Agent-Based Distributed Analytical Search

Subrata Das(✉), Ria Ascano, and Matthew Macarty

Machine Analytics, Cambridge, MA, USA
sdas@machineanalytics.com

Abstract. We describe here an agent-based Distributed Analytical Search (DAS) tool to search and query distributed "big data" sources regardless of data's location, content or format. DAS semantically analyzes natural language queries from a web-based user interface. It automatically translates the query to a set of sub-queries by deploying a combination of planning and traditional database query optimization techniques. It then generates a query plan represented in XML and guide the execution by spawning intelligent agents with various types of wrappers as needed for distributed sites. The answers returned by the agents are merged appropriately and return them to the user. We have demonstrated DAS using a variety of data sources that are distributed and heterogeneous. The tool is the prime product of our company with big enterprises as our target market.

1 Introduction

Big data is generally stored in relational databases, such as Oracle, DB2, SQL Server, and MySQL, and in data warehouses such as Terradata. This data is generally heterogeneous and distributed, making it difficult to query accurately and quickly for analytics (Das, 2014). Although big data environments are in the process of migrating to the scalable, fault-tolerant cloud environment, the cloud remains experimental in nature, due to its lack of adequate data security and the unrealized need for a query tools utilizing the Map Reduce paradigm (Dean and Ghemawat, 2008). As a result, data remains distributed in many formats, both structured and unstructured, and only non-essential data is currently stored in the cloud. What is needed is an approach to query distributed sources maintaining autonomy of individual data sources (Widom, 1996). We have developed an agent-based Distributed Analytical Search (DAS) tool to fulfill this gap.

DAS will allow end users to query distributed data sources in natural language without having to know the source formats and locations. In the government space, most distributed archives and databases, such as NASA's DAAC and the DoD's DCGS, are autonomously maintained. Additionally, our personal communications with personnel from big retailers such as Sears and Walmart reveal that their databases are also highly distributed and heterogeneous with less than 10% residing in cloud environments, and that it takes almost a day for an analyst to extract data from relevant sources after the request is placed. Our approach will allow analysts to query data sources directly in natural language and will reduce this one-day turnaround time to within seconds.

Y. Demazeau et al. (Eds.): PAAMS 2015, LNAI 9086, pp. 40–52, 2015.
DOI: 10.1007/978-3-319-18944-4_4

DAS searches distributed structured and unstructured “big data” sources by semantically analyzing natural language queries regardless of data’s location, content or format. DAS accepts natural language queries from a web-based user interface, deploying “intelligent agents” to scan unrelated data sources and return answers to support the decision-making process. DAS is format-agnostic. DAS allows users to perform distributed search within the cloud without users needing to already know the format or locations of individual data sources. In addition, it is not necessary for these data stores to be traditional relational, nor do they need to be on the same network. Agent-assembled data is analyzed for underlying trends. This is a non-trivial exercise, with agents building and executing queries based on natural language user input. Secured Agents will build temporary tables from multiple unrelated data sources by taking computations to data sources, thus avoiding large downloads. We are uniquely positioned in this market place.

In summary, DAS answers queries through the following stages:

- Accepts a search query from a user in natural language via a web interface.
- Automatically translates the query to a set of sub-queries by deploying a combination of planning and traditional database query optimization techniques.
- Generates a query plan represented in XML and guide the execution by spawning intelligent agents with wrappers as needed for distributed sites.
- Merges the answers returned by the agents and return them to the user.

Our approach is innovative because no other currently available technology can query distributed data sources, and its extreme need is justified above. Our natural language query translation, using hybrid deep linguistics processing and machine learning, and the plan generation along with XML representation and distributed execution, is unique and is Machine Analytics’ trade secrets.

The rest of the paper is organized as follows: Section 1 describes briefly the web-based querying interface and our approach to natural language query translation to SQL. Section 2 describes the query planning and optimization techniques. Section 3 describes in detail the agent-based query execution strategy. We conclude the paper with our future plan with DAS. For the purpose of illustrating DAS functionalities, throughout the paper we will be using a small example database consisting of two tables. Figure 1 shows the tables SALUTE and Mobility, with some sample rows as examples. These example tables are stored at multiple sites. The distributed query execution as described above therefore avoids downloading large volumes of Mobility and SALUTE (size-activity-location-unit-time-equipment) data records from these remote tables to the host site. An example query in this context that we will using throughout the paper is “Show Salute platforms from NAIs with mobility no go.”

