Endowing NoSQL DBMS with SQL Features Through Standard Call Level Interfaces

Óscar Mortágua Pereira, David Simões, Rui L. Aguiar
Instituto de Telecomunicações
DETI – University of Aveiro
Aveiro, Portugal
{omp, david.simoes, ruilaa}@ua.pt

***Abstract*— To store, update and retrieve data from database management systems (DBMS), software architects use tools, like call-level interfaces (CLI), which provide standard functionalities to interact with DBMS. However, the emerging of NoSQL paradigm, and particularly new NoSQL DBMS providers, lead to situations where some of the standard functionalities provided by CLI are not supported, very often due to their distance from the relational model or due to design constraints. As such, when a system architect needs to evolve, namely from a relational DBMS to a NoSQL DBMS, he must overcome the difficulties conveyed by the features not provided by NoSQL DBMS. Choosing the wrong NoSQL DBMS risks major issues with components requesting non-supported features. This paper focuses on how to deploy features that are not so commonly supported by NoSQL DBMS (like Stored Procedures, Transactions, Save Points and interactions with local memory structures) by implementing them in standard CLI.**

Keywords—NoSQL; SQL; databases; middle-ware; call level interfaces;software architecture.

I. INTRODUCTION

Critical data are mostly kept and managed by database management systems (DBMS). To store, update and retrieve data from DBMS, software architects use software tools to ease the development process of business tiers. Among these, we emphasize call-level interfaces (CLI) [1], which provide an API that allows an application to call methods that propagate to the database.

CLI try to build on the commonalities between DBMS and provide a set of methods that encompass these common aspects. Because all DBMS are inherently different, CLI have two main issues to deal with. Firstly, the way of accessing distinct DBMS is different (protocol, format, query language, etc.), which means every DBMS must have its own implementation, which converts the standard API calls to the proper DBMS format. Secondly, DBMS have different features and support different techniques. CLI try to encompass the most common and often seen capabilities, but some DBMS do not support all of them, while others can support features that CLI do not support. Most NoSQL DBMS, for example, do not support transactions, unlike most relational DBMS.

This paper focuses on how to handle this variety of features supported by different DBMS and focusing primarily on features provided by CLI but not supported by the DBMS. These consist on: 1) transactions, 2) the execution of database functions (like stored procedures) and, finally, 3) interactions with local memory structures, containing data retrieved from the database. We provide a framework that allows a system architect to simulate nonexistent features on the underlying DBMS for client applications to use, transparently to them. It is expected that this research can contribute to minimize the efforts of system architects when DBMS do not support what are considered key features.

The remainder of this paper is organized as follows. Section II presents the state of the art and Section III describes some key functionalities of a CLI (in this case, JDBC). Section IV formalizes our framework, Section V shows our proof of concept and Section VI evaluates our framework. Finally, Section VII presents our conclusions.

II. STATE OF THE ART

There is some work done to bridge the gap between NoSQL and SQL. There have been some solutions focused on providing JDBC drivers to particular DBMS, like [2]–[6], using the DBMS's own query language (usually SQL-like). The authors' approach is to create an incomplete JDBC implementation that delegates CLI requests to the DBMS API and converts the results of queries into JDBC's ResultSet.

There is also work done on translating SQL to the NoSQL paradigm [7]–[12], which allows clients to perform ANSI-SQL commands on NoSQL DBMS. These proposals create a SQL query interface for NoSQL systems, which allow SQL queries to be automatically translated and executed using the underlying API of the data sources.

Work has also been done in an attempt to standardize the access API for NoSQL DBMS. Atzeni et al. [13] propose a common programming interface to NoSQL systems (and also to relational ones) called SOS (Save Our Systems). Its goal is to support application development by hiding the specific details of the various systems. There is also research on cross-database tools that depend heavily on JDBC's features and that cannot be used with NoSQL because their implementations are not complete [14]. To the best of our knowledge, there has not been work done with the goal of implementing CLI's features on drivers and DBMS that do not support them. We expect that our framework positively contributes to overcome the gap between NoSQL and SQL.

(DOI reference number: 10.18293/SEKE2015-070)

III. BACKGROUND

Like previously stated, CLI try to build on the commonalities between DBMS and provide a set of methods that encompass these common aspects. These methods include, for example, reading data from the database, executing commands on it or performing transactions.

