RESEARCH ARTICLE

A Middleware Model for SQL to NoSQL Query Translation

Basant Namdeo[1]*, **Ugrasen Suman**[2]

1 Assistant Professor, International Institute of Professional Studies, Devi Ahilya University, Indore, India
2 Professor, School of Computer Science and IT, Devi Ahilya University, Indore, India

OPEN ACCESS

Received: 18-02-2022

Accepted: 16-03-2022

Published: 29.04.2022

Citation: Namdeo B, Suman U (2022) A Middleware Model for SQL to NoSQL Query Translation . Indian Journal of Science and Technology 15(16): 718-728. https://doi.org/10.17485/IJST/v15i16.2250

* **Corresponding author**.

basant_nd@yahoo.com

Funding: None

Competing Interests: None

Published By Indian Society for Education and Environment (iSee)

ISSN
Print: 0974-6846
Electronic: 0974-5645

Abstract

Objectives: To propose a suitable model for RDBMS SQL to NoSQL query translation, which works as a middleware between legacy applications and the NoSQL database. This model is expected to translate the SQL queries into NoSQL queries, and forward them to the NoSQL database for execution, and after the execution, the result received from NoSQL should be transferred to the legacy application. **Methods:** The proposed model is implemented in Java programming language for MySQL (RDBMS) to MongoDB document database query translation. The prototype translates the insert, update, delete, and select SQL queries into equivalent NoSQL query format for MongoDB document database. The middleware transforms the SQL queries to NoSQL query format, and returns the result to the legacy application, which they are expecting from the database. Performance of our model has been evaluated by executing SQL queries such as select, insert, update, delete (with simple and join queries) in Studio 3T, UnityJDBC driver for MongoDB, and our model SQL-No-QT. **Findings:** The study shows that the proposed model SQL to NoSQL Query Translation Model (SQL-No-QT) performs better in some cases. This model takes 7.5% less time compared to Studio3T, and 38.19% less time compared to UnityJDBC driver in executing select queries, and 78.82% less time compared to Studio3T in executing delete queries in big size database. This model also can execute the join SQL queries for insert, update and delete, which are not available in UnityJDBC driver for MongoDB. **Novelty:** This model works as a middleware between a legacy application and a NoSQL database, and it removes the need of developing whole new software for legacy application.

Keywords: Database reengineering; database; Query translation; NoSQL; RDBMS

1 Introduction

NoSQL is the term mainly used for the group of databases that do not follow the relational database model. Many companies developed their own product for such categories of databases, such as Amazon developed DynamoDB, Google developed Bigtable, Apache developed Cassandra, etc. Various types of NoSQL databases software

are available according to the data model used for storing information, such as document type, key-value, wide column, or graph[1]. High performance, scalability, and availability are the key characteristics of NoSQL databases[2].

NoSQL gives the freedom to define the database schema at runtime. Users can change the database schema as and when it is required, and it supports the big data too. Big data is the term or field in computer science that defines the large volume of structured or unstructured data. It may be received from the different number of sources in a very speedy way. It has three types of key concepts, i.e., volume, variety, and velocity (3 V's of big data)[3]. Initially, many organizations used RDBMS as database software for their application and other software. Latter, due to the rapidly growing database size because of business expansion, they have found that RDBMS databases are not able to fulfill their requirement. So they need to migrate their database to new database technology like NoSQL.

Leonardo et al.[4] presented a NoSQLayer framework, which supports the migration from relational to NoSQL DBMS. This framework has been divided into two parts; the first one is the data migration module and the second one is data mapping. Data migration module has been used for the migration of relational data into NoSQL data, and for that, it used Java metadata API to retrieve the table information of relational database objects. The data mapping module provides an abstraction layer between the application software and DBMS, which translates the SQL queries into NoSQL query format. They have built a mediator using MySQL proxy and a mapping module, which translates the SQL query into NoSQL query format and sends back the result received from NoSQL to the application software.

There is no common standard language available for interacting with NoSQL database systems. Typically, each NoSQL database software vendor provides its own query language API for interacting with its NoSQL database. So, the user has to program for a specific NoSQL database, and thus it reduces the portability. Many researchers have worked on this problem, proposed different middleware and translators for SQL to NoSQL. Zhang et al. [5] presented a Nomiddleware architecture which translates the SQL queries into NoSQL specific query, and forwards the result back to client application.

