4-14-2014

Comparison of NoSQL and SQL Databases in the Cloud

Dayne Hammes
Georgia Southern University, dh04788@georgiasouthern.edu

Hiram Medero
Georgia Southern University, Hiram_Medero@georgiasouthern.edu

Harrison Mitchell
Georgia Southern University, Harrison_L_Mitchell@georgiasouthern.edu

Comparison of NoSQL and SQL Databases in the Cloud

Dayne Hammes
Georgia Southern University
dh04788@georgiasouthern.edu

Hiram Medero
Georgia Southern University
Hiram_Medero@georgiasouthern.edu

Harrison Mitchell
Georgia Southern University
Harrison_L_Mitchell@georgiasouthern.edu

ABSTRACT

In this paper, we examine the design, execution, and subsequent performance between traditional Relational Database Management Systems (RDBMS) and modern NoSQL Database Systems, when implemented on a cloud server. While significant groundwork has been laid for this comparison, we will present two distinct scenarios for real-world comparison: one containing highly structured data, and one containing unstructured data. In addition to concerns of design, we will test our approaches to each model side-by-side in the same cloud environment.

Keywords

NoSQL, RDBMS, SQL, Cloud, Big Data, MongoDB, PostgreSQL, Elasticity, Structured Data, Unstructured Data

INTRODUCTION

The rise of cloud computing in the business consciousness has opened an ongoing debate regarding the pros and cons of using traditional Relational Database Management Systems or document and key-value based NoSQL (Not-Only SQL) implementations. Cloud computing was first utilized in the 1950's, but only after the dot-com bubble burst of the early 2000's did it grow into the standard we know today. Amazon's Elastic Compute Cloud (EC2) on the AWS platform was released in 2006 and took advantage of low-cost servers and high-bandwidth networks first offered by Grid and Utility computing methods.[1] The merits of reliability, scalability and accessibility of data quickly led the public to see the values of using Infrastructure-as-a-Service over dedicated or shared hosting solutions, and several copycat companies have sprung up in recent years as a result.

For database administrators, the challenge has become choosing which among a plethora of ever emerging database types to implement when hosting in the cloud. The relational model has its advantages in documentation, simplicity, familiarity, data integrity and reliability. SQL implementations in the cloud are capable of far more complex queries and aggregations than other solutions.[6] They support transactions to ensure only atomic changes are made to your data while keeping a single master copy by which all other copies replicate. The drawbacks come when a developer needs to scale outward across multiple servers. Standard RDBMS's have trouble with efficiently expanding a database due to their inherent complexity of organization.[2] Security is another concern for developers. RDBMS implementations in the cloud are vulnerable to SQL injection, something that NoSQL is able to avoid.

Not-Only SQL implementations are either key-value based or document-based and have grown in popularity because of their ease of scalability and improved security. NoSQL implementations are based on entities and support many of the functions of RDBMS such as sorting, indexing, projecting and querying but are not as reliable or effective when using complex database models. Transactions that guarantee atomic consistency are not supported and updating the database across multiple entities is an eventual process. Joins and ACID guarantees are traded off in favor of transaction speed. Scalability is where NoSQL earns its points, however, the quick propagation of new servers allows developers to rapidly grow or shrink their database to meet the demands of their users.[9]

PostgreSQL has become one of the most popular open-source implementations of RDBMS for online applications. An Object-Oriented RDBMS, like Postgres, is highly efficient with complex structured data models and queries. MongoDB is currently the most popular NoSQL solution largely because of its simplicity of use and efficiency in clustering data. Unstructured data such as that gathered by social media websites is presumed to be best handled by NoSQL implementations. PostgreSQL and MongoDB are both open source and will be the focus of this paper.

