Table of Contents

Chapter 6

6. High-performance design features, measurements and analysis

This chapter presents the common performance issues and high-performance design features in service-oriented, federated and interoperable GIS systems in which the interoperability is granted by structured data model. As the common data model, OGC [ogc] defined Geographic Markup Language (GML) [GML] is used. Developing a federated information system inspired us enhancing the whole system performance by applying novel parallel processing and caching techniques applied together in large scale interoperable information systems (see Chapter 6.3.2). In addition to this, we proposed some other innovative performance enhancement techniques (see Chapter 6.3.1) such as streaming data transfers, and enhanced parsing and rendering of semi-structured geo-data sets (GML). At the end of each chapter explaining these techniques, performance tests and analysis are provided.

The organization of the rest of the chapter is as follows. Chapter 1.1 summarizes and reviews the general performance issues of interoperable service-oriented GIS systems in which interoperability is granted by using XML-structured common data model and Web Services. Chapter 1.2 presents the limits of the ordinary GIS systems without having any performance enhancements which will be our comparison base for our proposed techniques and enhancements. Throughout the document, with the term "ordinary system" we mean a system built over naive approaches such as on-demand, single-threaded and no-caching systems. The last chapter (Chapter 6.3) explains our approaches to developing high performance GIS systems, and provides performance evaluations by comparing with the ordinary systems. We approach the performance issues from the two aspects. One is data-oriented and the other is federator-oriented. The data-oriented approaches deal with transferring large-sized XML structured data in common model, and high performance parsing and rendering algorithms. The federator-oriented approaches deal with the performance enhancement techniques based on data characteristics. For the infrequently changing archived data handling we propose pre-fetching technique (Chapter 6.3.2.1). On the other hand, for the frequently changing archived data, we propose a novel technique composed of client-based caching (Chapter 6.3.2.2) and parallel processing through query decomposition (Chapter 6.3.2.3).

6.1. General Performance Issues in Interoperable Service-oriented GIS

Performance issues in interoperable service-oriented GIS can be generalized into two groups:

- Issues regarding semi-structured data model (GML).

- Issues regarding domain specific data characteristics. In GIS, the data is described with location attribute defined in (x, y) coordinates. Based on the location value, the data is characterized as un-evenly distributed and variable sized. See *Figure 1*-b.

6.1.1. Using Semi-structured Data Model

Using semi-structured data model enables interoperability and inter-service communication. XML's emergence as the de facto standard for encoding tree-oriented, semi-structured data has brought significant interoperability and standardization benefits to distributed computing. On the other hand, performance has been still a persistent concern for large scale applications, because of the size issues and processing overheads [Lu2006]. The processing is detailed as parsing and differentiating (separating) the core-data from the attributes and other tags to create required application specific data formats.

Structured data representations enable adding some attributes and additional information (annotations) to the data. These attributes and additions are mostly due to the interoperability and security reasons. XML representations of data tend to be significantly larger than binary representations of the same data. The larger document size means that the greater bandwidth is required to transfer of data, as compared to the equivalent binary representations. The larger size often implies greater processing costs as well, since much of the overhead involved in communication processing is going to be based on the data volume.

There are two well-known and commonly-used paradigms for processing XML data, the Document Object Model (DOM) and the Simple API for XML (SAX). DOM builds a

complete object representation of the XML document in memory. This can be memory intensive for large documents, and entails making at least two passes through the data. SAX operates at one level lower. Rather than actually constructing a model in memory, it informs the application of elements through callbacks. This also requires at least two passes through the data. These are all expensive and resource (such as CPU and memory) consuming processes and they don't provide enough performance for the large scale applications.

In the document these issues are called data-oriented performance issues, and the proposed solution approaches are presented in Chapter 6.3.1

6.1.2. Though Data Characteristics and Attributes

The different domains have different data types having different characteristic to be handled. As an example, in GIS domain, which is our motivating domain, science applications need to manipulate geo-data. Geo-data is described with its location ((x, y) coordinates) on the earth. Based on the location attribute, geo-data is un-evenly distributed (such as human-population and temperature distributions) and variable sized. Because of these characteristics, it is not easy to implement some well-known performance enhancing techniques as applied in other science domains. Since it is not possible to know the work-load earlier, the classic load balancing algorithms do not work for the variable sized and unevenly distributed data. The work is decomposed into independent work pieces, and the work pieces are of highly variable sized. This issue is illustrated in Figure 1 for the case of using one-step-binary query partitioning based on the location attribute of the data. As it is illustrated in the figure, there are four worker nodes, and the worker node assigned to R2 gets the heaviest part of the total work, and

therefore the expected performance gain from usinf classic load balancing will not be obtained.

The geo-data is queried based on their attributes. Since all the data is described by their locations, in order to get the data sets falling in a specific region, the bounding box (bbox) values are used. The regions are defined in bboxes. A bbox defines a rectangular shape in a two-sim coordinate plane, and it is formulated as (minx, miny, maxx, maxy). For example, Figure 1 shows a region formulated in bbox value (a, b, c, d).

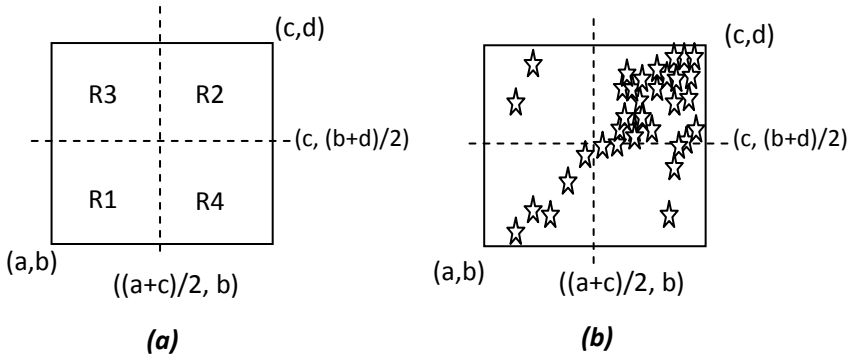

Figure 1: Unbalanced load sharing. Server assigned R2:"((a+b)/2, (b+d)/2), (c, d)" gets the most of the work.

These performance issues are dealt with in Chapter 6.3.2 which is called federator-oriented performance issues. We approach to the problem by keeping record for each client separately (we call it client-based caching) and utilizing locality and nearest neighborhood principles to share the work load to worker nodes as just as possible. Moreover, shares are created through query decomposition over bbox attribute of the main query.

In order to evaluate our proposed system design and performance enhancement techniques, we will be comparing the results with the baseline performance results given in the following chapter.

6.2. Ordinary GIS Systems Performance (baseline test results with naïve approaches)

In order to solve data and service heterogeneities for the GIS computation and data services OGC and ISO/TC-211 standards are used. These standards recommend using structured common data model called GML for the representation of location based geo-data. The standard bodies aim is to make the geographic information and services neutral and available across any network, application, or platform. Currently the two major geospatial standards organizations are the Open Geospatial Consortium (OGC) and the Technical Committee tasked by the International Standards Organization (ISO/TC211).

With the ordinary system we mean a GIS developed with widely used technique without using any novel advanced techniques to handle the data. Most of the implementations are based on single-threaded and on-demand processing. Deegree project [deegree] and Minnesota Map Server [minmapserv] can be given as sample projects. In order to compare and contrast our novel approaches to the ordinary systems approaches, we tested and presented their performance results at Table 1 and *Figure 3*.

Figure 2 shows the test setup for the system. This figure also illustrates a simple GIS system with major service components and data flow from the originating data sources to the end-users.

Figure 2:The ordinary system test set-up. Any-data is converted to common structured data (GML) and rendered as map images.

Over this setup system response time is measured and displayed in Table 1 and *Figure 3*. The average response times shown in the figures include times for querying, transforming, rendering and displaying spatial data. The average response time is formulated as below:

$$time_{(measured)} = time_{(result\ is\ displayed)} - time_{(client\ makes\ request)}.$$

Moreover, ($\boldsymbol{time_{(measured)}}$) can be further detailed as below (also see Figure 2):

- **[**$\boldsymbol{time_{(client\ makes\ request)}}$**.]** Client makes requests through the interactive smart map tools.
- WMS parse and render requests and define set of actions required based on the requests and its capabilities file.
- WMS Creates map images (from the returned datasets) and returns them to the clients:
    - Defines the set of WFSs [WFS] and other WMSs [WMS] to communicate with to build the response in accordance with its capability file and client provided parameters.
    - Creates requests for WFSs and other WSMs

- Invokes WFSs' *getFeature* Web Service interfaces for vector data encoded in Geographic Markup language (GML) [GML].
- Invokes other WMSs *getMap* Web Services for raster data rendered in map images
- Transferring GML data (feature collections) from WFS and WMS
- Parsing and rendering returned GML data sets
- Aggregating and overlaying layers according to the request and capability file.
- Sending the map images to the WMS Client.

- **[$time_{(map\ is\ displayed)}$]** Client shows the returned maps on his browser

Table 1: The round-trip times (or response times) of the ordinary system.

Data Size KB	Average Response Times (msec)	Log of Avg response msec	Standard Deviation
1	2,375.24	3.38	152.40
10	2,578.69	3.41	252.49
100	7,973.16	3.90	374.12
200	13,612.78	4.13	417.19
500	30,868.52	4.49	482.83
1000	59,635.69	4.78	343.76
5000	288,594.12	5.46	333.07

(a) (b)

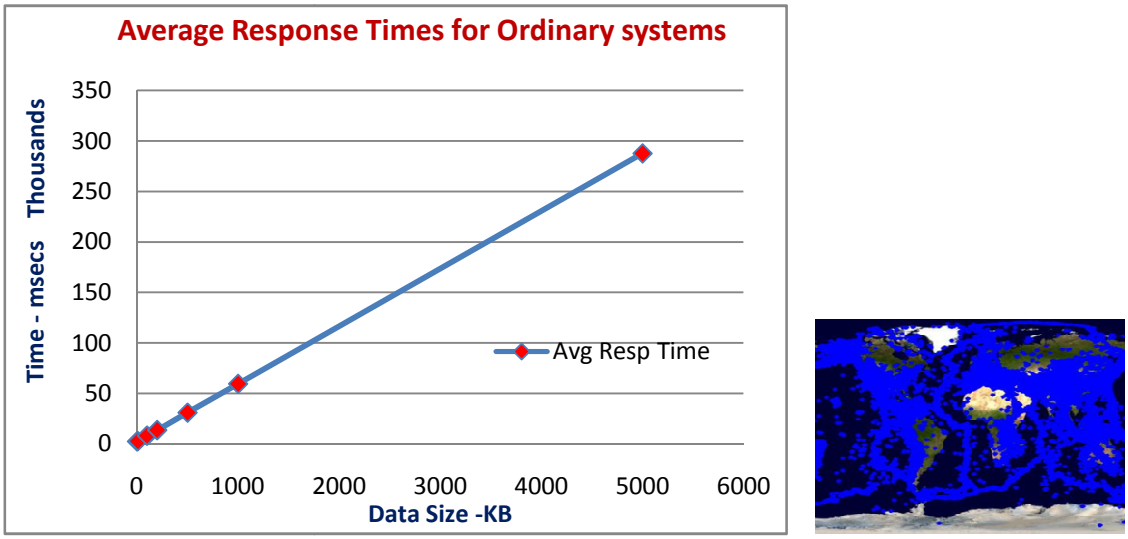

Figure 3: (a) Performance result of the ordinary system. (b) Sample output-seismic data is plotted over NASA Satellite map images

In order to be able to make more reasonable comparisons, we adjusted the timing values given in Table 1 by taking their logarithmic values and plotted them in Figure 4.

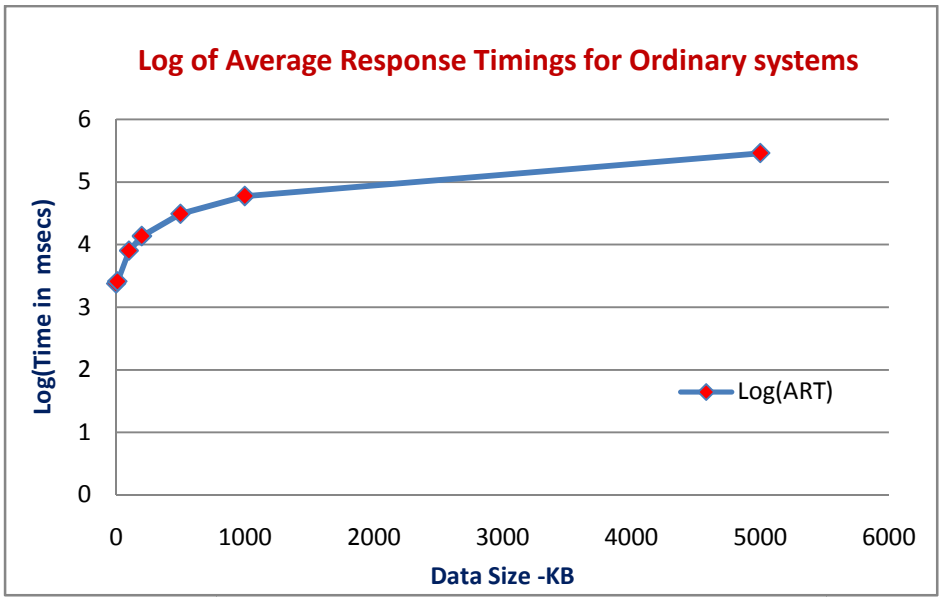

Figure 4: Adjusted performance values over *Figure 3* for the ordinary systems.

This performance results teach us valuable lessons in terms of the capabilities and limits of the general distributed and interoperable GIS systems. From the figure we draw following conclusions. First, for the small data payloads (less than 500KB) the response time is acceptable. However for larger data payloads the performance gets worse and the response time gets relatively longer. On the other hand, scientific applications require handling (transferring, parsing, rendering and displaying) large scale data.

From our experience we saw that depending on the total data size, over %90 of the $time_{(measured)}$ comes from the step called "transferring GML data (feature collections) from WFS and WMS". Because of that, even if we use the most efficient and fast parsing and rendering algorithms (such as using pull parsing or application specific XPath querying), it won't improve performance very much as long as the data transfer time still stays that much high as shown in the *Figure 3*.

