,UDBL,UppsalaUniversity,Sweden, http://user.it.uu.se/~udbl/amos/doc/amos_concepts.html, 2000

Appendix

A. One example of public statuses received from Twitter Streaming API

{"text":"@deano_BondiLG ur pretty handy in the kitchen tho arent you?? I
remember seeing u and Whippet cooking up a storm at Karrinyup that day...",
"contributors":"null",
"geo":"null",
"in_reply_to_screen_name":"deano_BondiLG",
"truncated":"false",
"id":"11238489534",
"source":"web",
"favorited":"false",
"in_reply_to_status_id":"11237710943",
"in_reply_to_user_id":"118339349",
"created_at":"Mon Mar 29 07:47:34 +0000 2010",
"place":"null",
"user":{"location":"Perth, WA",
"statuses_count":"260",
"profile_background_tile":"true",
"lang":"en",
"profile_link_color":"FF0000",
"id":"28741593",
"following":"null",
"favourites_count":"60",
"protected":"false",
"profile_text_color":"3D1957",
"description":"",
"verified":"false",
"contributors_enabled":"false",
"profile_sidebar_border_color":"65B0DA",
"name":"Fiona Mackenzie",
"profile_background_color":"642D8B",
"created_at":"Sat Apr 04 04:42:22 +0000 2009",
"followers_count":"30",
"geo_enabled":"false",
"profile_background_image_url":"http://s.twimg.com/a/1269387398/images/themes/theme10/bg.gif",
"url":"null",
"utc_offset":"28800",
"time_zone":"Perth",
"notifications":"null",
"friends_count":"166",
"profile_sidebar_fill_color":"7AC3EE",
"screen_name":"finona_99",
"profile_image_url":"http://a3.twimg.com/profile_images/542783733/Dean_and_I_normal.jpg"},
"coordinates":"null"}

B. AmosQL code to implement the JSON stream wrapper

```
create function init(charstring url,charstring name,charstring pass)-> Integer as foreign "JAVA:JsonWrapper/initConn";

create function jsonwrapper(Integer id,Record r)->Bag of Object as foreign "JAVA:JsonWrapper/jsonWrapper";

create function jsonstream(Integer id,Record r)->stream of Object as streamof(jsonwrapper(id,r));

/*recieve JSON stream from twitter*/
create function trackstream(Integer i,charstring k,Integer stop)->Stream of Record
as section(jsonstream(i,Record({"track",k})),0,stop);
```