Relational Databases and NoSQL in the Cloud

Relational Database Strengths

Relational databases have been the industry standard for several decades and with that comes years of experience and documentation from professionals who have been using the same paradigm for much of their career. A major strength to point out here is the maturity of relational databases. Years of improvements and tweaking by major for-profit players like IBM, Oracle, Microsoft and the open-source community virtually ensure that most avenues have been explored and optimized in the name of individual market-share. For example, the mathematical principles that underlie relational theory have been debated and sharpened for decades to provide a foundation that NoSQL implementations cannot match.[10] An example of improvement came with the introduction of the OODBM (Object-Oriented Database Relational Model), which addressed the need for a more programmatic flexibility to storing procedures and utilizing the benefits of object oriented theory. Another improvement for database administrators and developers is the level of support that comes with using a widely-known RDBMS. Widespread use over the decades means at some point, someone has come across your current problem and is able to offer a solution to start things back running. Perhaps understandably, the recent upsurge in NoSQL has many proponents of the relational model up in arms to defend its merits.

To start, in cloud implementations where consistency of data is paramount, RDBMS is universally considered the best choice. ACID compliance ensures transactions are completed in a single instance will give Relational databases an edge over their NoSQL counterparts when the need for consistency is imperative. Similarly, the atomic nature of relational databases guarantees that the database will be space-efficient compared to a document-based implementation. To provide an example, transactions are required to be completed successfully before changes are committed to the master database, and the master then updates the remaining copies (if they exist) across the cloud. RDBMS also offer greater control over structuring of data, enabling complex data models to be implemented in situations where this is necessary. The ability to create complex joins also gives a leg up over the competition, enabling the database administrator to formulate deeply complex queries for results that is not easily implemented by document and key-based databases.

In business application settings such as those used by banks and credit unions, ACID compliance is a must. Consistency of data is imperative for obvious reasons and the trade-off for quick availability offered by NoSQL would not make sense. The need for a complex system of accounts, customers, staff and branches means something more robust is called for when generating queries. Needless to say, the ability to perform joins between many tables would be indispensable for a database administrator dealing with intricate relationships. Thus for highly structured, complex settings it is assumed that RDBMS should be used over other NoSQL options.

Relational Database Weaknesses

One of the major arguments against RDBMS is a difficulty called impedance mismatch. The issues here arises when a programming language (typically object-oriented) has difficulty in finding a simple way to interact with a complex database structure. The problem called for a major shift in RDBMS and eventually the solution of OODBMS, referenced earlier, was introduced to bridge the gap. NoSQL implementations also claim to have an advantage over the relational model because of the simplicity of their API. [5][3]

Another drawback of choosing a RDBMS is the amount of time needed to design and normalize an efficient database and bring it to production. Primary Keys, Foreign Keys, Normal Forms, Data Types and similar design steps must be properly understood and applied to avoid mistakes and inconsistencies in data down the road. Furthermore, when designing the front-end program which rests on top of the database, complex queries and joins from multiple tables and columns may be necessary to get the required data in a single view. For business owners looking for minimal investment in database design and maintenance, these distractions will make them second guess whether a simpler, and most importantly, a faster deployed method would be beneficial.[2] Business owners must evaluate the best cost-return of their investment and oftentimes with complex relational databases, the owner(s) will need to hire both a database designer to effectively normalize and organize the resulting database, and a database administrator to maintain the inevitable technical issues that will arise after deployment. New and learning database admins will likewise be turned off by the complexity necessary to develop and maintain a relational database and have increasingly turned toward simpler implementations for administration.

The issue of scalability is perhaps the most-oft cited disadvantage of classic relational databases.[13] The consensus is that relational databases are able to scale just as effectively as their NoSQL counterparts, but only to a certain extent.[2] The main problem comes during updates to existing data. In an ACID-compliant relational database system, when someone accesses and updates a value within the master copy of the database, that value will be locked from updates while the transaction is duplicated across all other servers. It is only released when each individual copy is identical to keep with the requirements of consistency. When in situations where there are tens or hundreds of thousands of updates every minute and growing, this can quickly become a problem where database availability and speed suffer.

