SimpleSQL: A Relational Layer for SimpleDB

André Calil, Ronaldo dos Santos Mello

Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, Brazil.
{calil,ronaldo}@inf.ufsc.br

Abstract. This paper introduces SimpleSQL, a relational layer over Amazon SimpleDB, one of the most popular document-oriented cloud databases. SimpleSQL offers a SQL interface that abstracts any knowledge about data modeling, data persistence and data accessing at SimpleDB. This paper presents the architecture, data and operation mapping from a relational database to SimpleDB, as well as some experiments that evaluate query performance on accessing cloud data using SimpleSQL and using only SimpleDB. Our contribution is a solution for accessing SimpleDB through a relational layer, being the basis for a general approach to relational-to-(document) cloud mapping. The experimental evaluation shows that our solution is promising, since the over-head with data accessing through SimpleSQL is not prohibitive.

Keywords: SimpleDB, SimpleSQL, NoSQL, relational-cloud mapping, cloud database.

1 Introduction

The concept of software as a service has moving from an innovative paradigm to a business model during the last years. While the model of licensed and maintained soft-ware represents a high cost in terms of acquisition and maintenance to the organizations, software maintained by the service provider and charged according to the demand (pay as you go paradigm) [3], with contracts ensuring high availability and privacy, has become more and more attractive [8].

On following this paradigm, data storage and data management facilities have also being offered on cloud computing platforms [15]. This paradigm changes the existing database management system architectures to assign them some distributed system characteristics, like high availability and fault tolerance. Besides, this tendency has also raised new data models not compliant to the classical relational model [1]. These models are suitable to current Web applications and programming paradigms, which manage a large amount of data and transactions, being much more text- or object-oriented than record(relational)-oriented.

Examples of these new models are key-value collections, document-oriented or super-column [16]. Cloud database systems based on these models are known as Not only SQL (NoSQL). The main differential of these systems, if compared to relational databases, is to relax the overhead with consistency checking to increase data availability in a distributed scenario [9].

As these database systems are not relational, there is no support to the SQL standard, what makes more difficult to migrate and to adapt applications based on relational data and relational accessing. In order to deal with this problematic, this paper presents a relational layer, called SimpleSQL, for accessing SimpleDB [5], an Amazon's solution for data management on the cloud. We chose SimpleDB because it is one of the most famous databases based on the document-oriented model. This model provides a simple but efficient access method to large data sets. SimpleSQL, supports a simplified version of ISO/IEC SQL that allows data update operations and some query capabilities. On using SimpleSQL, a client application is isolated from SimpleDB access methods as well as its data model, providing a traditional relational interface to data on the cloud.

Besides storage and operation transparency for data on the cloud, SimpleSQL supports queries with joins, which is not a native capability of the access methods for NoSQL databases and is not specifically implemented at SimpleDB interface neither. Our layer is able to decompose a query that combines several tables through joins into a set of queries over single tables, to fetch the data that corresponds to each table from the cloud, and to combine them in order to generate the result set. A set of experiments shows that the overhead to process this kind of query, as well as other operations from SimpleSQL, is minimal. Details about these experiments and the design of the layer are given in the next sections.

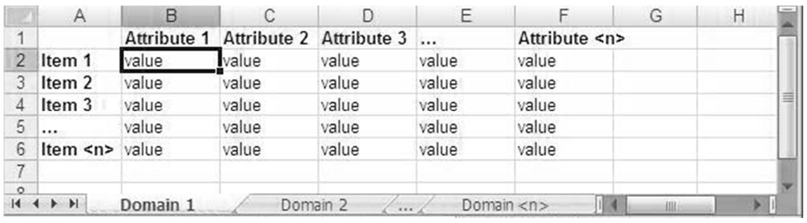
The rest of this paper is organized as follows. Section 2 presents SimpleDB and its data model. Section 3 presents SimpleSQL, its development and architecture, followed by the analysis of some experiments in Section 4. Section 5 presents related work and Section 6 is dedicated to the conclusion.

2 SimpleDB

Within the categories that describe NoSQL databases, the most noticeable are key-value data stores, that apply a dictionary structure to keep values at user-defined keys, and document oriented, that serializes objects as documents and keeps indexes for searching [9].

SimpleDB is an Amazon solution for data management on the cloud that follows the document-oriented model [9]. It is kept as a service, and data is automatically replicated over data centers at the same geographic region that the user selects during setup.