SALUTE

NAI	FROM	ACTIVITY	EQUIPMENT	TIME	SIZE
47	JSTARS	Milling	Vehicles	14:20	40-60
65	UAV	Emplaced	BMP	18:12	?
91	LRS	Meeting	AK 47	10:30	100-200
20	IMINT	Digging	Truck	05:10	1
...	...	...	...	...	...

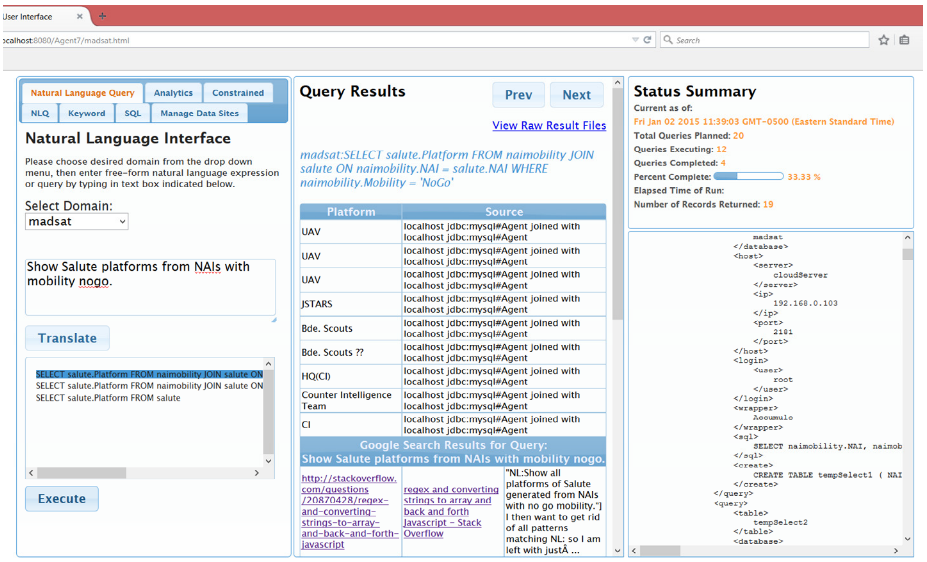
NAI-Mobility

NAI	Mobility
47	Slow Go
23	No Go
49	Go
43	Go
...	...

Fig. 1. SALUTE and Mobility tables with some example rows

2 User Interface and Natural Language Querying

Currently the user will find the web-based interface by visiting a URL. For example in a typical installation on one of our servers, the user can access interface and current functionality of DAS via the following URL: `192.168.0.101:8080/Agent7`. The user will be presented with the single page application a screenshot of which can be seen in Figure 2. The screenshot demonstrates the current iteration of the UI with control panel on the left.



Natural Language Interface

Please choose desired domain from the drop down menu, then enter free-form natural language expression or query by typing in text box indicated below.

Select Domain: madsat

Show Salute platforms from NAIs with mobility nogo.

SELECT salute.Platform FROM naimobility JOIN salute ON
SELECT salute.Platform FROM naimobility JOIN salute ON
SELECT salute.Platform FROM salute

Query Results

madsat:SELECT salute.Platform FROM naimobility JOIN salute ON naimobility.NAI = salute.NAI WHERE naimobility.Mobility = 'NoGo'

Platform	Source
UAV	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
UAV	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
UAV	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
JSTARS	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
Bde. Scouts	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
Bde. Scouts ??	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
HQ(CI)	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
Counter Intelligence Team	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent
CI	localhost jdbc:mysql#Agent joined with localhost jdbc:mysql#Agent

Google Search Results for Query:
Show Salute platforms from NAIs with mobility nogo.

http://stackoverflow.com/questions/20870428/regex-and-converting-strings-to-array-and-back-and-forth-javascript	regex and converting strings to array and back and forth Javascript - Stack Overflow	"NL:Show all platforms of Salute generated from NAIs with no go mobility."] I then want to get rid of all patterns matching NL: so I am left with justÂ ...