A hybrid database access layer has also been presented by some of the researchers. A hybrid database design with a relational database (RDB) and a NoSQL database supported by a data adapter system has been presented by Liao et al.[6] They provided a seamless technique for using RDB and NoSQL databases simultaneously with this data adaptor technology. Data adapter can read data from RDB or NoSQL, depending on the availability of data. Li et al.[7] also presented an integration model of relational databases and NoSQL data stores called MSI (multiple sources integration). This MSI interacts with both the relational and NoSQL database. Users can send the SQL queries to the MSI, and then it is translated to native NoSQL query for execution. SQL queries in this model can use both the NoSQL and relational database table in a single SQL query.

Ha et al.[8] have also discussed an approach to translate SQL queries from MySQL to MongoDB NoSQL database. They discussed four phases, first for parsing the incoming SQL queries, second phase for creating the dictionary of query parts, third phase for updating the dictionary for matching structure target database. This phase also determined the structure of the target database MongoDB. In the last fourth phase, a final query is generated by the process of checking the joining of each part in the dictionary. They only considered the select query for a translation task.

Some other types of query translation works have also been done by various researcher such as, SQL to XQuery translation[9], EOL (model-level queries) to SQL[10], SimpleSQL relational layer which works between application and SimpleDB[11], NoSQL2 which translate the administrative command of SQL such as truncate table, create table, etc. into NoSQL specific command[12]
.

Therefore, query translation from SQL to NoSQL databases has become a challenging domain of research. The new NoSQL database has come with its own query language for querying the database. Various new approaches and techniques have been proposed and evaluated for this purpose. The above literature survey reveals that query translation from SQL to NoSQL databases is an important part for those organizations which are planning to transform their business data from RDBMS to NoSQL and still want to run old application software. Various authors' proposed different methods for it, but the main focus of them are on converting simple select statement only. After studying the above literature, we have found that a major research gap is, most of the authors have only worked on the conversion of a simple select statement, and not on insert, update and delete query. To fill this research gap, we have proposed a model which considers the translation of insert, update and delete query with join statement, along with the simple select statement.

In this paper, we have proposed a middleware model SQL to NoSQL Query Translation Model (SQL-No-QT) for an automatic solution to the problem of SQL query translation to NoSQL (MongoDB) database. The paper is organized in the following way. Section 2 explains the proposed query translation model and algorithm. Experimental analysis is carried out for different database sizes with our proposed query translation models, which is discussed in Section 3. Finally, Section 4 presents the conclusion of this paper.

2 SQL to NoSQL Query Translation Model (SQL-No-QT

This query translation model is proposed as a middleware layer to provide a solution for legacy applications to work with the new NoSQL database without changing the database logic. This middleware layer works between the legacy application and the JDBC driver of the MongoDB database.

The architecture of the proposed model is presented in Figure 1. The model consists of five components; namely, lexical analyzer, syntax analysis or parser, query code generator, data dictionary, and result formatter. The legacy application sends the SQL queries to the model and receives the output from the model.

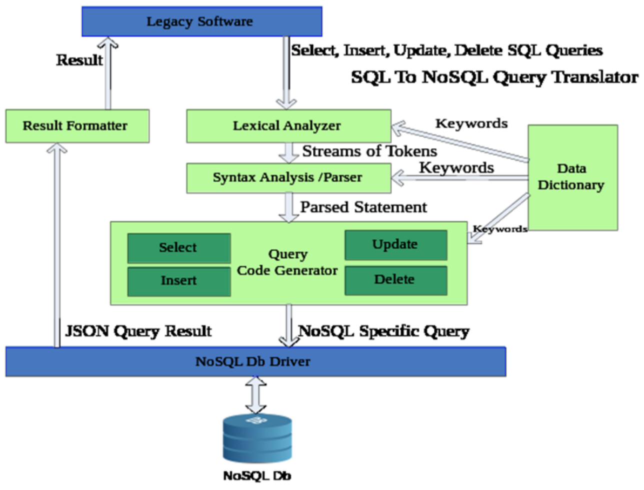


Fig 1. SQL to NoSQL Query Translation Model (SQL-No-QT)

The first component of the model, lexical analyzer, or say tokenizer receives the SQL (insert, update, delete and select) queries as input from the legacy software. Then, with the help of the data dictionary, it finds the tokens from the SQL string. Tokens can be keywords such as select, from, where , insert etc., or delimiters such as ,(comma), or identifiers such as table names, field names etc., or operators such as +(plus), - (minus) ,< (less than), > (greater than) , <= (less than or equal to), >= (greater than or equal to), != (not equal to) etc., or constant such as any number / string.