6.3. High Performance Design and Evaluation of the Proposed System

Our approaches to the performance issues are grouped into two. The first group of approaches deals with the general performance issues result from using semi-structured data encodings (such as GML), and large size data exchange, parsing and rendering (Chapter 1.3.1). The second group of approaches is regarding the federator oriented design and techniques to enhance the overall system performance (Chapter 1.3.2).

6.3.1. Data-oriented Enhancement Approaches

Distributed GIS systems typically handle a large volume of datasets. Therefore the transmission, processing and visualization/rendering techniques need to be responsive to provide quick, interactive feedback. There are some characteristics of GIS services and data that make it difficult to design distributed GIS with satisfactory performance. One of

them is that GIS services often transmit large resulting datasets such as structured data, images, or large files in tabular-matrix formats.

In order to provide interoperability and extensibility we use common data format represented and formulated in XML. This degrades the performance even worse for large scale applications. The major hurdle of the proposed federated GIS framework is encoding, transferring and rendering the data in common data model. In the following two sub-sections we present our approaches to these issues. One is regarding large scale structured data transfer (Chapter 6.3.1.1) and other is regarding the large scale data parsing (Chapter 6.3.1.2).

6.3.1.1. Streaming Data Transfer

Our experience shows that although we can easily integrate several GIS services into complex tasks by using Web Services, providing high-rate transportation capabilities for large amounts of data remains a problem because the pure Web Services implementations rely on SOAP [Donbox] messages exchanged over HTTP. This conclusion has led us to an investigation of topic-based publish-subscribe messaging systems for exchanging SOAP messages and data payload between Web Services. We have used NaradaBrokering [Pallickara2003] which provides several useful features besides streaming data transport such as reliable delivery, ability to choose alternate transport protocols, security and recovery from network failures.

Naradabrokering is a message oriented middleware (MoM) [Tran] system which facilitates communications between entities through the exchange of messages. This also allows us to receive individual results and publish them to the messaging substrate instead of waiting for whole result set to be returned.

In case of transferring the GML result set in the form of string causes some problems when the GML is larger than some amount of size (500KB see *Figure 3*-a). Since the WFS returns the resulting XML document as an <xsd:string>, this has to be constructed in memory and the size will depend on several parameters such as the system configuration and memory allocated to the Java Virtual Machine etc. Consequently there will be a limit on the size of the returned XML documents. For these reasons we have investigated alternative ways for data transport and, researched the use of topic based publish-subscribe messaging systems for streaming the data. Our research on NaradaBrokering shows that it can be used to stream large amount of data between nodes without significant overhead. Additional capabilities such as reliable messaging and support for different transport protocols already inherent in NaradaBrokering show that it is a powerful yet easy to integrate messaging infrastructure. For these reasons we have developed a novel Web Map Service and Web Feature Service that integrate OGC specifications with Web Service-SOAP [Donbox] calls and NaradaBrokering messaging system. Architecture is shown in Figure 5.

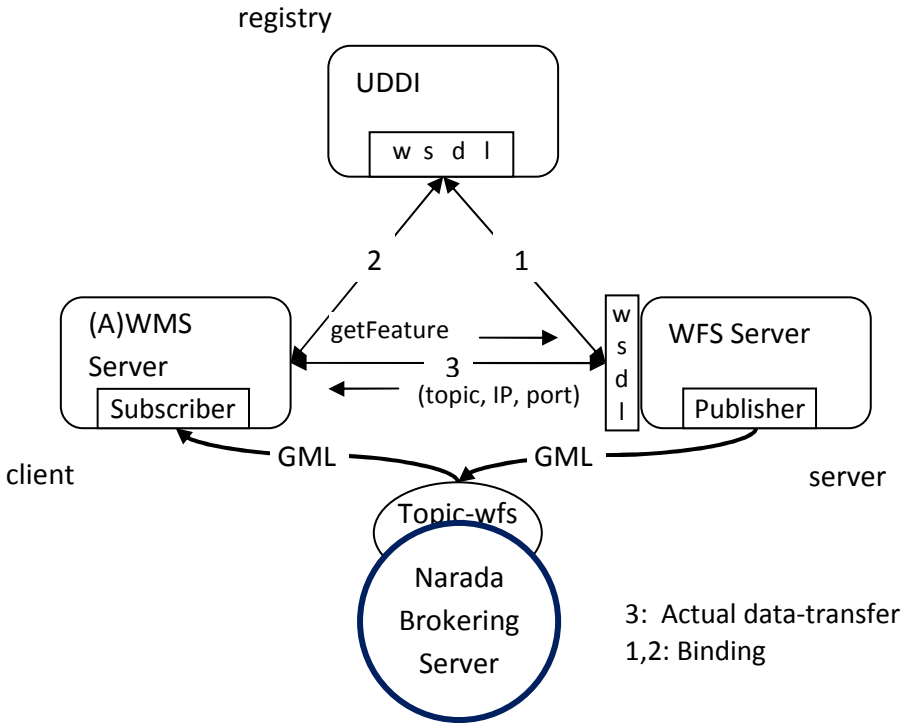

Figure 5: Streaming data transfer using Naradabrokering publish-subscribe topic based messaging middleware.

Connection lines 1 and 2, and UDDI (Universal Description, Discovery and Integration) [uddi] service are displayed in the figure for showing classic publish-find-bind triangle of the Web Service based Service Oriented architecture. We don't go into details of these interactions and UDDI registry service in this document but these can be summarized as following. WFS services publish their existence and service providing with their WSDL service description files (line-1). Clients (such as WMS) find appropriate WFS by searching UDDI registries (line-2). After finding appropriate service, clients are bind to that service by creating their client stubs. Instead of using lines 1 and 2, clients can also directly communicate with the services if they know the service's WSDL file earlier.

In case of streaming through Naradabrokering, the clients make the requests with standard SOAP messages (line-3) but for retrieving the results a NaradaBrokering subscriber class is used. Through first request to Web Service (called *getFeature)*, WMS gets the topic (publish-subscribe for a specific data), IP and port to which WFS streams requested data. Second request is done by *NaradaBrokering* Subscriber. In this way, even in the case of that the whole data is not received. WMS can draw the map image with the returned data. This depends on the WMS's internal implementation.

Table 2 gives a comparison of the streaming and non-streaming data access approaches for the different data sizes. These values are obtained by applying the proposed framework on Pattern Informatics (PI) [patterninfo] geo-science application using earthquake seismic data records. These are GML data access times including query conversion at WFS, result set conversion from database to GML and transfer times from WFS to federator or WMS.

Table 2: Data access times (from federator or WMS) while using (1) streaming and (2)non-streaming data transfer techniques.

	Streaming			Non-Streaming		
Data Size (KB)	Average Time for Streaming Transfer	Average Response Time	Standard deviation	Average Time Non-Streaming	Average Response Time	Standard deviation
10	31.3	2425	38	1518.8	3912.5	77
30	100	2661	27	1356.1	3917.1	38
100	320.1	2945	50	1473.8	4098.7	71

300	826.7	3405	48	1835.7	4414	39
1000	2414.2	4570	360	3506.8	5662.6	31

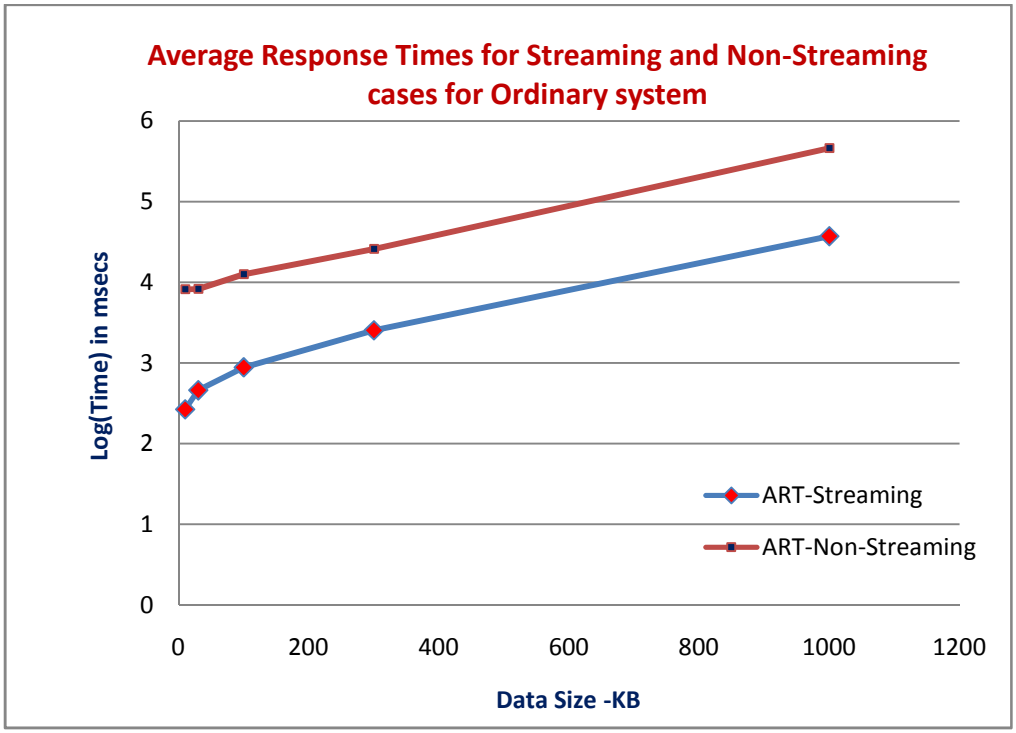

Figure 6: Comparisons of Streaming vs. Non-Streaming data response timings from source to federator or WMS.

We can deduce from the table that for the larger data sets when using streaming our gain is about 25%. But for the smaller data sets this gain becomes about 40% which is mainly because in the traditional Web Services the SOAP message has to be created, transported and decoded the same way for all message sizes which introduces significant overhead.

6.3.1.2. Pull Parsing and Application Specific Rendering

Proposed system includes data rendering/filtering tasks assigned to Web-based Map Services to create comprehensible data representations derived from the semi-structured common data (GML). These comprehensible representations are called maps. Regarding the rendering of large GML data and creating map images we use parsers.

There are three general parsing techniques proposed for processing XML structured data. These are document model, push model and pull model. There are also other hybrid alternatives built on these main approaches. In order to process data in XML structured common data model we use pull parsing technique.

Pull parsing, as exemplified by the XML Pull Parser [Alexander], is an efficient paradigm similar to SAX in that it does not build a complete object model in memory. It differs in that the tags and content are returned directly to the application from calls to the parser, rather than indirectly in the form of callbacks. The pull approach of this parsing model results in a very small memory footprint (no document state maintenance required – compared to DOM), and very fast processing (fewer unnecessary event callbacks - compared to SAX).

Pull parser only parses what is asked for by the application rather than passing all events up to the client application as SAX parsing does. You can see the article where pull parsing is compared with other leading Java based XML parsing implementations [Sosnoski].

Pull parsing does not provide any support for validation. This is the main reason that it is faster than its competitors. Since all the services are OGC compatible and created in Web

Service principles, validation is not necessarily needed. In OGC, services describe themselves by capability document and servers know each other by exchanging these document. If you are sure that data is valid (as in our case), or if the validation errors are not catastrophic to your system, or you can trust validity of the capabilities document of the server you are in contact, then using XML Pull Parsing gives the highest performance results. For example in communication between WFS and WMS, since it is known that WFS provides feature data in OGC's GML format [GML], it is very advantageous skipping validation and using "pull parsing".

For application specific comparison of Pull parsing and DOM see Table 3 and Figure 7. The performance values are measured in milliseconds and data sizes are in MBs. Performance test is done with 1GB allocated JAVA Virtual Machine. Dashed-line values in the table represent not-enough memory exceptions thrown. The figure illustrates the timing values for the data size till 100MB. Above this threshold value for the Virtual Machine allocated 1GB memory, DOM become useless.

Test case: For the XML data we use earthquake seismic data records encoded in GML. Each earthquake seismic record has some attributes and some geometry elements. In our tests we will parse the GML data in XML documents and extract the geometry elements. In case of DOM, parsing and extraction are done separate as it is shown in two columns in Table 3. In case of pull parsing, geometry data is extracted from GML with parsing and extraction applied all together.

Table 3: The performance values of DOM and Pull parsing (Xpp) over GML data. Dashed-line values imply memory exception.

	DOM (dom4j)					Pull Parsing	
Data Size (MB)	Average Parsing Time	StdDev	Avg Render Time	Average Total Time Pars/Rend	StdDev	Average Total Time	StdDev
0.001	394.29	18.68	75	469	21.32	15.59	0.87
0.01	429.32	36.46	65	494	20.87	72.81	7.41
0.1	484.41	18.18	141	625	23.04	183.06	23.25
1	663.94	18.09	96	760	31.58	270.47	40.09
5	1,247.00	36.74	175	1,422	47.66	671.74	76.05
10	2,126.63	20.73	1,430	3,557	61.51	1,025.67	51.49
100	1,159,614	13,122.6	----	----	----	7,059.72	93.16
150	----	----	----	----	----	11,047.89	107.80
200	----	----	----	----	----	14,949.12	253.15

As it is mentioned dashed lines in Table 3 represent memory exceptions. It means system does not have enough memory for completing its work. Since there is extreme performance difference between using DOM and pull parsing techniques, we plot their logarithmic values to illustrate the performance gains of using pull parsing more clearly.