SimpleDB data model is composed of domains, items, attributes and values, as shown at Figure 1.



	A	B	C	D	E	F	G	H
1		Attribute 1	Attribute 2	Attribute 3	...	Attribute <n>		
2	Item 1	value	value	value	value	value		
3	Item 2	value	value	value	value	value		
4	Item 3	value	value	value	value	value		
5	...	value	value	value	value	value		
6	Item <n>	value	value	value	value	value		
7								

Fig. 1. SimpleDB data model [6]

A domain is composed by a name and a set of items. Each item, in turn, has a set of attributes that are key-value pairs. The domain is the main entity for replication and performance issues. A user can have up to 250 domains, and each domain can grow up to 10 Gb, what is enough for most of the applications.

Data placement and sharing among domains is a database design issue. However, SimpleDB does not support queries that join data from different domains. In such a case, join operations must be made by user application. Because of this, the strategy for domain distribution must be chosen wisely. A high cost processing to distribute an item may compromise the performance of every single operation.

Items are composed by a name and a collection of attributes. As with domains, the name of an item must uniquely identify the entity. The collection of attributes describes its item. An attribute can handle multiple values for a given key and there is no requirement that all the objects of a given domain must have the same set of attributes. In fact, this flexibility follows the schema-free feature of NoSQL databases.

For consistency, SimpleDB guarantee that any write operation will update all the copies of an item, but it does not ensures that a reading operation will retrieve the last version of a given item. Given the delay to update all the copies of an item, a read operation may fetch an older value of an attribute [5]. As an alternative, it is possible to specify the desired consistency level for a reading operation. The default value is eventual consistency, which has the fastest response time.

The interface to access SimpleDB is an API developed with REST web services [11]. As it relays on HTTP requests, most of the current development frameworks are able to access the system. All reading and writing operations, and even the domain administration tasks, are performed with HTTP methods GET and POST, respectively. SimpleDB is available only as a service, being not licensed for local installations.

3 SimpleSQL

As one of the main cloud data management system, SimpleDB turns out as an option for fast setup and virtually no administration effort. However, the NoSQL paradigm is an obstacle to applications already developed with relational databases. To adapt a relational-based application to a cloud platform may incur in a large maintenance effort. In order to alleviate a situation like that, we propose an access layer that makes the translation of SQL requests to the SimpleDB API and returns data in a relational format. It is called SimpleSQL. In this first version, our layer is able to perform the four traditional manipulation operations: INSERT, UPDATE, DELETE and SELECT. This section gives details about its functionality and implementation.

SimpleSQL is developed over the Microsoft .NET Framework version 3.5, using C# 3.0 as programming language. Figure 2 shows the layer architecture, which highlights the three steps of an SQL command processing. Each step will be detailed on the following sections.

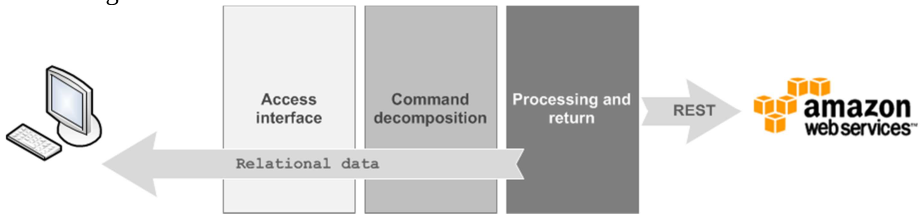


Fig. 2. SimpleSQL architecture

Table 3 presents the relationship between relational model concepts and SimpleDB data model. It summarizes the mapping strategy adopted by our approach in order to provide a relational view of cloud data presented at SimpleDB.

Table 1. Relationship between relational model and SimpleDB model

Relational	SimpleDB
Schema	Domain
Table	-
Table row	Item
Attribute	Attribute key
Value	Attribute value
Primary key	Item name

Although schema and domain can be said as equivalent, SimpleSQL does not support the schema qualifier at a command. Moreover, even though SimpleDB data model does not have a concept for the table entity, SimpleSQL does record the table name as an attribute, in order to keep the idea that an item has a type.

3.1 Processing Requirements

In order to connect to SimpleDB and identify the domains, SimpleSQL must receive the following information from the user:

- *Access Key*: access key of the user to its SimpleDB account. This information can be found when logged in at Amazon portal;
- *Secret Access Key*: secret access key, which is also found at Amazon portal;
- *Domains distribution*: if the user has more than one domain, it must provide to SimpleSQL a dictionary that uses the domain name as key and the list of its tables as value. If the user has only one domain, it may pass this single domain name instead.

