Relational Algebra for Heterogeneous Cloud Data Sources

Aspen Olmsted
Fisher College
Department of Computer Science, Boston, MA 02116
e-mail: aolmsted@fisher.edu

***Abstract*— Cloud computing has changed the way commonly used data is stored. Before the adoption of the cloud, most data was preserved in proprietary relational databases. Cloud services provide native storage for several complex data types including contacts, calendar events, tasks and form responses. Along with the cloud services the user is delivered mobile application synchronization, web application interfaces and guarantees of availability. Unfortunately, along with all the benefits of the native cloud data types comes complexity that leads to several difficulties. One difficulty is data queries that relate data from different heterogeneous data sources. In this paper, we develop a relational algebra that operates on two-dimensional data stored in many heterogeneous cloud formats. The relation algebra is exposed via web-services and allows a user to combine data from different data types and across domains.**

Keywords-Relational Algebra; Cloud Computing; Heterogeneous Data

I. INTRODUCTION

Relational algebra is a mathematical notation used for modeling the data stored in two-dimensional tables. Edgar F. Codd [1] created relational algebra to express the operations and operators used with relational databases to query data. Since its original inception, relational algebra has been extended to model query operations on many different data source structures from the original relational model. Two examples of extensions to the original relational algebra specification is an extension that allows operations on hierarchical data [2] and an extension that allows operations on semantic data [3].

In the same work [1], Codd also developed algorithms for reducing redundancy in the data model to ensure that data was kept correct and not lost from the update and deletion anomalies. At the time of Codd's work, access to computers to store databases was rare, and access to applications that manage data was even rarer. In today's world, most individuals carry at least one device with several databases on it. The same data is often stored on different machines in their office and their home. The redundant copies of the data lead to problems Codd could not have anticipated with his single data store where he applied his relational algebra operations.

Keeping the distributed data updated across all the diverse devices has been improved by the cloud. Often the data is stored in the cloud and changes made on mobile devices or desktops are bi-directionally synchronized with the cloud. Unfortunately, the data tends to be stored in different heterogeneous databases that are specialized for the type of data the application handles. An example of the diverse data source problem in the cloud is seen when looking at the three major cloud productivity app providers. Google G Suite [4], Microsoft Office 365 [5], and Zoho Docs [6] each have different data formats and application programmer interfaces (API) for emails, form data, calendar data. Each of these cloud office suites also provides the ability to store diverse two-dimensional data in spreadsheet files.

The organization of the paper is as follows. Section II describes the related work and the limitations of current methods. In Section III, we describe the relational algebra operators, we implemented for the cloud data sources. Section IV describes the different data sources we allowed as operands in our work. Section V describes some motivating examples of queries we developed using our relational algebra. Section VI drills into some specific details on how we programmed the relational algebra query engine and how data can be returned from the engine. In section VII, we talk about using the relational algebra to enforce consistency in the distributed data source. We conclude and discuss future work in Section VIII.

II. RELATED WORK

Garcia-Molina, Ullman, and Widom [6] spend several chapters in their database textbook discussing algebra on relations. The authors build upon the work originally developed by Codd [1]. They contribute several expressive additions to Relational Algebra in their book. One addition allows for a linear sequence of operations. The second addition utilizes relation algebra to express constraints on relations. We utilize their work to develop our cloud heterogeneous data source constraints.

Agrawal [2] extended relational algebra to handle hierarchical data. In Codd's original work and our work, we assume the data sources are two-dimensional tables where a column, or set of columns, of the table, relates to a column or set of columns, in a different two-dimensional table. Agrawal's work adds an operator to express hierarchical relationships and query the transitive closure of those relationships.

Cyganiak [3] took relation algebra and applied the operators to Resource Description Framework (RDF) triples. An RDF triple is a 3 part notation for expressing the subject object and predicate. An interesting piece of Cyganiak work is in a subsection of the paper that looks at extensions to relational algebra that would allow the operators to work on full RDF datasets. RDF datasets are collections of RDF graphs that would be distributed across the internet. Our work uses distributed data sources but does not limit the format to the same data structure as he did with RDF graphs.