Status Summary

Current as of:
Fri Jan 02 2015 11:39:03 GMT-0500 (Eastern Standard Time)
Total Queries Planned: 20
Queries Executing: 12
Queries Completed: 4
Percent Complete: 33.33 %
Elapsed Time of Run:
Number of Records Returned: 19

```
            madsat
        </database>
        <host>
            <server>
                cloudServer
            </server>
            <ip>
                192.168.0.103
            </ip>
            <port>
                2181
            </port>
        </host>
        <login>
            <user>
                root
            </user>
        </login>
        <wrapper>
            Accumulo
        </wrapper>
        <sql>
            SELECT naimobility.NAI, naimob
        </sql>
        <create>
            CREATE TABLE tempSelect1 ( NAI
        </create>
    </query>
    <query>
        <table>
            tempSelect2
        </table>
        <database>
```

Fig. 2. DAS web-based use interface

The user will select first a domain from a dynamic list of possible domains. This list is populated at load time based on output from the DAS application that the interface accesses via an AJAX call. Once a domain is selected, the user will begin typing a natural language query in the textbox below the domain selection. After a query has been completed the user will click the translate button displayed below the query textbox as shown in Figure 2. This initiates an AJAX call to the DAS translation class, which in turn makes calls to internal dependencies that will translate natural language query into SQL.

DAS automatically translates a natural language query to its equivalent SQL representation to be executed against structured data (e.g. Giordani and Moschitti, 2012). We are making use of the publicly available Stanford parser and the dependency relations (de Marneffe, et al., 2010) that it generates from a given sentence representing a user query in the context of a given database. The algorithm also makes use of the underlying database scheme and its content. The algorithm exploits the structure of the database to generate a set of candidate SQL queries, which we rerank with a heuristics based ranking algorithm developed in-house. In particular we use linguistic dependencies in the natural language question and the metadata to build a set of plausible SELECT, WHERE and FROM clauses enriched with meaningful joins.

Once the translation is complete, possible SQL queries are returned to the user via another AJAX callback. DAS returns all of the possible translations of the original natural-language-like query, using a proprietary algorithm to rank the translations. The list of translated queries that the user is presented with is displayed in rank order as shown in Figure 2. However, the "correct" translation in terms of relevancy is not always ranked highest due to the ambiguity of natural language.

The user can select any translation (first one is by default) by clicking on it and then click the execute button below the list of SQL translations. This action initiates AJAX calls to DAS classes responsible for planning and executing the query using direct cloud based queries to nodes on the network and agent based queries to nodes on the network where this is appropriate. At this point DAS starts by preparing an execution plan whereby subqueries are created and optimized prior to execution. In the UI presentation layer, the user is presented with the XML-based plan that DAS will execute. Figure 2 illustrates a portion of this plan, hiding a significant portion, which the user is able to scroll through both horizontally and vertically if desired to examine the order of execution.

Once the plan has been created by DAS, we will know how many queries will be executed at a maximum, and this number will be presented to the user. In some instances the number of queries planned will not be the same as the number of queries executed. This is primarily due to the fact that some nodes may be unavailable when contacted by an agent. Since it is a basic assumption that nodes will be or become unavailable for querying, DAS can and does continue the execution on available nodes. When this occurs we believe it is relevant to the user to know that not all nodes can be queried at the moment. The user is presented with a new statistic so s/he are alerted to the fact that not all queries will be executed and by extension that some nodes on the network are not available. However should unavailable nodes become available during the course of execution, they will be included in the execution. It should be possible to provide the user with a list of unavailable nodes in future iterations. In Figure 2, in the status summary panel, we can see that 20 queries were planned in this run, but only 8 actually executed, with partial results displayed in the Query Results panel.

Continuous communication between the UI and DAS is maintained via AJAX call throughout the execution process, and as soon as 200 results are available, they are displayed to the user. DAS continues to run and the user is presented with updates on the status of the query. When the user clicks the Next or Prev button a graphic is displayed to indicate that new results are being fetched.