As a non-functional requirement, the running environment of SimpleSQL must have access to Amazon website.

3.2 Access Interface

The access interface of SimpleSQL is composed of two methods: *ExecuteQuery*, that returns a *DataTable* object (a tabular structure), and *ExecuteNonQuery*, that returns a text (string). Both of them receive an SQL command as parameter.

As stated before, SimpleSQL supports the four traditional manipulation operations. However, as a scope restriction, each related command has constraints for the supported syntax at this first version:

- SELECT: supports queries for a single table or several tables using INNER JOIN. If a join is specified, all declared attributes must have the format *table.attribute*;
- UPDATE: supports updates to multiple entities, but without sub-queries. More than one attribute can be updated, with more than one filter. Updates without conditions are not supported;
- INSERT: supports insert of a single entity per command, without sub-queries or instructions like INSERT SELECT;
- DELETE: supports deletion of multiple entities, with multiple filters. Sub-queries and deletes without conditions are not supported.

3.3 Command Decomposition

The first processing step is to decompose the SQL command, converting it to the domain used by SimpleDB. In order to support it, each command has a regular expression that it was designed with two goals in mind: (i) to validate the command syntax, and (ii) to extract its elements from the command. Table 2 presents the commands and their related regular expressions.

The elements extracted from the commands are the parts that construct the command itself. In instance, for a SELECT command, SimpleSQL would extract the expected attributes, target table, joins and the set of conditions.

It is important to note at Table 2 the usage of named capture groups, denoted by the syntax `(?<group name>expression)`. Capture group is a regular expression technique for searching and retrieve text without the need for manually search the expected patterns [12]. By using this technique, it is possible to retrieve each command element easily, like target table name, list of expected attributes and filtering criteria.

SimpleSQL works with and abstract class named *Command*, which is specialized into the classes *Insert* and the abstract *ConditionedCommand*. *ConditionedCommand* is inherited and implemented by *Update*, *Select* and *Delete*. Figure 3 presents a short class diagram for SimpleSQL solution domain.

Table 2. DML operations of SimpleSQL and its regular expression

Operation	Regular expression
INSERT	`^(?:\s*(?i:INSERT INTO)\s+)(?<table>[^\(]+)(?:\(\s*)(?<attributes>(?:\w+\s*)(?:\,\s*\w+\s*)*)(?:\)\s+)(?:(?i:VALUES)\s+\(\s*)(?<values>(?:[^\,]+|[^\)])+\s*(?:\,\s*[^\)])*)(?:\))$`
UPDATE	`^\s*(?i:UPDATE)\s+(?<table>\S+\s+)(?i:SET)\s+(?<attributes>(\s*\S+\s*)\=(\s*\S+\s*)(,(\s*\S+\s*)\=(\s*\S+\s*))*)+(?<condition>((?i:where)|(?i:and)|(?i:or))(\s+\w+\s*)(=|<|>|<=|>=|<>|(?i:in)|(?i:not in))((\s*\(?)(\s*\w+\s*)(,\s*\w+\s*)*(\s*\)?)))*$`
DELETE	`^(?:\s*(?i:DELETE FROM)\s+)(?<table>\S+\s+)(?<condition>(?:(?i:where)|(?i:and)|(?i:or))(?:\s+.+\s*)(?:=|<|>|<=|>=|<>|(?i:in)|(?i:not in))(?:(?:\s*\(?)(?:\s*.+\s*)(?:,\s*.+\s*)*(?:\s*\)?)))*$`
SELECT	`^\s*(?i:SELECT)\s+(?<attributes>(\S+\s*)(,\s*\w+\s*)*)\s+(?i:FROM)\s+(?<table>\s*\S+\s*)\s+(?<join>(?i:inner join)\s+(?<toTable>\s*\S+\s*)\s+(?i:on)\s+(?<fromKey>\s*\S+\s*)\s*=\s*(?<toKey>\s*\S+\s*)\s+)*(?<condition>((?i:where)|(?i:and)|(?i:or))(\s+.+\s*)(=|<|>|<=|>=|<>|(?i:in)|(?i:not in))((\s*\(?)(\s*.+\s*)(,\s*.+\s*)*(\s*\)?)))*$`