A syntax analyzer will be used as the second component of the model. It receives the string of tokens from the lexical analyzer and checks the syntax of the SQL queries by using grammar. Grammar for the simple select, insert, update and delete SQL statement can be defined as given in Tables 1, 2, 3 and 4 respectively.

Table 1. Select Statement Grammar

<SelectQuery>	=>	SELECT <SelFieldList> FROM <fromTableList>[INNER JOIN <TableName> ON <joinCondition>] [WHERE <condition>][GROUP BY <groupFieldList> [HAVING <condition>]] [ORDER BY <SortFieldList> [DESC | ASC]]
<SelFieldList>	=>	<Field> | <AggreFunc> [, SelFieldList >]
<AggreFunc>	=>	AVG(<Field>) | COUNT(<Field>) | COUNT(DISTINCT <Field>) | MAX(<Field>) | MIN(<Field>) | SUM(Field>)
<fromTableList>	=>	<TableName> [,<fromTableList>]
<condition>	=>	<condition> | <condition> and <condition> | <conditon> or <condition> | <Field><operator><Field> | <Field> in <SelectQuery>
<operator>	=>	< | > | <= | >= | !=
<joinCondition>	=>	< joinCondition > | < joinCondition > and < joinCondition > | < joinCondition > or < joinCondition > | <Field><operator><Field>
<SortFieldList>	=>	<Field> | <AggreFunc> [, <SortFieldList>]

The output of the syntax analyzer is a parsed statement, which creates the various objects of select, insert, update and delete queries. The next part of the model is Query Code Generator. Here, in this phase objects that belong to different SQL queries (select, insert, update and delete), received from the previous phase, are passed to different modules for query code generation

Table 2. Insert Statement Grammar

<InsertQuery>	=>	Insert into <TableName> [(FieldList)] [[VALUES (<ValueList>)] | [<SelectQuery>]]
<ValueList>	=>	constant [,<ValueList>]
<FieldList>	=>	<Field> [,FieldList>]

Table 3. Update Statement Grammar

<Update-Query>	=>	Update <TableName> Set <AssignmentList>[<TableName> INNER JOIN <TableName> ON <joinCondition>] [WHERE <condition>]
<Assign-mentList>	=>	<Field> = Value[, <AssignmentList>]

Table 4. Delete Statement Grammar

<Delete-Query>	=>	Delete [[<TableName>] from [<TableName> INNER JOIN <TableName> ON <joinCondition>]] | [from <TableName>] [WHERE <condition>]

for the target NoSQL database. Here, an equivalent query is generated for its corresponding SQL query. Suppose, there are relational tables as discussed in[13]. So, we can translate our SQL queries into NoSQL MongoDB query format as shown in Table 5. Here, we take two types of SQL queries for each select, insert, update and delete SQL query, one for a single table and the other for two tables (e.g. join, etc.)

Table 5. SQL to NoSQL query equivalent

Select Query	
SQL	Select stu_rollno, stu_name from StudMast where stu_rollno = 101010
NoSQL	db.StudMast.find({stu_rollno:101010},{stu_rollno:1,stu_name:1})
SQL	select StudMast.stu_rollno,StudMast2020.s_name from StudMast inner join StudMast2020 on StudMast.stu_rollno = StudMast2020.s_no where StudMast.stu_rollno = 101010

NoSQL	db.StudMast.aggregate([{"$project":{
	"_id": NumberInt(0),
	"StudMast":"$$ROOT"
	}},
	{ "$lookup":{
	"localField":"StudMast.stu_rollno",
	"from":"StudMast2020", "foreignField":"s_no",
	"as":"StudMast2020" }
	},
	{
	"$unwind":{
	"path":"$StudMast",
	"preserveNullAndEmptyArrays":false
	}},
	{
	"$match":{
	"StudMast.stu_rollno": 101010
	}
	},
	{
	"$project":{
	".StudMast.stu_rollno":"$StudMast.stu_rollno",
	"StudMast2020.s_name":"$StudMast2020.s_name"
	}
	}]);

Insert Query	
SQL	insert into StudMast(stu_rollno, stu_name) values(101010, "Mahesh")