 ISBN: 978-1-61208-703-0

III. CLOUD RELATIONAL ALGEBRA OPERATORS

Relational algebra is a set based mathematical model that provides a notation for performing operations on sets and producing sets as results of the operations. In this work, our cloud-based relational algebra includes the same operators as traditional relational algebra, but instead of operating on sets, our algebra operates on bags or multisets. A bag or multiset is a generalization of a set that allows duplicate instances of elements in the bag. We choose to work with bags for performance reasons as we do not have control of the data sources in the cloud and they may contain duplicate elements. We also use bag operations for performance reason. The deduplication process of a bag is at best a $N\log_2 N$ operation.

Table I shows our relational cloud algebra operators mapped from the original relational algebra. Our implementation was developed as functions in Google App Script. The signature of each function contained the same number of arguments as the original relational algebra but implemented as parameters to the function. Each function also returns a two-dimensional relational structure that can be passed into any of the other cloud relational functions as an input for the relation. The consistent return type allows the operators to be combined to produce complicated queries.

A. Selection

The cloud selection function is called by passing in a condition as a string and a relation. The algorithm iterates over the tuples in the relation and returns all the tuples where the condition evaluates to true.

B. Projection

The cloud projection function is called by passing in a comma-delimited list of columns in the first argument as a string. The second argument is the original relation data source that holds the data columns. The result is a new table with just the columns specified.

TABLE I
RA CLOUD OPERATORS

Relational Operator	Example	Name	Cloud Function
Σ	R1 := σ_C (R2)	selection	R1 = ra_select(c,R2)
Π	R1 := π_L (R2)	projection	R1 = ra_project(L,R2)
⋈	R3 := R1 $\bowtie_C$ R2	theta join	R3 = ra_theta(R1,c,R2)
⋈	R3 := R1 ⋈ R2	natural join	R3 = ra_natural(R1,R2)
P	R1 := ρ_L (R2)	rename	R1 = ra_rename(L,R2)
X	R3 := R1 X R2	product	R3 = ra_product(R1,R2)
∩	R3 := R1 ∩ R2	intersection	R3 = ra_intersect(R1,R2)
∪	R3 := R1 ∪ R2	union	R3 = ra_union(R1,R2)
—	R3 := R1 —R2	difference	R3 = ra_diff(R1,R2)

C. Product

The cloud production function takes the two arguments passed in and creates a cartesian product of the tuples in the first argument and the tuples in the second argument. The result is a new two-dimensional multiset with the schema made up of the combination of the columns from the two input datasets. The multiplicity of the result relation is the number of tuples in the first argument data source multiplied by the number of tuples in the second argument data source.

D. Theta Join

The cloud theta join function is invoked by passing in three arguments. A new data source is passed out of the function with the same schema as if the first and third arguments were passed to the product function. The multiplicity is reduced by applying a filter to the product. The filter condition is specified in the second argument to the function.

E. Natural Join

The cloud natural join function is similar to the theta join function except no condition is specified. The caller specifies the two input sources and the data is joined based on equal column names between the two sources.

F. Rename

The cloud rename function is used to change the names of the columns in the data source. The primary purpose of the operator is to ensure as a predecessor to a natural join or an intersection, union or difference operation.

G. Intersection

The intersection cloud function takes two data sources as arguments and finds the tuples that exist in both data sources. The schema of both data sources needs to match so that the tuples can be compared. The result relation has the columns of one of the input data sources and the tuples that were in common between the two data sources. Often in bag relational operations, the intersection operator produces a set. Our implementation produces a bag for the performance reasons given earlier.

H. Union

The union cloud function takes two data sources as arguments and combines the dataset. The schema of both data sources needs to be identical so that the tuples can be combined. The result relation has the schema of one of the input data sources and all the tuples that were in both of the two input data sources. Often in bag relational operations, the union operator produces a set. Our implementation produces a bag for the performance reasons given earlier.

