Querying Combined Cloud-Based and Relational Databases

Minpeng Zhu and Tore Risch
Department of Information Technology, Uppsala University, Sweden
Minpeng.Zhu@it.uu.se Tore.Risch@it.uu.se

Abstract—**An increasing amount of data is stored in cloud repositories, which provide high availability, accessibility, and scalability. However, for security reasons enterprises often need to store the core proprietary data in their own relational databases, while common data to be widely available can be stored in a cloud data repository. For example, the subsidiaries of a global enterprise are located in different geographic places where each subsidiary is likely to maintain its own local database. In such a scenario, data integration among the local databases and the cloud-based data is inevitable. We have developed a system called BigIntegrator to enable general queries that combine data in cloud-based data stores with relational databases. We present the design and working principle of the system. A scenario of querying data from both kinds of data sources is used as illustration. The system is general and extensible to integrate data from different kinds of data sources. A particular challenge being addressed is the limited query capabilities of cloud data stores. BigIntegrator utilizes knowledge of those limitations to produce efficient query execution.**

Keywords: cloud data repository; relational database; data integration; Bigtable;

I. INTRODUCTION

Cloud based repositories such as Google's Bigtable [1] allow widely accessible distributed data stores to be queried by the query language GQL [7]. This is done by web-based applications managed by the Google App Engine (GAE) [10]. GAE provides an application environment and query language to manage data stored in Google's cloud. It is easy to write web-based applications that access and update these cloud-based databases.

Cloud repositories such as Google's Bigtable are particularly useful to store data that has to be globally available. For example, in industrial settings, machines such as engines, trucks, cutting tools, etc., produce many different kinds of data and the machines are often geographically widely distributed and maintained locally. To check that distributed equipment works properly, it is crucial to analyze its working status by searching the data produced by the equipment. Since the equipment is widely distributed, properties about the equipment should be stored in an environment that provides high availability and universal access, such as a cloud-based data store.

Relational database systems (RDBMSs) have the limitation that they must run in some central server site and therefore require substantial maintenance efforts to provide high availability. As an alternative approach, we propose to store common data, for instance equipment properties, in a cloud-based data store, such as Bigtable. By using such a cloud service the data becomes universally available and can easily be maintained. However, the data stored in the cloud often needs to be combined with data stored in regular databases. For example, cloud-based data is used for finding the locations of a particular machine, while the information about the machines' operating environments is stored in local relational databases. A maintenance engineer may wish to make queries combining relational data with the cloud-based data. To enable this, there is need for a system supporting queries combining cloud-based data and data in relational databases. We have developed such a system, BigIntegrator, to transparently process queries combining data stored in Bigtable data stores and data stored in relational databases.

BigIntegrator utilizes a novel query processing mechanism to provide easy extension of data integration from different kinds of data sources. The mechanism is based on plug-ins called *absorbers* and *finalizers*. The limited expressiveness of GQL has to be taken into account by BigIntergrator's query processor, which is the challenge being addressed by the absorbers and finalizers.

GQL has some similarities with SQL but has very limited query expressions in order to provide for scalable processing. BigIntegrator can process queries to such data sources with limited back-end query languages support. The absorber and finalizer for Bigtable data sources know the limitations of GQL and will pre and post-process those operations that cannot be processed by the data sources. For this, BigIntegrator generates integrating execution plans containing calls to relational databases, Bigtable data stores, and local operators.

In summary the contributions of our work are:

- The BigIntegrator system provides query capabilities over combined cloud-based and relational databases.
- A novel query processing mechanism based on plug-ins for absorbers and finalizers is developed to allow easy extensions for each new kind of data source that provide a restricted query language.

978-1-4577-1637-9/11/$26.00 ©2011 IEEE

- A client-server architecture for scalable querying of Bigtable data repositories is developed.

The rest of the paper is organized as follows: Section II discusses related work. Section III illustrates the system by a scenario from an industrial equipment point of view. Section IV overviews the BigIntegrator system architecture and describes its query processing. Conclusions and future work are described in section V.