Google's BigTable implementation in 2006 spurred the popularity of NoSQL databases and perhaps best outlines the shortcomings of relational database scalability. Google needed a solution for its ever-growing collection of semi-structured data that was distributed across multiple data centers worldwide. The relational model they had been using was unable to accommodate such a large pool of data efficiently enough, and Google developed BigTable as a document-based database that now handles most of their infrastructure.[6]

2.3 NoSQL Database Strengths

NoSQL databases follow the BASE (Basically available, Soft state, Eventually consistent) paradigm and have become the preferred method for developers when the need for scalability, availability and simplicity is paramount. NoSQL is considered the best use for unstructured data because it deals with documents organized into entities rather than tables with column-row restrictions. Each entity can be described as a single unit of information coded into byte-data for storage. The key-value or document nature of the database design means NoSQL is schema-less although some proponents claim there is always an implied schema which is not as rigid as that of relational databases.[3][4]

Scalability was the driving force behind the development of NoSQL databases. The entity nature of the design means single documents or key-value pairs can be moved among different servers without affecting the integrity of the database as a whole. This makes for easy horizontal scaling and redundancy of data. Companies like Google, Yahoo and Amazon have all adopted this method to deal with the ever-increasing amount of data collected by their systems.

Availability is assumed to be of utmost importance in NoSQL implementations. Redundant replication of documents across the cloud helps protect against data loss as well as ensuring a faster communication between server and workstation. This works best with read requests in systems like social media websites, where worldwide users demand access to similar data across the globe. Updating documents are done through a method known as eventual consistency.[7] The nearest document is first updated and saved, then progressively replicated across all other copies in the cloud. This saves in write time where a relational database would wait for all instances of the data to update before saving the update.

NoSQL developers love the simplicity of developing and deploying with open-source implementations such as MongoDB or CouchDB. Design of a database is minimal and a document can contain any type of information as long as it remains accessible by its key-value or document number. For programmers, NoSQL APIs to query and update data are simpler to structure and than the often complex queries required in traditional SQL RDBMS. Perhaps the most important aspect is the speed with which developers can go from concept to production. Implementations like MongoDB are designed for quick setup to help developers realize their proof-of-concept in comparatively minimal time.[6] Maintenance after deployment is also kept to a minimum and usually dependent on the reliability of the cloud servers that data is stored on.

For websites dealing with unstructured and semi-structured data, it is usually easier to store all the info needed in a single document. For example, The Guardian newspaper uses a NoSQL implementation on their website's backend to store all information of a single article including title, links, content and comments. The advantage is seen when compared to a relational database where they would perhaps need to query several different tables to acquire the data for a single page.

2.4 NoSQL Database Weaknesses

An obvious weakness in the NoSQL approach is the unreliable nature of data availability. Eventual consistency means that two people with access to different copies of the same document may see updated and non-updated versions if accessed shortly after a write operation. Another inherent flaw in the eventual consistency model is the case in which the servers that contain updated data crash before non-updated servers are able to synchronize with the new information. This is unlikely due to the speed of write but an issue that should be weighed when considering data integrity and reliability.

Redundancy of data and the storage of unstructured data can be a heavy financial burden on business owners given the storage space required to maintain a reliable set of documents.[7] The cost-benefit of this must of course be weighed against the costs of hiring a developer and system administrator to maintain a traditional relational database.

NoSQL as a technology has been around for less than a decade and still very immature in comparison to its relational counterpart. Though growing in adoption, NoSQL databases make up just a small fraction of the total number of implemented databases, with the result being less documentation and a system that has not had the benefit of years of improvement and developing. Should a business owner decide to hire a NoSQL developer to install and maintain a database, (s)he would have difficulty finding an experienced professional given the technology's young age.[8]

The biggest weakness of any database can be set out using the CAP theorem. The theorem states that any database implementation must choose two of three properties:

Consistency, Availability or Partition Tolerance.[5] RDBMS implementations tend to concentrate on Availability of and Consistency while NoSQL implementations favor Consistency and Partition Tolerance. Each has its advantages in a given situation and we aim to test the performance of both in structured and unstructured scenarios.

