Baltic J. Modern Computing, Vol. 10 (2022), No. 2, pp. 121–131
https://doi.org/10.22364/bjmc.2022.10.2.02

Data Model for Rich Time Series Data and Chameleon Query Language

Marta Jadwiga BURZANSKA *

Nicolaus Copernicus University, Torun, Poland

quintria@mat.umk.pl

Abstract. Nowadays time-stamped data is being generated by a variety of applications and stored in a variety of database systems. Depending on the data structure and the importance of the time aspect those database systems may be either general-purpose or time-series oriented. Most popular time-series DBMS use only simple data types for non-time values, whereas general-purpose databases usually lack specialized query methods designed for time-series aspects. Each of those solutions has some drawbacks as far as data handling is concerned. Extending SQL with built-in time series focused components enables the combination of relational queries with document and time-series-oriented queries. At the same time, the popularity of SQL among data analysts and data scientists would give them the benefit of an easy start. This paper presents such an extension and introduces ChQL, a query language designed to work with both document data and multivariate time series. This language is designed to imitate the syntax of the Python language, so that it could be easily integrated as a library into object-oriented languages such as Python, Javascript, or PHP. The main architectural concept behind the data model for both languages is to use a mixed data model based on both document and time-series storage. This paper is focused on the languages time-series related features and their architectural consequences

Keywords: query languages, data model, time series, ChQL, SQL

1 Introduction

The analytical needs of modern medium and large companies are on a steady rise and demands on real-time generated data analysis are increasing. The analysis of time series is often a central issue in economic research and many other scientific applications. Companies wish to adapt machine-learned prediction models in response to changing user behavior, public safety organizations monitoring weather or seismic activity need quick and reliable warning systems, producers of semi-autonomous machines need to

* The research presented in this paper was supported by the Polish National Centre for Research and Development (NCBiR) under Grant No. POIR.01.01.01-00-1205/18

monitor them and gather data for further research and enhancements. Those are just a few examples where efficient reliable database systems working with time-stamped data are in high demand. But nowadays a good database management system is not enough. We live in a world of Big Data, where the most valuable is the knowledge and skills of data analysts and data scientists. And the two most popular computer languages that these specialists utilize in their everyday work are SQL and Python (Haq et al. (2020)). This is one of the reasons that many NoSQL systems, apart from a vendor-specific query language, also adapt parts of the SQL language as an alternative querying method. Database systems designed for time-series data usually have their own query language, sometimes heavily influenced by SQL syntax (Sharma (2020); Rith et al. (2014)). But this approach, intended for people fluent with SQL, has some serious drawbacks. Despite similarities, data analysts still need to learn new structures, search for specific functions and, when faced with a problem, look for help in a limited user community or documentation. A much better approach has been adapted by TimescaleDB, which is based on the PostgreSQL DBMS. Timescale developers took the full PostgreSQLs SQL dialect and extended it with only a few necessary capabilities by either implementing user-defined functions or adding a group of additional clauses or keywords (Mazak et al. (2020); Borkar et al. (2016)). However, despite SQLs popularity and expressiveness, it is well known that more complex queries are difficult to construct and maintain. Also, solving very complex issues quickly becomes a tedious task. Another aspect worth considering is the variety of collected data and common changes of the sales models which require database modification. A plain multivariate time-series model may not be enough. After considering and testing out different solutions for their clients the developers at Synerise S.A. company have decided on creating a novel DBMS ”Chameleon”. It is intended as an in-memory distributed database with a hybrid document-multivariate time-series model which can serve multiple purposes. This data model required a query language capable of dealing at the same time with JSON-style documents and with time-series data with all its complexities. Such language should be easy enough for a data analyst to quickly master it, but also for software developers to be able to effortlessly incorporate it into their products. And yet, those two groups usually have different skill sets and different preferences when it comes to computer languages. After long consideration, the development team has decided to implement a dual-language approach. In order to give data analysts a quick start on our product, we have adapted an SQL dialect based on the 2013 standard with some enhancements like simplified Row Pattern Matching, JSON support, and time-series-specific elements. And for those more accustomed to programming languages, we provide a python-like ChQL query language with the same expressiveness. Both approaches are intended to complement each other.