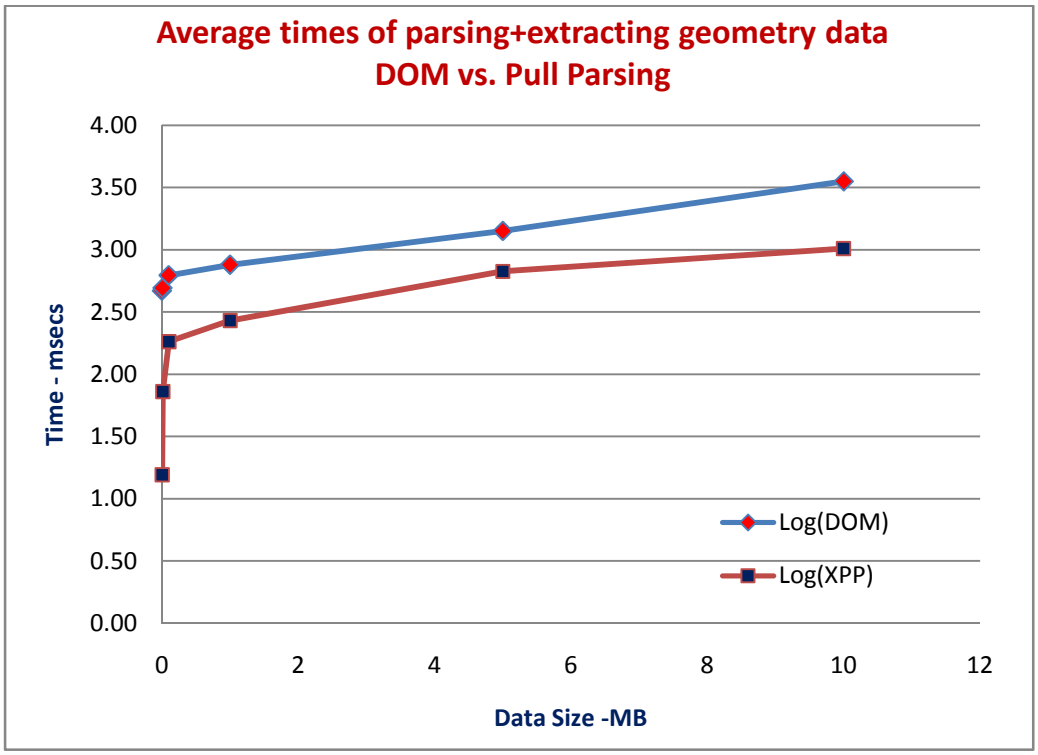

Figure 7: Performance comparison of two XML data processors, pull parsing and Document Object Model by using dom4j.

6.3.1.3. Overall Performance Evaluations over data-oriented performance enhancement approaches

This chapter presents overall performance gains obtained by applying data-oriented performance enhancement techniques mentioned in previous chapters (Chapter 6.3.1.1 and Chapter 6.3.1.2). We also compared the performance results with the baseline performance results given in *Figure 3*.

We again use the system test set-up shown in Figure 2.

Table 4: The performance results in average timings.

	Average Timings				
Data KB	Data Capturing	Map Rendering	Total time Map Creation*	map images' transfer time	Response time for end-users**
10	797.85	927.35	1741.17	61.88	1808.13
100	1384.86	1168.29	2567.35	62.22	2635.46
500	3770.16	1153.96	4934.94	60.15	5001.29
1000	6794.94	1360.41	8155.35	68.38	8225.73
5000	31237.41	2116.12	33350.80	70.26	33419.31
10000	61777.20	2675.87	64441.96	62.15	64506.78

** Total time for map creation = WFS to WMS data capturing + Map Rendering.*
*** Response time for end-users = Total time for map creation at WMS + map images transfer time to end user*

Table 5: The standard deviation values for the average timings given in Table 4

	Standard Deviations				
Data KB	Data Capturing	Map Rendering	Total time Map Creation	map images' transfer time	Response time for end-users
10	48.39	123.29	132.36	26.33	140.32
100	73.86	383.61	384.61	21.90	313.48
500	80.81	230.33	234.03	20.74	238.94
1000	93.24	207.60	199.49	24.59	200.27
5000	211.45	346.06	432.43	22.19	394.48

10000	152.54	252.97	279.04	18.64	283.24

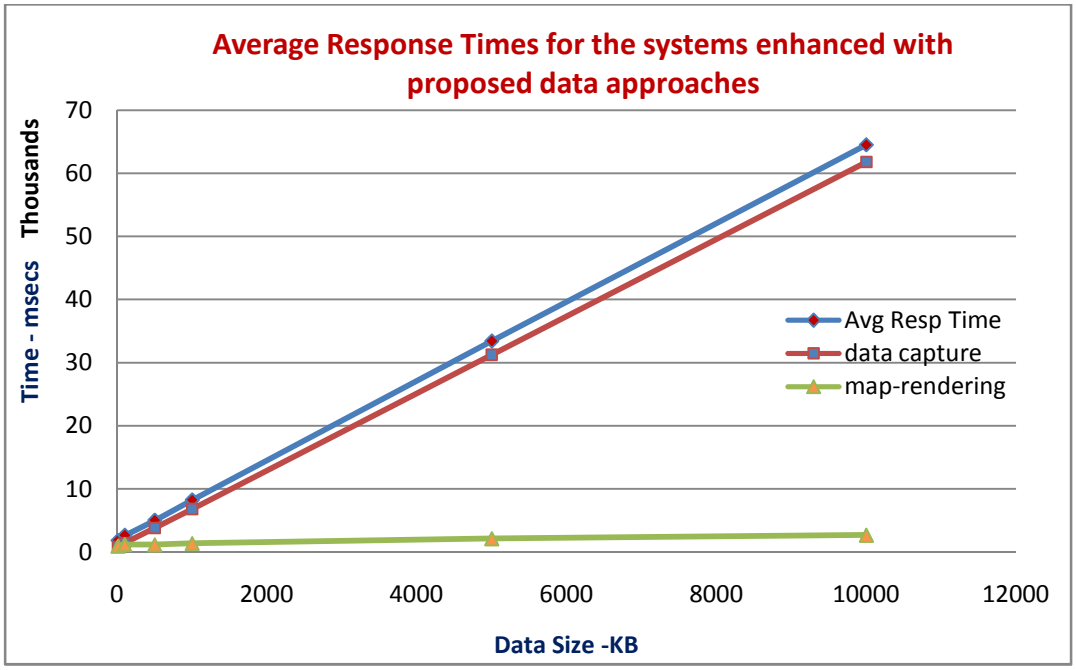

Figure 8: Average response, data capturing and map rendering timings for different data sizes. The values are obtained over the enhanced system with the proposed data-oriented techniques.

Table 6: The comparison of average response times: Enhanced systems vs. naive systems.

	Naïve Approaches		Enhanced with proposed data approaches	
Data KB	Response Time (msec)	Standard Deviation	Response Time (msec)	Standard Deviation
10	2578.69	252.49	1808.13	140.32
100	7973.16	374.12	2635.46	313.48
500	30868.52	482.83	5001.29	238.94
1000	59635.69	343.76	8225.73	200.27

5000	288594.12	333.07	33419.31	394.48
10000	574825.16	836.46	64506.78	283.24

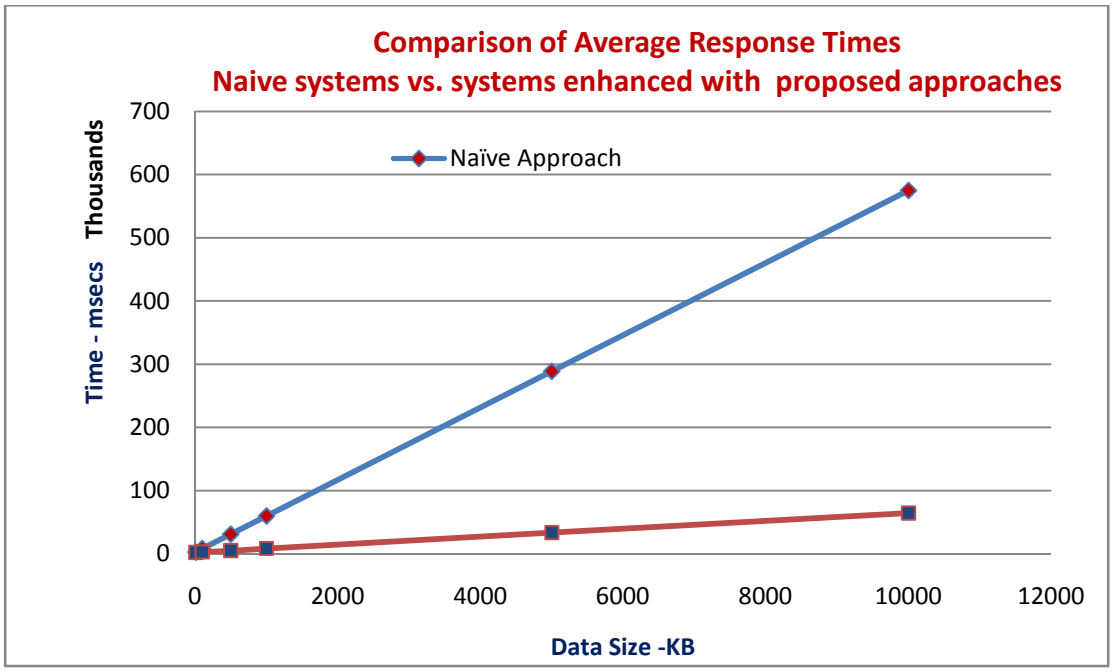

Figure 9: The comparison of average response times: Naïve systems vs. enhanced systems with the proposed data-oriented performance enhancement techniques (Chapter 6.3.1.1 and Chapter 6.3.1.2).

We still need to improve the system performance to make it applicable to high performance GIS applications requiring quick response times such as early warning systems and crisis management. In order to improve the performance further, we propose federator-oriented performance enhancement techniques in the following chapter.

6.3.2. Federator-oriented Performance Enhancement Approaches

The federator in the proposed federated GIS system inherently enables load balancing and parallel processing and this helps with enhancing the overall system performance.

This chapter presents the techniques and system design to develop high performance federated GIS system through the federator.

The system design changes depending on the characteristics of the data application use. For the infrequently changing data (static archived data) we propose pre-fetching (Chapter 6.3.2.1) technique. For the frequently changing data (similar to the real-time data) we propose a novel technique composed of client-based caching and parallel processing through query decomposition (in Chapter 6.3.2).

In summary, pre-fetching is purely for overcoming the natural bandwidth problem, caching helps the system with preventing to redo the jobs of querying and rendering, and parallel processing helps with workload sharing and parallel job run. Depending on the data characteristics, federator uses only one or the combination of these techniques. These techniques will be explained in the following sections with their performance evaluations and analysis.

6.3.2.1. Pre-Fetching

In the proposed integration framework we deal with the archived data in GML format. Archived data does not change often. Therefore, it is not reasonable transferring and rendering the same data again and again for every request coming from the different or even the same users. In order to solve this problem we propose pre-fetching. Pre-fetching is used to overcome the performance degradation of transferring large sized data from source (database) to destination. It also indirectly enables getting rid of the data transformation overhead at WFS. As it is mentioned before, WFS transform any-data kept in databases into common data model (GML) every time it gets a request.

Pre-fetching is briefly defined as getting the data before it is needed. We accomplish the pre-fetching by the data transfer technique explained in Section 6.3.1.1. The general architecture for the pre-fetching is shown at Figure 10. A performance result of the pre-fetching and comparisons to the on-demand fetching techniques are displayed in Figure 11 and Figure 12 respectively. Since pre-fetching is independent of the real-time application and run in an asynchronous manner, it does not degrade the proposed framework's overall performance. It's running times defined by the periodicity parameter of the Pre-fetching module (PM) (see Figure 10).

The OGC's standard WMS and WFS specifications are based on HTTP Get/Post methods, but this type of services have several limitations such as the amount of data that can be transported, the rate of the data transportation, and the difficulty of orchestrating multiple services for more complex tasks. Web Services help us overcome some of these problems by providing standard interfaces to the tools and applications we develop.

As in the proposed data exchange framework defined in Section 6.3.1.1, the pre-fetching module make the requests with standard SOAP messages but for retrieving the results a NaradaBrokering subscriber class is used. Through the "*getFeature"* interface of WFS Web Services, pre-fetching module gets the topic name (publish-subscribe for a specific data), IP and port on which WFS streams the requested data. Second request is done by NaradaBrokering Subscriber using the returned parameters. GML data is provided by streaming WFS (implemented by G. Aydin) [Vretanos]. It uses standard SOAP messages for receiving queries from the clients; however, the query results are published (streamed) to a NaradaBrokering topic as they become available. In order to do that, we define the "task" and "timer". Task defines pre-fetching job, and timer defines the

running periodicity of the task. Different data might have different periodicities set. Pre-fetching is done over the critical data. The critical data is the GML data affects the performance because of their sizes.

There will be two separate locations for the pre-fetched data. One is temporary into which pre-fetched data is stored. Another is stable which will be used for serving the clients' requests. Even if the system is busy with the pre-fetching job, it keeps itself up and running for the clients by using the stable storage. When the data transfer is done to the temporary location, all the data at that location will be moved to stable location. Reading and writing the data files at the stable locations will be synchronized to keep the data files consistent. This cycle is repeated at some time intervals pre-defined by periodicity parameter of Pre-fetching Module (PM).

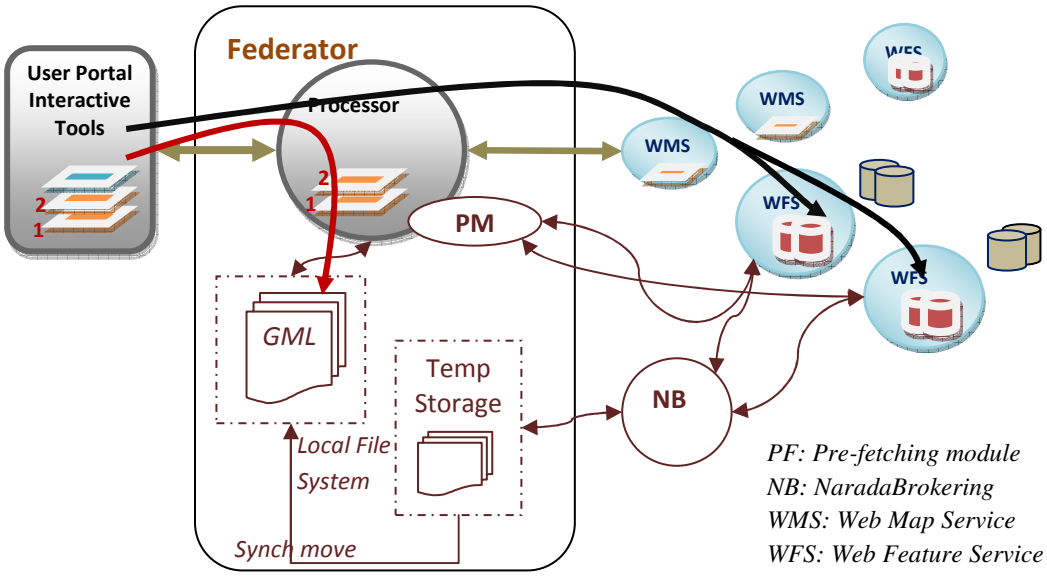

Figure 10: Pre-fetching architecture embedded to the federated GIS system

In order for the pre-fetching algorithm to work properly, pre-fetching module fetches the data as a whole; no constraint should be defined in the query. On the other hand, the requests from clients contain some query constraints. These queries and their constraints are handled at the A WMS side. Queries are processed by using parser techniques and XPATH queries over the pre-fetched data.