Based on the number of queries that will execute, the user is also presented with a near-real-time "percent complete" statistic and graphic. This graphic and number are replaced with the word "Completed" once all results are available. It should be possible to provide the user with an estimate of time to completion as well. The user is currently given the ability to toggle through results by means of "Next" and "Prev" buttons. Additionally a link is provided to the directory where result files are stored, should the user with to view or download raw result files.

3 Query Planning and Optimization

Query planning (Das et al., 2002 & 2005) involves generating a set of sub-queries from a given user query based on the data source locations that have parts of the required information to answer the query. The optimization process then generates an efficient ordering of execution among these sub-queries. We first create an example to illustrate the concept of query planning and optimization.

Once a natural language query is translated into its equivalent SQL query, we automatically decompose the output SQL query into a query plan composed of subqueries to be executed at distributed sites where data reside. Our implementation makes use of the two tables, Sites and Columns. The table Sites stores the physical location of tables and the table Columns stores the descriptions of columns and the user privileges.

We have focused on planning and optimizing "select-before-join" type of queries as shown below. Below is an example of this type of planning and optimization. The query here (a translation of the original query posed in natural language via the web interface) finds the equipment/vehicles that are operating in a 'no go' named area of interest (NAI):

```
select s.equipment, t.mobility
from s in salute, t in nai-mobility
where s.location = t.location and t.mobility = 'no go'
```

The optimization technique helps to identify the selection sub-query as follows to generate a temporary intermediate relation:

```
select t.mobility
from t in nai-mobility
where  t.mobility = 'no go'
```

The executive agent sends an agent to execute the query at the site where terrain mobility information by NAIs is located. The results are then carried by two other agents in a temporary relation to the two sites of the SALUTE databases. The same query that are executed at the two SALUTE data sites are as follows:

```
select s.equipment, temp.mobility
from s in salute
where s.location = temp.location
```

The results are brought back by the agents and merged and presented to the user via the user interface. This kind of optimization avoids downloading the join relations to the user's local environment.

Our target is general query planning and optimization beyond just the limited optimization described above. Consider a family of surveillance platforms (e.g., JSTARS, UAV, and AWACS) and assume that an extraordinary tactical event is reported (e.g., enemy tank T-80 is identified at the named area of interest NAI-68) in the SALUTE format prepared from the UAV mission during the interval (t1, t2). For an analysis through comparison, the analyst needs to access the intelligence data of that location for the interval (t1, t2) from other surveillance platforms as well as the information about terrain and weather during that period. The query involves access from various repositories containing intelligence and environmental data. A high-level user query to retrieve only the intelligence data in this regard will look like the following:

```
select s.*
from s in salute
where s.location = 'NAI-68' and
s.time =< t1 and t2 =< s.time
```

Note that neither the repository nor the wrapper is mentioned in the query. If salute0 and salute1 are the only two tables respectively at repositories r0 and r1 containing SALUTE reports from the surveillance platforms, the above query will be translated as follows:

```
select s.*
from s in union {salute0, salute1}
where s.location = 'NAI-68' and
s.time =< t1 and t2 =< s.time
```

Given the fact that repositories r0 and r1 are at different locations, the following two sub-queries will be generated corresponding to the above query:

```
select s.*                           select s.*
from s in salute0                    from s in salute1
where s.location = 'NAI-68' and      where s.location = 'NAI-68' and
s.time =< t1 and t2 =< s.time        s.time =< t1 and t2 =< s.time
```

The above two sub-queries will be executed in parallel through wrappers w0 and w1 respectively. Not every sub-query will return a result, because the SALUTE report within a repository might not contain a reading of the surveillance platform s at that particular time interval (t1, t2). We generate an efficient query execution order based on several traditional query optimization strategies including a typical "select before join" type

4 Agent-Based Query Execution

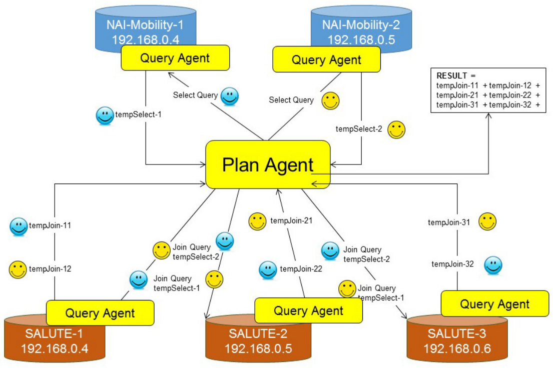


Fig. 3. Plan Agent spawning Query Agents processing information from several databases