TESTING

Considering the strengths and weaknesses of RDBMS and NoSQL, we have devised two data models, one containing highly structured banking data, and containing less structured data from an online blog. We have set up two cloud servers to host each scenario, each implementing the data model in both MongoDB and Postgres. The servers used are the most basic Ubuntu Linux servers offered by Amazon Web Services, Ubuntu 12.04.3 LTS. We have loaded each database with identical data.

Aside from testing the capabilities of both database engines to complete bulk writes and reads, we have chosen two practical problems to solve on both data models. We will document the design of the solutions and the execution time for these queries, in order

Bulk Operations

The initial test for each database was a script containing basic Create, Read, Update, and Destroy operations.

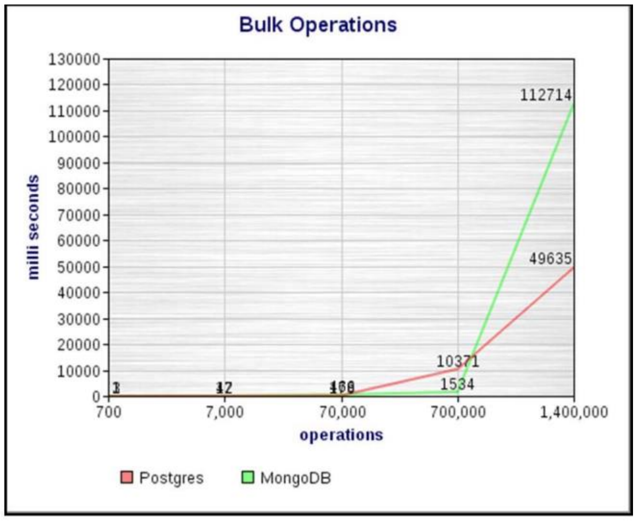


Figure 3.1 Graph showing operations per millisecond. Scripts cycled through CRUD operations.

Highly Structured Banking Data

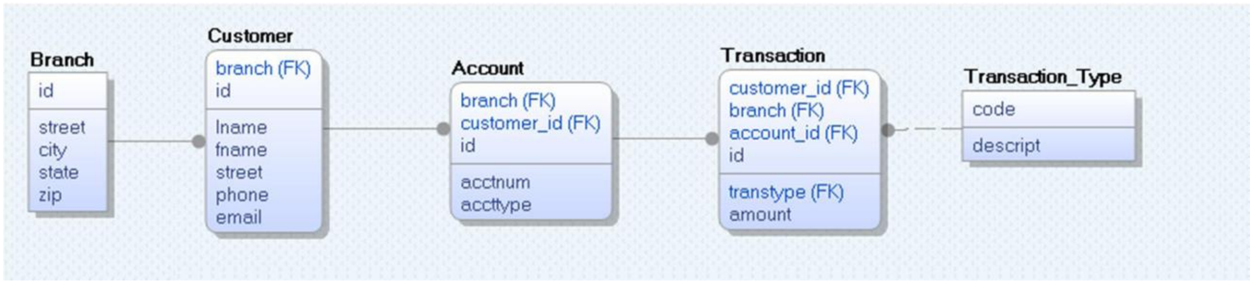


Figure 3.2: Entity Relation Model for Banking Data Model.

RDBMS Approach

Problem 1: Return the customers who bank in the state of Georgia and have deposits over $3000.