6.3.2.1.1. Fetching module (PM)

The pre-fetching module (PM) is composed of two components. One is “timer” defining the periodicity that PF will be running, and other is “task” defining what to do. The periodicity should not be less than the time to transfer one set of critical data. Assigning a periodicity at PM is the most critical task. This is defined under the considerations of data characteristics and developer’s experience on the domain specific application.

Since the system is developed in JAVA, we use Timer and TaskTimer JAVA class libraries to implement the routinely running pre-fetching module.

Here is the “task” defined in a pseudo code:

```
public void pseudo_TASK() {
    Vector CDMdataList = new Vector();
    CDMdataList = getListPerformanceCritical_GMLDataNames();
    String tempDatastore = applpath + "/prefetchedData";
    String stableDatastore = applpath + "/prefetchedDataUsed";

    //Fetching all the data in CDM format (GML) - with NB
    fd.FetchDataWithStreaming( NBip,NBport,NBtopic,
                    wfs_address,tempDatastore,CDMdataList );

    //After pre-fetching is done move the data to stable storage
```

```
        fd.moveData(tempDatastore, stableDatastore);
    }
```

We also define timer determining the periodicity of task to run. The below sample code sets the periodicity of "task" defined above to 3 days. It means PF will be running once every three days.

```
        Timer timer = new Timer();
        timer.schedule(task, 0, 40000);
```

Timer class schedules the specified task for repeated fixed-delay execution, beginning after the specified delay. Subsequent executions take place at approximately regular intervals separated by the specified period.

There are two concerns in developing an efficient pre-fetching architecture. First one is limited storage capacity for a node. The size of the pre-fetched data is constrained by local node's storage capacity. Second one is regarding the pre-fetched data characteristics. Some archived data is updated so often that they look like real-time data. In that case, pre-fetching becomes unfeasible and cannot be benefited. For this type of data (archived but updated frequently), we propose a novel parallel processing approach applied together with the caching (see Chapter 6.3.2.3).

6.3.2.1.2. Performance Evaluation

We test the proposed pre-fetching technique over the proposed federated GIS system by using real-world Pattern Informatics (PI) geo-science applications (see Figure 10). PI is an earthquake forecasting application and uses archived earthquake seismic records stored at WFS as feature collections encoded in GML (XML encoded structured data model for geo-data).

We basically test the system as illustrated in Figure 10. Red-curve (short) illustrates the round-trip path for the pre-fetching and black-curve (long) illustrates the round-trip path for the on-demand fetching. For the simplicity we will be using only one critical data to apply pre-fetching.

In summary, we give the performance results for the proposed pre-fetching approach and compare it with the ordinary on-demand fetching approach in Figure 12 and Table 8. In case of on-demand fetching approach, one end is database and other end is user browser (see the black (dark)-curve in Figure 10). Performance results show the response times.

Table 7: Performance results for the response times when the pre-fetched data is used.

GML Data Size MB	Average Processing	StdDev	Average Transfer	StdDev	Average Response	StdDev
0.01	19,215.60	477.71	46.30	15.39	19,261.90	481.57
0.1	19,040.74	670.65	71.57	29.74	19,112.30	673.69
0.5	19,191.24	630.50	31.24	8.30	19,222.48	631.35
1	19,387.64	307.45	39.84	10.01	19,427.48	305.94
5	20,107.54	514.46	38.46	10.66	20,146.00	516.50
10	20,113.19	548.52	52.71	27.13	20,165.90	546.53
50	22,830.33	505.86	52.19	15.88	22,882.52	509.98
100	22,934.52	598.25	55.90	12.66	23,990.43	603.59

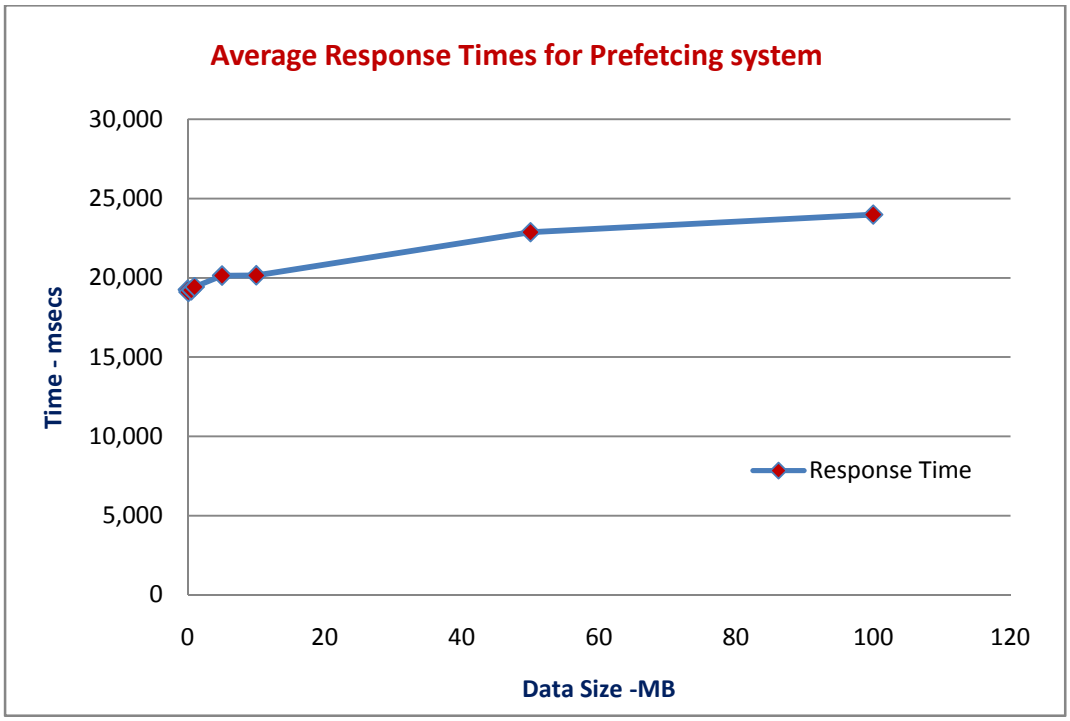

Figure 11: Performance of the pre-fetching technique

Table 8: Comparison of the pre-fetching (Figure 11) and ordinary (on-demand fetching) techniques

Data Size MB	Average Response Pre-fetching	StdDev	Average Response On-demand	StdDev
0.01	19,261.90	481.57	1,808.13	140.32
0.1	19,112.30	673.69	2,635.46	313.48
0.5	19,222.48	631.35	5,001.29	238.94
1	19,427.48	305.94	8,225.73	200.27
5	20,146.00	516.50	33,419.31	394.48
10	20,165.90	546.53	64,506.78	283.24
50	22,882.52	509.98	316,906.00	623.08
100	23,990.43	603.59	643,344.00	548.65

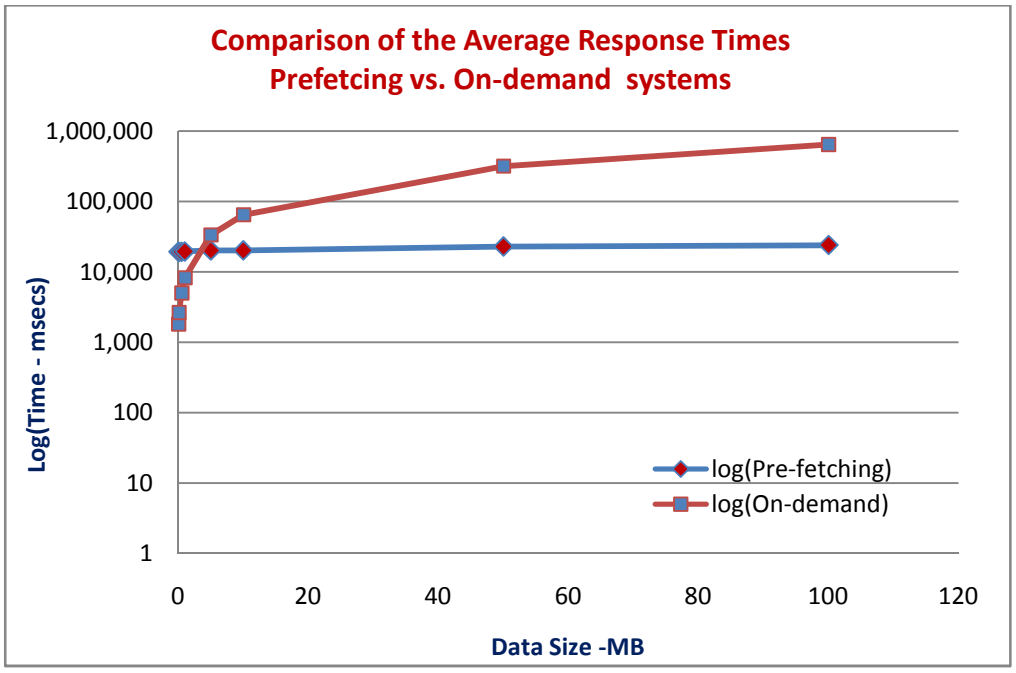

Figure 12: Performance comparison of the map rendering in the proposed GIS system with pre-fetching and ordinary ways.

As it is expected the pre-fetching increased the performance and responsiveness of the system for accessing, querying and rendering archived data. Compared to on-demand fetching (ordinary), pre-fetching removes the times spent on conversion (from database to GML at WFS side) and transferring GML data. In case of cascaded data (going through multiple chained services to access the original data source), performance gains even becomes much larger. Furthermore, the higher the data size, the higher the performance gains.

Our criterion for selecting the technique to apply depends on two measurements. One is the minimum time required to fetch a whole critical data from the source and another is the time periodicity in which data is updated in its storage. If the data changes less than a time periods in which whole critical data is fetched, then the data is called frequently changing.

6.3.2.2. Client/Session-based Dynamic Caching

We allocate separate chunk of caching area for the clients and each client is served from its own allocated area. Client's cache is updated with the data used for serving that client's last query. Server differentiates the clients based on their IDs defined in the request.

In this context, we use client and session interchangeably. One client might have more than one session by assigning different IDs to his messages to the server. For example, when event-based interactive mapping tools are used, those IDs are assigned automatically whenever user opens a new browser.

We introduced this novel idea for performance reasons. It removes the repeated jobs and helps efficient load balancing over the un-predicted workload by utilizing the locality [Denning] and nearest neighborhood [Belur] principles. It also helps us finding out the best efficient number of partitions for parallel processing and reducing the overhead timings for handling unnecessary number of partitions. Locality principle in this context is explained as following. If a region has a high volume of data, then the regions in close neighborhood also expected to have high volume of data. The simplest example to give is the distribution of human population data across the earth. The urban areas have higher human population than the rural areas, and oceans (2/3 of the world) have no human populations etc.

For large scale applications it might be impossible to cache whole data at intermediary servers to lower the response times. Furthermore, keeping data at different places force application developers to be more careful to keep the data consistent. It also brings maintenance and handling costs to the service administrators. Instead of doing this, we

propose a selective client-based dynamic caching. In the following chapter we explain the architectural details about how to develop such a framework.

6.3.2.2.1. Architectural Details

Architecture is based on recently used data sets and clients requesting them. The research issues this chapter deals with are summarized as (1) how server differentiate the clients and (2) what to cache and how to cache.

What to cache: Maps are composed of multiple layers and each layer is created from different data set such as satellite map layer, state boundaries layer and earthquake-seismic layer. The proposed caching is applied to the selected layers. These layers are defined as critical in server's properties file.

How to cache: each critical data is cached in the common data model (GML format) instead of ready to use image tiles. The reason behind this is that the proposed GIS framework allows attribute-based querying/display and data-mining. It is not just for displaying based on location attribute. In order to accomplish this, the data/layer needs to be cached with its geometry and non-geometry elements together with the core data. By doing this, even if client changes its queries in terms of attributes system utilizes the cached data as long as queries and cached data bboxes are intersects.

For each separate session (differentiated by their IDs defined in the request message), there will be separate set of cached data. Cached data is upgraded at every request from the same session.

Since the proposed federated GIS is interacted through browser-based interactive decision making tools over the integrated data views, the remaining of the chapter first

gives the details about how to set browser-based session ID to the SOAP message and forward it to the server, and then how to keep track of separate clients' session information at the server.

The proposed interactive event-based client tools are developed in Apache Tomcat [apache] Servlet container and pages are developed with Servlet and Java Server Pages (JSP). JSP defines session ID whenever a user opens a page to interact with the federated GIS system. A session is normally stored in a cookie which is available to all windows in the browser. The system access this ID by *session.getId()*. This returns a string unique user ID (uuid) which can be used for application specific purposes.

Whenever federated GIS client interacts with the system through federator, it sets its browser's session ID to the header of its SOAP messages sent to the Web Service. All the requests coming from the same browser has same session ID. Session IDs are created when the browser is opened and kept same until it is closed. Each browser has a separate and unique session ID. By setting this session ID to the header of SOAP messages federator can distinguish what client (browser) makes the requests and check its cached data and session information stored before.

