International Journal of Computer Systems (ISSN: 2394-1065), Volume 2– Issue 3, March, 2015
Available at http://www.ijcsonline.com/

Loose Coupling between Cloud Computing Applications and Databases: A Challenge to be Hit

Ebtehal Nassar, Ehab Ezzat and Sherif Mazen

Information systems Department,
Cairo University, Giza- Egypt

Abstract

One of the main characteristics of database driven applications is the strong tight coupling between application and database. Thus, any required change in the database schema leads to change in database access layer. Problems elicit during design phase (specifically in database schema defining and representation), development phase, and maintenance. If the database is shared between different applications, applying changes will be more difficult. The available solutions proposed adding a middle layer that receives query and returns results in a unified form. This research focuses on the limitations of tightly coupled architecture. In addition, it hits the challenge of loose coupling between cloud computing applications and databases by encapsulating the database access code and queries in a separate layer.

Keywords: Tight and loose coupling, cloud computing, and application architecture.

I. INTRODUCTION

The term tight coupling means a type of coupling in which software components/layers are dependent upon each other. This means that if any one of these components is modified the other components also should be adapted to be able to deal with this modifications [1]. Also, if the codes are duplicated in different places this is a facet of tight coupling, because changes in codes will need modification in different places or components.

The default design of - database driven- application is to put database access code and queries inside the application implementation files, so if any changes required in the database (database refactoring, database type change or maintenance), the application will be affected and most probably it may need to be rebuilt.

Maintenance of application for bug fixing need changes in application code and it will be hard to identify if the bug exists in database access code or in business logic code. If the application is distributed the problem will be more complicated; as changes must be applied to all parts of the application. This means that there is a tight coupling between application and database and this introduces a lot of problems if the database change is required.

As web applications need to be loosely coupled to be rapidly and easily scaled up [2], we try to apply this concept on the relationship between application and database. Since cloud computing depends on the paradigm of offering everything as a service by supporting loosely coupled components there is a need to make cloud components, as loosely coupled as possible [3].

This paper is categorized as follows, in section II we will introduce the main problems of tight coupling and how these problems exist between the application and the database, and how loose coupling will help in making the change of database types or schemas easier and faster.

In section III, we briefly present the researches that try to decrease the change in application if database changed. In section IV the proposed solution is highlighted, introduces how to use a middle layer as an approach to achieve loose coupling between applications and databases. In section V, a comparison between the proposed solution and the existing database driven applications (three tier architecture) is presented. Finally section VI concludes the paper and drives the future work.

II. TIGHT VS. LOOSE COUPLING

Different researches compared tight coupling versus loose coupling according to different perspectives. One of these comparisons is presented in [4]. This comparison measure tight coupling between software components/layers according to the following perspectives:

- Physical coupling (it considered there is tight coupling if there is a must to have a direct physical link between the components is required)
- Communication style, if the communication is synchronous this means that each component will wait to the other and this gives tight coupling.
- Type system, if it‟s an interface semantics it considered tight coupling because the interface semantics must be defined between the communicating components.

- Interaction pattern, the interaction pattern is considered tightly coupled if it‟s designed as OO-style of interaction; it must know how the object relates to other objects not only the logic of the desired object.
- Control of processes is considered tightly coupled if the processes are managed centrally.
- The service discovery and binding, it‟s related to the Service Oriented Architecture, if the service discovery is statically bounded it considered tight coupling case.
- Platform dependencies, it‟s considered tightly coupled if there is dependence between application/component and the platform it will run on.

This comparison is summarized in Table I.

TABLE I. TIGHT VERSUS LOOSE COUPLING

Level	Tight Coupling	Loose Coupling
Physical coupling	Direct physical link required	Physical intermediary
Communication style	Synchronous	Asynchronous
Type system	Strong type system (e.g., interface semantics)	Weak type system (e.g., payload semantics)
Interaction pattern	OO-style navigation of complex object trees	Data-centric, self-contained messages
Control of process logic	Central control of process logic	Distributed logic components
Service discovery and binding	Statically bound services	Dynamically bound services
Platform dependencies	Strong OS and programming language dependencies	OS- and programming language independent

Also there is a special comparison presented by Cesare Pautasso and Erik Wilde in [2] to measure coupling level for web services, this comparison is specified to the web services, it measures coupling degree by testing different facets like: discovery of the service, identification, binding and others. These facets and the degree of coupling for each one, are presented in table II.