II. RELATED WORK

There are several cloud-based storage systems available, such as Dynamo [8], PNUTS [5], and Bigtable [1]. These systems have very limited query languages as a compromise for very high scalability. The restricted queries do not allow joins and there are restrictions on how to specify the query conditions. In contrast, the BigIntegrator pushes as much query processing as possible to the data sources and compensates the lacking query capability of a data source by doing post-query processing with its own query engine. Similar approaches can be applied on [8, 5] as well.

Some cloud-based storage systems such as Cloudy [3] provide rather complete SQL capabilities. It offers key-value, SQL, and XQuery interfaces to manipulate its cloud data. Microsoft SQL Azure [2] offers full SQL language support for its cloud-based relational database. Unlike Cloudy and SQL Azure, the purpose of BigIntegrator is to allow joining of data from a restricted cloud-based data store such as Bigtable with relational databases, by generating execution plans that combine queries sent to the data sources.

Unlike classical work on mediator/wrapper techniques over conventional databases such as [4], BigIntegrator provides data integration between cloud-based data repositories and relational DBMSs. Furthermore, a novel query plug-in mechanism based on absorbers and finalizers is developed to provide easy extensions for new kinds of data sources providing restricted query languages.

To conclude, most work on cloud-based databases concentrates on providing scalability, availability and consistency as storage services inside a cloud. No other system addresses the problem of integrating data from cloud-based databases having restricted query languages with relational databases. We show the extensibility of the system and the advantages of its novel query plug-in mechanisms.

III. SCENARIO

In this section, we present a scenario combining data from Bigtable and a local relational database. An enterprise is responsible for maintaining geographically widely distributed industrial equipment. Some generally available data about the equipment is stored in a cloud repository, while data about local personnel is in relational databases. BigIntegrator enables queries combining these databases.

The database schema for the cloud based database is shown in Fig. 1 and for the relational one in Fig. 2. The cloud table *Machine(Model, Name, Manufacturer)* stores general data about industrial machines such as its model identifier, name, and manufacturer. The table *Site(SID, Name, Country, Region)* stores information about each site such as site ID, its name, and the country and region where it is located. The table *MachineInstallation(MID, Model, SID)* stores information about each installation of a machine at some site, i.e the identifier of the machine, its model, and the identifier of the site where it is located (*SID*). The attribute *Model* is foreign key from *MachineInstallation* to *Machine* and the attribute *SID* is foreign key from *MachineInstallation* to *Site*. The tables *Machine*, *Machineinstallation*, and *Site* provide globally accessible common data and are therefore stored in the cloud.

A country maintains its local personnel database in the relational database in Fig. 2. The table *Operator(PID, Name, Skill, Operates)* stores the identifier of a machine operator along with his name, specialty, and the machine he is currently operating. The attribute *Operates* is foreign key from the local database table *Operator* to the cloud database table *MachineInstallation*. Fig. 3 shows all the tables in this scenario with populated data.

Machine(Model, Name, Manufacturer)
MachineInstallation(MID, Model, SID)
Site(SID, Name, Country, Region)

Figure 1. Cloud database schema

Operator(PID, Name, Skill, Operates)

Figure 2. Relational database schema

Model	Name	Manufacturer
1	M1	Volvo
2	M2	Volvo
3	M3	Volvo
4	M4	Volvo
5	M5	Volvo

Machine table

SID	Name	Country	Region
1	Uppsala	Sweden	Uppland
2	Chengdu	China	Si Chuan
3	Campinas	Brazil	Sao Paulo
4	Chapaevsk	Russia	Samara
5	Monki	Poland	Bialystok

Site table

MID	Model	SID
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	1	1
7	2	2
8	3	3
9	4	4
10	5	5

MachineInstallation table

PID	Name	Skill	Operates
1	John	Operation	1
2	Oliver	Operation	2
3	Bruce	Operation	3
4	Carl	Operation	4
5	Thomas	Operation	5
6	Jens	Operation	6
7	Lucas	Operation	7
8	Alex	Operation	8
9	Ryan	Operation	9
10	Wes	Operation	10