Data manipulation commands are usually called 'CRUD Expressions', which stand for Create, Read, Update and Delete Expressions, and represent the most common ways to handle data in a DBMS. CLI also usually allow the modification of data on local memory structures, modifications which are propagated to the database transparently, without a client having the need to execute any CRUD expression.

While most full-fledged DBMS have several complete CLI implementations (Microsoft SQL Server, MySQL, among others), some relational DBMS do not (SQLite, for instance) and most NoSQL DBMS do not either.

A. Java Database Connectivity

The Java Database Connectivity (JDBC) [15] is a CLI API for Java. Because of Java's portable nature, it has been the most popular development language for NoSQL DBMS and, as such, JDBC is the most popular CLI for NoSQL DBMS, even though it is oriented towards relational DBMS.

JDBC Drivers typically return "Connection" objects, which are then used to perform operations on the database. The "Connection" object has a given set of capabilities, which include the creation of CRUD statements to be executed on the database, the creation of statements that call functions inside the DBMS (like Stored Procedures) and the usage of transactions (with commits, roll-backs and save points).

Associated with connections, are ResultSets (RS), which are local memory structures retrieved with "select" queries and representing rows on the database. These use cursors to iterate through their set of data and also allow a set of capabilities, which include retrieval of values from the current row and, if the RS is defined as 'updatable', the insertion or deletion of a row and the modification of the current row's values. These interactions are going to be referred to as 'Indirect Access Mode (IAM) Interactions' through the remainder of this paper.

Listing 1 shows the creation of a statement *stmt*, the retrieval of data from table *table1* and how it is kept in the RS (*rs*). Applications are then allowed to update their content. In this case the attribute *attributeName* was updated to *value* and then the modification was committed. We can see how the update is done without the use of any CRUD expression.

```
stmt = conn.createStatement();
rs = stmt.executeQuery ("select * from table1" );
rs.update("attributeName", value)
rs.commit();
```

Listing 1. A query and the update of a value using JDBC.

The features that the driver supports can be further grouped by category: statements (with or without parameters), execution of database functions (stored procedures or user-defined functions), transactions (and save points), iteration through RS, retrieval of values from RS and IAM interactions.

Some of these features are implemented by all drivers (executing statements on the database, for example). However, the execution of database functions, transactions, save points or IAM interactions is not implemented by some DBMS, depending on their architecture or features. These categories are, then, the focus of this paper.

IV. IMPLEMENTATION FORMALIZATION

To implement these features, there are several options. The first is to create another driver, wrapping the original one, where the methods call the original methods or implement those not supported; the second is to have a server-side middleware layer that intercepts the CLI calls, allows the supported ones and redirects the non-supported ones; the third is to have the clients connecting to the server through a regular socket connection and the server either forwards those requests to a JDBC driver connected to the DBMS or it executes functions from our framework.

While wrapping the driver in another may seem the simplest option (clients can simply use the driver as they usually would, as there is no need for middleware layers to intercept the driver requests or for clients to change the way they connect to the DBMS), it presents some security vulnerabilities, which will be explained further ahead, and also forces the clients to use the modified driver. The second option is the most transparent for clients, but forces a complex implementation on the server, to intercept the JDBC calls and act accordingly, in an imperceptible way for the client. The third option eliminates the need for clients to have any CLI dependency on their code and the server merely acts as a relay from clients to the DBMS. This makes for a simpler implementation of the server logic, but is not transparent to clients. The last two approaches are similar, consisting in a middleware layer able to identify client requests, and any of them are viable. It's up the each system architect to decide which approach suits his needs the best.

For the remainder of this paper, the middleware layer (intercepting the CLI calls) where the extra logic is implemented will be referred to as the "barrier". All the client requests must go through the barrier to access the database. It is able to intercept requests and, instead of forwarding them to the DBMS, provide its own implementation and return the appropriate results to the clients, transparently.

A. Execution of Database Functions

A Stored Procedure (SP) is a subroutine available to applications that access a relational DBMS. Typical use for SP include data validation (integrated into the DBMS) or access control mechanisms. Furthermore, they can consolidate and centralize logic that was originally implemented in applications. Extensive or complex processing that requires execution of several SQL statements is moved into stored procedures, and all applications call the procedures. SP are similar to User-Defined Functions (UDF), with a few minor

differences (how many arguments are returned, ability to use try-catch blocks, among others).