TABLE II. COUPLING FACETS

	facet	Tight coupling	Loose coupling
1	Discovery	Registration	Referral
2	Identification	Context-based	Global
3	Binding	Early	Late
4	Platform	Dependent	Independent
5	Interaction	Synchronous	Asynchronous
6	Interface orientation	Horizontal	Vertical
7	Model	Shared Model	Self-Describing Messages
8	Granularity	Fine	Coarse
9	State	Shared, Stateful	Stateless
10	Evolution	Breaking	Compatible
11	Generated code	Static	None/Dynamic
12	Conversation	Explicit	Reflective

A. Problems of tight coupling

Tight coupling arises in different categories of computer science; in application structure, coupling exists if there is duplication of written codes inside different parts of the application. Another factor that can be tested to check tight coupling is the control of process logic. If the control of process is managed by a central process this will lead to tight coupling [4].

One of the most important factors used for testing application architecture, is measuring tight coupling. For example tight coupling is considered one of the main drawbacks of client/server architecture, it was concluded from need for control the communication‟s ends between client and server to get the client work with the server software. So if the programmer needs to update or change something in the server software s/he should apply changes to all clients deal or communicate with this server, and this is one of the very clear tight coupling problems [5].

Another tight coupling problem was detected in the service composition, in this process tight coupling detected from the ability to extract the description language of the web services, and the implementation language details of them. Using tight coupling service composition limits the creation of user- generated services and contents [6].

B. Advantages of loose coupling

One of the main advantages of use loosely coupled architecture is to makes the application appears more agile and enables faster change. Also it increases system maintainability [4]. The current development of services resides on the concept of loose coupling [2]. Service Oriented Architecture (SOA) depends on the loose coupling of components as it makes the creation of systems easier by composing services together and it enables the services and components update without disrupting other components that interacts with these changed components [7].

Loose coupling implies that services share a small set of assumptions so the impact of change is very limited, and therefore the services can be considered independently. According to that the loosely coupled systems can be easily scaled.

In usage of network, developers used the concept of loose coupling to make applications dependent from the network protocols. This helps applications to use network infrastructure without need to write specific codes for specific network protocol which give the ability to switch between different protocols without affecting the application code [4]. This is showed in figure 1.

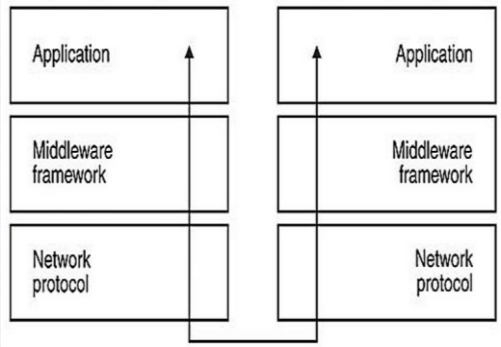

Figure.1 A communication middleware framework to isolate the application developers from the details of the network protocol [4]

As previously mentioned, the use of loosely coupled architecture helps in systems‟ easy maintenance and scaling up, which is one of the main requirements of web applications and specifically on cloud environment [8]. Also the loose coupling becomes more important according to the need of satisfying the user requirements. And as the requirements on the cloud environment change rapidly the loose coupling architecture may be the best solution.

C. The coupling between application and database

According to the different facets of coupling exist in the software architecture we try to measure the coupling degree between applications and database. First, in the application design process, one of the main steps is to create the database schema because the application development will rely on it.

The second issue, after the application being built if there is a need of database refactoring [9] the application developer should change the database access code related to the changed parts to be able to deal with the database changes.

According to the new paradigm of using NoSQL stores, if there‟s a need to use a NoSQL store for an application uses a relational database, we will need to change all database access code. As these issues show a very clear type of tight coupling there is a need to find a solution to ensure loose coupling between application and database.