Local table Operator

Figure 3. Scenario database schema

The following is an SQL query to BigIntegrator that combines data from the cloud-based tables at a data source named *A* (the Bigtable data source) and the relational database table at data source named *B* (the country's local data source):

```
select i.Mid, o.Name
from Machine_A m, MachineInstallation_A i, Site_A s, Operator_B o
where m.Name = 'M1' and
      m.Manufacturer like 'V%' and
      s.Region = 'Uppland' and
      s.Sid = 1 and
      m.Model = i.Model and
      i.Sid = s.Sid and
      i.Mid = o.Operates
```

The query retrieves identities of machines of model "M1" along with the operators' names, where the machines' manufacturer names starts with "V", the machines are installed in the region "Uppland", and the site code is equal to one.

Every time a data source is accessed the system automatically generates a set of relations called the *source predicates* representing the collections inside the source. The source predicates are references as tables in the SQL queries. In the example there are the source predicates *Machine_A, MachineInstallation_A, Site_A,* and *Operator_B*. The name of each collection in a source named *X* is suffixed by "*_X*". After the query is executed, BigIntegrator returns the following query result:

1, John 6, Jens

IV. SYSTEM OVERVIEW

The BigIntegrator system architecture is shown in Fig. 4. The system contains two sub-systems: The *RDBMS wrapper* and the *Bigtable wrapper*. A wrapper needs to be implemented for each *kind* of data source to be queried from the BigIntegrator system. The RDBMS wrapper generates SQL queries sent to a back-end RDBMS, while the Bigtable wrapper generates GQL queries to data stored in Bigtable.

The system receives SQL queries, which are processed to generate a query execution plan that contains calls to the underlying relational and Bigtable databases. The wrapper modules have plug-ins that know how to generate queries to each kind of data source.

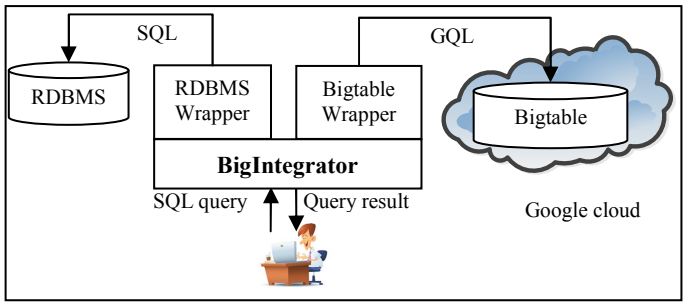

Figure 4. BigIntegrator architecture

A. BigIntegrator Wrappers

Fig. 5 shows the components of a wrapper definition for a BigIntegrator data source.

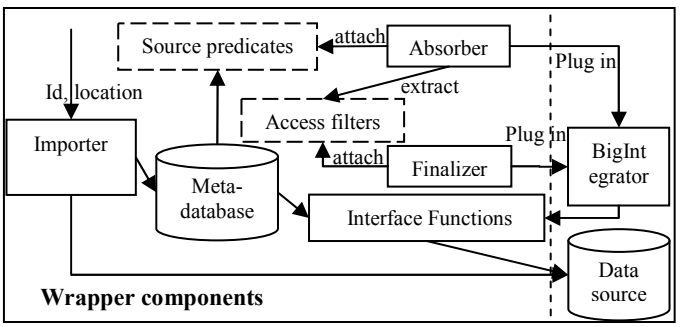

Figure 5. The wrapper components

For each new kind of data source the components *importer*, *absorber*, *finalizer*, and *interface function* have to be developed. Once a wrapper is defined any data source of that kind can be wrapped by creating a source identifier *id* for the source and then calling a system procedure *import(id,location),* which accesses the location and imports system catalog data to the local *meta-database*. When a data source is wrapped it can be used in SQL queries and joined with other wrapped data sources.

Each data source can contain many collections presented to the system as *source predicates*. The importer creates the source predicates and stores them in the local meta-database. Each wrapper has one *absorber,* which is a plug-in that from a user query extracts a subquery, called the *access filter*. It selects data from a particular source predicate, based on the capabilities of the source. Each wrapper also has a *finalizer,* which is a plug-in that translates each access filter in the plan to an algebra operator called an *interface function,* specific for each kind of source. The interface function sends a query to the data source (i.e. a GQL or SQL query).