This query could practically be used for a bank to suggest financial products to customers, or to decide who qualifies for loans and lines of credit. We solved this problem multiple joins.

```
SELECT DISTINCT a.customer_id
FROM branch b JOIN (customer c JOIN (account a JOIN transaction t
ON a.branch = t.branch AND a.id = t.account_id)
ON c.branch = a.branch AND c.id = a.customer_id)
ON b.id = c.branch
WHERE amount > 3000 AND transtype = 'DP' AND state = 'Georgia'
```

Figure 3.2.1.A: This query took 448 milliseconds.

Problem 2: Aggregate all the transactions in the database. For this query, we used two nested sub-selects in the FROM clause, we returned an aggregation of the transaction amounts, based on the type of transaction.

```
select a - b
from
(select sum(amount)as a from transaction where transtype = 'DP') as x,
(select sum(amount)as b from transaction where transtype = 'CH') as y;
```

Figure 3.2.1.B: This query took on average 7959 milliseconds.

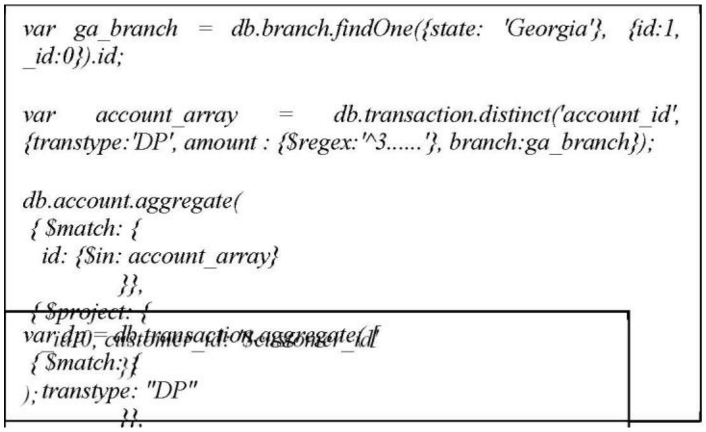
NoSQL Approach

Problem 1: Return the customers who bank in the state of Georgia and have deposits over $3000.

MongoDB does not support joins or references to another collection directly within a query. For the MongoDB equivalent of this query, we reference stored procedures and attributes in our queries.

Conclusions

Postgres has the advantage of having the join operation, which connects relations and stores query results easily. Unless the data structure is heavily nested, Mongo relies on stored functions to reference other collections. Practically, for web applications, Mongo's apparent disadvantage may not be as prominent, when considering the ease at which Mongo queries can be built into javascript and other scripting languages.

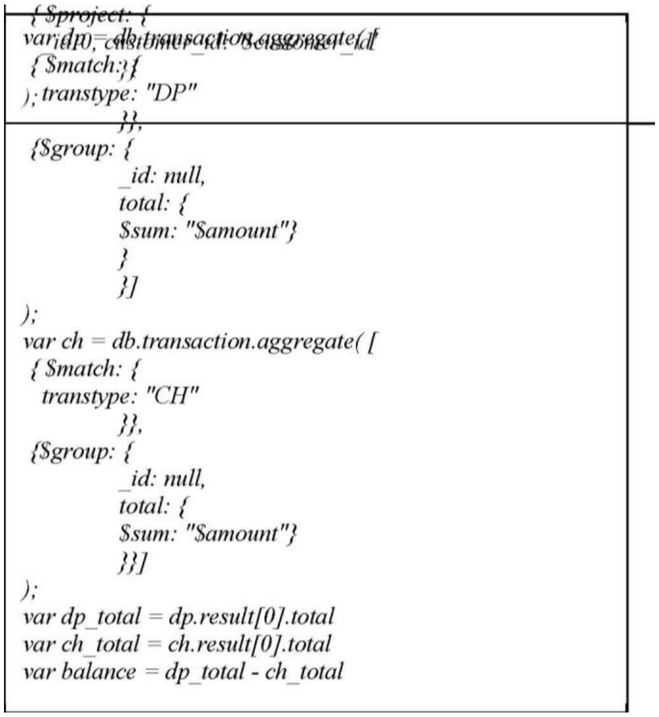


```
var ga_branch = db.branch.findOne({state: 'Georgia'}, {id:1,
_id:0}).id;

var account_array = db.transaction.distinct('account_id',
{transtype:'DP', amount : {$regex:'^3......'}, branch:ga_branch});

db.account.aggregate(
 { $match: {
  id: {$in: account_array}
          }},
 { $project: {
  _id:0, customer_id: '$customer_id'
var dp = db.transaction.aggregate( [
 { $match: {
); transtype: "DP"
          }},
```