If a DBMS does not allow the definition of SP or UDF, these can be implemented on the barrier as a server-side function that calls a group of SQL statements and operations, which are executed together and, therefore, simulate a SP. By doing so, it is possible to simulate most of the behaviors of SP or UDF.

To detect when the functions that simulate SP should be called, there are multiple ways. A simple one would be to give the client the ability to call a SP by the use of a keyword (e.g., *exec storedProcedure1*), where the SP name would be the function name. On the barrier, when the *exec* keyword was detected, a function with the same name as the one requested would be called with the arguments supplied and the results would be returned to the client.

B. Transactions

A transaction symbolizes a unit of work performed within a database, and treated in a coherent and reliable way independent of other transactions. Transactions in a database environment have two main purposes: to provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure; to provide isolation between programs accessing a database concurrently.

A database transaction, by definition, must be atomic, consistent, isolated and durable (ACID). In other words, transactions provide an "all-or-nothing" proposition, stating that each work-unit performed in a database must either complete in its entirety or have no effect whatsoever. Furthermore, the system must isolate each transaction from other transactions, results must conform to existing constraints in the database, and transactions that complete successfully must get written to durable storage.

The implementation of transactions is a complex engineering problem, heavily dependent on the DBMS architecture. We present a solution that works with most DBMS, but which also depends on the database schema. Our proposal is defined by, after a transaction has been started, executing the statements in the usual manner, but registering them in a list. If a rollback is ensued, using the list, the changes are undone and return the database to its original state. The implementation of transactions inherently involves the implementation of the ACID properties to a group of statements. Consistency and durability cannot be implemented on the barrier, because these are guaranteed by default by the database itself.

To implement atomicity, along with a list of all the executed actions, there is a need for a list of all the statements that reverse those actions, hereafter referred to as the list of *reversers*. All *inserts* are reversed with a *delete*, all *deletes* with an *insert*, *updates* with *updates* and *selects* do not have to be reverted. To reverse the performed actions, the reverser list of actions must be executed backwards.

One needs to pay attention to the database schema and, if an *insert* triggers other *inserts* (for logging purposes, for example), all of their reversers must be added to the reverser list. The same happens for cascading *updates* and *deletes*. These kinds of mechanisms are mostly common in relational databases, where transactions are natively supported, so we expect few practical cases where these become relevant.

As an example, imagine a simple transaction consisting of a bank transfer: money is withdrawn from *Account A* and deposited in *Account B*. The money in *A* cannot fall under 0 and the transaction first deposits the money in *B* and then withdraws from *A*. Currently, *A* has 40€, *B* has 0€ and the transaction is executed for a transfer of 50€. When the deposit is made, *B* has 50€ and *A* still has 40€. Here, the increment is registered in the barrier and the reverser (subtracting 50€) is also registered. Then, the transaction tries to withdraw 50€ from *A* but it fails, because the value would go below 0. Here, the transaction is rolled back and the actions in the reverser list would be executed, subtracting the money added to *B* and ending the transaction.

The fact that CRUD expressions are kept on the barrier also has an advantage when implementing transactions. If they were on the client-side, inside the JDBC driver, it would be the client to keep a list of the reversers needed in case of a rollback. If indeed there was a need for a rollback, the client might not have had the permissions to execute those actions and, therefore, could not rollback. To solve this, special permissions would need to be set for this case and that could lead to vulnerabilities that an attacker could take advantage of.

Formally, our definition states that a transaction is composed of actions (which trigger cascading actions), which affect data in the database. Atomicity in a transaction can be implemented if and only if: for any action in any transaction, all the cascading actions can be found; for any action (or cascading action) in any transaction, there is a reverser; the execution of a reverser undoes all and only the changes made by the original action.

Implementing isolation can be done through the use of a single lock (a semaphore or a monitor), which serializes multiple transactions. This concept can be further extended with multiple locks (for example, one for each table), which would allow concurrent transactions if these transactions interacted with (in this example) different tables. Multiple locks can, however, lead to deadlock issues; to avoid them, either one of the transactions has to be reverted (deadlock avoidance/detection) or the locks must all be done at the start of the transaction and must occur in an ordered manner (deadlock prevention).