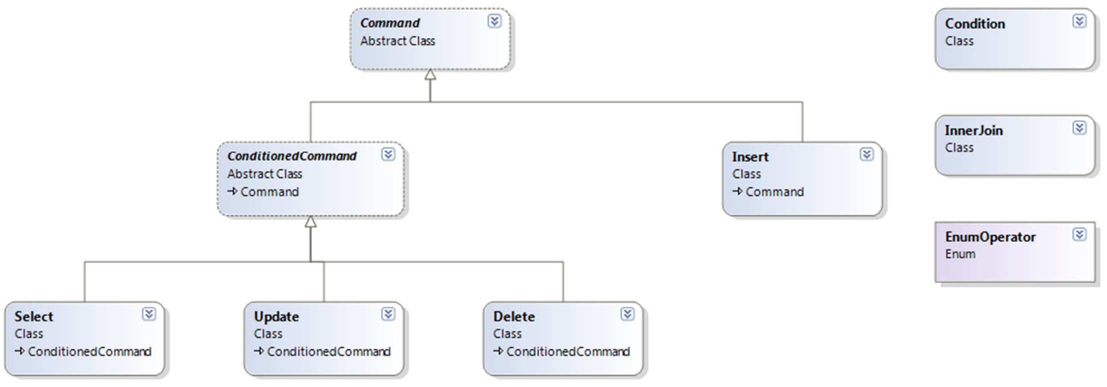


Fig. 3. Class diagram for SimpleSQL domain

Classes *Condition* and *InnerJoin* are used to represent, respectively, filtering criteria (condition) and joins. The enumerator *EnumOperator* is used at the condition to indicate which operator should be applied.

3.4 Processing and Return

Once identified a command and its components, SimpleSQL translate it to a SimpleDB REST method call. All the commands begin with the identification of the target SimpleDB domain from the target table, extracted from the command. DELETE and UPDATE commands return the number of affected items. INSERT returns the result of the operation (success or fail) and SELECT returns the fetched data in a table structure using .NET class *DataTable*.

INSERT

One table tuple corresponds to one item at SimpleDB schema. Thus, one INSERT command generates one item. When starting this command processing, SimpleSQL checks if the number of columns is equal to the number of values.

Besides the given attributes, SimpleSQL will add one attribute to the item with the format *SimpleSQL_TableName*, in order to keep the same name of the target table. The name of the item, that is a required field at SimpleDB model, is filled with an instance of a global unique identifier (GUID) [15].

UPDATE and DELETE

The filter list at the condition part of these commands is extracted and processed like a simple SELECT (a query without joins) to fetch the items to be updated or removed.

In case of a DELETE, every retrieved item is removed as an isolated operation. In case of an UPDATE, the attributes to be updated are identified as well as their new values. For every retrieved item, if the item has that given attribute (as it is schema-free, items of the same type may not have the same attributes), its value is updated.

Both of these commands return the total of affected items.

SELECT

When receiving a query command, SimpleSQL extracts the list of expected attributes, the target table, joins and the filters. If there are joins, SimpleSQL will split them into single simple queries. It means that, using the *table.attribute* notation, SimpleSQL identifies the expected attributes and the condition of each joined table. After retrieving the return of each single query, there will be created a *DataTable* with the schema of the expected return, the list of retrieved items are joined using the foreign keys of the relational schema and the return table is be filled. It is recommended that all tables joined in a query have at least one condition, preventing SimpleSQL from retrieving a big amount of data.

For each query sent to SimpleDB, SimpleSQL will append at least one condition to filter the *SimpleSQL_TableName* attribute, in order to avoid retrieving items of other tables that could have attributes with the same name.

When queries are processed at SimpleDB, the response is a collection of *Item*. SimpleSQL iterates through all attributes of every item retrieved. The name of the attribute is validated against the expected attributes and added to the final list of the returning attributes. The returning *DataTable* is loaded with the values of the selected attributes. On this way, each retrieved *Item* represents a line at the returning table, and its schema is made by the union of the expected attributes. If retrieved items do not have the same schema, the respective cells become null at the returning table.

Another aspect of SimpleDB is that the response of any query is restricted to 1MB size. This means that not all the resulting items of a query will be sent on the first response. The full result is split, and SimpleDB sends a *NextToken* value, so the user can re-issue the query along with this token, to fetch the next part. SimpleSQL has a recursive method that keeps requesting a query until the full response has been collected.