Continued on next page

Table 5 continued

NoSQL	db.StudMast.insertOne({stu_rollno:101010, stu_name: "Mahesh"})
SQL	Insert into StudMast (stu_rollno, stu_name) select s_no, s_name from StudMast2020
NoSQL	db.StudMast2020.find().forEach(function(doc){ db.StudMast.insertOne({stu_rollno:doc.s_no, stu_name: doc.s_name}) })
Update Query	
SQL	update StudMast set stu_name = "Ramesh" where stu_rollno = 101010
NoSQL	db.StudMast.updateMany({stu_rollno:101010}, {$set:{ stu_name: "Ramesh"}})
SQL	Update StudMast set stu_name = StudMast2020.s_name from StudMast inner join StudMast2020 on StudMast.stu_rollno = StudMast2020.s_no
NoSQL	db.StudMast2020.find.forEach(function(doc){ db.StudMast.updateMany({stu_rollno:doc.s_no}, {$set:{stu_name:doc.s_name}}) })
Delete Query	
SQL	Delete from StudMast where stu_rollno = 101010
NoSQL	db.StudMast.deleteMany({stu_rollno:101010})
SQL	delete StudMast from StudMast inner join StudMast2020 on StudMast.stu_rollno = StudMast2020.s_no where cur_termno = 1
NoSQL	db.StudMast2020.find({cur_termno:1}).forEach(doc){ db.StudMast.deleteMany({stu_rollno:doc.s_no}) }

The translated NoSQL query is then forwarded to the NoSQL database driver for execution. Result received from the NoSQL database driver after execution is then forwarded to the Result Formatter module, which is responsible for converting the result into the format that the legacy application is expecting from the database.

In the next part of this section, we describe the algorithm of the query code generator module of the model.

```
Algorithm: Query Translation - parsedStatement - parsed object for SQL statement.
Output: NoSQLCmd – A NoSQL command
Begin
  var cmdType = parsedStatement.commandType;
  var NoSQLCmd;
  if cmdType= "select" then
  Begin
    NoSQLCmd = Call getSelectCmd(parsedStatement);
  End
  else if cmdType= "insert" then
  Begin
    NoSQLCmd = Call getInsertCmd(parsedStatement);
  End
  else if cmdType= "update" then
  Begin
    NoSQLCmd = Call getUpdatetCmd(parsedStatement);
  End
  else if cmdType = "delete" then
  Begin
    NoSQLCmd = Call getDeleteCmd(parsedStatement);
  End
  return NoSQLCmd;
End Algorithm ;
```