Because the DBMS does not support transactions natively, reverting one is a heavy process, and it can lead to starvation, depending on which transaction is selected to be rolled-back. The second option, however, decreases the system concurrency and also implies knowing a priori all the tables where changes will be made, which might not be possible.

As an example of the first solution, consider *Transaction A*, which wants to change *Table t1* and *Table t2*; and *Transaction B*, which wants to interact with *Table t2* and *Table t1*, in the opposite order. When the transactions start, both try and lock their first table. Then, one of them, let's say *A*, tries to lock the second table and blocks (because the other transaction, *B*, has

that table locked). When *B* tries to lock its second table, a deadlock situation is detected (because *A* has that table locked) and one of the transactions is rolled back. At that point, the remaining transaction can proceed (because there are no locks on any of the tables now, except its own) and when it is finished, the rolled-back transaction can proceed as well.

As an example of the second solution, consider the same situation. When the transaction starts, both transactions try and lock both tables. To avoid deadlocks, the locks must be done in an ordered manner. In this case, they could be done alphabetically, and not in the order the transactions use them. Both transactions would try to lock *t1* and then *t2*.

The level at which the locks are implemented is also an important choice. With higher levels, implementation is easier, performance is better but concurrency is worse. As an example, imagine a database-level lock. This single lock allows only a single transaction at a time. The cases where such implementation would work in a practical manner are very few. SQLite is one of them, given it is a local file meant to be used by a single process at a time.

Locks at table level, for example, would have better concurrency; clients can perform transactions on different tables at the same time. However, with many clients or very few tables, this level might still be too restrictive. Some NoSQL DBMS may not, however, have the concept of 'tables'.

Relational DBMS use row-level locks on transactions, which are ideal in the sense that many clients can perform transactions on the same table, just not on the same piece of data they are handling. However, some DBMS may not support row distinction and, inherently, may not support row-level locks. Some NoSQL DBMS also feature millions of rows, which could lead to severe performance issues.

C. Savepoints in Transactions

Assuming transactions have been implemented, the ability to create a save point in a transaction and to roll back to that save point is a simple matter of defining points in the reverser list and only reverting the actions and freeing locks up until that point.

D. IAM Interactions

IAM interactions on a RS consist on the update of values in a row and on the insertion or deletion of rows. By default, a RS's concurrency type is *read only* and does not allow any of these. If it does, its type is *updatable*. To create a RS that allows IAM interactions, a client must specify it when creating the statement object to execute CRUD expressions on the database.

The barrier can intercept the creation of this statement object and, if the *updatable* type is not supported, wrap the RS that is generated inside our framework's RS, which simulates the necessary behaviors to allow the insertion, update and deletion of rows. This RS is the one supplied to the client, where he will be able to execute IAM interactions as usual.

Our first approach was the following: when clients attempt to perform actions on the RS (say, inserting a new row), the actions would be converted and executed like a normal query and the RS would be reset to show the new changes. This had a noticeable performance decay (performing a CRUD expression for the action and another to update the RS) and led to problems when multiple clients were querying the same tables, due to the fact that by resetting the RS, we were re-querying the table fetching results affected by other clients.

Because of this, we followed a different approach where our original RS is never changed (and where we do not have to re-query the table). Values that are updated or inserted are converted to a CRUD expression, inserted in the table and kept in memory. If the client tried to access those values, our framework would present them from memory, without the need to query data from the table. Deleted rows are kept track off and ignored when iterating through the values.

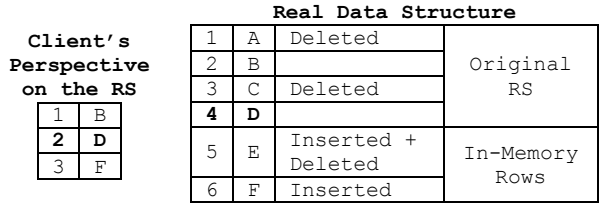

Figure 1. Our data structure for IAM interactions with row 2 highlighted.