Next section describes an experiment that validates and evaluates the performance of SimpleSQL.

4 Experimental Evaluation

We based our experiments on a relational data sample about the entrance exams of our University (UFSC - Universidade Federal de Santa Catarina). This sample consists of six tables, representing the candidates, their course choice, their exam results and what event (specific exam) they were associated to. Figure 4 presents the relational schema of the considered data sample, which counts more than 500k tuples for all tables.

The experiments have been processed in the following environment:

- Dell Vostro 3550 notebook;
- Intel Core i5-2430M processor;
- 6GB DDR3 1066mHz RAM;
- 10Mbps ADSL2 internet connection.

As for the SimpleDB settings, all data was stored at a single domain, located at the USA East region. Data was loaded to SimpleDB as a part of the experiments, as shown on the following sections.

We evaluate the performance of two operations: INSERT and SELECT. The first one was chosen in order to evaluate the processing time spent to load a large data volume. We also chose SELECT operation in order to evaluate the processing time for a set of different queries in terms of complexity.

For each operation, we compare processing time using the SimpleSQL layer as well as the time spent using only SimpleDB .NET API. The results are detailed in the next sections.

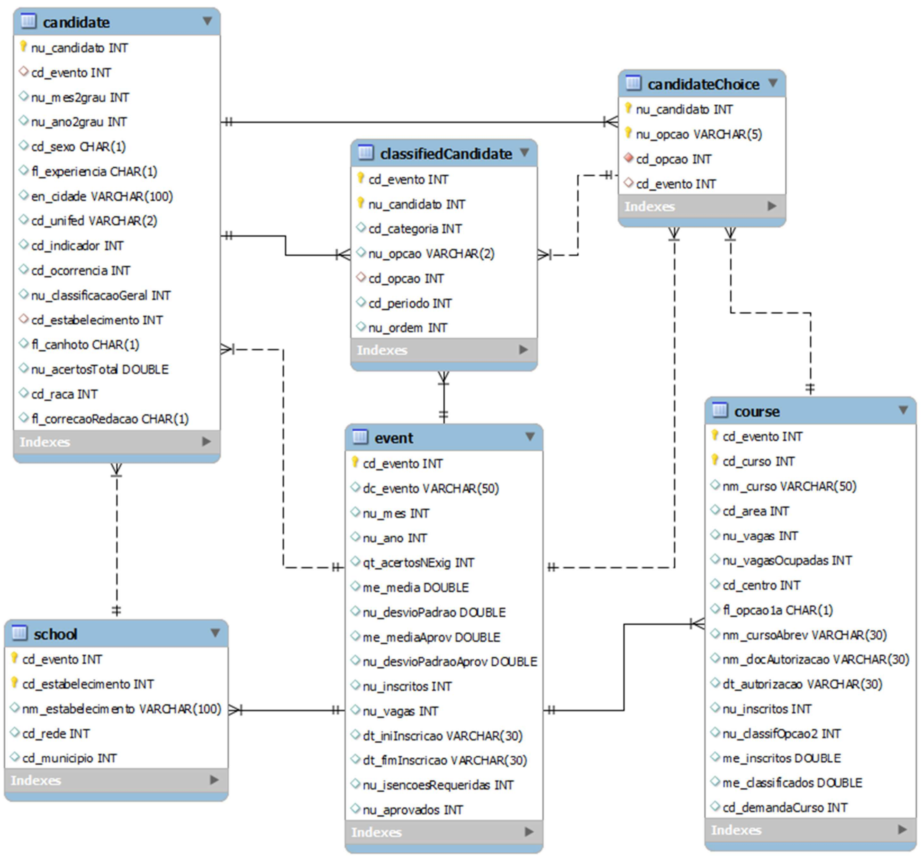


Fig. 4. Relational schema used in the experiments

4.1 INSERT Operations

We ran the INSERT operations to load data into two tables. Table 3 shows the processing time for SimpleSQL and SimpleDB, as well as the average of tuples inserted by minute.

Table 3. Results of INSERT operations

Table	Mode	# tuples	Duration	Average (tuples/min)
candidate	SimpleSQL SimpleDB	50000	03:29:24 03:19:24	238.78 250.75
candidate Choice	SimpleSQL SimpleDB	100000	07:37:08 07:07:32	218.75 233.90

Table 3 shows an overhead for SimpleSQL, which is expected. However, the increasing in processing time was less than 5% for both runnings, and the difference in the average number of inserted tuples was almost the same. It reveals that the introduction of the SimpleSQL layer does not compromise SimpleDB performance and scalability.