This paper presents the overview of our data model and the key concepts of both the extended SQL dialect and the ChQL language. The focus is placed on complex problems and explaining some decisions behind specific solutions. This paper is organized as follows: Section 2 gives an overview of the data model. Section 3 introduces the time-series related extensions to SQL dialect and discusses the use of the Row Pattern Recognition (RPR) extension. Section 4 contains an overview of the ChQL query lan-

guage and shows its correlation with our SQL dialect. Section 5 contains conclusions and future work.

2 Data Model

This section presents the models and concepts that form the foundation of Synerises Chameleon DBMS project and its query languages. It starts with a presentation of a motivating example and then proceeds to the technical details.

2.1 Motivating example

One of our baseline motivating examples deals with a hypothetical retail chain. Each store gathers customer receipts info as a JSON document. When a customer shows their loyalty card, their receipt is stored in a time series labeled as purchase with customers uid. Also, the customer has a loyalty app. The basic customer data (uid, name, phone no, etc.) is stored as a single JSON document per uid. The app gathers info on products viewed by the customer as a time series with values such as uid, timestamp, product id, time spent viewing.

After launching a campaign in the loyalty app that showed a selected discounted product, the retail chain wishes to analyze how many of the customers that had viewed the product, have bought it in the week span.

2.2 Hybrid data model

The above example illustrates the main idea behind the Chameleon DBMS: its user should not be confined only to a time-series database functionality, nor to a plain document-type DB. It is often the case that both functionalities are needed to a similar extent. Therefore we propose a hybrid data model so that our clients are not forced to use dual DBMS architecture and to migrate data between systems. Most architectural concepts come from the document storage domain.

2.2.1 Data types Besides the basic data types like *int*, *double*, *string* or *boolean*, Chameleon DB also provides support for arrays of elements with different data types. Our DBMS also supports a range of data types for storing date and time values. The most important is the *timestamp* type, which is used while defining time series. In order to store document data, Chameleon supports a JSON-based *document* type. An instance of this type is a tree-like structure, with named nodes on each level. Node names on a specific level are unique. Node values may be of any type supported by Chameleon DB. Also, the order of nodes is not preserved.

Example 1. A sample time-stamped document stored as a single record

```
{"uid": "24b7afdd-bfd2-4823-86b4-bb0cf4eec96f",
"event_id": 123469146807, "timestamp": 12357684,
"seriesuid": "purchases", "params": {
```

```
   "item": [ {"sku": "code2", "unitprise": 453,
      "name": "xyz1", "quantity": 2.0},
{"sku": "code10", "unitprise": 662,
  "name": "xyz2", "quantity": 1.5} ],
 "sum": 2229.0},
"guid": "med22-pl", "action_id": 6}
```

2.2.2 Data structures Basic data types are used to store singular data, however, to organize it, more complex structures are required. In Chameleon DB, each database instance consists of *collections*. A collection is a set of data with a predefined structure. In some way, it resembles a table in a relational DBMS. The collection structure must be defined by an ordered, non-empty list of named key attributes and a list of named value attributes. Each attribute has a specific type - one of the basic types or a document type. A collection must have a defined primary key. According to intuition, the primary key is defined by an ordered, non-empty subset of the collection's attributes, and it should uniquely identify a database record.

A collection may also have an *ordering key* defined. The ordering key consists of one or more of the collection's attributes. It determines the logical and physical ordering of the database records. The ordering key does not have to be unique. A typical example is the timestamp of the event described by the database record. However, this is not the only possible data type. All basic data types may be used, and it is easy to imagine using a string type representing product names.

A very useful feature known from relational DBMSs is a view. Its equivalent in ChameleonDB database is called a *projection*. It is a defined operator which selects a set of collections attributes. It is also capable of selecting a set of documents nodes.