Here is the pseudo code briefly explaining the steps:

```
WMSServicesSoapBindingStub binding;
binding = (WMSServicesSoapBindingStub)
    new WMServiceLocator().getWMSServices(newURL( service_address));
String sessionID = session.getid();
String channel_name = "WMS_getMap_Request";
//Add SessionID to the SOAP message's header
```

```
binding.setHeader(service_address, channel_name, sessionID);
//See Appendix xx for the sample GetMap request
Object value = binding.getComprehensibleData(getMap_request);
```

Whenever a user access the system through the same browser its session number will be the same and federator keeps its local data and actions in the system differentiated based on its unique session ID.

In order to implement dynamic client-based caching we keep static table keeping updated session information about each active client. This table is called *MapTable* and each entry represents a client. Each entry keeps unique user identification number (uuid) and its dynamic session information. Dynamic session information for each client is kept as an instance of a class called *FormerRequest*. It has four attributes as listed below.

MapTable: Client-id session tracking Obj

uuid-1	FormerRequestObj1
uuid-2	FormerRequestObj2
…..	……

FormerRequest Class attributes

```
String uuid;    //unique-user-id
String bbox;    //bounding box of the last request
Double density;  //data size falling into per unit square.
Vector [] feature_data;  //geometry elements of the last request used to plot map
```

Density is used to find out allowable largest bbox area to be assigned to a thread for parallel processing. Details about the load balancing and parallel processing are given in Chapter 6.3.2.3.2.

1.3.2.2.2. Why Client-based Dynamic Caching

The fundamental concept behind the caching is removing the resource consuming repeated jobs and serving the client from the ready to use data sets kept in local storages. In case of map rendering process, ready to use data sets are map images. Google Map Servers [Googlemap] are the best examples for caching map images to provide high performance map services. They keep the data as ready to use map images chunked in tiles. Each tile is defined by its x,y coordinates and a corresponding zoom-level (18 different zoom levels). They formalize the accepted requests (in terms of parameters), and responses in terms of the tile compositions. Their major concern is developing high performance map services. In order to do that, they introduced AJAX (Asynchronous JavaScript and XML) [Ajax] for client/server communications and used locally stored static map images.

However, Google Map's static caching approach (tiling) would not work in case of considering (1) data's dynamic and distributed characteristics, and their various heterogeneous formats; and (2) seamless addition of new data sources rendered as layers and overlaid with other layers in various combinations and orders.

Google Map Servers provide two unique layers, satellite and Google map, and one hybrid layer as overlay of those two. Maps are served from three groups of tiles corresponding to these layer sets. In order to highlight the limitations of their algorithms, let's assume they

provide three unique layers instead of two. Let's say layer names are 'a, b and c. Then server would need to have 7 different tile groups as named a, b, c, ab, ac, bc and abc.

In summary, for the N number of unique layers, the required number of tile groups is calculated as below. It is sum of all k-subset combinations in which k gets the values from 1 to N.

$$\sum_{k=0}^{N} \mathrm{C}\binom{N}{k}, \qquad \mathrm{C}\binom{N}{k} = \frac{N!}{(N-k)!\,k!}$$

In case of 10 unique layers (N=10), the number of tile groups would be 1023. Moreover, in each tile group there are thousands of tiles, and each tile in the group has different copies for 18 different zoom levels. As the layer number increases, the number of required tile groups increases dramatically and at some point it becomes impossible to store that much tiles in a single storage with current possible technologies.

Client-based dynamic caching approach: We allow all the data to be kept at their original resources and integrated to the system through standard service API, communication messages and in expected common data formats. This enables extensibility and interoperability, easy data handling/maintenance, and workload and data sharing. We do on-demand data fetching and rendering. Instead of caching whole combinations of data sets we fetch and cache the data based on client's actions (locality and nearest neighborhood principles). Clients are given the flexibility to compose their own maps based on their applications' requirements. The framework also enables attribute based querying of the data integrated to the system through the common data model carrying both content and presentation features of the data.

With the client-based caching, besides removing the repeated processing jobs, we utilize the locality principles and develop efficient load balancing algorithm for sharing unpredicted workload among the worker nodes. We will show how to use this approach for load balancing in the following section.

6.3.2.3. Load-balancing through Query Decomposition and Parallel Processing

The parallel processing is implemented based on the main query partitioning. Each partition is assigned to separate thread of work. The number of partitions and their sizes are defined by using locality principles. Locality information is obtained from the cached data kept for the same session and user. See the Chapter 6.3.2.2.

Federator apply the parallel processing for creating multi-layered map images corresponding to un-cached queried region. Since all the data in the system is geo-referenced and queried in ranges defined by bounding boxes (defining coordinates of rectangles in the form of (minx, miny, maxx, maxy)), we do range query partitioning to implement parallel processing.

Parallel processing algorithm has three parts in order and closely related. These are listed below.

1. *Cached-data extraction and Rectangulation (Chapter* 6.3.2.3.1*)*
2. *Query decomposition over un-cached data regions in rectangle regions created at step1. (Chapter 6.3.2.3.2)*
    - *If there is no cache utilized decomposition will be applied to main query*
3. *Parallel-processing for sub-queries created at step2.(Chapter 6.3.2.3.3)*

In order to make these concepts more clear I give the illustration of these steps and their relations in Figure 13. Figure shows a map image composed of two layers. One is NASA satellite base map layer, and other is a layer showing earthquake seismic records (in blue dots). (a) shows partially overlapping of cached data and the main request bboxes. (b) Shows cached data extraction and rectangulation for the remaining part in the main query. (c) Shows partitioning of the rectangles from (b) based on the locality information obtained (explained in Chapter 6.3.2.3.2) from the cached data. All the rectangulated regions from (c) will be assigned to a thread to created map images as final responses.

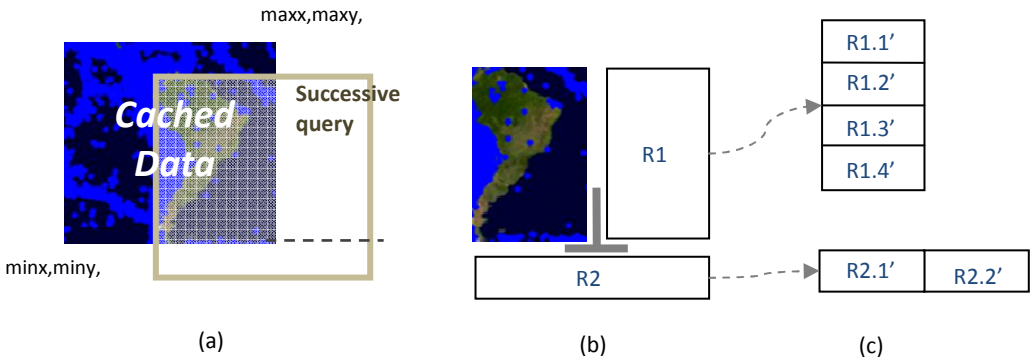

Figure 13: (a) Cached data extraction, (b) rectangulation, and (c) query decomposition/partitioning for parallel processing.

6.3.2.3.1. Cached-data Extraction and Rectangulation

According to OGC standards in GIS domain, queries are created with location parameter and location is defined in bounding box (bbox) formats. bbox is a formula defining the region as a rectangle through coordinates of bottom left corner and top right corner. Example: Q(minx, miny, maxx, maxy).

After extraction of cached data falling in the main query range, the remaining of the main query needs to be converted to the rectangular shapes in order to create valid sub-queries in the ranges defined by the bboxes (see *Figure 13* -b). This is why we make rectangulation after cached data extraction from queried-region.

The cached data extraction and rectangulation algorithm changes depending on the positions of bboxes of the main query and cached data region against to each-other. The main query and cached data bboxes can be positioned to each other in four possible ways (see Figure 14).

Notation to be used for representing bboxes: Main query bbox is described as (minx, miny, maxx, maxy) and Cached data bbox is described as ($minx^c$, $miny^c$, $maxx^c$, $maxy^c$)

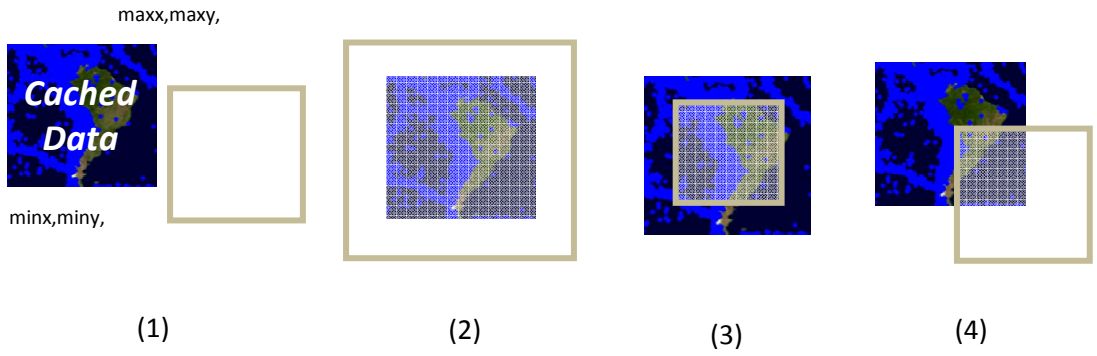

Figure 14: Positioning of the successive main query and stored client-based cached data

- Positioning-1: (No rectangulation). The main query and cached data do not overlap in anyway. In this case "cache data extraction and rectangulation" process is going to give only one rectangle which is the main query bbox.
- *Positioning-2*: The main query covers cached data (zoom-out action):

Rectangles: R1: minx, miny, $minx^c$, maxy R3: $minx^c$, $maxy^c$, $maxx^c$, maxy

R2: max^c, miny, maxx, maxy R4: $minx^c$, miny, $maxx^c$, $miny^c$

- *Positioning-3*: (No rectangulation). The main query falls in cached data (zoom-in action)

  Rectangles: This case enables the fastest response. There is no need for query partitioning and data transfer from WFSs. It just uses cached GML to create map image based on the bboxes values of main query. A lot of performance gains.

- Positioning-4: The main query partially overlaps with cached data (move action).

  This case is also explained in *Figure 13*.

  Here is the formula of the rectangles for a specific case of partial overlapping of cached data bbox and main query bbox (Figure 14-4):

  Rectangles: R1: minx, miny, maxx, $miny^c$ and R2: $maxx^c$, $miny^c$, maxx, maxy

  In this case, there are four different sub-cases depending on the movement directions. These are (1) down-right, (2) down-left, (3) up-right, and (4) up-left. The *Figure 13* illustrates the down-right case, and the rectangles above belong to his case. The rectangles for the other cases are also created similarly.

The rectangles obtained in this section go through the decomposition process explained in the following chapter.

6.3.2.3.2. Query Decomposition

This chapter explains how to determine the number of partitions, and how to partition the rectangles to assign to the separate threads to create map images in parallel processing.

There two ways we propose. One is naïve approach, just partition into equal sizes (Chapter 6.3.2.3.2.1). The other is smart approach partition the queries according to the previous query's bbox values and utilizing the locality principles. But because of the overhead timings and costs we need to define the best partition number to decompose the main query. In order to do that we propose smart query decomposition using client-based caching algorithm defined in Chapter 6.3.2.3.2.2.

6.3.2.3.2.1. Blind Query Decomposition

If there is no cached data available for the client, in other word rectangles are coming from positioning-1 explained in the previous chapter, then we use blind partitioning. In all the other cases we use smart decomposition technique explained in the following chapter.

Blind query decomposition is a static approach, it just chunks the query area (represented in bbox) into equal sized regions in terms of bbox values without identifying identity of client. Partition number is pre-defined and does not change at run-time.

6.3.2.3.2.2. Smart Query Decomposition Using Client-based Caching

Instead of decomposing the main query into predefined static number of sub-regions, we utilize the neighborhood and locality principles through client-based caching and figure out the most efficient number of partitions changing based on the data returned and cached at last time.

Here we explain how to define the number of partitions (i) and how to decompose the query (ii).

i. Determining the partition number:

In order to define the partition number we use locality principles. Locality principle in this context is explained as following. If a region has a high volume of data, then the regions in close neighborhood also expected to have high volume of data. Example is the human population data. The urban areas have higher human population than the rural areas. The oceans (2/3 of the world) have no populations etc.

We partition the rectangles into equal regions in the form of bboxes, because we don't know the size of the data falling in that region before getting it. In order to define the size (in bbox) we use cached data sizes expressed in bbox and KB. We assume (by using locality) cached data density is similar to the main request density, and by using the threshold value and un-cached main request part we calculate the partition number (P_n) as below:

Cached data bbox area = $(maxx^c - minx^c)*(maxy^c - miny^c)$

Density of cached-data: $dcd = \frac{Cached-data\ size\ in\ KB}{Cached\ data\ bbox\ area}$

Cached-data size in KB and bbox values are obtained from the client-based caching.

Allowable largest area to assign: $lat = \frac{threshold\ data\ size}{density\ of\ cached\ data} = \frac{thr}{dcd}$

Threshold data size is a static value pre-defined in server's properties file for the corresponding critical data.

P_n*: the number of partition* $= \frac{rectangule\ query's\ bbox\ area}{lat}$

If P_n is less than 1 then, don't make partition. In contrast, if it is bigger than 1, then partition into P_n regions. The following section explains how to partition a rectangle into P_n number of regions.

ii. Query decomposition of the rectangulated regions with P_n:

After getting P_n value in previous step, we cut the region into P_n number of sub-regions in the form of bbox values.

Here, we explain how to partition a given rectangle into P_n number of bboxes. There are two alternative techniques here, one is partitioning the rectangle vertically and the other is partitioning it horizontally.

In case of horizontal partitioning the step value is calculated as below, and partitioning is done along the Y-coordinate. (See Figure 15)

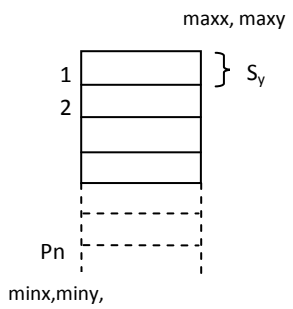

Figure 15: Partitioning a rectangle along the coordinate-y

Calculating the bboxes of the partitioned regions:

*for (i=0; i<P_n*s_x; i=i+s_x;)*

print (minx-i, miny, maxx-(i+s_y), maxy) ;

$$S_y = \frac{(maxy - miny)}{Pn}$$

6.3.2.3.3. Parallel-Processing

The proposed parallel processing is based on range-query (defined in bbox) decomposition we call it partitioning. The partitioning is done with the locality principles to share the workload to the threads to reduce the response times.