C. Java class implementation for Json stream wrapper

```
/**
 * JsonWrapper uses Httpclient to request resource from certain url.
 * Then processes the requested resource which is in JSON format to
 * Amos Vector data type
 * @author Bo Yang
 */
import callin.AmosException;
import callin.Connection;
import callin.Oid;
import callin.Scan;
import callin.Tuple;
import callout.CallContext;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collection;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.commons.httpclient.Credentials;
import org.apache.commons.httpclient.HttpClient;

import org.apache.commons.httpclient.HttpStatus;
import org.apache.commons.httpclient.HttpURL;
import org.apache.commons.httpclient.NameValuePair;
import org.apache.commons.httpclient.URIException;
import org.apache.commons.httpclient.UsernamePasswordCredentials;
import org.apache.commons.httpclient.auth.AuthScope;
import org.apache.commons.httpclient.methods.*;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import org.json.JSONTokener;

public class JsonWrapper {

    private AuthScope authScope;
    private HttpURL url = null;
    private String baseUrl = null;
    private String username = null;
    private String password = null;
    private static ArrayList<HttpClientUrl> arraylist = new ArrayList<HttpClientUrl>();
    private static PostMethod postMethod = null;

    /**
     * Extracts the host and post from the baseurl and constructs an
     * appropriate AuthScope for them for use with HttpClient
     */
    private AuthScope createAuthScope(String baseUrl) throws AmosException {
        AuthScope authscope = null;
        try {
            url = new HttpURL(baseUrl);
            authscope = new AuthScope(url.getHost(), url.getPort(), "realm");
        } catch (URIException ex) {
            Logger.getLogger(JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
            throw new AmosException(ex.getMessage());
        }
        return authscope;
    }

    /**
     * Process JSONObject to Amos Record
     */
    public Tuple processJson(CallContext cxt, JSONObject jsonobject, Tuple t) {
        JSONArray keys = jsonobject.names();
        for (int i = 0; i < keys.length(); i++) {
            try {
```

```
            //put the key of JSONObject into Record
            t.setElem(i * 2, keys.getString(i));

            //put the corresponding value of JSONObject into Record
            if (jsonobject.get(
                        keys.getString(i)).getClass().getName().equals("org.json.JSONObject"))
             {
                Scan s;
                Tuple temp1 = new Tuple(1);
                JSONObject temp = jsonobject.getJSONObject(keys.getString(i));
                temp1.setElem(0, processJson(
                                                cxt, temp, new Tuple(temp.names().length() * 2)));
                s = cxt.connection().callFunction(
                                                "vector.make_record->Record", temp1);
                t.setElem(i * 2 + 1, s.getRow().getOidElem(0));

             } else if (jsonobject.get(
                                keys.getString(i)).getClass().getName().equals("org.json.JSONArray"))
                {
                JSONArray temp = jsonobject.getJSONArray(keys.getString(i));
                t.setElem(i * 2 + 1, processArray(cxt, temp, new Tuple(temp.length())));
            } else {
                //System.out.println(jsonobject.get(keys.getString(i)).getClass().getName());
                t.setElem(i * 2 + 1, jsonobject.getString(keys.getString(i)));
            }
        } catch (AmosException ex) {
            Logger.getLogger(
                                    JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        } catch (JSONException ex) {
            Logger.getLogger(
                                    JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    return t;
}

/**
 * Process JSONArray to Amos Vector
 */
public Tuple processArray(CallContext cxt, JSONArray jsonarray, Tuple t) {
    for (int i = 0; i < jsonarray.length(); i++) {
        try {
            if (jsonarray.get(i).getClass().getName().equals("org.json.JSONObject")) {
                JSONObject temp = jsonarray.getJSONObject(i);
                t.setElem(i, processJson(
                                                cxt, temp, new Tuple(temp.names().length() * 2)));
            } else
                        if (jsonarray.get(i).getClass().getName().equals("org.json.JSONArray"))
                        {
                JSONArray temp = jsonarray.getJSONArray(i);
                t.setElem(i, processArray(cxt, temp, new Tuple(temp.length())));
            } else {
                t.setElem(i, jsonarray.getString(i));
            }

        } catch (AmosException ex) {
            Logger.getLogger(
                                JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        } catch (JSONException ex) {
            Logger.getLogger(
                                JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    return t;
}

/**
 * Initialize the httpclient by setting credentials and creating Authorization Scope.
 * Then add the httpclient and url into an arraylist
 */
```

```
public void initConn(final CallContext cxt, final Tuple tpl) throws AmosException {

    Boolean find = false;
    this.baseUrl = tpl.getStringElem(0);
    this.username = tpl.getStringElem(1);

    this.password = tpl.getStringElem(2);
    int status;
    authScope = createAuthScope(baseUrl);
    HttpClient httpClient = new HttpClient();
    Credentials creds = new UsernamePasswordCredentials(username, password);
    httpClient.getHttpConnectionManager().getParams().setSoTimeout(60000);
    httpClient.getState().setCredentials(authScope, creds);
    httpClient.getParams().setAuthenticationPreemptive(true);

    GetMethod get = new GetMethod(baseUrl);