III. RELATED WORK

Nowadays most of the applications going to use NoSQL stores, but if application is built on a relational database it will need to be rebuilt to access NoSQL stores [10]. One of the main benefits we may get by make loose coupling between application and database is to replace relational databases with NoSQL stores without application change.

A good work was previously done to present NoSQL stores [11], [12], [13]. But each one of these NoSQL stores has its own structure and to use it, it‟s required to build the application according to it.

The existing solutions to minimize the change in the application upon the database change are presented in this section.

A. Migration process from relational to NoSQL databases

The usage of cloud computing applications exceeds the capabilities of the relational databases; these applications need to accumulate and analyze a huge amount of data daily. The usage of relational databases to complete these tasks faces a lot of challenges and problems. These problems may arise from the need of data sharding. A research is done to give guidelines to migrate data from relational databases to NoSQL stores to pass these problems and get the opportunity to get the benefits of NoSQL stores. [16]

B. Extending MySQL into NoSQL

Another approach to overcome the need to change everything to switch from SQL to NoSQL is to extend MySQL into NoSQL; this approach provides the usage of Object Relational Mapping (ORM) to map objects from the developer code to tables in the database. Then partition these tables into database nodes.

They designed a unique ID schema that‟s optimized for the primary key look up operations. The shared key information is encoded into its data ID. By using this technique, queries with data ID can go directly to the target node without the need to know the key that the data is distributed on.

A data access framework on top of MySQL is provided to guarantee a fast and robust data access with availability and scalability. The target of this framework is to achieve data access with high scalability and availability; those targets are achieved by the following:

- Horizontally portioning of tables into different database nodes to have high scalability.
- Automation of the procedures for master failover and providing an online tool for shared rebalancing to have high availability.
- Leverage MySQL storage engine for robust data management.
- Design of a unique ID lookup operation to get fast data access with multiple data nodes.
- Support of HandlerSocket plug-in that gives the ability to read requests routed to replica nodes; this will support fast access on the level of a single node.

As presented in figure.2 the architecture of the system is based on portioning the database tables horizontally into multiple database nodes. The tables‟ rows are separately stored based on a certain key. The key systems are done using hash-based sharding and range-based sharding. [15]

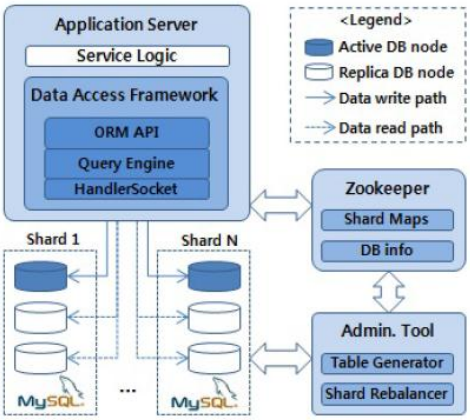

Figure.2 The structure of extinding MySQL database into NoSQL

C. SQL Processing Data in NoSQL

An approach of SQL processing Data in NoSQL is presented in [18].

This approach presents a method to have SQL-like commands to manipulate data on NoSQL stores. It processes structured data in NoSQL stores using MapReduce [19]. It transforms relational databases into a NoSQL structure; and the data is manipulated by a series of MapReduce functions then integrated into a framework to provide SQL-like queries. This solution provided its own MapReduce codes to process different ANSI- SQL queries.

D. Universal Query Language (UQL)

A trial for having a unified query language is introduced in [20]. By considering all objects are in the form of Unified State Model (USM) which is a universal model of objects; and introduces a universal query language to deal with this model of objects. It deals with objects as a set of three components: atomic values, external names, and identities. The value of an object can be a set of objects; this means that the object can be composite object.

UQL operators are as follows: renaming, flatting, mapping, evaluating, getting k-th subobject, filtering, nesting, cloning, product, grouping, transposing and folding.