The final step in carrying out a user's request for data is performed by the Query Execution module. The Query Execution module controls all aspects of agent creation, migration, data retrieval, and collaboration. These topics will be discussed in the following subsections. The module receives a list of sub-queries from the Planning and Optimization systems and generates a series of mobile agents to carry out these sub-queries. For each agent, the module creates an itinerary of the various sites to be visited and the data retrieval and processing tasks to be executed at each site. Each mobile agent is then spawned and the system waits for the return of each agent with its associated data. Upon return, the system performs any required data joining, processing, and formatting before displaying the results to the user.

Our mobile agent approach as shown in Figure 3 created multiple Plan Agents and Query Agents as part of the Query Execution module. These mobile agents were built on top of the Aglets 2.02 API along with Tahiti server running on the Java 1.7. But we now have replaced Aglets with our in-house mobile agent platform. Aglets is a Java mobile agent platform and library. An aglet is a Java agent that is able to autonomously and spontaneously move from one host to another. The Plan Agents and Query Agents inherit the properties of an Aglet.

Different types of execution mobilities exist (Jansen and Karygiannis, 1999) corresponding to the possible variations of relocating code and state information, including the values of instance variables, the program counter, execution stack, etc. For example, a simple agent written as a Java applet has mobility of code through the movement of class files from a web server to a web browser. However, no associated state information is conveyed. In contrast, Aglets, developed at IBM Japan, builds upon Java to allow the values of instance variables, but not the program counter or execution stack, to be conveyed along with the code as the agent relocates. A stronger form of mobility allows Java threads to be conveyed along with the agent's code during relocation. DAS design allows relocation of code information and state information.

Detailed architectural diagrams of the Query Execution module will be shown and discussed in the next subsection.

4.1 Query Execution Architecture

Figure 4 (left) shows the diagrams of the Query Execution Module. The two main parts are JSP Server and the Aglets Agent Servers. The Query Execution Module integrates with the Web-based Analyst Interface component and the Planning and Optimization System Module. A user-submitted natural query will be processed by the JSP Server and passed on to the Planning and Optimization systems.

Planning and Optimization systems are customized Java Objects that can process the transformation from a Natural Language Query and produce a plan of action in XML format. The user may then choose a desired transformation SQL and pass it back to the JSP Server to create a plan of action in XML format. The XML file that was created will be processed by the Plan Agent as shown in Figure 4 (right). The figure also shows the roles of the Agents that were customized from the Aglets API. The Plan XML file was read and processed. The Plan Agent creates Query Agents based on the number of queries obtained from the plan XML file. This XML file contains a plan of action created from a catalogue of available databases. Changing the availability of databases in the catalogue will reflect on the plan created in XML.

The Query Agents are then dispatched to the remote computers containing the desired databases. The Query Agents perform all computations locally where the databases reside. Query Agents can be sent to remote machines and process SQL commands to different databases on those machines. The databases that we used for testing were MySQL and Derby. One of the advantages of using agents is that the database needs not be open to outside connection. Since the agent had been sent to the remote machine, the agent has the ability to query the database locally. Query Agent also has the ability to create temporary database tables and carry out any standard SQL command.

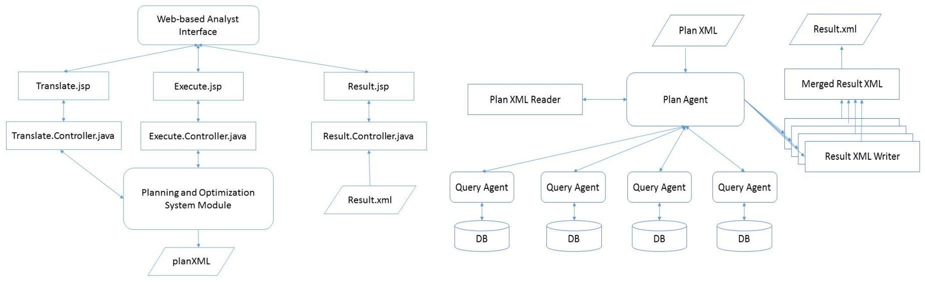


Fig. 4. (left) Query Execution architecture part 1 (JSP Server); (right) Query Execution architecture part 2 (Aglets Agent Servers)