B. The BigIntegrator query processor

The steps of the query processor in BigIntegrator are shown in Fig. 6.

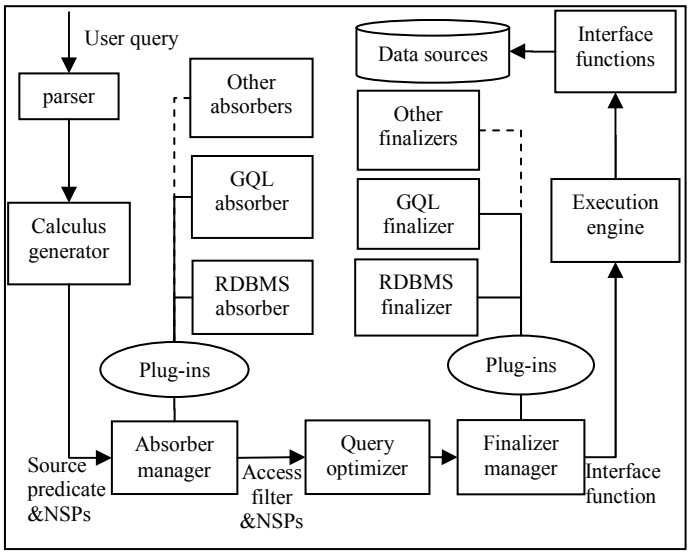

Figure 6. Query processing in BigIntegrator

The *parser* translates the SQL query into a parse tree, which the *calculus generator* transforms into a Datalog [6] query. The Datalog query will contain both source predicates and *non source predicates (NSPs)*. The *absorber manager* takes the Datalog query and, for each source predicate referenced in the query, calls the corresponding absorber of its wrapper. In order to replace the source predicate with an access filter, the absorber collects from the query the source predicates and the possible other predicates, based on the capabilities of the data source. The *query optimizer* reorders the access filters and other predicates to produce an algebra expression containing calls to both access filters and NSP operators. The

finalizer manager takes the algebra expression and, for each access filter operator referenced in the algebra expression, calls the corresponding finalizer of its wrapper. The finalizer transforms the access filters into interface function calls. To access the different data sources, the *execution engine* interprets the finalized algebra expression calling the interface functions.

The example query is transformed by the parser and calculus generator into the following Datalog query:

```
Query1(mid, name3) :-
Machine_A(model,name1,manufacturer) AND
MachineInstallation_A(mid,model,sid)AND
Site_A(sid, name2, country, region) AND
Operator_B(pid,name3,skill,operates)AND
name1 = 'M1' AND
manufacturer like 'V%' AND
region = 'Uppland' AND sid = 1
```

The NSPs are in bold phase. Unique variable names are generated when needed, e.g. *name1, name2* and *name3*.

In this example, the GQL absorber for the source predicate *Machine_A(model, name1, manufacturer)* will absorb *name1 = 'M1'* since the predicate = can be handled by a GQL data source and both predicates share the same parameter *name1*.

The capabilities of a data source can vary widely, e.g. joins are allowed in RDBMS data sources but not in Bigtable data sources. If joins are allowed, as in SQL, an access filter is formed as a conjunction of all relational source predicates and supported NSPs. If joins are not allowed, as for Bigtable sources, each source predicate forms its own access filter based on GQL language constraints.