An example of mapping SQL into UQL:

Assuming the following relational database schema with two tables:

emp: empno, firstname, lastname, salary, deptno

dept: deptno, deptname, location

SQL query to be operated:

SELECT firstname, lastname

FROM emp

WHERE lastname = 'Schmidt'

Assuming the state of this database is „o". The form of this query in UQL will be:

$$map(\ filter_{name='firstname' \vee name='lastname'}(\ filter_{\exists on: name(on)='lastname' \wedge value(on)='Schmidt'}(\ flatten(\ filter_{name='emp'}(o)\ )\ )\ )\ )$$

By using this approach all database types will be treated by the same way. As all data objects will be mapped to a universal state model and can be queried by a universal query language.

E. SQL++

A new query language was developed to solve this problem by writing queries that can run on SQL and NoSQL databases it"s called SQL++ [14]. This query language"s purpose is to write queries that can be useful for writing software that interoperates between different NoSQL databases or between SQL and NoSQL databases.

SQL++ is a data model and query language that can deal with SQL and NoSQL databases. Its target is to decrease the change in the application if we need change in the database and make the application deal with different types of databases; and proposing a unifying query language to deal with SQL and NoSQL databases with the same code.

It uses a middle layer to execute queries over different types of databases and mash up results send back to client.

On SQL"s end, a database has a fixed schema comprising flat tables, where a table is a set of homogeneous tuples and each tuple is a set of scalar attributes. The SQL++ data model is designed as a superset of both SQL"s relational tables and JSON.

SQL++ has its own data model like:

```
any

{ location: string, * }

{
  location: string, readings: [
    {
      time: timestamp, ozone: number|string,
      no2?: number, co?: number
    }
  ]
}
```

It deals with data as a JSON (JavaScript Object Notation) object [21]. SQL++ introduced a framework for their middle layer (Forward middleware).

It's showed in figure 3.

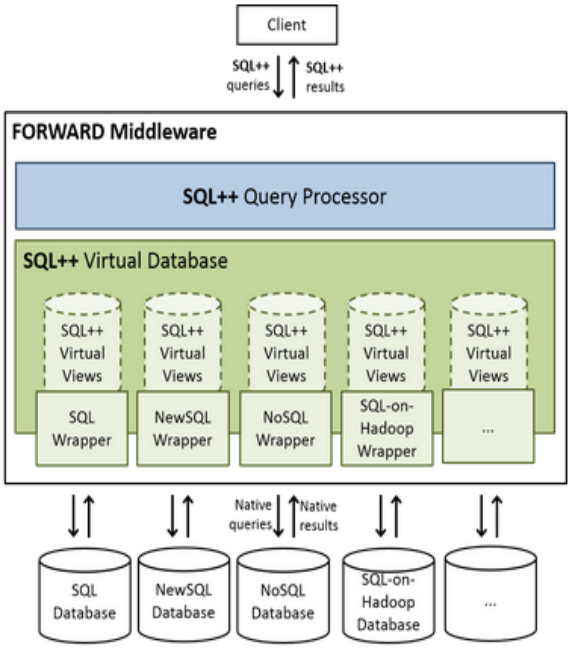

Figure.3 SQL++ structure [14]

IV. THE PROPOSED APPROACH LOOSE COUPLING MIDDLE LAYER

According to the tight coupling between application and database we try to move the database access codes to a separate layer. So, if any changes needed to be applied on database like: database refactoring or database type change, the database access code will be updated in one place only which is the middle layer. This means that the application itself will not be affected with any changes in the database.

As presented in figure.4 the structure of our design is to have a middle layer between the application and the database; compared to the three tier architecture presented in figure.5 where the database access codes are stored inside the application itself. The middle layer will be responsible for dealing with the database and return the results to the application.

The database access codes, metadata and queries will be written inside this layer. In this case the application will call a function in the middle layer; this function will send query to the Database Management System, get results and return to the application. Inside the function in the middle layer the DB connection code and queries will be written.

By applying this design, the application will not need to know the structure of the used database and will not be affected if we need to change the database schema or type used. The application won't include any queries or metadata about the database.