 ISBN: 978-1-61208-703-0

TABLE II
RA CLOUD DATA SOURCES

Data Source	schema	Ownership
Spreadsheets	Dynamic	Distributed
Forms	Dynamic	Personal
Contacts	Static	Personal
Events	Static	Distributed
RSS Feeds	Static	Distributed
RESTful data	Dynamic	Distributed

I. *Difference*

The cloud difference function takes two data sources as arguments and finds the tuples that exist in the first data source but not in the second data source. The schema of both data sources needs to be identical so that the tuples can be compared. The result relation has the columns of one of the input data sources and the tuples that were in the first data sources but not the second data source. Often in implementations of relational operations on bags, the difference operator produces a set. Our implementation produces a bag for the performance reasons given earlier.

IV. CLOUD HETEROGENEOUS DATA SOURCES

Cloud service providers offer native storage for many different sources of data a user may want to query. In our implementation, we supported several different types of data sources. The data sources either had a fixed static schema or the schema was dynamic based on the configuration of the data source. Some examples of fixed static schemas are Rich Site Summary (RSS) feeds, events and contacts. With other data sources, the schema varies based on the specific relation queried. These sources include spreadsheet data, form data, and web services. Table II shows the breakdown of the schema types for the different cloud data sources.

Some of the schemas for the different data types are fixed while others are pulled from the metadata of the data. The

TABLE III
RSS SCHEMA

Field	Data Type
Title	String
LinkScheme	String
LinkHost	String
LinkPath	String
LinkQueryString	String
PubDate	String
Description	String
GUID	String

TABLE IV
EVENTS SCHEMA

Field	Data Type
Id	String
Title	String
Description	String
StartTime	Date
EndTime	Date
AllDay	Boolean
Recurring	Boolean
Location	String

data sources that cross domains require the unique address of the data source. RSS Feeds are a data source that can cross domains and have a fixed schema. When an RSS data source is used as the operand of an operation, we prepend a string of "rss=" followed by the feed URL. An example operand of type RSS would be expressed as "rss=http://today.cofc.edu/category/news-briefs/feed/."
Table III shows the fixed schema for all tuples in a relation of type RSS. Each RSS tuple has a title, date of publication and description, which are the typically displayed by an RSS reader. There is also a link field returned by an RSS feed. We parse the link into four parts: scheme, host, the path and query string. The scheme is the protocol that is used to reference the link. The host is the website where the link is hosted. The path identifies a specific resource at the website, and the query string is a set of key-value pairs that are sent as parameters to the resource. We parse URL into the separate components so that each component can be easily joined to other data sources in relational algebra queries.

The Calendar datasource is a fixed schema source that reads data that is stored in the Google GSuite. The calendar data source is broken into two different relations. The first relation has the primary event details for events on the calendar. The second relation has the guests that are linked to the event. Table IV displays the schema for the event component of the data source, and TABLE V shows the schema for the event guest data. We separated the calendar data into two sources to normalize the guest email addresses. The email address is often a unique identifier in cloud data sources. Having the email addresses in a normalized relation will allow for easy joining to other data sources. When a calendar data source is used as an operand, the operand is passed as a string with a prefix of "calendar=" followed by

TABLE V
EVENT GUEST SCHEMA

Field	Data Type
EventId	String
Name	String
Email	String

 ISBN: 978-1-61208-703-0

TABLE VI
CONTACTS SCHEMA

Field	Data Type
Id	String
FullName	String
GivenName	String
MiddleName	String
FamilyName	String
Initials	String
Prefix	String
Suffix	String
MaddenName	String
NickName	String
ShortName	String
HomeAddress	String
HomeAddressIsPrimary	Boolean
WorkAddress	String
WorkAddressIsPrimary	Boolean
Company	String
JobTitle	String
AssistantPhone	String
CallBackPhone	String
HomePhone	String
WorkPhone	String
MobilePhone	String
Page	String
HomeFax	String
WorkFax	String
HomePage	String

the name of the calendar. The events data source allows data to come from different ownership. To access a calendar owned by a different user B, user A would need to share the calendar with user B. An example operand of type event would be expressed as "calendar=US Holidays." To query the guests of an event, you would prefix the relation with "calendarguests=." An example operand of type event would be expressed as "calendarguests=US Holidays."