The access filters are represented as Datalog rules. In the example there will be one access filter created for each of the source predicates *Machine_A* (filter *F1*), *MachineInstallation_A* (filter *F2*), *Site_A* (filter *F3*), and *Operator_B* (filter *F4*):

```
F1(model,name1,manufacturer):-
  Machine_A(model,name1,manufacturer) AND
  name1='M1'

F2(mid,model,sid):-
  MachineInstallation_A(mid,model,sid)AND
  sid = 1

F3(sid,name2,country,region):-
 Site_A(sid,name2,country,region) AND
 region ='Uppland' AND sid =1

F4(pid,name3,skill,operates):-
Operator_B(pid, name3, skill, operates)

Query1(mid,name3):-
F1(model,name1,manufacturer) AND
F2(mid,model,sid) AND
F3(sid,name2,country,region) AND
F4(pid,name3,skill,operates) AND
manufacturer like 'V%'
```

The possible NSPs are placed in all the access filters for which they have a shared source predicate parameter. For example, *Sid = 1* is placed in both *F2* and *F3*. In other word, the NSPs can be absorbed into one or several access filters.

If an NSP cannot be placed in any access filter, it will remain as a separate predicate in the query and post-processed by BigIntegrator. In the example, *Manufacturer like 'V%'* remains as a separate predicate even though it shares variable *manufacturer* with the GQL access filter *F1,* since GQL does not support *like* predicates. An absorber contains rules about what NSPs can be absorbed into the access filter according to the query capability of the data source. GQL queries have the following restrictions [7]:

> Suppose A, B, and C are attributes names of a table in a GQL data source, and x, y, a, and z are constants or strings. Then the following *where* clauses of a GQL query are allowed:
> where A = x
> where A < x
> where A > x and A < y
> where A > x and A < y and B = z
> where A > x and A < y and B = z and C = a etc

Accordingly, we define the following heuristic algorithm for the GQL absorber:

1. Absorb all equalities having one variable in common with the source predicate while the other parameter is known.
2. Absorb the first inequality having one variable in common with the source predicate also having the other parameter known.
3. If an inequality is absorbed in 2. then also absorb the first inverse inequality for the same variable.

Unlike GQL, SQL can handle joins. Therefore, the absorber for the RDBMS wrapper absorbs several source predicates to produce joins. This is not elaborated here.

The access filters (F1, F2, F3 and F4) and the NSPs that cannot be absorbed into any access filter, are combined into a conjunctive form and sent to the *query optimizer* for optimization. A greedy query optimization method [6] is employed to find an optimized plan fast.

The finalizer manager takes the optimized algebra expression and, for each access filter referenced in the algebra expression, calls the finalizer of the access filter's wrapper. The finalizer translates the access filter into an interface function call to the source.

In the final plan, BigIntegrator's query execution engine calls the interface functions. An interface function sends the query to a data source for execution. For the example query, the finalizer manager finalizes the query execution plan shown in Fig. 7. Bind joins [12] in this example combine each result tuple of F3 and F5 as the input for F2.

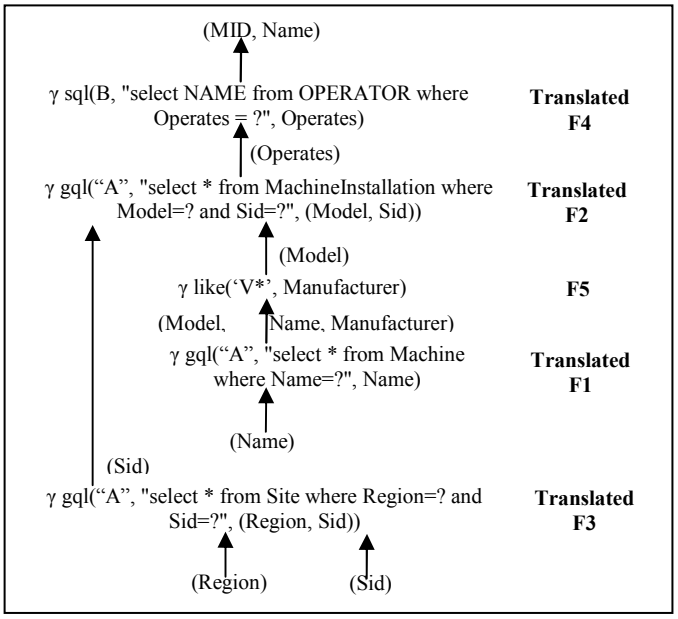

Figure 7. Example query execution plan

The execution plan contains several algebra expression with calls to the *apply* operator γ [11].