Figure 1 shows an example of our data structure. When the client requested the RS, rows *A* to *D* were queried. The client inserted *E* and *F* and deleted *A*, *C* and *E*. Rows *E* and *F* are kept in memory, in an array. Rows *A*, *C* and *E* are flagged as deleted. When the client requests the row with index 2, which corresponds to the value *D*, our implementation iterates through the RS, ignoring deleted rows, until we reach the intended row. With this implementation, there is no unnecessary performance decay (there is no need to re-query the data) and there are no concurrency issues (each client can modify their own RS and their inserted/deleted values do not affect the other clients' RS). This behavior mimics a relational driver implementation's behavior.

V. Proof of Concept

This section describes how the mentioned features were implemented.

A. Execution of Database Functions

To define a SP in a common DBMS, an administrator needs to define four aspects: the name, the input, the output and the actual function of the SP. As such, it is expected that the same aspects must be defined to implement SP on the barrier.

By defining an abstract class *Barrier_CallableStatement* (implementing the *CallableStatement* class), which takes as input a JDBC connection, a name String and an array of arguments (that can be either input or output), the SP framework is defined. To specify the SP, a developer instantiates this abstract class and implements the *execute()* method, which will contain all the SP logic and is the only method that needs to change depending on the SP and the underlying database. As such, all four original aspects are defined and the execution of a SP can be intercepted by the

framework, which will then execute the custom implementation, instead of trying to run it on the database, which would throw an error.

As an example, Listing 2 shows a stored procedure *getEmpName*, defined in MySQL, which returns the name of an employee based on his ID, by querying a table *Employees*, with the fields *id* and *name*.

```
SELECT
CREATE PROCEDURE 'Emp'.'getEmpName'
   (IN EMP_ID INT, OUT EMP_NAME VARCHAR(255))
BEGIN
   SELECT name INTO EMP_NAME
       FROM Employees WHERE ID = EMP_ID;
END
```

Listing 2. Stored Procedure in MySQL.

The usage of this SP in a Java client with a JDBC connection is shown in Listing 3. A *CallableStatement* is created from the connection object with the SP invocation SQL string. The input and output parameters are defined, the procedure is executed and output parameter is read. We can see that there are two separate definitions of the same procedure, one in the database and one in the client. Because the SP and the barrier are in the same place, this redundant definition should not be needed. When implementing a SP, a developer extends it to the *Barrier_CallableStatement* class and defines the number of arguments and the SP name. The execute method contains all the logic (reading input, processing and setting the output).

```
CallableStatement stmt = connection.prepareCall
   ("call EMP.getEmpName (?,?)");
stmt.setInt(1, employeeID);
stmt.registerOutParameter(2, VARCHAR);
stmt.execute();
employeeName = stmt.getString(2);
```

Listing 3. Invocation of the SP in a Java Client.

The usage of this class is quite similar to the original invocation of the SP and is shown in Listing 4. There is no need to register which parameters are *output* and, in this case, there was no need to refer to the SP name. The barrier, however, keeps a list of the implemented SP and, when it detects a command like *exec getEmpName*, matches the desired SP, executes it and returns the corresponding results.

```
CallableStatement stmt = new SP_getEmpName(conn);
stmt.setInt(1, employeeID);
stmt.execute();
employeeName = stmt.getString(2);
```

Listing 4. Invocation of the SP implementation in a Java Client.

B. Execution of Transactions

Transactions are implemented with an abstract class, just like SPs. Each implementation depended on the underlying DBMS and the methods that must be overridden are the methods that return the reversers. When the execution of a statement is requested, the reverser is determined and the corresponding lock is activated. Then, the statement is executed and the reverser is added to the list of actions in the current transaction. The commit statement releases the locks being used and clears the list of reversers.

In case it is not possible to find the reverser (for example, if the row about to be inserted is not unique and there is no way to delete this specific row, then there is no reverser to be found), an exception is thrown and the statement is not executed. If the statement's execution throws an error, the reverser is not added to the list. A rollback executes all the reversers in the list backwards and clears the list.

If deadlock is detected, one of the transactions is rolled-back. The choice of which transaction is selected can be random, by most recent transaction (first come, first served logic), by which transaction detected the deadlock or by which transaction is easiest to rollback (while better on performance, can lead to starvation). The ease of rollback can be determined by the size of the actions list or, if actions have different impacts, by the calculation of the impact of all the actions currently in the list.