It should be noted here, that Chameleon DB is designed with centralized in-memory storage enhanced with a capability of placing indexes on any part of a collection, including any node of a document. Also, when a collection is both timestamp-ordered and contains document data, the system may choose to compress on-the-fly JSON type fields of a record.

3 Querying Chameleon DB

While analyzing different use cases it quickly became apparent that the potential users of Chameleon DB would mostly fall into one of two categories. Either they could be programmers, skilled in object-oriented programming languages like C♯, Java, Python, or JavaScript, or they could be data analysts, working on their daily basis with SQL and Python/R languages (for building machine learning models). Those two groups have different predispositions and habits. Many developers, when given a choice, prefer to communicate with a DBMS using either some form of an ORM library or a query language resembling one of the most popular programming languages. Therefore many NoSQL DBMSs provide a query language resembling either JSON documents or a scripting language (Sharma (2020); Ramesh et al. (2016)).

On the other hand, one of the basic skills required from a data analyst is proficiency in SQL. Unfortunately, this language, by design, has limited capabilities when it comes

to handling other data structures than those from relational DBMSs. Thus, a lot of vendors decide to implement a query language resembling, to a lesser or greater extent, SQL (Rith et al. (2014); Borkar et al. (2016)). The degree of similarity usually depends on the similarity of their data model to the relational model.

In ChameleonDB it has been decided that only one approach is not enough. The main assumption was that two user profiles will form the majority. Firstly there will be users fluent with SQL but not so much in other technologies. And yet, some problems will still require advanced algorithmics where SQL would be troublesome. Those problems will be solved by the second group of users - software developers. Therefore ChameleonDB implements a dual approach: SQL dialect extended with basic functionalities required when dealing with time-series data and ChameleonQL (ChQL in short) - an interpreted python-style object-query language.

3.1 SQL dialect for time series

When analyzing different approaches to SQL-based query languages for non-relational databases we may encounter a lot of various designs. From limited versions based on the sole core of the SQL to dialects with proprietary extensions highly influencing the shape of queries. Usually, those extensions are necessary to unlock the full potential of a DBMS. However, considering that this query language is dedicated to analysts proficient in standard SQL, the differences are often problematic and unintuitive, and solutions known to users are not available. Also, since such language exists only in one DBMS, there is limited access to professional help at user-forums and a limited amount of learning materials. Thus, in ChameleonDB our goal was to limit such extensions to a bare minimum. The design choices were made while constantly bearing in mind the profile of the target user - a data analyst or a business representative, not fluent with advanced algorithmics. Our user should be able to start querying a database without having to read through stacks of documentation.

The core syntax is based on the SQL:2016 with SQL/JSON standards (Michels et al. (2018)). In a use case, where a user would not require time-series specific functionalities, a query on a collection would not be any different from a query on a single table. However, to address the aspect of time series and to utilize the collection ordering aspect some additional functionality had to be added.

When dealing with time series not only date parts extraction is needed. It is often necessary to aggregate parts of a series based on a specific time interval. For example, one may need to calculate the number of page visits in a span of 10 minutes or to find the minimum oxygen blood saturation level every 30 seconds. After careful consideration, it has been decided to simply extend the well-known *EXTRACT, ROUND, CEIL* and *FLOOR* functions with the possibility of defining intervals of any length required by the user.

Examples of such calls are:
ROUND(timeVal, 'm', 15) - rounding with interval of 15 min, *EXTRACT(timeVal, 'day')* - extracting weekday name from date, *FLOOR(timeVal, 's', 30, startingPoint)* - assigns a timestamp value calculated by adding 30 second intervals to a startingPoints timestamp value.

Another functionality, useful when dealing with ordered data series, is extracting a given element. Thus Chameleons SQL dialect includes functions: *FIRST, LAST, GET-NTH(n, valNode)*. In the last case, the 'n' value may be positive or a negative, where, for example, -2 would mean the second last element

3.2 Row Pattern Matching

A very important part of time series analysis is finding patterns. SQL:2016 standard introduced the Row Pattern Recognition feature. It heavily utilizes regular expression and currently is available in only a selection of database systems (like Oracle or Apache Flink). The main clause for this feature is MATCH_RECOGNIZE extending the From clause. The main ideas behind this feature are to logically partition and order the data, define patterns using variables and regular expressions which are matched against a sequence of rows. An example of a query containing RPM is presented in Example 2 and is taken from (Michels et al. (2018)).