The contacts data source is a large schema with many de-normalized columns. We chose to leave the table mostly de-normalized except pulling out the contact emails into their own relation. Table VI shows the schema for the contact relation. In the schema, addresses and phone number types are represented by distinct attributes. The phone and address fields were not used in any of our test cases that involved join operations. For simplicity, we left the phone and address fields de-normalized. A future version may add additional cloud data sources where the address or phone number is a primary key, and we will want to normalize this data. Table VII shows the schema for the contact emails. When a reference to a contacts data source is used as an operand in a relational algebra operation, the operand is expressed with a fixed string of "contacts." The primary cloud providers of contact services do not support distributed contacts, so there is one single relation that holds the contacts. If the relational algebra operation should operate on the contact emails, then the operand is expressed as the fixed string of "contactemails."

TABLE VII
CONTACT EMAILS SCHEMA

Field	Data Type
ContactId	String
Type	String
Email	String
Primary	Boolean

The spreadsheets data source use the first row of the data range to specify the schema to be used. In our experimental implementation, we assume the data range starts in cell A1 on the first tab of the spreadsheet. Future implementations could enhance the spreadsheet functionality to specify specific tabs and specific cell ranges. As you will see in our example distributed queries our primary goal with spreadsheet queries was to allow queries across ownership. So in the relational algebra operations, the operand can specify wildcards to include many spreadsheets stored in the same folder. With the Google GSuite, each spreadsheet can be owned by a different user. The user executing the relational algebra can locate the shared files into their own folder.

Form data behaves almost identically to spreadsheet data in our implementation. The Google GSuite stores form data as spreadsheet data so when a form is queried in a relational algebra operation the first row in the spreadsheet is the form questions. Again our implementation assumes all questions are included in the data set by starting the data from cell A1 in the backend spreadsheet.

```
[  { country: 'China',         population: 1379510000 },
   { country: 'India',         population: 1330780000 },
 { country: 'United States', population:  324788000 },
  { country: 'Indonesia',     population:  260581000 },
   { country: 'Brazil',        population:  206855000 }]
                                ];
```

Figure 1. Example REST data

TABLE VIII
SAMPLE POS

Semester	Class	Area
Fall 17	CSIS602	Core
Fall 17	CSIS603	Core
Fall 17	CSIS614	Cybersecurity
Spring 17	CSIS601	Core
Spring 17	CSIS604	Core
Spring 17	CSIS631	Cybersecurity
Summer 18	CSIS638	Elective
Summer 18	CSIS649	Elective
Fall 18	CSIS632	Cybersecurity
Fall 18	CSIS641	Cybersecurity
Fall 18	CSIS618	Elective

Our implementation of restful web-services made some simple assumptions to ensure success in the first version. The first assumption is that there is no authentication. The second assumption was that data is returned in JavaScript Object Notation (JSON) format. The data returned from the web-service must be an array of JavaScript objects, each with the same properties. In the industrial world, these restrictions are too high to successfully include data from many 3rd party vendors, but it was good enough for our implementation to prove that web-service data could be included in the relational algebra operations. Figure 1 shows a sample JSON array of countries along with its population that can be processed by our relational algebra.

V. EXAMPLE DISTRIBUTED QUERIES

The first example query we tested with our cloud relational algebra was querying of graduate student program of study (POS) plans that were stored in individually owned spreadsheets in the cloud. Each student in the graduate program keeps a spreadsheet they have shared with the program director. The MS degree requires each student to take eleven classes to complete their degree. Four of the classes are core classes, so all students are required to take these classes. There are some additional four classes to represent the focus area, so students choose a focus area and have a set of classes to choose from to meet the focus requirement. The final three courses are electives that can be taken as a thesis option. Table VIII shows a sample POS for a typical student with a focus on cybersecurity. The shared links are stored in a single cloud directory named "GradSchool." Each spreadsheet is given a name that starts with the student's name followed by the letters "POS." The spreadsheets have three columns. The semester a student plans to take a course is the first column. The course they plan to take is the second column. The third column holds the POS category the course is fulfilling.