Figure 3.2.2.A: This query took 2456 milliseconds to complete



```
 { $project: {
var dp = db.transaction.aggregate( [
 { $match: {
); transtype: "DP"
          }},
 {$group: {
          _id: null,
          total: {
          $sum: "$amount"}
          }
          }]
);
var ch = db.transaction.aggregate( [
 { $match: {
  transtype: "CH"
          }},
 {$group: {
          _id: null,
          total: {
          $sum: "$amount"}
          }}]
);
var dp_total = dp.result[0].total
var ch_total = ch.result[0].total
var balance = dp_total - ch_total
```

Figure 3.2.2.B: This query executed in an average of 8897 milliseconds.

Unstructured Data

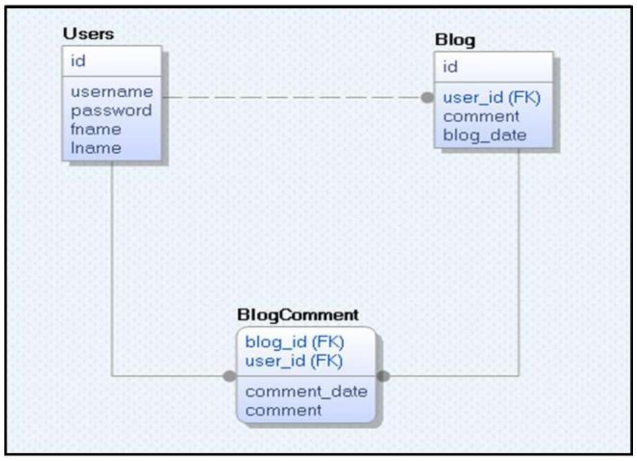


Figure 3.3: Entity Relation Model for Blog Data Model.

RDBMS Approach

Problem: finding a sub-string within user populated data.

For an application such as a blog or social media a large part of the data is usually populated with user input. There is no guarantee as to what this data may contain. For a user to be to search for content in an application with large amounts of text, an efficient string search function is required.

```
select id from blog where comment like '%quasi eligendi%';
```

Figure 3.3.1: This SQL string-search query take on average1061 milliseconds.

NoSQL Approach

```
db.blog.find({comment: /.*quasi eligendi.*/i}, {_id:0, id:1})
```

Figure 3.3.2.A: This Mongo string-search query through the entire collection takes on average 4784 milliseconds.

The alternative is to use MongoDB's indexing function, which is simple to implement. To apply an index on the comments field of a blog is a heavy, one-time expense of 186,433 milliseconds; however, the payoff is an even faster query, which can be applied over the same field indefinitely by the user.

```
db.blog.ensureIndex( { comment: "text" } ).explain()
db.adminCommand({ setParameter: 1, textSearchEnabled: true })
db.blog.runCommand('text', {search: '.*quasi eligendi.*'} )
db.blog.runCommand('text', {search: '.*repellendus eos fugit.*'} )
db.blog.runCommand('text', {search: '.*repellendus eos fugit.*'} )
db.blog.runCommand('text', {search: '.* quo ex sequi.*'} )
```

Figure 3.3.3.B: After the indexing, which takes on average 186433 milliseconds, the text search commands can run at an average of 150 milliseconds, and as fast as 103 milliseconds.