    get.setDoAuthentication(true);

    try {
        status = httpClient.executeMethod(get);
        //System.out.println(status);
        if ((status != 200) && (status != 406)) {
            throw new AmosException(
                    "Got status " + get.getStatusText());
        }

    } catch (IOException ex) {
        Logger.getLogger(JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        throw new AmosException(ex.getMessage());
    } finally {

        get.releaseConnection();
    }

    if ((status == 200) || (status == 406)) {
        for (int i = 0; i < arraylist.size(); i++) {
            if ((arraylist.get(i) == null) && (!find)) {
                arraylist.remove(i);
                arraylist.add(i, new HttpClientUrl(httpClient, baseUrl));
                find = true;
                tpl.setElem(3, i);
                cxt.emit(tpl);
            }
        }

        if (!find) {
            arraylist.add(new HttpClientUrl(httpClient, baseUrl));
            tpl.setElem(3, arraylist.size() - 1);
            cxt.emit(tpl);
        }
    }
}

/**
 * Send a name value pair to url and process the recieved data
 */
public void jsonWrapper(final CallContext cxt, final Tuple tpl) throws AmosException {
    Scan s;
    Tuple temptpl = new Tuple(1);
    Tuple temptpl2 = new Tuple(1);
    Collection<NameValuePair> namevaluepair = new ArrayList<NameValuePair>();
    int snumber = tpl.getIntElem(0);

    if (arraylist.isEmpty()) {
        throw new AmosException("No connection initialized!");
    } else if ((snumber >= arraylist.size()) || (arraylist.get(snumber) == null)) {
        throw new AmosException("Connection is not defined or has been released");
    } else {

        HttpClient httpclient = arraylist.get(tpl.getIntElem(0)).httpClient;
    Oid oid = tpl.getOidElem(1);
    temptpl2.setElem(0, oid);
    s = cxt.connection().callFunction("Record.all_value->Object", temptpl2);
    while (!s.eos()) {
        Tuple row;
        String name;
        String value;
        row = s.getRow();
        name = row.getStringElem(0);
        s.nextRow();
        row = s.getRow();
        value = row.getStringElem(0);
        namevaluepair.add(new NameValuePair(name, value));
        s.nextRow();
    }

    try {
        postMethod = new PostMethod(arraylist.get(tpl.getIntElem(0)).baseurl);
        postMethod.setRequestBody(namevaluepair.toArray(
                                          new NameValuePair[namevaluepair.size()]));
        httpclient.executeMethod(postMethod);
        if (postMethod.getStatusCode() != HttpStatus.SC_OK) {

            throw new AmosException(
                    "Got status " + postMethod.getStatusText());
        }

        InputStream is = postMethod.getResponseBodyAsStream();
        if (is != null) {
            JSONTokener jsonTokener = new JSONTokener(
                    new InputStreamReader(is, "UTF-8"));
            String jsontype = jsonTokener.nextValue().getClass().getName();
            if (jsontype.equals("org.json.JSONObject")) {
                while (true) {
                   JSONObject jsonObject = new JSONObject(jsonTokener);
                   temptpl.setElem(0, processJson(cxt, jsonObject, new Tuple(
                                                       jsonObject.names().length() * 2)));
                    s = cxt.connection().callFunction(
                                              "vector.make_record->Record", temptpl);
                    tpl.setElem(2, s.getRow().getOidElem(0));
                    cxt.emit(tpl);
                }
            } else if (jsontype.equals("org.json.JSONArray")) {
                while (true) {
                    JSONArray jsonarray = new JSONArray(jsonTokener);
                    tpl.setElem(2, processArray(
                                              cxt, jsonarray, new Tuple(jsonarray.length())));
                    cxt.emit(tpl);
                }
            } else {
                throw new AmosException("Data is not in JSON format ");
            }
        }
    } catch (JSONException ex) {
        Logger.getLogger(
                        JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        throw new AmosException(ex.getMessage());
    } catch (IOException ex) {
        Logger.getLogger(
                        JsonWrapper.class.getName()).log(Level.SEVERE, null, ex);
        throw new AmosException(ex.getMessage());
    } finally {
        // Abort the method, otherwise releaseConnection() will
        // attempt to finish reading the never-ending response.
        // These methods do not throw exceptions.
        postMethod.abort();
        postMethod.releaseConnection();
    }
}
```

```
    }

    /**
     *Abort the method and finish reading the never-ending response
     */


    /**
     * Get the number of available connections
     */
    public void available(final CallContext cxt, final Tuple tpl) throws AmosException {
        if (arraylist.isEmpty()) {
            throw new AmosException("No connection available!");
        } else {
            Boolean find = false;
            for (int i = 0; i < arraylist.size(); i++) {
                if (arraylist.get(i) != null) {
                    find = true;
                    tpl.setElem(0, i);
                    cxt.emit(tpl);
                }
            }
            if (!find) {
                throw new AmosException("No connection available!");
            }
        }
    }

    public static void main(String argv[]) throws AmosException {
        Connection.initializeAmos(argv);
        Connection theConnection = new Connection("");
        theConnection.amosTopLoop("JavaAmos II"); // Enters the AmosQL top-loop

    }
}

class HttpClientUrl {

    public HttpClient httpClient = null;
    String baseurl;

    public HttpClientUrl(HttpClient httpclient, String Url) {
        this.httpClient = httpclient;
        this.baseurl = Url;
    }
}
```