Listing 5 shows an example transaction in a Java client. The database has a table *tb*, on which are inserted two tuples, *A* with ID=1 and *B* with ID=2. The *A* value is committed and therefore, is stored in the database. The *B* value is rolled-back and is not stored in the database. Assuming the table was empty at the start of the transaction, by the end of the transaction, a query should show only a single value, *A*.

```
conn.setAutoCommit(false);
try (Statement stmt = conn.createStatement()) {
   stmt.execute("insert into tb values (1, 'A')");
   conn.commit();
   stmt.execute("insert into tb values (2, 'B')");
   conn.rollback(); }
conn.setAutoCommit(true);
```

Listing 5. A simple transaction in a Java client.

As before, a transaction using our framework is expected to function in a similar manner. Listing 6 shows the same transaction, using our framework for SQLite. The creation of the *Barrier_Transaction* object matches the setting of *Auto Commit Mode* to *false* in Listing 5 and it handles the creation of the statement object. Then, *A* is inserted and committed, *B* is inserted and rolled-back and the transaction is closed, which matches the setting of *Auto Commit Mode* to *true*.

```
Barrier_Transaction trans =
   new Barrier_TransactionSQLite (conn);
trans.execute("insert into tb values (1, 'A')");
trans.commit();
trans.execute("insert into tb values (2, 'B')");
trans.rollback();
trans.close();
```

Listing 6. A transaction using our Framework.

In a SQL compliant DBMS, when each *insert* action is requested, the corresponding *delete* action is created. For the *A* value, for example, the reverser is *delete from tb where id=1 and name='A'*. On DBMS with different query languages (like Hive), the parsing and creation of reversers would be different. Hence the fact that each DBMS and each schema have its own implementation of the *Barrier_Transaction* class; schemas with trigger actions need different implementations from schemas without them.

There is also a need for a client-wide lock system to be deployed to enforce isolation, as well as a system to prevent

deadlocks when handling concurrent transactions. Corbett et al. [16] have shown that there are many different solutions for deadlock detection, both distributed and centralized. In our case, the barrier layer acts as a centralized lock system to guarantee isolation among transactions and, as such, it makes sense to use a centralized deadlock prevention mechanism. We have used table-wide locks with MySQL and Hive and row-level locks with Redis and MongoDB.

When a client performs an action during a transaction, the appropriate reverser is found. Immediately after it is determined, the lock is requested to the *Concurrency Handler* (CH), which requires two things: the URI of the lock (in this case, table names or row keys) and the URI of the requesting process. The CH uses semaphores as locks and creates them as transactions request them. In other words, the first time a client requests the lock for table *t1*, that semaphore is created. Any following requests for that table use that semaphore. This removes the need for our framework to know the database schema and be flexible for any lock-level.

The CH does not lock the semaphore immediately. Before doing so, it checks whether a deadlock situation would be created. It does so by using a graph structure that represents subjects (each transaction) and objects (each table/row) and checking for cycles. If a cycle were to be created by this lock request, that a deadlock situation would emerge [17].

Figure 2 shows an example using the previously mentioned example of transactions *A* and *B* trying to lock tables *T1* and *T2*. We can see that we have a deadlock situation. *B*'s request to *T1* leads to its owner, *A*, which has requested *T2*, which belongs to *B*. In our implementation, this situation would never be reached. Assuming *A* requested *T2* before *B* requested *T1*, when *B* made its request, the cycle would be revealed and the transaction would be restarted. When it rolled-back, its locks would be released, which would allow *A* to proceed. When *A* finished, *B* would be able to lock both tables and execute as well.

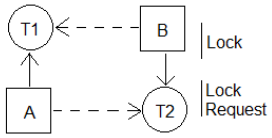

Figure 2. A graph representation of a deadlock situation.

While this example only features two subjects and two objects, the concept can be easily extended for multiple subjects and objects. By solving the deadlock issues, the use of these locks enforces isolation among each transaction. Given that the transactions cannot access another transaction's table/row, then values being read, modified, deleted or created are safe from concurrent modifications.