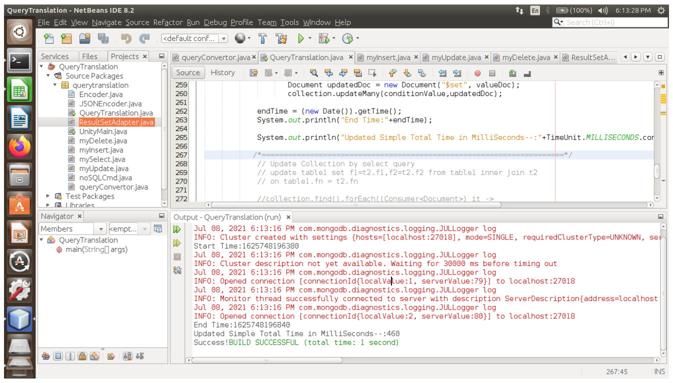
The above algorithm receives the parsed statement as input from the previous phase of the model i.e. parser. We have used JsqlParser[14] as a parser for SQL queries. This algorithm first checks its command Type property of the parsed Statement object, and according to its property value, calls a separate function or module for select, insert, update and delete SQL queries. Algorithms for select, insert, update and delete queries modules are defined separately in getSelectCmd, getInsertCmd, getUpdateCmd, and getDeleteCmd function respectively as follows. Each function then returns the converted NoSQL query command. This NoSQL query command is then forwarded to the NoSQL database driver. The result received from the NoSQL database driver is then forwarded to the Result Formatter module, which simply converts the JSON result received from the NoSQL database driver into Tabular format. This tabular format data is then sent back to legacy software.

```
function getSelectCmd (parseStatement)
Begin
    var cmd ;
   var selectCmd = parseStatement.selectCmd;
   if selectCmd.fromTable.count = 1 then //Only One Table in Select Command
    Begin
      var fromTable1 = selectCmd.fromTables[0];
      var selectList = selectCmd.selectList.getPropertyValueList();
      var whereList = selectCmd.whereList.getPropertyValueList();
      cmd = "db."+fromTable1 + ".find({"+whereList+"},{"+selectList+"})";
    End
    else // more than 1 tables are in Select Command
    Begin
      var fromTable1 = selectCmd.fromTables[0];
      var fromTable2 = selectCmd.fromTables[1];
      var project1, project2, lookup, unwind, match ;
      project1 = " '_id': NumberInt(0), ' "+fromTable1+" ': '$$ROOT' ";
      lookup = " 'localField' : ' "+fromTable1+ "." + fromTable1.joinFieldName+" ', ";
      lookup+= " ' from': '"+fromTabl2+" ', 'foreignField':' " + fromTable2.joinFieldName+" ', ";
      lookup+ =" 'as':' " +fromTable2+" ' ";
      unwind = " 'path': '$"+fromTable1+" ', 'preserveNullAndEmptyArrays':false ";
      match = selectCmd.whereList;
      project2 = selectCmd.selectList;
      cmd = "db." + fromTable1 + ".aggregate([{'$project' : {"+ project1+ "}},";
      cmd + = " { '$lookup' : {" + lookup + "}},";
      cmd + = " { '$unwind' : {" + unwind + "}},";
      cmd + = " { '$match' : {" + match +"}},";
      cmd + = " { '$project' : {" + project2 +"}}])";
    End
    return cmd;
End getSelectCmd;
function getInsertCmd(parseStatement)
Begin
  var cmd;
  var insertCmd = parseStatement.insertCmd;
  if insertCmd.table.count= 1 then // Only one table in insert command
  Begin
      var tableName = insertCmd.table[0];
      var columnList = insertCmd.columnList;
      var insertData = insertCmd.valueList;
      cmd = "db."+tableName+".insertMany("+insertData+")";
  End
  else // more than 1 tables are in Insert Command
  Begin
      var selectTable = insertCmd.table[1];
      var insertTable = insertCmd.table[0];
      var propertyValueList= "";
```

```
        cmd = “db.”+selectTable+ “.find().forEach(function(doc){”;
        cmd+=“db.”+insertTable+ “.insertOne({ “ ;
        for(var x = 0 ; x< selectTable.selectList.count;x++)
        Begin
          propertyValue+=insertTable.getProperty[x] + “:doc.” + selectTable.getProperty[x]+”,”
        End
        propertyValue.removeLastChar();
        cmd+=propertyValue+”})”;
        cmd+=”)”;
    End
**End getInsertCmd;**
**function getUpdateCmd**(parseStatement)
**Begin**
    var cmd;
    var updateCmd = parseStatement.updateCmd;
    if updateCmd.table.count= 1 then // Only one table in update command
    Begin
        var tableName = updateCmd.table[0];
        var whereList = updateCmd.whereList.getPropertyValueList();
        var updateData = updateCmd.setList.getPropertyValueList();
        cmd = “db.”+tableName+”.updateMany({“+whereList+”},{“+updateData+”})”;
    End
    else // more than 1 tables are in Update Command
    Begin
        var jTable = updateCmd.table[1];
        var uTable = updateCmd.table[0];
        var propertyValueList=””;
        var joinedConditionList = “”;
        var conditionList= “”, normalCondition= “”;
        cmd = “db.”+jTable+ “.find().forEach(function(doc){”;
        cmd+=”db.”+uTable+”.updateMany({“;
        /*Creating Where Condition*/
        for(var x = 0 ; x< jTable.conditionList.count; x++)
        Begin
          joinedConditionList+=uTable.getProperty[x] + “:doc.” + jTable.getProperty[x]+”,”
        End
        joinedConditionList.removeLastChar();
        normalCondition = uTable.whereList.getPropertyValueList();
        conditionList = normalCondition + “,” + joinedConditionList ;
        cmd+=conditionList+”},{”;
        /*Creating Update Set List*/
        for(var x = 0 ; x< jTable.updateList.count; x++)
        Begin
          propertyValue+=uTable.getProperty[x] + “:doc.” + jTable.getProperty[x]+”,”
        End
        cmd+=propertyValue+”})”;
        cmd+=”)”;
    End
**End getUpdateCmd;**
**function getDeleteCmd**(parseStatement)
**Begin**
    var cmd;
    var deleteCmd = parseStatement.deleteCmd;
    if deleteCmd.table.count= 1 then // Only one table in delete command
    Begin
        var tableName = deleteCmd.table[0];
        var whereList = deleteCmd.whereList.getPropertyValueList();
        cmd = “db.”+tableName+”.deleteMany({“+whereList+”})”;
    End
    else // more than 1 tables are in delete Command
    Begin
        var jTable = updateCmd.table[1];
        var dTable = updateCmd.table[0];
        var joinedConditionList= “”;
        var jconditionValue = “”;
        jcondition = jTable.whereList.getPropertyValueList();
        cmd = “db.”+jTable+ “.find({“+jcondition+”}).forEach(function(doc){”;
        cmd+= “db.”+dTable+ “.deleteMany({“;
        for(var x = 0 ; x< jTable.whereList.count; x++)
        Begin
          joinedConditionList+=dTable.getProperty[x] + “:doc.” + jTable.getProperty[x]+”,”
        End
        normalCondition = dTable.whereList.getPropertyValueList();
        conditionList = normalCondition + “,” + joinedConditionList ;
        cmd+=conditionList+”})}”
    End
**End getDeleteCmd;**
```