We designed custom codes with the assumption that we have sufficient privileges to modify one or more databases involved in the query as well as permissions to read the corresponding tables across the network. These written codes have automated access to user defined queries obtained from the Planning and Optimization systems. The combined processed results, according to the query plan, from heterogeneous data from multiple sources are sent back to the Plan Agent, who will then save them into an XML format. The resulting XML files are visualized as single or multiple merged results.

4.2 Agent Creation

Plan Agent was created by inheriting the properties of an Aglet. The Aglet class is provided by the Aglets API. Aglets need to be hosted by an agent host such as a Tahiti server. Plan Agent was instantiated within an Aglet Context that performs the role of sending messages to other Agents. The Aglet Context was created by the Tahiti Server which has a network daemon whose job is to listen to the network for other agents. Incoming agents are received and inserted into the context by the daemon. The Context provides all agents with a uniform initialization and execution environment.

One of the challenges faced by mobile agents is that a Tahiti server with the ability to host a query agent needs to be present in the destination database machine. To respond to this challenge we have developed the option for the plan agent to create query agents that will have the ability to query multiple databases without dispatching them to the different machines. Having this additional feature will enable the Plan Agent to combine results from machines that can host Agents as well as machines that cannot host Agents. One of the test scenarios involved multiple databases residing on different servers with different database software. For example we have four servers on machines a, b, c, and d, with MySQL, Derby, CloudBase, and Accumulo, and with different operating systems such as Ubuntu and Windows. Different servers refer to distinct physical machines or multiple virtual machines using different physical hardware. The key point is that there are multiple installations of the database software.

Another challenge that the query agents face includes what to do if the destination machine is temporarily unavailable. Should the program crash, or should it proceed and produce partial results from other available machines? The Query Agents had been designed to detect if the destination machine specified on the plan XML file has become unavailable. The program will complete, produce fewer results, and report to the user that some database sites are unavailable. Destination machines may become unavailable due to loss of network connections, availability of the Tahiti Servers, power outage, or incorrect or change of user access at a particular site, among other reasons. The Query Agent will send a message to the Plan Agent regarding the unavailability of the destination host. The final result will consist of a single result XML file containing the merged results obtained from the available databases. The user interface will display the results on the web browser as a result table. Status reports including availability or unavailability of databases have been made available to the user. Sources of information are also displayed as part of the results.

4.3 Agent Migration

The Plan Agent can create, monitor, coordinate, retract, dispatch, and dispose Query Agents as needed. A Query Agent can be dispatched to a specific host (which itself hosts a database on the network) to visit and perform a specific function, computation, or query. Once an agent completes its tasks, it can send messages to other agents to perform other tasks such as creating temporary database tables or merging query results from different database tables. Agents also send messages to other agents to verify that they have reached their destinations and have completed their tasks. The Plan Agents have the ability to decide what path to take and what actions to perform as they gather data from the nodes that the Query Agent visits.

The Plan Processor reads XML files and stores the information in the form of Serialized objects (Java classes that can be converted into bytes and be sent over the wire). The instance of this class is saved and can be restored upon arrival to a destination. Serialization allows the persistence of an object from memory to a sequence of bits, and deserialization enables the reading of the data to recreate the object.

Plan Agent will create multiple Query Agents that can calculate and carry vital information while “hopping” to and from different machines. The number of Query Agents created depends upon the number of queries in the XML document. Multiple queries can be processed in parallel or sequentially in a distributed manner. Query Agents are deployed to different machines based on the plan XML file to process information from the remote databases. MySQL and Derby Test Databases were configured and used for testing.

4.4 Agent Retrieval

Using agents, it is possible to leave data where it resides and to only extract the required data on demand. The user writes a query in his own words and submits it using the web based user interface. From the user's perspective, one query produces one combined answer and the complexities of the process have been hidden. The original data has not been moved nor modified. Only relevant data had been extracted and passed through the network.