4.2 SELECT Operations

SimpleSQL has also been evaluated by running simple queries (SELECT commands without joins, issued to a single table), as well as with one complex query with three joins and four tables. Table 4 presents the simple in SimpleSQL and SimpleDB syntax, and the number of retrieved tuples, while Table 5 presents the average processing time of each query. We execute each query three times.

Table 4. Evaluation of simple queries

Query	SimpleSQL	SimpleDB	# tuples
1	`SELECT nu_candidate, cd_race FROM candidate WHERE en_city like 'FLORIAN%'`	`SELECT nu_candidate, cd_race FROM do-main1 WHERE en_city like 'FLORIAN%'`	34678
2	`SELECT nu_candidate, cd_race FROM candidate WHERE cd_gender = 'F'`	`SELECT nu_candidate, cd_race FROM do-main1 WHERE cd_gender = 'F'`	58410
3	`SELECT * FROM curse WHERE cd_area = 1 and nm_curse like 'ENGE%' and nu_places >= 100 AND nu_applicants > 1000`	`SELECT * FROM do-main1 WHERE cd_area = '1' and nm_curse like 'ENGE%' and nu_places >= '100' AND nu_applicants > '1000'`	58

Table 5. Average duration time for each query

Query	SimpleSQL	SimpleDB
1	00:02:22	00:01:34
2	00:03:09	00:02:33
3	00:00:03	00:00:02

Table 5 shows that the overhead of SimpleSQL in comparison to SimpleDB was not superior to 40% for all queries. We consider these results acceptable, given the large volume of data to be accessed, specially for Candidate table, that holds around 160K tuples.

On considering complex queries, SimpleDB does not have the concept of *type* of items (tables) and does not support the JOIN operator. Because of this SimpleDB is able to filter only the relevant data on each single table, being the application system responsible to perform the joins.

On the other hand, SimpleSQL was designed to support complex queries. The processing steps it performs are the following:

- *Split*: the command is split into simple SELECTs, i.e., SELECTs without JOIN. SimpleSQL identifies the expected attributes and conditions of each individual table when performs the necessary splits;
- *Access*: each individual SELECT command is submitted to SimpleDB;
- *Transform*: the resulting set of each individual command is transformed to the relational schema as viewed by SimpleSQL;
- *Join*: the transformed tables are combined according to the join conditions to generate the resulting table.

On this way, the access step is the only step that is available at SimpleDB API. The other steps are new features implemented at SimpleSQL.

Based on this, in order to compare the performance of SimpleSQL and SimpleDB, the time of each step has been gathered, and the access step is directly compared with SimpleDB time. Table 6 shows the proposed complex query: the original query, as processed by SimpleSQL and the decomposed queries that are submitted to SimpleDB.

Table 7 presents the average amount of time spent in each processing step, as executed by SimpleSQL and SimpleDB. We also execute three times this query

Table 6. SELECT command in the original form and in SimpleDB syntax

SimpleSQL	SimpleDB

`SELECT classifiedCandi-date.nu_order, candida-te.en_city, school.nm_school, event.dc_event FROM candi-date INNER JOIN school ON candida-te.cd_school = school.cd_school INNER JOIN classifiedCandidate ON can-didate.nu_candidate = clas-sifiedCandi-date.nu_candidate INNER JOIN event ON classified-`	`SELECT nu_order FROM do-main1 WHERE cd_event = ‘25’`
	
	 `SELECT en_city FROM do-main1 WHERE cd_event = ‘25’`
	
	 `SELECT nm_school FROM do-main1 WHERE cd_event = ‘25’`
	
	 `SELECT dc_event FROM do-main1 WHERE cd_event =`

`Candidate.cd_event = event.cd_event WHERE event.cd_event = 25 AND school.cd_event = 25 AND classifiedCandi-date.cd_event = 25 AND can-didate.cd_evente = 25`	‘25’

According to Table 7, we note that *Access* step is the most expansive step, and has more overhead for SimpleSQL because each simple query must be separately submitted. However, the steps that are only performed by SimpleSQL are fast, as expected, and the sum of their processing times does not overcome the access time.