This section explains how to create sub-queries corresponding to the partitions, and how to assign the sub-queries to threads and assemble the results.

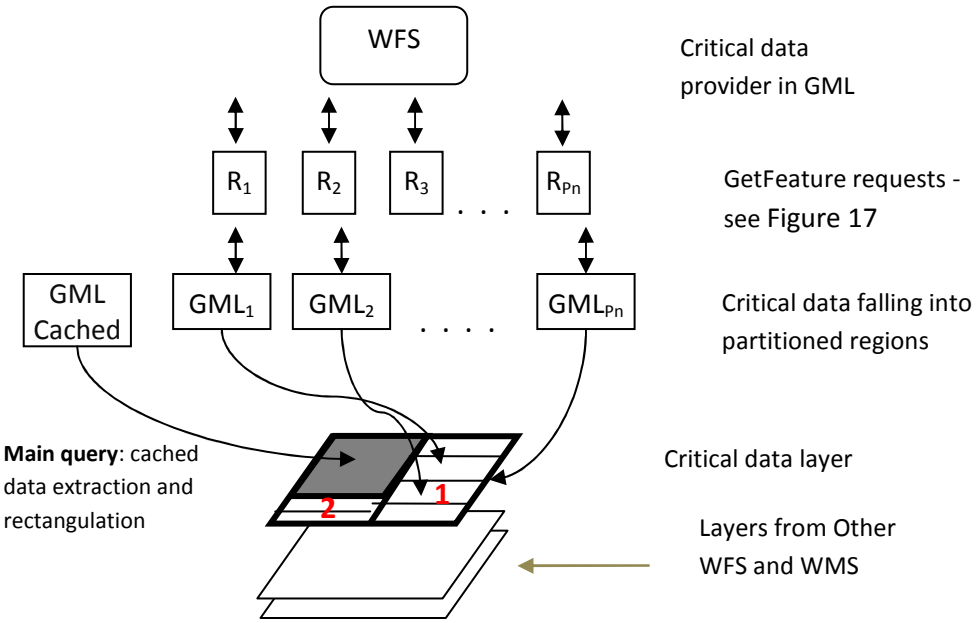

Figure 16: Parallel processing and caching architecture in brief. See also *Figure 13*.

These issues are illustrated in Figure 16 above. In this specific example, main query includes three separate layers, and one of them is created with the critical data encoded in common data model, GML. The rectangulated regions 1 and 2 in the main query are determined by the cached-data extraction and rectangulation processes explained in Chapter 6.3.2.3.1. Grey region in the main query overlaps with the cached data. There is no need for data transfer for this region. This is obtained from the cache. For the other parts not overlapping with the cache (region 1 and 2), the system makes parallel processing for data access, query and plotting after creating partitions.

i. Creating the queries for the partitions.

Throughout the rectangulation and partitioning, the only changing attribute of the main query is the bbox coordinate value. These are calculated in the previous chapter.

Based on the set of bbox values obtained at the end of partitioning process (ii) we need to create sub queries. Each partition is differentiated by only their bbox value, and they go through the query creation process. AWMS creates *getFeature* requests corresponding to these rectangles based on their bounding boxes. Other parameters and attributes required for creating *getFeature* request are obtained from the main query. All the parameters, attributes and their values (except for bbox values) will be the same for all the getFeature requests created for the partitions.

An example case of decomposing a rectangle obtained by rectangulation process and creating parallel queries is illustrated at Figure 17. In this example, rectangle is partitioned into 5 regions vertically.

$$P_n = 5 \quad \text{and} \quad s_y = \frac{(\text{maxy} - \text{miny})}{Pn} = \frac{(45-40)}{5} = 1$$

You can see a sample getFeature created for bbox value "-110, 35, -100, 40" request at Figure 18.

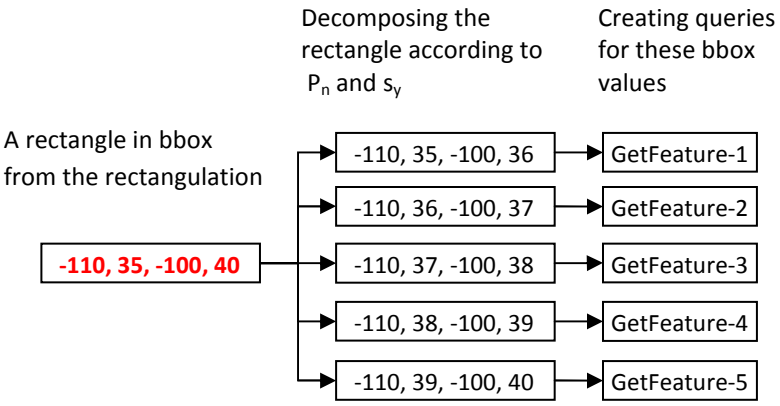

Figure 17: Example scenario of the partitioning a region into 5 sub-regions through the bbox value of a rectangle.

```
<?xml version="1.0" encoding="iso-8859-1"?>
 <wfs:GetFeature outputFormat="GML2" xmlns:gml="http://www.opengis.net/gml"
xmlns:wfs="http://www.opengis.net/wfs" xmlns:ogc="http://www.opengis.net/ogc">
      <wfs:Query typeName="global_hotspots">
       <wfs:PropertyName>YEAR</wfs:PropertyName>
       <wfs:PropertyName>MONTH</wfs:PropertyName>
       <wfs:PropertyName>DAY</wfs:PropertyName>
       <wfs:PropertyName>LATITUDE</wfs:PropertyName>
       <wfs:PropertyName>LONGITUDE</wfs:PropertyName>
          <wfs:PropertyName>MAGNITUDE</wfs:PropertyName>
       <ogc:Filter>
        <ogc:BBOX>
          <ogc:PropertyName>coordinates</ogc:PropertyName>
          <gml:Box>
           <gml:coordinates>-110,35 -100,40</gml:coordinates>
          </gml:Box>
        </ogc:BBOX>
       </ogc:Filter>
      </wfs:Query>
      <wfs:Query typeName="global_hotspots">
       <ogc:Filter>
        <ogc:PropertyIsBetween>
         <ogc:Literal>MAGNITUDE</ogc:Literal>
         <ogc:LowerBoundary>
          <ogc:Literal>7</ogc:Literal>
         </ogc:LowerBoundary>
         <ogc:UpperBoundary>
          <ogc:Literal>10</ogc:Literal>
         </ogc:UpperBoundary>
        </ogc:PropertyIsBetween>
       </ogc:Filter>
      </wfs:Query>
</wfs:GetFeature>
```

Figure 18: Sample GetFeature request for the partitioned region of bbox (-110, 35 -100, 40). Request is done for global hotspot (earthquake seismic data)

ii. How to assign the sub-queries to threads and assemble the results.

Sub-queries created at previous step are assigned to separate threads to capture the GML data from WFS and process the corresponding map pieces. Partitions are assigned to worker nodes through separate thread of works in round-robin fashion [tanenbaum].

Let's say PN is the partition number and WN is the number of WFS worker nodes.

$$share = base\,(\frac{PN}{WN})$$

Share is the number of partitions each worker node is supposed to get.

$$rmg = WN - \ base\,(\frac{PN}{WN})$$

rmg is the remaining of the PN/WN division. If there is no remaining every worker node is assigned share number of partitions. Rmg is dofferent from 0 then partitions are assigned to worker nodes as below:

The first rmg #of WN are assigned share+1 number of partitions, and remaining WN are assigned share number of partitions.

Figure 19 illustrates the algorithm over a case of seven partitions and three WFS worker nodes (called WFS-1, WFS-2 and WFS-3). So, the algorithm's parameters would be

share = base (7/3) = 2 and *rmg* = 3 – 2 = 1;

So WFS-1 is assigned 3 (*share+1*) partitions through thread-1, 4 and 7,

WFS-2 is assigned 2(*share*) partitions through thread-2 and 5

And finally WFS-3 is assigned 2 (*share*) partitions through thread-3 and 6.

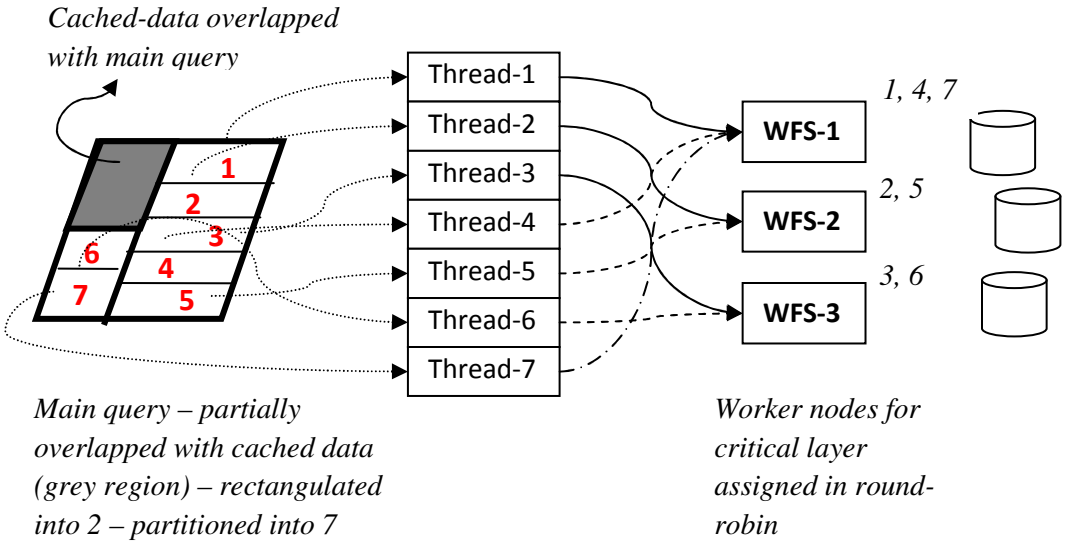

Figure 19: Assigning partitions to threads and capturing/processing in parallel

Each query corresponding to the partitions are assigned to the threads. Threads are responsible for interacting with the WFS and getting the requested data to create map images for the partition. After getting the data, federator starts rendering and plotting the critical data over the other layers by parsing and extracting the geometry elements in returned GML.

6.3.2.3.4. Overall Performance Evaluation

Performance will be evaluated in three possible generalized situations categorized based on the cached data utilization. These are:

a. *No usage of cached data*

b. *Complete usage of cached data. No need for parallel processing.*

c. *Partial usage of cached data*

Here is the performance test setup:

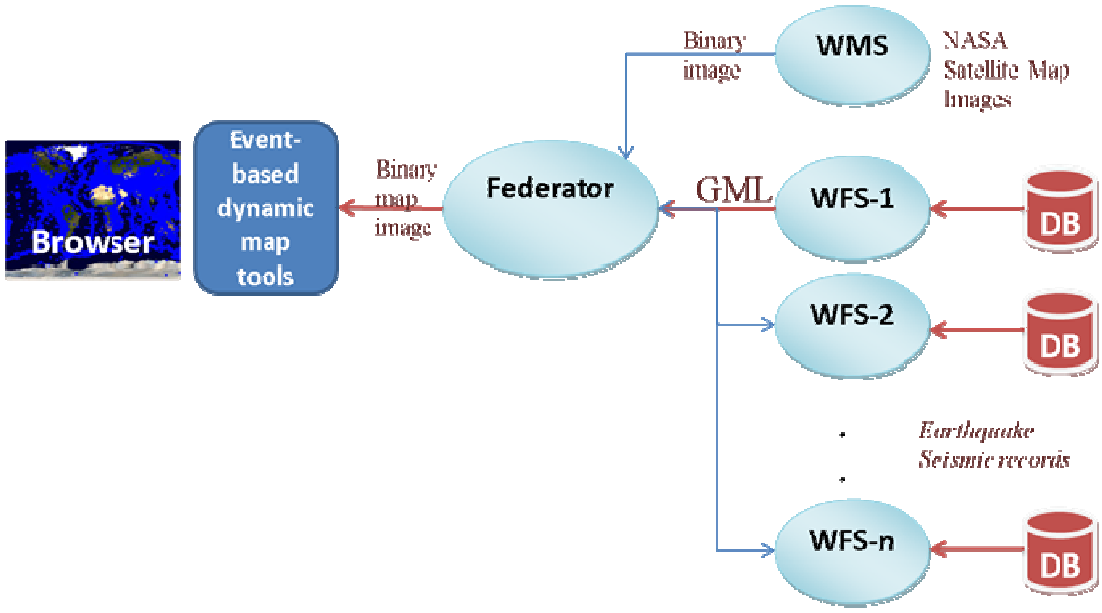

Figure 20: Test setup for federator oriented approaches.

According to the tes purposes and test setup environment we have six different kinds of servers. These are Web Map Server (WMS), the WMS-extended federator, Web Feature Service (WFS), MySQL-database, NaradaBrokering messaging middleware and browser/event-based interactive mapping tolls client. We also integrated the third party OGC compatible WMS servers such as NASA WMS providing satellite map images from OnEarth project and Google Map servers providing Google maps.

Every machine (on which servers are deployed) has 2 Quad-core Intel Xeon processors running at 2.33 GHz with 8 GB of memory and operating Red Hat Enterprise Linux ES release 4. Machines are in Local Area Network (LAN).

a. ***No*** *usage of cached data:*

This case happens when the query bbox don't not overlap with each other. In this case there is no need to cached-data extraction and rectangulation, because there is only one rectangle which is the main query to partition. Here, we show performance gaining by using parallel processing through query decomposition. In order to make performance evaluations, we test the system with different (2, 10 and 20) levels of partitions and assign them to separate individual threads for creating map images in parallel.