Several databases were loaded with gigabytes of data. A Plan Processor Java Object was designed and implemented to enable carrying huge data streams across the wires. A new scenario was developed and a series of tests were carried out to query new tables containing large amounts of data with a huge number of results that were carried across the wires. The testing was successful and gigabytes of data were obtained from a remote computer.

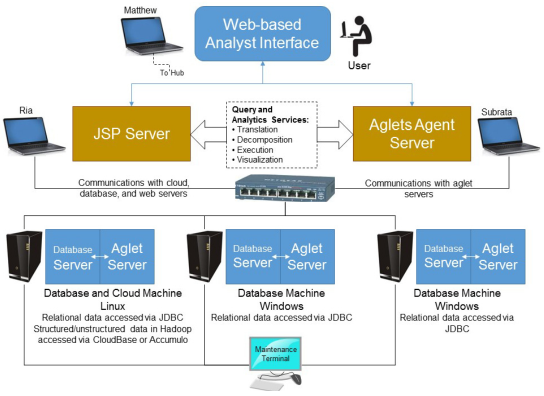
The Plan Agent has the ability to create Query Agents that can travel autonomously through the network, providing an increased fault tolerance. The agents' ability to travel through the network and carry data along with them enables these agents to individually process queries in parallel and/or in sequence. The query execution module will not crash with a single point of failure and the query process may continue even if individual machines fail or become unavailable.

New computers or new database source may be added to the network. This feature offers better scalability of the module. We have created a data site table stored where users may add or delete existing data sources. The Plan Agent has the ability to automatically increase the creation of Query Agents that can be dispatched to different computers. The ability to have the Query Agents travel through the system and execute their code using the host's resources allows for dynamic load sharing and automatic data processing.

5 Experimental Results

Figure 5 shows the DAS demo implementation environment that we have created. We have set up three database servers to emulate storing and serving big data from a variety of environments, including Hadoop-based cloud and a traditional database server. These servers are connected via a router providing fixed IP addresses to these servers, thus creating local area network. The servers are connected by a common maintenance terminal for configurations.

We have also developed the option to directly query the databases specified on the plan XML document without sending



Matthew
Web-based Analyst Interface
User
To Hub
Ria
JSP Server
Query and Analytics Services:
• Translation
• Decomposition
• Execution
• Visualization
Aglets Agent Server
Subrata
Communications with cloud, database, and web servers
Communications with aglet servers
Database Server
Aglet Server
Database and Cloud Machine
Linux
Relational data accessed via JDBC
Structured/unstructured data in Hadoop accessed via CloudBase or Accumulo
Database Machine
Windows
Relational data accessed via JDBC
Database Machine
Windows
Relational data accessed via JDBC
Maintenance Terminal

Fig. 5. Agent collaboration

the Agents to the remote locations. A comparison between direct querying and the sending the Agents was developed. Sources of Query Agent delay were found and the code has been restructured to eliminate or minimize runtime inefficiency.

The table below shows a comparison between distributed and centralized database as well as direct parallel querying and sending the Agents to remote locations. It took less than half the time to retrieve 2.1 million records from three distributed databases than the same amount of records from one centralized database. There is not much difference between the direct parallel approaches as opposed to sending the mobile agents remotely.

	Direct	Mobile Agent
Centralized DB (2.1M records returned)	7 min 43 sec	7 min 32 sec
Distributed in 3 DB (0.7M each) (2.1M records returned)	3 min 22 sec	3 min 12 sec

Plan Agents and Query Agents have been configured to run on both Windows and Ubuntu Operating systems. MySQL containing huge amounts of data has also been installed on both platforms. The Ubuntu machine had been expanded from its current capacity of 30 gigabytes storage to 450 gigabytes of space to accommodate big data for traditional SQL databases and Hadoop, CloudBase, and Accumulo. Precautions were taken to ensure that the original data were protected, and the expansion was been carried out without loss of data. Precautionary measures included backing up relevant files and information. Testing is being done on different machines to ensure that the DAS system can operate in a heterogeneous environment.