3 Results and Discussion

In order to test the validity and performance of the proposed model, we have developed a prototype using the Java programming language. A screenshot of the prototype is shown in Figure 2 . We have used MongoDB as a NoSQL database in the model and JSqlParser[14] as a parser for SQL queries. JSqlParser is an open-source library that translates SQL queries into the hierarchy of Java classes like Select, Insert, Update, Delete, etc. These classes have various methods and properties for getting the where conditions, table names, order by values of various classes.


QueryTranslation - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
queryConvertor.java
QueryTranslation.java
myInsert.java
myUpdate.java
myDelete.java
ResultSetA...
Source History
Output - QueryTranslation (run)
267:45 INS

```
                Document updatedDoc = new Document("$set", valueDoc);
                collection.updateMany(conditionValue,updatedDoc);

        endTime = (new Date()).getTime();
        System.out.println("End Time:"+endTime);

        System.out.println("Updated Simple Total Time in MilliSeconds--:"+TimeUnit.MILLISECONDS.cor

       /*====================================================================*/
        // Update Collection by select query
        // update table1 set f1=t2.f1,f2=t2.f2 from table1 inner join t2
        // on table1.fn = t2.fn

        //collection.find().forEach((Consumer<Document>) it ->
```

```
Jul 08, 2021 6:13:16 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Cluster created with settings {hosts=[localhost:27018], mode=SINGLE, requiredClusterType=UNKNOWN, se
Start Time:1625748196380
Jul 08, 2021 6:13:16 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Cluster description not yet available. Waiting for 30000 ms before timing out
Jul 08, 2021 6:13:16 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:1, serverValue:79}] to localhost:27018
Jul 08, 2021 6:13:16 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Monitor thread successfully connected to server with description ServerDescription{address=localhost
Jul 08, 2021 6:13:16 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:2, serverValue:80}] to localhost:27018
End Time:1625748196840
Updated Simple Total Time in MilliSeconds--:460
Success!BUILD SUCCESSFUL (total time: 1 second)
```

Fig 2. Java Program Screen Shot NetBeans IDE forSQL-No-QT

Qualitative comparison of our SQL to NoSQL query translation model (SQL-No-QT Model), and other frameworks is shown in Table 6. This qualitative comparison covers main features such as SQL query supported, database used, and methodology used by various techniques or frameworks.

Here, in quantitative analysis, two types of database size are considered in terms of the number of records for checking the performance of the model. The first database has approximately 0.4 million records, and the second database has approximately more than 1 million records. Performance of the executing SQL queries such as select, insert, update, delete (with simple and

Table 6. Comparison of various methods

	SQL Query Supported	**Database**	**Methodology**
NoSQLayer [4]	select, insert, update, delete (join not allowed in insert, update and delete)	MongoDb	—
MSI [7]	select, insert, update, delete (join not allowed in insert, update and delete)	Document Type (No Specific) Only Framework	Map Reduce
Studio 3T	select with join	MongoDB	—
UnityJDBC	select, insert, update, delete (join not allowed in insert, update and delete)	MongoDB	MongoDB Java Library
SQL-No-QT Model	select, insert, update, delete with join	Document Type, MongoDB	Direct Query Translation

join queries) in Studio 3T [15], UnityJDBC [16,17] driver for MongoDB and our model is shown in Figures 3, 4, 5 and 6 respectively. Currently Studio 3T is not supporting insert, update and delete SQL queries, so we executed the native NoSQL query in Studio 3T for them. This model takes 7.5% less time compared to Sudio3T, and 38.19% less time compared to UnityJDBC driver in executing select queries, and 78.82% less time compared to Studio 3T in executing delete queries in big size database. It also takes 1.25% less time compared to Studio 3T in executing insert join query, and UnityJDBC is not supporting the join in insert query.