To determine the demand for a specific course, we can write a relational algebra expression that uses a combination of the selection and projection operations. Figure 2 shows the cloud relation algebra that will return a two-dimensional array of the student's email address and the class they want to take in the "Summer 18" semester. Once the result data is returned to a cloud spreadsheet, a pivot table can be used to display the course demand.

Figure 3 extends this example by performing a theta join operation on the students taking summer classes with their contact information. For the implementation, two theta joins are applied. The first join is done by the owner of the spreadsheet to the email of the contacts. The second join is completed from the contact email id to the contact id. This query is only possible because the normalization that was performed on the contact data described earlier. A student may have both a home and work email address, and it is not known which email is the owner of the spreadsheet data.

The second example query we wanted to handle with our cloud relational algebra was also related to student data. In this example, we want to combine the results from a student survey on happiness in the program with the events the student attended and the classes they took during the semester. Figure 4 shows a part of our final query. We start by extracting the classes from the program of study spreadsheets with the selection operation on the semester = "Fall 17". We apply a theta join to the results of the selection operation with the cloud form named "SpringStudentSurvey." The results of the join operation are then theta joined to the event guests in the "studentevents" calendar. In the final step, the guest is theta joined with the event they attended. Normally, the attributes would be

```
ra_project("owner, class",ra_select("semester=Summer  18","Spreadsheet=GradSchool/*POS"))
```

Figure 2. RA to retrieve students taking summer classes

```
ra_theta(ra_theta(ra_project("owner, class",ra_select("semester=Summer
18","Spreadsheet=GradSchool/*POS")),"owner=email",contactemails),"contact_id=id",contacts)
```

Figure 3. RA to retrieve student summer contact info

```
ra_theta(ra_theta(ra_select("semester=Fall 17","Spreadsheet=GradSchool/*POS"),"owner=username", ra_project("opinion",
"Form=SpringStudentSurvey")),"email=username","calendarguests=studentevents"),"eventid=id","calendar=
studentevents")
```

Figure 4. RA to retrieve student survey and student data

 ISBN: 978-1-61208-703-0

TABLE IX
Sample Form Constraint Table

Field	RA	Invalid	Message
1	ra_select("semester=*Field1*","Spreadsheet=Rooms")	Empty	Please choose a valid room
2	ra_select("semester=*Field2*","Spreadsheet=Resources")	Empty	Please choose a valid resource

minimized with another projection operation, but for simplicity, we left projection out of Figure 4.

The three queries presented in this section are just a small representation of the types of queries that can be performed with the heterogeneous data.

VI. IMPLEMENTATION

As discussed in the earlier sections we developed our solution using Google Application Script (GAS) [7]. The GAS environment was designed to allow a developer to extend G Suite [4], Google's suite of cloud office applications. The programming environment concepts are similar to the Microsoft VBA programming functionality included in Microsoft Office [8]. At the time of our experimentation, Microsoft did not offer a similar programming environment for their cloud office suite Office 365 [5].

We felt the best storage location for the result of the cloud relational algebra operations was in Google spreadsheets. Google spreadsheets can be enhanced with GAS to allow custom functions. Unfortunately, for security reasons, Google does not allow spreadsheet custom functions to access external data. We decided to implement our solution as a web-service that could be called from any programming language but also imported into a Google cloud spreadsheet using the "importdata" function.

VII. CONSTRAINTS ON HETEROGENEOUS DATA SOURCES

In this section, we build on the work previously described to query data using relational algebra. We extend the work to guarantee consistency in data entry in the cloud by expressing constraints utilizing the heterogeneous data. The general idea is that we want to express a constraint that evaluates to true before allowing new data to be saved.

Since the heterogeneous data is entered into the different native cloud applications directly, we have limited ability to intercept the request and execute our relational algebra queries. Google has exposed some Triggers in the GAS framework that do allow us to intercept the save request we can use for validation.

Google Forms and Calendars are the two applications that currently provide triggers that allow us to intercept the data save and validate that the relational algebra evaluates to true. We are hopeful that more triggers will be exposed via the API and we can extend our constraint work.