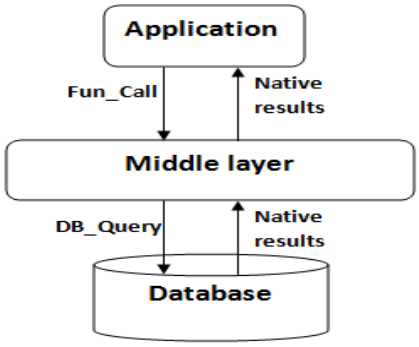

Figure.4 the structure of the proposed design

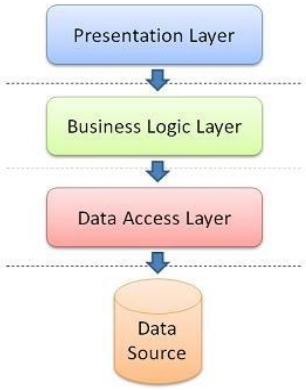

Figure.5 the three tier architecture [17]

Using the middle layer will help in communication with the database without the need to know the database type, server or schema. And this increases maintainability; interoperability and flexibility of database refactoring and change without affecting applications.

V. THE PROPOSED LAYER DIFFERENCE

As presented in the related work section, the existing solutions depend on one of the coming approaches : the first one is trying to migrate data from relational to NoSQL stores or vice versa, like in "Migration process from relational to NoSQL databases" [16], the second one is to extend MySQL database into a NoSQL store to get the benefits of NoSQL stores as presented in "Extending MySQL into NoSQL"[15], with the same concept "SQL Processing Data in NoSQL" [18] is to process SQL data in NoSQL stores.

Another approach is to have a universal query language to run on different types of databases like relational and NoSQL stores. These solutions are presented in "Universal query language" [20] and "SQL++" [14].

The target of all these solution is to decrease the change in application when there is a need to change database type. But the problem still exists because the database access code written inside the application code; any changes in the database like: database refactoring, database type change will need changes in the application code.

SQL++ (the unified query language) project used a middle layer to process database queries on different types of databases and mash up results from different stores; but it still has the same problem as other solutions; because the data access codes are written inside the application. The queries must be written inside the application and with well known of database schema.

To the best of our knowledge, there is no solution presented to use a programming middle layer to be used between the application and database to handle the queries and guarantee the loose coupling between the application and the database. A comparison between the application with the use of a middle layer and the database driven application (three tier architecture) is presented in table III.

TABLE III. COMPARISON BETWEEN DATABASE DRIVEN APPLICATION AND USING A MIDDLE LAYER

Comparison elements	Database driven application (three tier architecture)	The application with the use of middle layer
Knowledge of the database schema during the design phase	Must know the database schema and type	Application : No need
Knowledge of the database schema during the development phase	Must know the database schema and type	Application: No need
Maintenance (Bug fixes)	There will be difficulty in deciding in which part the bug exists	Application: if the bugs are in the application the change will be done once
Database refactoring (example: normalization of tables, drop column, make a nullable column not null ..)	It˝s a must to change the database access code related to the changed tables	Application: No changes needed
Change database type from relational to object oriented or NoSQL	It˝s a must to change all database access code inside the application to apply the changes	Application: No changes needed

According to comparison between the database driven application (three tier architecture) and the application with a middle layer, we need to measure the coupling degree between the application and the database which is presented in table IV.

TABLE IV. COMPARISON BETWEEN DATABASE DRIVEN APPLICATION AND APPLICATION WITH A MIDDLE LAYER WITH RESPECT TO COUPLING FACETS

Coupling facets	Database driven architecture	Application with a middle layer
Physical coupling	Loose Coupling	Loose Coupling
Communication style	Tight coupling	Tight coupling
Type system	Tight coupling	Loose Coupling
Interaction pattern	Tight coupling	Loose Coupling
Control of process logic	Tight coupling	Loose Coupling
Service discovery and binding	-	-
Platform dependencies	Tight coupling	Loose Coupling

As introduced in table IV this is a comparison between the application with a middle layer, and the traditional database driven application, using the coupling measurement by Dirk Slama, Karl Banke, Dirk Krafzig in [4].