Conclusions

Considering the initial bulk operation benchmark documented in [section 3.1], as well as the performance with the Banking Data, the advantage leans towards Postgres as the more efficient database engine. However, blind operations with no practical application do not decide a comparison for two different methods of design and use. Upon reflection of the tests applied to these two very distinct systems, it is apparent that more research is necessary. MongoDB has a framework capable of very efficient optimizations, like indexes, complex aggregation, and even Map Reduce. SQL-based Relational Database Management Systems like Postgres are still powerful and sometimes under-utilized engine capable of extreme complexity. Without considering the specific application for data definition and data querying, it is still hard to answer the question of supremacy. For the common programmer, the decision may even come down to a preference between procedural and functional programming. For the more informed subject, the decision will likely depend on the client specification for what specific features an application's data should support.

ACKNOWLEDGMENTS

Our thanks to Postgres and MongoDB for providing open source database software.

REFERENCES

[1] The History of Cloud Computing & AWS Services. Retrieved November 15, 2013 from http://www.newvem.com/cloudpedia/the-history-of-cloudcomputing/

[2] David DeWitt, Avrilia Floratou, Jignesh Patel, Nikhil Teletia, Donghui Zhang. Can the Elephants Handle the NoSQL Onslaught? *Proceedings of the VLDB Endownment* 5, 12 (August 2012), 12 pages.

[3] Fowler, Martin. 2012. Introduction to NoSQL. Video. (19 February 2013). Retrieved December 1, 2013 from http://www.youtube.com/watch?v=qI_g07C_Q5I

[4] Marin Fotache, Dragos Cogean. 2013. NoSQL and SQL Databases for Mobile Applications. *Informatica Economica* 17, 2 (February 2013), 14 pages. DOI:10.12948/issn145305/17.2.2013.04

[5] Greg Burd. 2011. NoSQL. (October 2011), Retrieved November 22, 2013 from https://www.usenix.org/legacy/publications/login/201110/openpdfs/Burd.pdf

[6] Rick Cattell. 2010. Scalable SQL and NoSQL Data Stores. *SIGMOD Record* 39, 4 (December 2010), 16 pages. DOI:10.1145/1978915.1978919

[7] Ioannis Konstantinou, Evangelos Angelou, Christina Boupouka, Dimitrios Tsoumakos, Nectarios Koziris. On the Elasticity of NoSQL Databases over Cloud Management Platforms. *CIKM '11* Proceedings of the 20th ACM international conference on Information and knowledge management (2011), Pages 23852388. DOI:10.1145/2063576.2063973

[8] Michael Stonebraker. Stonebraker on NoSQL and Enterprises 54, 8 (August 2011), 3 Pages. DOI:10.1145/1978542.1978546

[9] Chi-Hung Chi, Guillaume Pierre, Zhou Wei. Scalable Join Queries in Cloud Data Stores. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012), Pages 547-555. DOI:10.1109/CCGrid.2012.28

[10] Jeff Cogswell. 2012. SQL vs. NoSQL: Which is Better? (July 2012) Retrieved November 28, 2013 from http://slashdot.org/topic/bi/sql-vs-nosql-which-is-better/

[11] Carlo Curino, Djellel Difallah, Andrew Pavlo, Philippe Cudre-Mauroux. Benchmarking OLTP/Web Databases in the Cloud: The OLTP-Bench Framework. *CloudDB '12* (October 2012) 4 pages. DOI:10.1145/2390021.2390025

[12] S. Strauch, O. Kopp, F. Leymann, F. Unger. A Taxonomy for Cloud Data Hosting Solutions. *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference,* (December 2011) Pages 577-584. DOI:10.1109/DASC.2011.106

[13] Roberto Zicari. Measuring the scalability of SQL and NoSQL systems. (May 2011) Retrieved November 23, 2013 from http://www.odbms.org/blog/2011/05/measuring-the-scalabilityof-sql-and-nosql-systems