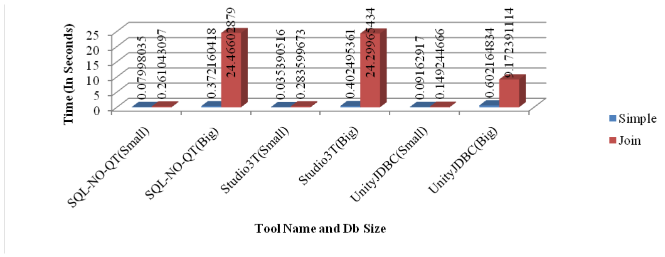


Fig 3. Performance Analysis (Select Query).

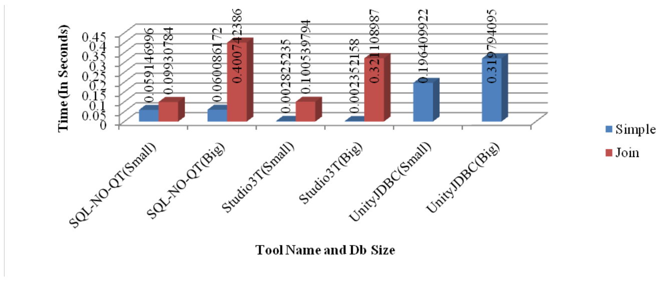


Fig 4. Performance Analysis (Insert Query).

After analyzing the results of different SQL queries execution times in the different databases, it is observed that SQL to NoSQL Query Translation Model (SQL-No-QT) performs better in some cases such as simple query in big data, join delete query in big data in terms of the number of time taken to perform SQL query translation and getting the result back from the database. It is also observed that the proposed model performs better for large database sizes too, and from the comparison shown in Table 6, it is observed that, our model can also perform the task of translation of insert, update, and delete queries with join statements, which are not supported by other tools or models.

4 Conclusions

This study has proposed an SQL to NoSQL query translation model, which can be used as middleware between the NoSQL database and legacy application. A model SQL-No-QT is presented here, which efficiently converts its SQL queries into NoSQL

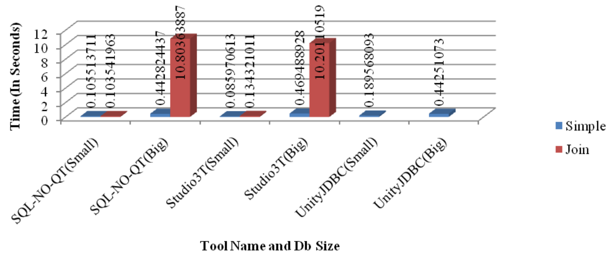


Fig 5. Performance Analysis (Update Query).

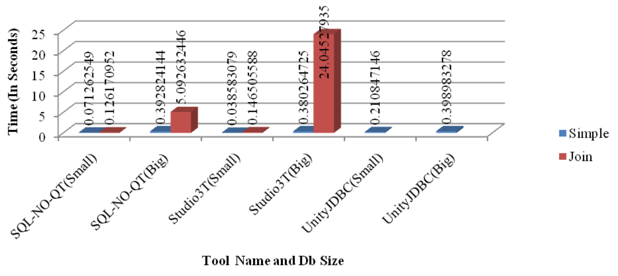


Fig 6. Performance Analysis (Delete Query).

queries and forwards them to the NoSQL database driver for execution. After executing the queries, it sends back the result to legacy software through the result formatter module of the model. This model has five components; namely, lexical analyzer, syntax analysis or parser, query code generator, data dictionary, and result formatter. It takes the SQL queries as input from the legacy software. This model efficiently translates the set of SQL queries into NoSQL queries, specifically for the MongoDB database. It gets the result from the NoSQL database, and that result is forwarded to the legacy application in the format required by it. The study shows that the proposed model SQL to NoSQL Query Translation Model (SQL-No-QT) performs better in some cases. This model takes 7.5% less time compared to Sudio3T, and 38.19% less time compared to UnityJDBC driver in executing select queries, and 78.82% less time compared to Studio3T in executing delete queries in big size database. This model also can execute the join SQL queries for insert, update and delete, as compared to UnityJDBC and Studio 3T.