The interface function *gql* is an interface function with the signature:

gql(Charstring dsn, Charstring query, Tuple params) -> Stream result

The *gql* function sends a parameterized GQL *query* with parameter *params* to the Bigtable data source *dsn* for execution and returns a stream of tuples, *result*. The *"?"* in a GQL string is substituted with a corresponding parameter value.

Analogously the interface function *sql* has the signature:

sql(Relational ds, Charstring query, Vector params) -> Stream result

The function *sql* sends a parameterized SQL *query* with parameter *params* to RDBMS data source *ds* for execution and returns a stream of tuples, *result*.

In the execution plan the interface function call *gql("A", "select * from Site where Region=? and Sid=?", (Region,Sid))* returns a stream of tuples *(Sid)*. The interface function *gql("A", "select * from Machine where Name=?", Name)* returns a stream of tuples *(Model, Name, Manufacturer)*. The *like* operator returns the filtered stream of tuples *(Model)*. Each combination of tuples from *(Model, Sid)* is input for the interface function call *gql("A", "select * from MachineInstallation where Model=? and Sid=?", (Model, Sid))*, producing a stream of tuples *(Operates)*, which is fed to the interface function call *sql(B, "select NAME from OPERATOR where Operates = ?", Operates)*, producing the final result.

The BigIntegrator automatically generates algebra operators for the NSPs that can't be absorbed into any access filter to post-process them by its query engine. For example, *like('V*', Manufacturer)*. This compensates for the lack of a *like* function in GQL.

C. The Bigtable wrapper

1) Architecture

The Bigtable wrapper includes the server and client components shown in Fig. 8.

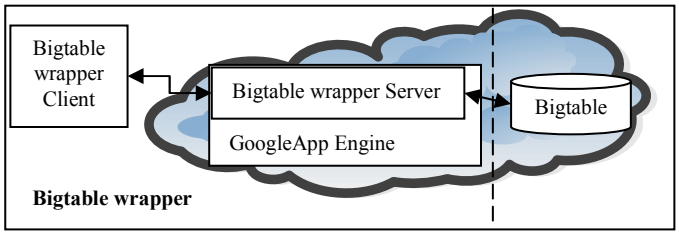

Figure 8. Bigtable wrapper architecture

The Bigtable wrapper server is a web application written in Python served by GAE. It manages the requests from the Bigtable wrapper client. The http protocol is used for the communication between the client and the server. The interface function sends a query request to the server, which forwards the GQL query to Bigtable using the Python Datastore API [9]. The Bigtable wrapper server then sends back the query result to the Bigtable wrapper client.

GAE limits the size of a query result. This is a problem when a GQL query returns a large result. Another problem is that there is a 30 seconds limit on the response time for a request. This is a problem if the server is running longer time than the limit or returns a too large result. Therefore the server delivers the query results in chunks. This is implemented through the cursor facility of the Python Datastore API. Fig. 9 illustrates the Bigtable wrapper client and server communication.

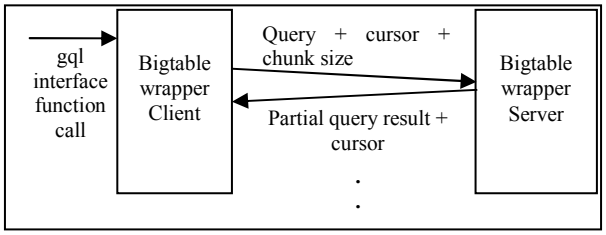

Figure 9. Bigtable wrapper client-server communication

For a given *gql* interface function call, the client sends the GQL query, the cursor information, and the chunk size to the server. The Bigtable wrapper server retrieves the chunks one by one by several *next* requests from the Bigtable wrapper client until the entire result is transmitted to the client. To be able to separate cursors from different queries the cursor handle is shipped back with each result and used in the *next* calls to move the cursor forward.

2) The Bigtable wrapper client and server components

Fig. 10 illustrates the Bigtable wrapper client and server components.