C. Save Points

A client can set a save point in a transaction and roll-back only up to that save point, which allows for fine-grained control when handling transaction exceptions. Listing 7 shows a transaction that inserts 3 values but only rolls-back one of them (value *B*). Our save point implementation is based in the *Barrier_Transaction* class and, logically, depends on each underlying DBMS. To use save points, a client executes all the methods, just like previously shown, on the *Barrier_Transaction* object.

```
setAutoCommit(false);
try (Statement stmt = conn.createStatement()) {
   stmt.execute("insert into tb values (1, 'A')");
   conn.setSavepoint("savepoint_one");
   stmt.execute("insert into tb values (2, 'B')");
   conn.rollback("savepoint_one");
   stmt.execute("insert into tb values (3, 'C')");
   conn.commit();
}
conn.setAutoCommit(true);
```

Listing 7. A transaction with savepoints in a Java client.

D. IAM Interactions

Interactions on a RS imply that the RS has been requested with the *updatable* type, which enables them. By default, the type is *read only*. Listing 8 shows how a Java client can create a RS, update the third row, insert a new one and delete the second one.

```
Statement stmt = connection.createStatement(
   ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(
   "SELECT * FROM person");
```

Listing 8. A Java client creating a RS to perform IAM interactions.

Because it is our aim to provide as much transparency as possible, the biggest difference is the object request, which uses our wrapper class, as shown in Listing 9. We do not need to specify the type (*updatable* or *read only*) because we assume the DBMS only supports *read only*.

```
ResultSet rs = new Barrier_ResultSetSQLite(
   connection, "SELECT * FROM person",
   ResultSet.TYPE_SCROLL_SENSITIVE);
```

Listing 9. A Java client creating a RS with our SQLite implementation.

Our implementation depends on the underlying DBMS, because it depends on the query syntax, as previously stated.

VI. Evaluation

To demonstrate the soundness of our approach, we have selected four DBMS with different paradigms: SQLite, a relational DBMS; Hive v1.0, a NoSQL DBMS; MongoDB v3.0.2, a document-oriented DBMS; and Redis, one of the most popular key-value DBMS. We expect that our concepts are general enough to be adapted to most NoSQL DBMS. As a basis for comparison, we also used a full-fledged relational DBMS, MySQL, which served as a comparison basis between our barrier implementation and an actual database engine implementation.

The lock-levels were set as tables for all tests, although Redis and MongoDB could use row IDs. Because Redis does not provide a functional and up-to-date JDBC driver, we developed our own driver, which uses the Redis Java API and converts a simple query language into Redis' operations.

The choice of which DBMS to use was done taking into account two main aspects: diversity (it is our goal to show that our concept works with any kind of DBMS, and so it is

important to have both relational and non-relational DBMS, as well as different NoSQL paradigms) and popularity (it is important to choose widely used DBMS).

We tested our framework in a 64-bit Linux Mint 17.1 with an Intel i5-4210U @ 1.70GHz, 8GB of RAM and a Solid State Drive. All the databases were deployed locally, including Hive, which was set-up together with Hadoop as a single-node cluster in this machine. The tests performed include the insertion, update and deletion of values both outside and inside a transaction from our framework.

		SQLite		MongoDB		Hive	
Op.	Rows	Off	On	Off	On	Off	On
Insert	100	749	754	120	189	2642k	2780k
	500	3699	4031	420	1051	X	X
	1000	7907	8494	718	2309	X	X
Update	100	755	758	111	138	3038k	3120k
	500	4025	4096	731	1158	X	X
	1000	8248	8423	2010	3325	X	X
Delete	100	737	746	65	103	2919k	3080k
	500	3648	3784	403	761	X	X
	1000	7502	7775	1123	2018	X	X
Select	100	7	8	81	79	160k	161k
	500	105	107	425	422	X	X
	1000	295	292	1135	1097	X	X

Table 1. A comparison of times taken (in ms) to perform operations in different DBMS with our framework's transactions enabled and disabled.

Tests (shown in Table 1) show an expected performance decay on all databases. In SQLite, the decay amounts to approximately 8% of the original time taken for the insert operation, 2% for the update operation and 3% for the delete operation. In MongoDB, the decay is much stronger, with over 200% decay for inserts, 60% for updates and 80% for deletes. For Hive, tests could only involve up to 100 rows, due to time restraints. However, Hive shows good results of about 5% decay in inserts, 3% in updates and 5% in deletes. Tests for MySQL and Redis were not considered to have relevant information and were not included.