References

1) Namdeo B, Suman U. Performance Analysis of Schema Design Approaches for Migration from RDBMS to NoSQL Databases. *Advances in Data and Information Sciences*. 2020;94:413–424. Available from: https://doi.org/10.1007/978-981-15-0694-9_39.
2) Namdeo B, Suman U. A Model for Relational to NoSQL database Migration: Snapshot-Live Stream Db Migration Model. In: 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS). IEEE. 2021;p. 199–204.
3) Patel AA, Dharwa J, and. Graph Data: The Next Frontier in Big Data Modeling for Various Domains. *Indian Journal of Science and Technology*. 2017;10(21):1–7. Available from: https://dx.doi.org/10.17485/ijst/2017/v10i21/112828.
4) Rocha L, Vale F, Cirilo E, Barbosa D, Mourão F. A Framework for Migrating Relational Datasets to NoSQL 1. *Procedia Computer Science*. 2015;51(1):2593–2602. Available from: https://dx.doi.org/10.1016/j.procs.2015.05.367.
5) Zhang C, Xu J. A Unified SQL Middleware for NoSQL Databases. *Proceedings of the 2018 International Conference on Big Data and Computing*. 2018;p. 14–23. Available from: https://doi.org/10.1145/3220199.3220212.
6) Liao YT, Zhou J, Lu CH, Chen SC, Hsu CH, Chen W, et al. Data adapter for querying and transformation between SQL and NoSQL database. *Future Generation Computer Systems*. 2016;65:111–121. Available from: https://dx.doi.org/10.1016/j.future.2016.02.002.
7) Li C, J G. An integration approach of hybrid databases based on SQL in cloud computing environment. *Software: Practice and Experience*. 2019;49:401–423. Available from: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2666https://doi.org/10.1002/spe.2666.
8) Ha M, Shichkina Y. The Query Translation from MySQL to MongoDB Taking into Account the Structure of the Database. In: 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus). IEEE. 2021;p. 383–386. Available from: https://doi.org/10.1109/ElConRus51938.2021.9396591.
9) Ali A, Ibrahim M. NoSQL Database Query Generation using an Automated Approach. *Artificial Intelligence Mod Syst*. 2017;1(1):32–41. Available from: http://jdconline.net/aims/archive/1(1)/1104.pdf.
10) X DC, Sagardui G, Trujillo S. MQT, an approach for runtime query translation: From EOL to SQL. In: Brucker AD, Dania C, Georg G, Gogolla M, editors. CEUR Workshop Proceedings. 2014;p. 13–22. Available from: http://ceur-ws.org/Vol-1285/paper02.pdf.

11) Calil A, Mello RDS. SimpleSQL: A Relational Layer for SimpleDB. In: European Conference on Advances in Databases and Information Systems. Springer, Berlin, Heidelberg. 2012;p. 99–110. Available from: https://doi.org/10.1007/978-3-642-33074-2_8.
12) Adriana J, Holanda M. NoSQL2: SQL to NoSQL Databases. In: Advances in Intelligent Systems and Computing. Springer International Publishing. 2018;p. 938–948. Available from: https://doi.org/10.1007/978-3-319-77712-2_89.
13) Namdeo B, Suman U. Schema design advisor model for RDBMS to NoSQL database migration. *International Journal of Information Technology*. 2021;13(1):277–286. Available from: https://dx.doi.org/10.1007/s41870-020-00515-8.
14) Jsqlparser - Home. . Available from: http://jsqlparser.sourceforge.net/.
15) The Professional Client, IDE & GUI for MongoDB | Studio 3T. . Available from: https://studio3t.com/.
16) UnityJDBC - Multiple Database SQL Querying, Reporting, Development, Virtualization. . Available from: https://www.unityjdbc.com/.
17) Lawrence R. Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB. In: 2014 International Conference on Computational Science and Computational Intelligence. IEEE. 2014;p. 285–90. Available from: https://doi.org/10.1109/CSCI.2014.56.