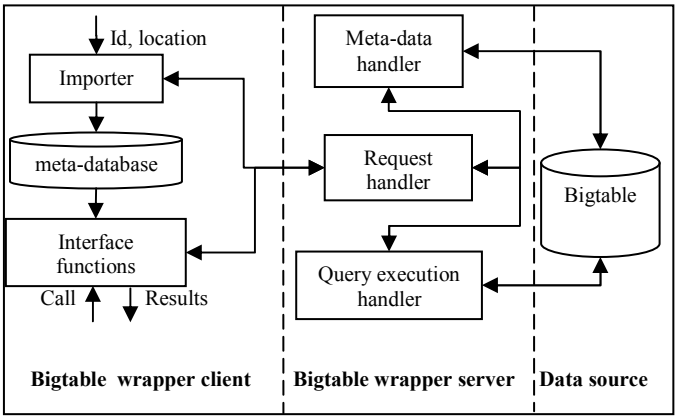

Figure 10. Bigtable wrapper client and server components

Every web application in GAE has its own application identifier (e.g. *http://application-id.appspot.com/*), which is specified when the application is created. A Bigtable wrapper server is a GAE web application and therefore has a unique URL. The location (URL) is used by the importer of the Bigtable wrapper client to establish an http connection to the Bigtable wrapper server. The *importer* first sends a request to the Bigtable wrapper server to collect the meta-data of the Bigtable database. The *request handler* routes the request to the *meta-data handler*. The retrieved meta-data is sent back to the importer by the request handler. The importer stores the meta-data (e.g. source predicate definitions) in the client's *meta-database*. The request handler passes query requests to the *query execution handler*, which calls the Python Datastore API to execute the GQL query. The query results are then sent back to the client through the request handler.

V. CONCLUSIONS AND FUTURE WORK

We presented the BigIntegrator system, which enables SQL queries joining data stored in a Bigtable data repository and in local relational databases. A novel query processing mechanism based on plug-ins for absorbers and finalizers implements extensions for each new kind of data source having limited query capabilities. We presented the architecture of the system. The Bigtable wrapper provides communication between a client computer running the BigIntegrator engine and a Bigtable wrapper server managed by GAE running in a cloud. A communication mechanism provides streamed communication between the Bigintegrator system and the Bigtable wrapper server.

As future work, we plan to evaluate the scalability of the system and develop strategies to improve the system's performance by parallelization.

ACKNOWLEDGMENT

This work is supported by the Swedish Foundation for Strategic Research under contract RIT08-0041.

REFERENCES

[1] F. Chang et al, "Bigtable: A distributed storage system for structured data," in OSDI, 2006, pp. 205-218.

[2] D. Campbell, G. Kakivaya and N. Ellis, "Extreme scale with full SQL language support in Microsoft SQL Azure," in SIGMOD, 2010.

[3] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser, "Cloudy: A modular cloud storage system," in *Proc.* VLDB 2010, Vol. 3, No. 2.

[4] V. Josifovski, P. Schwarz, L. Haas, and E. Lin, "Garlic: A new flavor of federated query processing for DB2," in ACM SIGMOD, 2002.

[5] B. F. Cooper et al, "PNUTS: Yahoo!'s hosted data serving platform," in VLDB, 2008.

[6] W. Litwin and T. Risch, "Main memory oriented optimization of OO queries using type data log with foreign predicates," in IEEE Transactions on Knowledge and Data engineering, 1992, pp. 517-528.

[7] The GQL reference web page, published online http://code.google.com/appengine/docs/python/datastore/gqlreference.html

[8] G. DeCandia et al, "Dynamo: amazon's highly available key-value store," in SOSP, 2007.

[9] The Python Datastore API reference web page, published online http://code.google.com/appengine/docs/python/datastore/

[10] Google App Engine, published online http://code.google.com/appengine/docs/whatisgoogleappengine.html

[11] G. Fahl and T. Risch, "Query processing over object views of relational data," The VLDB Journal, Vol. 6 No.4, November 1997, pp. 261-281.

[12] L. Haas, D. Kossmann, E.Wimmers, and J. Yang, "Optimizing queries across diverse data source," in *Proc*. VLDB 1997, Athens, Greece.