Example 2. SQL:2016 Row Pattern Recognition example

```
SELECT M.Symbol, M.Matchno, M.Startp, M.Bottomp,
M.Endp, M.Avgp FROM Ticker
MATCH_RECOGNIZE
 ( PARTITION BY Symbol
   ORDER BY Tradeday
  MEASURES MATCH_NUMBER() AS Matchno,
A.Price AS Startp,
LAST (B.Price) AS Bottomp,
LAST (C.Price) AS Endp,
AVG (U.Price) AS Avgp
ONE ROW PER MATCH
AFTER MATCH SKIP PAST LAST ROW
PATTERN (A B+ C+)
SUBSET U = (A, B, C)
DEFINE /* A defaults to True, matches any row */
B AS B.Price < PREV (B.Price),
C AS C.Price > PREV (C.Price) ) AS M
```

In simple cases, like the one presented above, the naming of parameters is self-explanatory. However, the details may quickly become complicated. Let us take a closer look at the PATTERN element. Patterns are built according to POSIX regular expression definition. They may contain a number of modifiers. For example, the expression: `^X{1,3} Y|(Z+) $(V|W{3,})` means that a pattern should start with one to three elements of X condition matching rows, followed by a single Y condition matching row or at least one row matching the Z condition and ending with either exactly one V row or at least 3 W rows. And here we are faced with a difficult design choice. Row Pattern Recognition is a necessary functionality when dealing with data series. But it is also a well known fact, that regular expressions tend to quickly become overly complicated and difficult to maintain. It takes a good programmer to fluently work with them.

While designing ChameleonDB a series of use cases that require RPR feature have been analyzed. And after very careful consideration it has been decided against the implementation of this form of queries in ChameleonDB. The base assumption for its query languages is that the SQL dialect would be mostly used by non-programmer users, whereas users more skilled in algorithmics would be provided with a more powerful, object-oriented query language. The target SQL users should be given basic row pattern matching capabilities, but presented in a very readable, easy to master, form. Thus the idea to enhance the WHERE clause with a THEN parameter which may be used to specify the order of conditions for a pattern. Our version of the query from Example 2 would take the form presented in Example 3.

Example 3. Row pattern matching example for ChameleonDB

```
SELECT Symbol, FIRST(T.A.Price) as Startp, LAST(T.B.Price)
AS Bottomp, LAST (T.C.Price) AS Endp, AVG (Price) AS Avgp,
FunnelCount(), FunnelTag
FUNNEL UNBOUND
FROM Ticker T
WHERE True AS 'A',
   THEN  price < PREV(price) AS 'B',
  THEN  price > PREV(price) AS 'C'
GROUP BY Symbol, FunnelTag
ORDER BY Tradeday
```

The Order By clause is optional when the queried collection has an ordering key. What needs to be explained here is the FUNNEL clause. It is an optional clause that actually serves multiple purposes. It is often the case where business analysis creates conditional funnels where the first level of a funnel shows elements matching a first of the set of conditions, the second level - those matching first and a second condition, and so on. Using the funnel clause gives access to functions: FunnelCount resulting in a JSON document whose node names are the step aliases, and node values store the number of rows matching the step. FunnelTag is the sequential number representing which pattern in a series is currently displayed (since a pattern may occur multiple times in a data series). When users would like to utilize the ALL ROWS PER MATCH functionality, they may simply remove FunnelTag property from the group by clause. The FUNNEL clause comes with 3 modifiers: UNBOUND, BOUND and OVERLAP. UNBOUND means that a pattern is matched when at least the first two conditions match (A and B in the above example). It may be useful when looking for customers, who have viewed a promotional item in a loyalty app and made some purchases within a span of a week, but we wish to check how many of them have bought the item in question. The BOUND modifier means that all conditions must be met in order to match a pattern. But the OVERLAP modifier is the most interesting one.