VI. CONCLUSION AND FUTURE WORK

This paper has shown that the tight coupling between application and database leads to major problems during maintenance, development, design, database schema defining and representation. In this research a loosely coupled architecture was proposed to ensure that any change in database schema or type will not affect the application that uses this database.

We tried to show the trials that are done to solve the problem of need to application˝s change according to any database˝s change. And we declared that according to our research there is no solution presented to guarantee the loose coupling between application and database.

Our proposed solution is to have a middle layer to separate the database access code from the application. This will lead to a high degree of loose coupling between applications and databases. As a future work to the proposed solution; architecture for the middle layer will be presented and implemented. The proposed layer will be implemented to work on relational databases and NoSQL stores.

REFERENCES

[1] Tight coupling, (Accessed 2015-2-22) http://www.webopedia.com/TERM/T/tight_coupling.html

[2] C. Pautasso, E. Wilde, New York, NY, USA, 18th international conference on World Wide Web: Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design, 2009, 911-920

[3] S., P., and S. Capelli. "A practical and automated approach for engineering service-oriented applications with design patterns." Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International. IEEE, 2014.

[4] D. Slama, K. Banke, D. Krafzig, Service Oriented Architecture: Inventory of Distributed Computing Concepts, chapter 3, 2004.

[5] J. Bloomberg, R.Schmelzer, Service Orient or Be Doomed!: How Service Orientation Will Change Your Business, 2006.

[6] K., S. , et al. "Loosely coupled service composition for deployment of next generation service overlay networks." Communications Magazine, IEEE 50.1 (2012): 62-72.

[7] MongoDB organization, MongoDB , (Accessed 2015-3-30) http://www.mongodb.com/

[8] A. grawal, D., et al. "Data management challenges in cloud computing infrastructures." Databases in Networked Information Systems. Springer Berlin Heidelberg, 2010. 1-10.

[9] P. Sadalage, ThoughtWorks, Refactoring Databases: Evolutionary Database Design, (Accessed 2015-2-22)
http://databaserefactoring.com/

[10] R.Cattell, New York, NY, USA, Scalable SQL and NoSQL data stores in ACM SIGMOD 39(4),2010, 12-27.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver and R. Yerneni, PNUTS: Yahoo!˝s Hosted Data Serving Platform, VLDB Endowment 1 (2), 2012,1277-1288.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshallvand W. Vogels , New York, NY, USA, twenty-first ACM SIGOPS symposium on Operating systems principles, 205-220, Dynamo: Amazon˝s Highly Available Key-value Store, 2007

[13] F. Chang, J. Dean, S. Ghemawat, C. Hsieh, Deborah A. Wallach, M.Burrows, T.Chandra, A.Fikes, and R. E. Gruber, Seattle, WA, OSDI'06: Seventh Symposium on Operating System Design and Implementation, Bigtable: A Distributed Storage System for Structured Data, 2006

[14] K. Ong, Y. Papakonstantinou , R. Vernoux, ,cornell university library, The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases, 2014.

[15] S., H., et al. "An Object-oriented Approach for Extending MySQL into NoSQL with Enhanced Performance and Scalability." DBKDA 2014, The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications. 2014.

[16] G., A., et al. "Building an Experiment Baseline in Migration Process from SQL Databases to Column Oriented No-SQL Databases." J Inform Tech Software Eng 4.137 (2014): 2.

[17] F. Normen, November 2008, Using Web Services in a 3-tier architecture, (Accessed 2015-3-30) http://weblogs.asp.net/fredriknormen/using-web-services-in-a-3-tier-architecture

[18] P., W. "A Method of SQL Processing Data in NoSQL." 2nd International Conference on Soft Computing in Information Communication Technology. Atlantis Press, 2014.

[19] D.Jeffrey, and S.Ghemawat. "MapReduce: a flexible data processing tool." Communications of the ACM 53.1 (2010): 72-77.

[20] Wi.niewski, Piotr, and K.Stencel. "Universal query language." (2012).

[21] JSON organization, Introducing JSON, (Accessed 2015-3-30)
http://json.org/