Because queries are an integral part of the transaction process, the decay is directly related to the ratio between the time taken for queries and operations for each DBMS. This explains why MongoDB has a much stronger decay than SQLite or Hive.

Tests were also conducted in regards to database-stored functions, rollbacks and IAM interactions. The tests show that the performance decay is directly related to the performance of a CRUD expression on the database: if a statement takes 10 seconds, an IAM interaction will also take 10 seconds, plus a residual processing time (about 5 to 10 microseconds). The same relation exists for rollbacks and stored procedures which involve operations in the database.

VII. Conclusion

We have proposed a framework that implements some features on a JDBC driver that are not usually implemented using NoSQL drivers. Our proposal includes a model to use our framework in a way that allows concurrent clients to perform atomic and isolated transactions, as well as IAM interactions and database functions, like stored procedures. We have proven our concept with SQLite, Hive, Redis and MongoDB, and we expect our model to be general enough that it can be extended to other DBMS, relational or NoSQL.

Our performance results show that the use of our framework can be suitable for a real-life scenario. However, work is underway to perform a more in-depth performance evaluation of the different DBMS, with different test conditions, which will be adequate to each DBMS's architecture and design and provide a more insightful analysis. Work is also underway to add fault tolerance to our proposal; our framework does not currently provide atomicity in case of hardware failures.

In conclusion, our framework positively contributes to overcome the gap between NoSQL and SQL. It helps system architects to simulate key relational DBMS features on NoSQL databases that do not natively support them and eases the transition from a DBMS to another, by abstracting underlying features of the DBMS.

References

[1] ISO/IEC, Information technology -- Database languages -- SQL -- Part 3: Call-Level Interface (SQL/CLI). 2008.

[2] R. Öberg, "Neo4j JDBC," 2015. [Online]. Available: https://github.com/neo4j-contrib/neo4j-jdbc. [Accessed: 11-Mar-2015].

[3] E. Horowitz, "MongoDB JDBC," 2010. [Online]. Available: https://github.com/erh/mongo-jdbc. [Accessed: 11-Mar-2015].

[4] R. Felix, "CouchDB JDBC," 2009. [Online]. Available: https://github.com/fellix/couchdb-j.

[5] Apache, "HBase JDBC," 2011. [Online]. Available: http://www.hbql.com/examples/jdbc.html.

[6] Apache, "Hive JDBC," 2014. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/HiveJDBCInterface.

[7] W.-C. Chung, H.-P. Lin, S.-C. Chen, M.-F. Jiang, and Y.-C. Chung, "JackHare: a framework for SQL to NoSQL translation using MapReduce," Autom. Softw. Eng., vol. 21, no. 4, pp. 489–508, 2014.

[8] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira, "An effective scalable SQL engine for NoSQL databases," in Distributed Applications and Interoperable Systems, 2013, pp. 155–168.

[9] J. Taylor, "Querying a not only structured query language (nosql) database using structured query language (sql) commands." Google Patents, 18-Dec-2013.

[10] A. Calil and R. dos Santos Mello, "SimpleSQL: a relational layer for SimpleDB," in Advances in Databases and Information Systems, 2012, pp. 99–110.

[11] R. Lawrence, "Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB," Computational Science and Computational Intelligence (CSCI), 2014 International Conference on, vol. 1. pp. 285–290, 2014.

[12] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacigümüş, "Partiqle: An elastic SQL engine over key-value stores," in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. 629–632.

[13] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to non-relational database systems: The SOS platform," in Advanced Information Systems Engineering, 2012, pp. 160–174.

[14] B. M. Clapper, "SQLShell." 2012.

[15] Oracle, "JDBC Overview." [Online]. Available: http://www.oracle.com/technetwork/java/overview-141217.html. [Accessed: 09-Mar-2015].

[16] C. Corbett, "Evaluating Deadlock Detection M e thods for Concurrent Software," vol. 22, no. 3, 1996.

M. Singhal, "Deadlock detection in distributed systems," Computer, vol. 22. pp. 37–48, 1989.