With Google Forms a trigger can intercept the post and run the related relational algebra constraint. Since all the relational algebra operators return a two dimensional set of data, we assume an empty set is false and any data returned is true. If an empty set is returned, then the form is not submitted. Instead, the user is redirected to a new URL with the fields of the form prefilled. The field with the error is replaced with an error message stored in the constraint setup. TABLE IX shows our implementations simple constraint setup for form submissions. The first column in the table identifies the field in the form that is checked. The second column holds the relational algebra. The third column specifies how validity is identified. The validity column is expressed as invalidity to simplify the rule definition as it is often expressed as an empty set on the return of the relational algebra. The fourth column holds the message that is to be displayed in the pre-filled form the user is redirected to if there is a constraint violation. TABLE IX was used with a simple example form for event booking that ensured that the room specified in field 1 and the resource specified in field two where valid. To be valid, they had to exist in specific cloud spreadsheets.

The calendar triggers were designed to solve the problem of synchronization of events between multiple calendars. Because of this design, there is not a way to intercept a call before the data is persisted. Several of the Google cloud products support time-driven triggers. Time-Driven trigger functions are similar to a CRON [10] job that runs a script based on a specific time. Time-driven triggers let scripts execute at a particular time or on a recurring interval, as frequently as every minute or as infrequently as once per month. The challenge with a time-driven trigger is how to know what data has changed since the last time the trigger fired. The calendar triggers solve this problem by passing a list of events that have changed since the last trigger fired. We decided not to implement relational algebra constraints that based on the calendar triggers or the time-driven triggers because it would require human interaction after the data is persisted to fix the constraint issue.

VIII. CONCLUSIONS AND FUTURE WORK

Based on our research, we believe the use of native heterogeneous cloud data sources will continue to grow and replace the proprietary relational data sources that people and organizations have come to rely upon for joining data into queries for analysis and constraints. This work demonstrates a successful implementation of the low-level relational algebra operations and provides some successful use cases of our tool. The hooks available to enforce constraints based on the relational algebra are inadequate at this point as demonstrated in our discussion. We hope the cloud vendors can be enticed to provide programmatic hooks before all data persistence operations in the future. Our future work will

 ISBN: 978-1-61208-703-0

expand our use cases and provide a native front end to the relational algebra web-services we provided.

REFERENCES

[1] E. F., "A relational model of data for large shared data banks," *Communications of the ACM,* vol. 13, no. 6, pp. 377-387 , 1970.

[2] R. Agrawal, "Alpha: an extension of relational algebra to express a class of recursive queries," *IEEE Transactions on Software Engineering,* vol. 14, no. 7, pp. 879 - 885, 1988.

[3] R. Cyganiak, "A Relational Algebra for SPARQL," HP Labs, Bristol, UK, 2005.

[4] Google, "Get Gmail, Docs, Drive, and Calendar for business.," [Online]. Available: https://gsuite.google.com/. [Accessed 18 September 2018].

[5] Microsoft, Inc., "Modernize the workplace with Office 365," [Online]. Available: https://www.microsoft.com/en-us/CloudandHosting/office365.aspx. [Accessed 18 September 2018].

[6] Zoho, Inc, "Your personal file manager," [Online]. Available: https://www.zoho.com/docs/. [Accessed 18 September 2018].

[7] H. Garcia-Molina, J. Ullman and J. Widom, Database Systems: The Complete Book, Pearson, 2008.

[8] Google, "Google Apps Script," [Online]. Available: https://developers.google.com/apps-script/. [Accessed 18 September 2018].

[9] Microsoft, Inc., "Getting Started with VBA in Office," 7 June 2017. [Online]. Available: https://docs.microsoft.com/en-us/office/vba/library-reference/concepts/getting-started-with-vba-in-office. [Accessed 18 September 2018].

[10] Wikipedia Foundation, Inc., "Cron," [Online]. Available: https://en.wikipedia.org/wiki/Cron. [Accessed 24 September 2018].

 ISBN: 978-1-61208-703-0