We first present the performance values in average response times detailed in "data capture timing", "map rendering timing" as displayed in Table 9 and Table 10. In this context, response times (total map creation time) are divided into three measured items. First is DC (data capturing from WFS to WMS/federator), second is MR (Map rendering at WMS/federaator), and third one is map images' transfer time from federator to event-based dynamic map clients for end-users. Third one is not shown in the analysis but can be derived from the table by below formula for each data size separately.

$$\textit{Map images' transfer time} = RT - (DC+MR)$$

Here, for the partitioning, since there is no cached-data to be utilized we use blind partitioning technique given in Chapter 6.3.2.3.2.1.

Table 9:Average times for data capturing, map rendering and overall response for different number of partitioning and different data sizes.

	Average Timings								
Size	2 Threaded			10 Threaded			20 Threaded		
MB	*DC	*MR	*RT	DC	MR	RT	DC	MR	RT
0.01	769.9	813.3	1,728.3	1,385.5	891.6	2,329.5	2,423.3	1,041.4	3,589.1
0.1	1,161.0	829.6	2,031.4	1,712.3	994.3	2,760.0	2,483.4	1,077.1	3,629.4
0.5	2,664.5	958.1	3,672.7	2,488.5	999.7	3,460.4	2,628.1	1,194.6	3,759.4
1	5,749.8	1,172.7	6,977.0	3,440.9	1,140.5	4,640.5	3,820.4	1,382.9	5,268.8
5	20,350.4	1,707.0	22,108.0	15,036.9	1,627.6	16,725.4	14,390.5	1,680.5	16,148.0
10	45,072.8	2,499.0	47,639.1	20,517.3	2,518.1	23,118.4	22,060.3	2,637.6	22,800.1
50	247,321.8	11,839.6	259,341.7	192,592.8	11,894	204,727	111.753	8,890.2	120,822

*TMC (Total Map Creation Time) = Data Capture (*DC) + Map Rendering (*MR)*
**RT = TMC + Map Images' Transfer Time*

Table 10: Standard deviation for data capturing, map rendering and overall response for different number of partitioning and different data sizes.

	Standard Deviations								
Size	2 Threaded			10 Threaded			20 Threaded		
MB	*DC	*MR	*RT	DC	MR	RT	DC	MR	RT
0.01	92.52	103.67	164.37	91.09	85.34	131.46	281.40	206.17	482.77
0.1	99.03	81.87	123.44	89.56	107.28	104.35	177.48	297.37	312.16
0.5	94.35	140.09	193.28	97.66	81.44	120.24	117.43	149.45	124.11

1	101.61	191.37	211.20	90.05	86.35	106.42	108.09	210.27	223.71
5	99.43	154.33	287.72	190.37	77.25	201.62	265.17	277.07	488.09
10	131.44	420.00	509.01	973.42	137.81	941.83	582.05	261.82	706.62
50	312.2	5,208.7	5,395.6	1,852.5	5,639.1	5,676.4	1,154.5	245.9	1,182.4

*TMC (Total Map Creation Time) = Data Capture (*DC) + Map Rendering (*MR)*
**RT = TMC + Map Images' Transfer Time*

Since the major bottleneck of the performance is transferring GML data, we first demonstrate the performance enhancement in data transfer (see Table 11 and Figure 21). The data is transferred from the databases through WFSs to the federator (or WMS). The measured transfer times are in milliseconds.

The data capturing times in the below table are obtained from Table 9 except for single-thread column's values. They are obtained from Table 4.

Comparison of data transfer times based on partition number and data sizes

Table 11: Data transfer times for different levels of partitioning and data sizes.

Data Size MB	Data Capture Comparisons			
	Single-thread*	2-thread	10-thread	20-thread
0.01	797.85	769.94	1,385.50	2,423.30
0.1	1,384.86	1,160.95	1,712.27	2,483.36
0.5	3,770.16	2,664.47	2,488.47	2,628.10
1	6,794.94	5,749.79	3,440.89	3,820.36
5	31,237.41	20,350.38	15,036.95	14,390.50

10	61,777.20	45,072.75	20,517.26	22,060.27
50	308,671.63	247,321.80	192,592.80	111,753.20

*Values for single-thread are obtained from first column of Table 4.

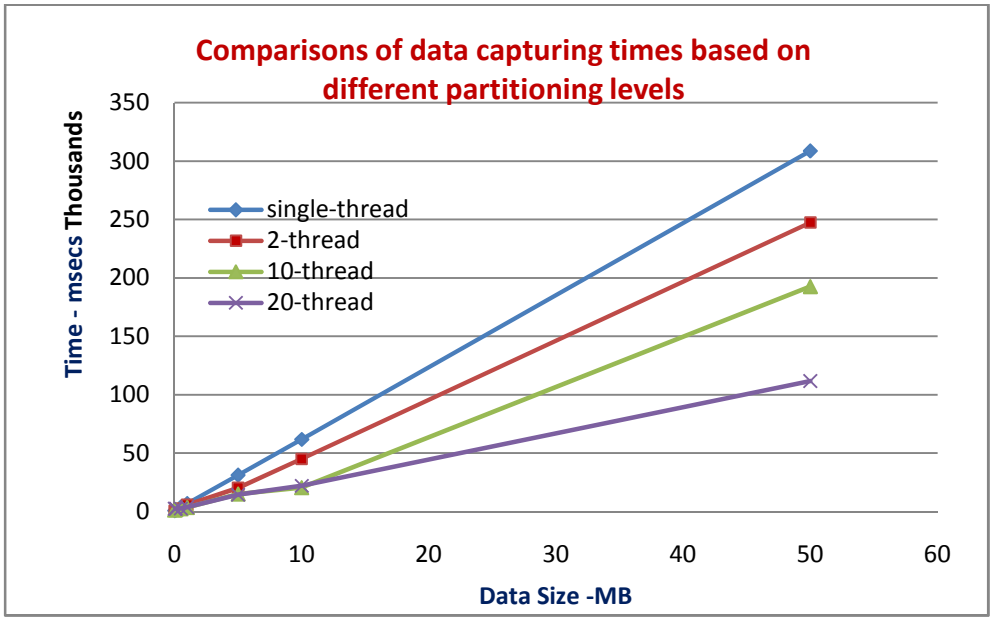

Figure 21: Comparison of average data transfer times for various levels of data sizes and partitioning level.

Comparison of response times based on partition number and data sizes

Table 12 compare the performance values for different levels of partitions and shows what partition level gives the best result for corresponding data size. According to the table, for the data sizes less than 100KB partitioning into two gives the best result. Using more than two partitions degrade the response times because of the overhead times. For the overhead times and analysis see Table 14 and Figure 24. For the sample cases of

partitioning levels and given data sizes, we also present the best partition numbers in the last column.

Table 12: Average response times for different data sizes and partition levels, and listing of best partitions for each data sizes.

Data Size	Response Time Comparisons				
MB	Single-thread	2-thread	10-thread	20-thread	Best Partition
0.01	1,808.13	1,728.28	2,329.50	3,589.10	*2*
0.1	2,635.46	2,031.35	2,760.00	3,629.36	*2*
0.5	5,001.29	3,672.74	3,460.40	3,759.40	*10*
1	8,225.73	6,977.00	4,640.53	5,268.79	*10*
5	33,419.31	22,107.95	16,725.37	16,148.00	*20*
10	64,506.78	47,639.10	23,118.42	22,800.13	*20*
50	316,906.39	259,341.67	204,727.93	120,822.00	20

For small sizes of data such as less than 500KB, high number of partitioning does not help in performance increase, instead degrade it. As you see in the table for small size of data partitioning into 2 give sthe best result. That is because of the overhead times coming from partitioning, sub-query creation and finally merging the results to create final response. For more information about overhead times see Table 14 and Figure 24.

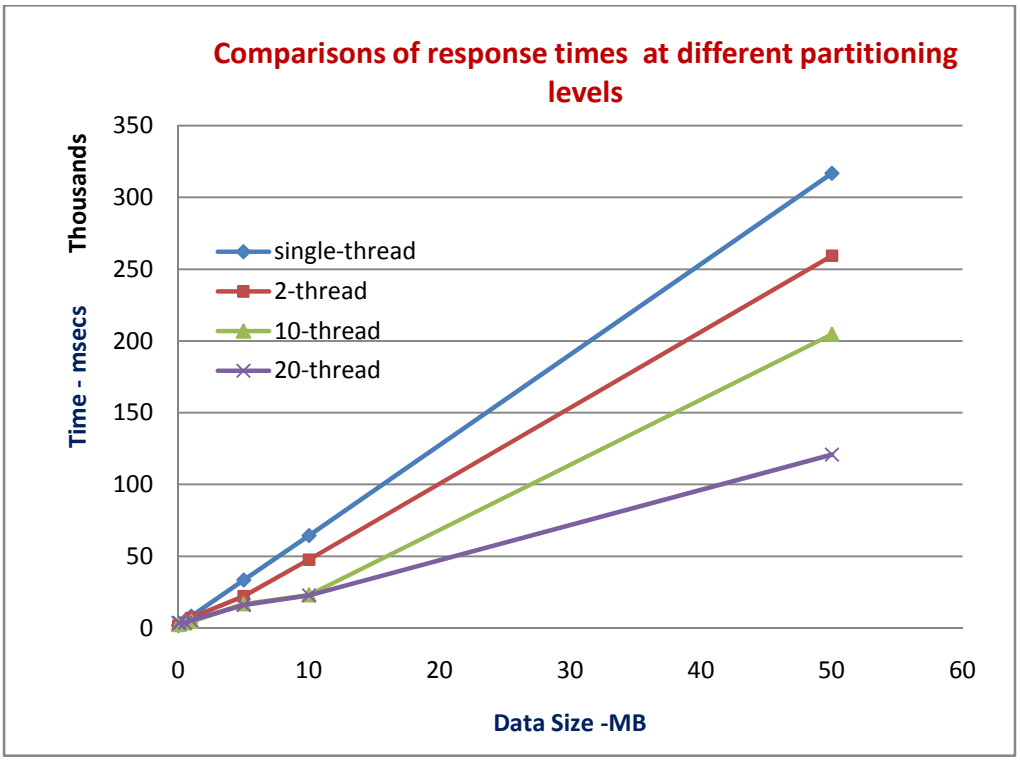

Figure 22: Comparison of average response times for different partitioning and data sizes.

When you compare the Figure 21 and Figure 22, you will think that they are same but it is not. They look similar because of that data capturing/transfer is the dominant value in the response times, and in some cases almost %90 of response times comes from data transferring times.

Comparison of Response times: Naïve approaches vs proposed enhanced approaches

Table 13 and Figure 23 show the striking performance enhancement in response times for the overall architecture. To make it more clear, for 10MB of data size, proposed architecture is almost 30 times faster than the architecture developed with naïve approaches.

Table 13: Response times comparison values - Naïve approach and the proposed approach at different partitioning levels.

Data Size MB	Response Time Comparisons			
	Naïve approach	2-thread	10-thread	20-thread
0.01	2,578.69	1,728.28	2,329.50	3,589.10
0.1	7,973.16	2,031.35	2,760.00	3,629.36
0.5	30,868.52	3,672.74	3,460.40	3,759.40
1	59,635.69	6,977.00	4,640.53	5,268.79
5	288,594.12	22,107.95	16,725.37	16,148.00
10	574,825.16	47,639.10	23,118.42	22,800.13

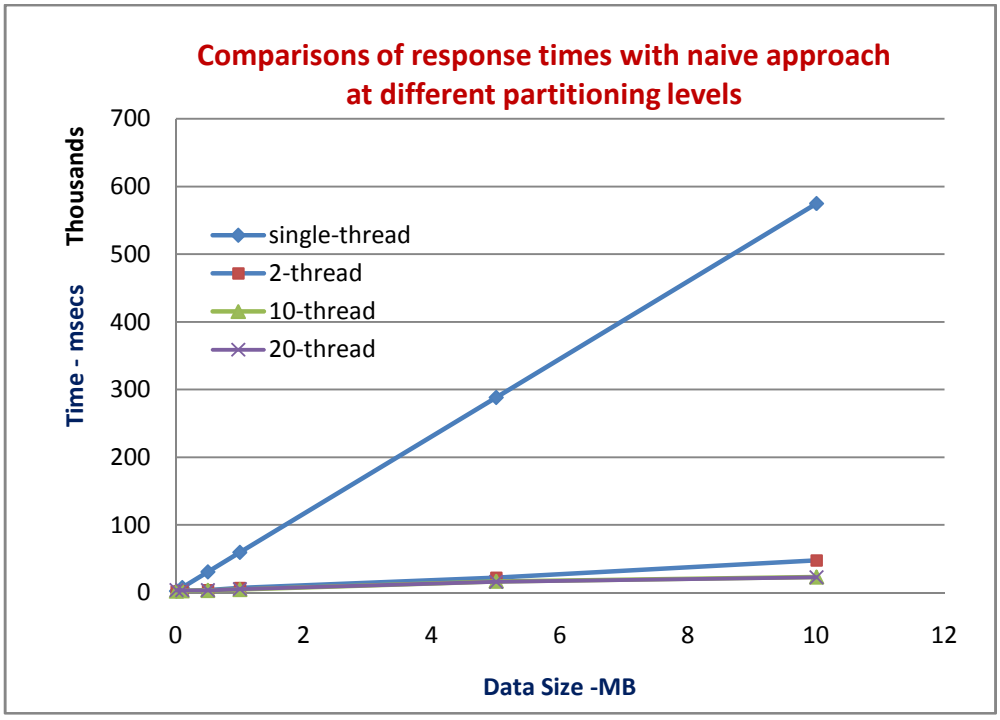

Figure 23: Comparison of response times at different partitioning levels – Naïve approach vs. proposed approach.

From the figure we see that the performance does not increase in the same ratio at which the thread number increases. That is because of the overheads resulted from mainly the query decomposition and assembling the result sets for the main query etc. Moreover, the figure shows that the higher the data size the larger the performance gains.

Detailed overhead timings:

Table 14 and Figure 24 present overhead times of the proposed federator oriented parallel processing technique in map rendering. The major overheads are grouped into three. These are partitioning, sub-query creation, and merging the sub-results to the partitions to create final output.

Table 14: Overhead times due to making partitioning for parallel processing at various partitioning levels.