Table 7. Average duration time for each processing step for the complex query

Step	SimpleSQL	SimpleDB
Split	00:00:03	
Access	00:18:17	00:18:05
Transform	00:02:23	
Join	00:04:08	

5 Conclusion

The availability of database management systems as a service brings many benefits, like costs reduction and less concerns about database administration. However, most of cur-rent data-centered applications use relational databases, being necessary to provide a bridge between traditional relational data access to data stored in the cloud, which adopts different data models.

We contribute with this problematic by proposing SimpleSQL, a specific solution for mapping a relational schema and some relational operations to SimpleDB, a document-oriented database. Despite of the focus on a specific cloud database, we argue that our solution is a basis for a general mapping approach between these two data models, which is one of our future studies. The intention is to provide a standard access interface and data representation, allowing SimpleSQL to support other NoSQL databases, and providing freedom of choice to the user.

As shown in Section 4, an experimental evaluation shows that SimpleSQL adds a small processing overhead if compared to pure SimpleDB requests, but this overhead does not represent an obstacle to its adoption. For insert operations, the processing time over the relational layer was less than 5%. For simple queries, could be observed an increase of at most 40% to the total time, which is expected, given that SimpleSQL must process the data retrieved from the cloud and convert them to a relational schema. These results indicate that the use of SimpleSQL is not prohibitive in terms of performance. A higher overhead was perceived to a complex query test, which indicate that the execution of the extra SimpleSQL must be optimized, despite of the fact that the sum of these extra overhead was lower than the time spent by SimpleSQL to access data through SimpleDB.

There are some works regarding the disposition of a relational at a distributed environment [17][18][19]. However, as far as we could search, there are no other papers proposing a relational interface to a non-relational database system, such as SimpleSQL. We novel not only to provide insert and query relational operations over SimpleDB, but also to support data joins.

We also intend to design and execute other experiments with data sets of different sizes in order to evaluate, in a fine-grained way, the SimpleSQL performance. An extension to give a wider support to SQL standard syntax is of our interest too.

References

1. Abadi, D. J.: Data management in the cloud: Limitations and op-portunities. IEEE Data Eng. Bull., 32:3-12 (2009).
2. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D. J., Rasin, A., and Silberschatz, A.: Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. PVLDB, 2(1):922-933 (2009).
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. UC Berkeley Reliable Adaptive Distributed Systems Laboratory (2009).
4. Perguntas frequentes sobre o Amazon SimpleDB, http://aws.amazon.com/pt/simpledb/faqs/
5. Amazon SimpleDB, http://aws.amazon.com/simpledb/
6. Amazon SimpleDB, Getting Start Guide, http://docs.amazonwebservices.com/AmazonSimpleDB/latest/GettingStartedGuide/Welcome.html?r=1
7. Amazon Web Services .NET SDK, http://aws.amazon.com/pt/sdkfornet/
8. Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst., 25(6):599–616 (2009).
9. Cattell, R.: Scalable SQL and NoSQL Data Stores. SIGMOD (2010).
10. G. Coulouris, J. Dollimore, T. Kindberg: Distributed Sys-tems: Concepts and Design, 5^{a} edition. Addison-Wesley, (2011).
11. Fielding, R. T.: Architectural Styles and the Design of Network-based Software Architectures. University of California (2000).
12. Friedl, J. E. F.: Mastering Regular Expressions, 2^{a} edition. O'Reilly (2002).
13. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, and partition-tolerant web services. ACM SIGACT News 33 (2002).
14. Pritchett, D.: BASE, an ACID alternative. ACM Queue (2008).
15. MSDN Library, Guid Structure, http://msdn.microsoft.com/en-us/library/system.guid%28v=vs.90%29.aspx

16. Sousa, F. R. C., Moreira, L. O., de Macêdo, J. A. F., Javam, C. M.: Gerenciamento de Dados em Nuvem: Conceitos, Sistemas e Desafios. Em Tópicos em sistemas colaborativos, interativos, multimídia, web e bancos de dados. Sociedade Brasileira de Computação (2010).
17. Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, Nickolai Zeldovich: Relational Cloud: a Database Service for the cloud. CIDR 2011:235-240 (2011).
18. Campbell, D. G., Kakivaya, G., Ellis, N: Extreme Scale with Full SQL Language Support in Microsoft SQL Azure.
19. Amazon Relational Database Service, http://aws.amazon.com/pt/rds/.