The ability to have multiple clients querying from different browsers or different machines has been designed and implemented. A unique session directory is created when a user chooses a particular translated query to execute. The plan XML document and all the other relevant documents that are related to this particular query will be contained in this unique session directory. Relevant files include the status XML file and the partial and merged results.

The limits of our system have been continuously subjected to stress testing by sending huge data results across the wires. Gigabytes of data have been loaded across several data sources. Up to 3 million result objects per remote data source have been sent through the wires. There were no issues with using the mobile Agents and we run into heap space issues with the direct approach. Major refactoring was implemented to accommodate the migration of huge data results into different machines. We continue to encounter heap space issues as we increase data and several steps were taken to improve. Memory management is continuously monitored and managed.

The DAS Agents are constructed as lightweight processes, so that each process tests a single vulnerability. As new vulnerabilities are detected and tests for these vulnerabilities are developed, new agents can be added to the test suite. As the system configuration changes, some agents can be retracted or disposed of if they are no longer needed. Test suites can be fine-tuned for each individual node depending on its configuration. This increases the efficiency of the testing as tests are performed only when and where they are needed. A lightweight agent architecture makes the test suite configurable for heterogeneous environments.

6 Conclusions and Future Directions

Our agent-based approach to distributed analytical search offers several advantages: 1) Databases need not be open to outside connections. Since the agent has been sent to the remote machine, it has the ability to query the database locally; 2) Network bandwidth usage is reduced because the Mobile agent moves computation code to where the data resides; 3) The agents do not require a continuous connection between machines and the clients can dispatch an agent into the network when the network connection is healthy, and then it can go off-line. The network connection can be reestablished later when the result from the remote host is ready; and 4) Agents operate asynchronously and autonomously and the user doesn't need to monitor the agent as it roams the internet. This saves time for the user, reduces communication costs, and decentralizes network structure.

Future developments include researching possible security issues. We will investigate the possibility of creating cooperating agents that can help reconfigure the network to deny network services to certain nodes until they have been confirmed to be in a safe state. Query Agents can monitor network events and cooperate with the Plan Agents. For example, if one of the Agents detects suspicious activity on one computer and notifies the rest of the network, the other agents may decide to challenge the nodes by modifying the rights given to those agents.

Real time status reports will be continuously improved. We are researching means to show the user a more detailed report on why data may or may not be available as well as how long it will take to get data. The percentage of completion will be calculated as well as information on particular queries that will be abandoned because of the unavailability of the database or its dependent database. The detailed status report will also show whether an agent was available in the remote machine or a direct query had been implemented.

More testing will be developed to ensure the robustness of the application. New scenarios will be created for testing and more data sources will be explored, including finding data that are publicly available through the internet. Different testing mechanisms will be studied in more detail to enable the system to have flexible capabilities. New scenarios will be considered to test the limits of performance. Simultaneous querying using multiple client machines will be tested and smarter Agents will be designed and developed to operate on both Windows and Ubuntu Systems.

References

Das, S., Shuster, K., Wu, C.: ACQUIRE: agent-based complex QUery and information retrieval engine. In: Proc. of the 1st Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, July 2002

Das, S., Shuster, K., Wu, C., Levit, I.: Mobile Agents for Distributed and Heterogeneous Information Retrieval. Journal of In Retrieval **8**, 383–416 (2005). Springer Science

Das, S.: Computational Business Analytics. Chapman and Hall/CRC Press (2014)

de Marneffe, M.-C., et al.: Stanford typed dependencies manual: Revised for Stanford Parser v. 1.6.5 (2010)

Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communication of the ACM **51**(1), 107–113 (2008)

Giordani, A., Moschitti, A.: Generating SQL queries using natural language syntactic dependencies and metadata. In: Bouma, G., Ittoo, A., Métais, E., Wortmann, H. (eds.) NLDB 2012. LNCS, vol. 7337, pp. 164–170. Springer, Heidelberg (2012)

Jansen, W., Karygiannis, T.: Mobile Agent Security, NIST Special Publication 800-19 (1999)

Widom, J.: Integrating Heterogeneous Databases: Lazy or Eager?. ACM Computing Surveys **28** (1996)