Partition Number	Partitioning		Sub-Query Creation		Merging of partitions	
	Avg	StDev	Avg	StDev	Avg	StDev
5	51.28	14.74	161.67	25.32	27.00	12.88
10	58.65	15.16	421.55	63.98	44.26	23.44
15	60.15	19.74	720.35	102.87	64.90	23.77
20	68.75	21.75	1,058.84	199.49	118.90	25.53
25	69.05	15.98	1,366.10	198.37	131.88	30.59
30	85.42	30.04	1,837.16	343.26	170.00	30.56

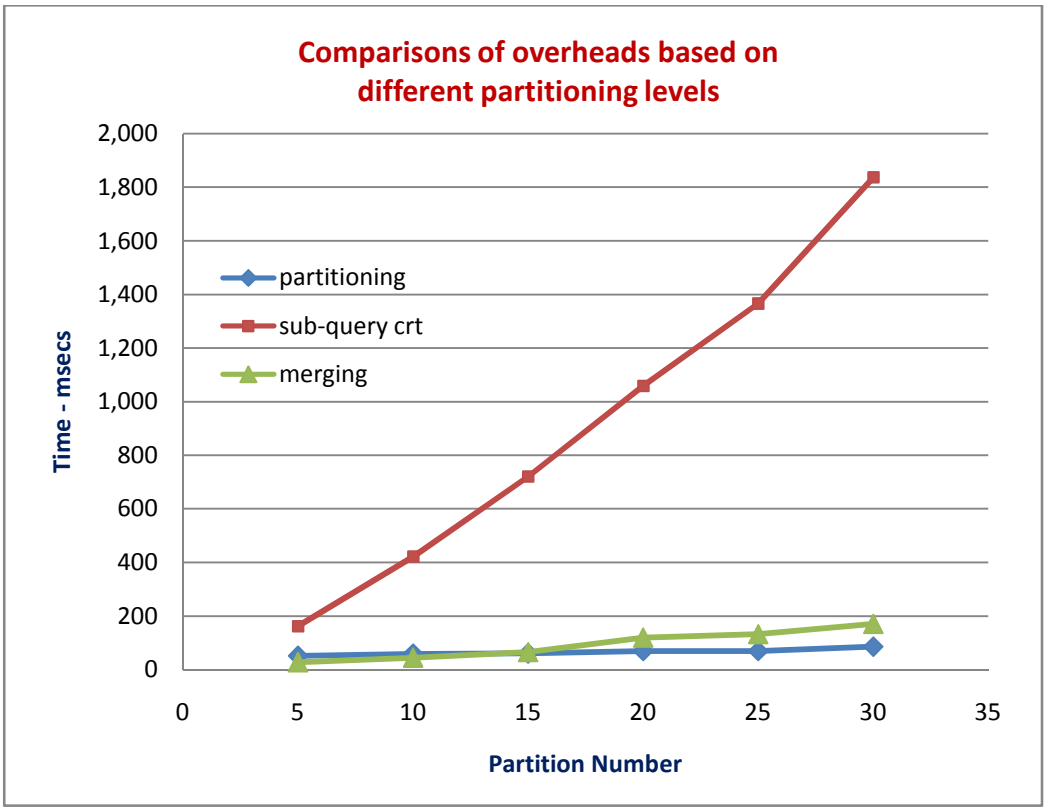

Figure 24: Parallel processing overheads based on different levels of partitioning.

b. ***Complete*** *cached data utilization*

In this case there is no need for rectangulation, main query decomposition and threaded parallel processing. This case happens when the user query a smaller region falling in the previous map he got on his browser. It is mostly caused by zooming-in action. In this case, the cached data is enough for responding to the main request, and no other cascaded requests are needed. The Federator renders the map just by using the cached data. The only task needed is the cached data extraction and overlaying (plotting) over the other requested layers.

This case's performance results are almost same as the pre-fetching techniques'. Please see Table 7 and Figure 11 to get an idea about the performance enhancement.

c. *Partial cached data utilization :*

This case happens when the user moves (or drag and drop) the map to another region or makes zooming-out. In other words, when the user makes successive requests and their bbox values partially overlap. As explained before in *Figure 13* and Figure 14 in Chapter 6.3.2.3, if only one point of main request falls in bbox boundaries of the cached-data, they are called as partially overlapped.

In order to simplify the analyzing we give a sample scenario: 1/2 of main query overlaps to the cached data and remaining data is obtained and processed with 10-threaded parallel processing.

Table 15 shows the average timing values for the selected sample bbox values and data sizes, and Table 16 shows the corresponding standard deviations. The cached data is accessed and processed in a way similar to the way of processing the pre-fetched data. In order to access the remaining data we decompose the query into 10 and assign each query to 10 separate threads to create corresponding map images created from the data captured from databases through WFS services.

The first column of the table shows the data sizes of GML data to be captured to create map images. The data size values given in the parenthesis are cached data sizes which are actually half of the requested data size.

Table 15: Performance results for the sample case scenario in which half of the data is provided by the cached data, and other half is obtained from WFSs and processed by 10-thread parallel processing.

	Sample bounding boxes after rectangulations on which partitioning is done				Average Timings		
GML Data Size -MB*	minx	miny	maxx	maxy	cached-data processing	On Remain 10-thread	Total Average Response
0.01(0.005)	-121.58	34.55	-121.45	34.69	734.33	2,360.86	3,095.19
0.1(0.05)	-121.65	34.36	-120.78	35.00	768.33	2,808.40	3,576.73
0.5(0.25)	-118.68	34.21	-118.39	34.50	782.45	2,939.32	3,721.77
1(0.5)	-119.16	34.21	-118.25	35.12	851.33	3,460.40	4,311.73
5(2.5)	-120.83	32.07	-117.15	36.18	1,209.79	10,084.79	11,294.58
10(5)	-120.83	32.07	-115.70	36.70	1,646.32	16,725.40	18,371.72

*requested data size (cached data size)

Table 16: The standard deviations for the average times given in Table 15.

	Standard Deviation		
GML Data Size -MB	cached-data processing	remaining-data access and proc	Total Response
0.01(0.005)	62.45	141.77	204.22
0.1(0.05)	66.37	217.43	283.80
0.5(0.25)	79.56	130.85	210.41
1(0.5)	72.21	120.24	192.45

5(2.5)	68.52	245.07	313.59
10(5)	94.57	201.62	296.19

In Table 17, we compare the average response times of the given sample case with other two group of response times obtained by not using cached data. First group's response times are obtained by using 10 threaded parallel processing and second group's are obtained by using single thread.

Table 17: Comparison of the response times for the hybrid (caching and parallel processing) and ordinary non-caching single-threaded system.

	Comparison of the response times					
GML Data MB	Half cached/ 10 thread		NO Cached /10 thread		NO Cached /Single thread	
	Avg. Time	StdDev	avg time	std dev	Avg. Time	StdDev
0.01	3,095.19	204.22	2,329.50	131.46	1,808.13	140.32
0.1	3,576.73	283.8	2,760.00	104.35	2,635.46	313.48
0.5	3,721.77	210.41	3,460.40	120.24	5,001.29	238.94
1	4,311.73	192.45	4,640.53	106.42	8,225.73	200.27
5	11,294.58	313.59	16,725.37	201.62	33,419.31	394.48
10	18,371.72	296.19	23,118.42	941.83	64,506.78	283.24

As it is shown in first two lines of Table, there is no gain of using parallel processing with caching for the small sizes of data. In such cases, total overhead sometimes get higher than the total response times of single threaded cases. This problem is solved by

using a threshold value to define if the partitioning is needed or not. This technique is also explained in 6.3.2.3.2.2.

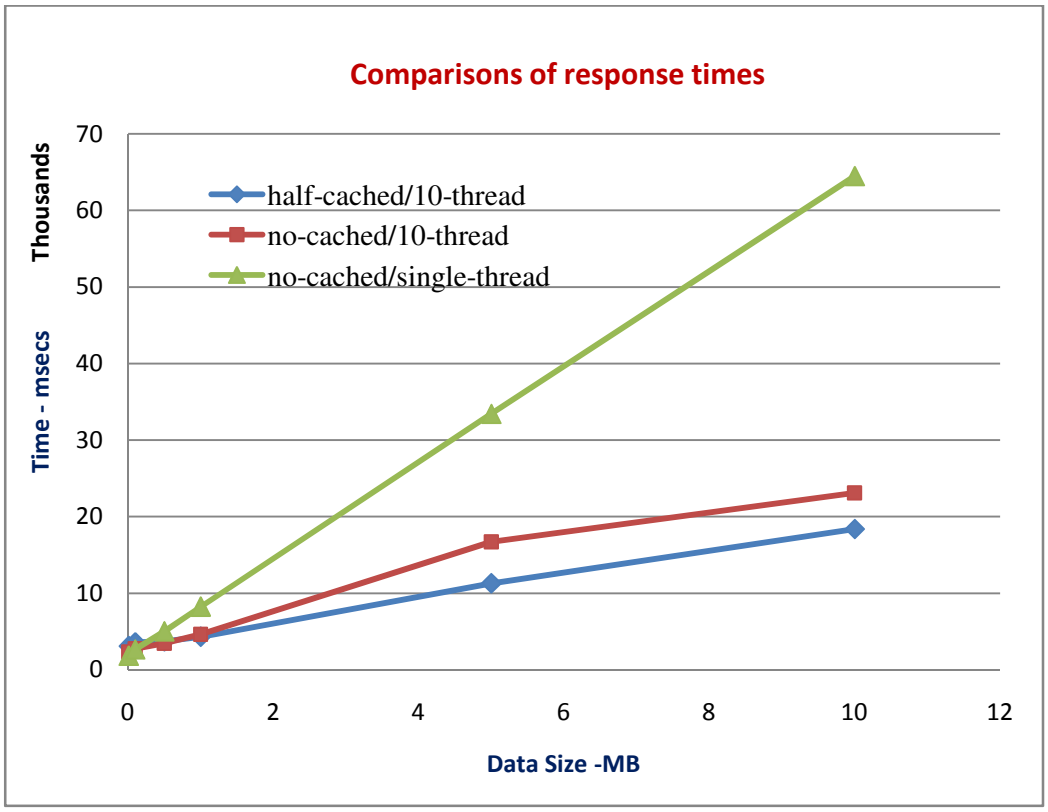

Figure 25: Illustrating the performance enhancement of using caching with parallel processing with ½ cached data case.

As it is shown the in Figure 25, for the given test scenario (1/2 of main query overlaps to the cached data and remaining data is obtained and processed with 10-threaded parallel processing) proposed system is more than 3 times faster than the single threaded system. As the data size increases, that ratio increases.

When we compare the enhanced system's performance result with the naïve approaches' performance result given in Table 1 and *Figure 3*. We see that using parallel processing

and caching techniques make the system almost 30 times faster than the naive approaches for the given specific case scenario.

As the data size and/or density of data falling per unit square increase, the performance gaining from using the proposed technique increases.

REFERENCES

[Lu 2006] Wei Lu, Kenneth Chiu, and Yinfei Pan, "A Parallel Approach to XML Parsing". In *the 7th* IEEE/ACM International Conference on Grid Computing, *2006.*

[Pallickara2003] Pallickara S. and Fox G., "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids" ACM/IFIP/USENIX, Rio Janeiro, Brazil June 2003.

[Donbox] Don Box, David Ehnebuske, Gobal Kakivaya, Andrew Layman, Dave Winer., Simple Object Access Protocol (SOAP) Version 1.1, May 2000.

[Sosnoski] Sosnoski, D. "XML and Java Technologies", performance comparisons of the Java based XML parsers. Available at http://www-128.ibm.com/developerworks/xml/library/x-injava/index.html

[Alexander] Aleksander Slominski. XML Pull Parser, visited 04-15-02. http://www.extreme.indiana.edu/xgws.

[Vretanos] Vretanos, P. (ed.), Web Feature Service Implementation Specification (WFS) 1.0.0, OGC Document #02-058, September 2003

[GML] Cox, S., Daisey, P., Lake, R., Portele, C., and Whiteside, A. (eds) (2003), OpenGIS Geography Markup Language (GML) Implementation Specification. OpenGIS project document reference number OGC 02-023r4, Version 3.0.

[WMS] de La Beaujardiere, J., Web Map Service, OGC project document reference number OGC 04-024. 2004.

[WFS] Vretanos, P. (2002) Web Feature Service Implementation Specification, OpenGIS project document: OGC 02-058, version 1.0.0. Volume,

[Booth]Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. "Web Service Architecture." W3C Working Group Note, 11 February 2004. Available from http://www.w3c.org/TR/ws-arch.

[Tran] Tran, P., Greenfield, P., and Gorton, I., *Behavior and Performance of Message-Oriented Middleware Systems.* . Proceedings of the 22nd international Conference on Distributed Computing Systems, ICDCSW. 2002.

[uddi] Bellwood, T., Clement, L., and von Riegen, C., UDDI Version 3.0.1: UDDI Spec Technical Committee Specification http://uddi.org/pubs/uddi-v3.0.1-20031014.htm. 2003.

[ogc] The Open Geospatial Consortium, Inc. web site: http://www.opengeospatial.org

[deegree] deegree project web site available at http://deegree.sourceforge.net/

[minmapserv]University of Minnesota Map Server, available at http://mapserver.gis.umn.edu/

[patterninfo] Tiampo, K. F., Rundle, J. B., McGinnis, S. A., & Klein, W. Pattern dynamics and forecast methods in seismically active regions. Pure Ap. Geophys. 159, 2429-2467 (2002).

[Denning] P.J. Denning and S. C. Schwartz, Properties of the working-set model. Communications of the ACM, 15(3), March 1972.

[Belur] Belur V. Dasarathy, editor (1991) *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*, ISBN 0-8186-8930-7.

[Ajax] Nicolas Serrano, Juan Pablo Aroztegi, Ajax Frameworks for Interactive web apps, IEEE Software Magazine V24n5 (Sep/Oct 2007) pp12-14.

[Googlemap] Project web site is available at http://code.google.com/apis/maps/index.html

[tanenbaum] Andrew S. Tanenbaum, Modern Operating Systems, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2001.

[apache] Apache Tomcat, http://tomcat.apache.org/.