Let us consider a case when we want to find drivers who tend to drive recklessly. Let us define reckless driving as firstly with accelerating above 8 $\frac{m}{s^2}$, at some point exceeding the speed of 100 $\frac{km}{h}$ and then hard braking when acceleration drops below -9 $\frac{m}{s^2}$. However, there may be points in time when two of those three conditions are met at the same time. What is more, there may not be a point in time where the speed is

above 100 and neither of the two acceleration conditions are met (driver accelerated up to 120 $\frac{km}{h}$ and then hit the brakes). This situation is very hard to define in SQL:2016 RPR standard, because a row is tagged with the first condition tag it matches. Example 5 shows a fragment of an SQL:2016 query corresponding to the proposition in Example 4.

Example 4. Example of the usage of the FUNNEL OVERLAP modifier

```
SELECT CAR_ID FROM CARS
FUNNEL OVERLAP
WHERE  ACCEL > 8 AS A
THEN SPEED > 100  AS B
THEN  ACCEL < -9 AS C
GROUP BY CAR_ID
```

Example 5. Fragment of a RPR query matching with Example 4

```
SELECT * FROM CARS
PARTITION BY CAR_ID
PATTERN A+ B+ C* D+ E+.
/*A-Accel, B-Accel and Speeding, C-Speeding, D-... */
WHERE A.SPEED <= 75 mph AND A.ACCEL > 8
  AND B.SPEED > 75 mph AND B.ACCEL > 8
  AND C.SPEED > 75 mph  AND C.ACCEL >= -9 AND C.ACCEL <= 8
  AND D.SPEED > 75 mph AND D.ACCEL < -9
  AND E.ACCEL < .
```

FUNNEL clause may also be used without the THEN subclauses. With Bound and Unbound parameters it may be combined with the AND operator and thus becoming a simple business funnel representation similar to the one from the New Reliq DBMS. The Funneltag variable would then contain the number of logical expressions evaluated to the True value. The query with the Unbound modifier would firstly return those records, for which the Funneltag value is the lowest (but greater or equal to 2, according to the abovementioned Unbound modifiers properties), unless specified otherwise in the Order by clause.

Usage of the Bound modifier is no different from not using the FUNNEL clause at all. The requirement to use the Funnel clause is that the WHERE clause should contain only conjunctions of logically evaluated expressions.

As for the general approach, it has been decided for ChameleonDB to implement the SQL grammar compliant with the SQL:2016 core specification, with the Funnel clause being the only major modification. The time-series specific functionality, like calculation of missing integer values in a time series, or timestamp rounding to a given level of precision, is accessible through built-in function (like estimate_missing() and round_to_interval() for the abovementioned cases).

4 Chameleon Query Language

Chameleon QL (ChQL for short) was designed with professional programmers in mind. It is based on the Python language design so that it would be as easily used as a stan-

dalone query language and adapted as a library for Python or JavaScript. The basic data types match those mentioned in Section 2. It has been decided that a data type should be associated with a value and not with a variable (similarly to the Python approach), the need arose to provide a distinction between string, timestamp, and date values. One approach would be to use special objects (like in Pythons Pandas library), but in ChQL it has been decided that another approach would be more convenient for a user. In ChQL the user needs to provide a simple one-letter indicator to distinguish between a timestamp (e.g. t402993000) or a date (e.g. d1995-02-04 22:45:00).

In order to work with simple data collections we designed the *filter* function. It corresponds to the WHERE clause in SQL. An example of usage would be:

```
 Buses.reading.filter(sensorName==’brake’ and MaxPower>50).
(sensorName, sensorNo)
```

The Buses in this example is a collection with one of its fields, called reading, being a JSON document storing information gathered during some buses route at some point in time. As may be seen here, ChQL uses the dot operator in an extended way in comparison to the Python language. It is not simply an operator used to access fields and methods of an object, but rather an operator that evaluates the right expression in the context of the whole left-side expression. Such an approach has firstly been used in the SBQL proposed by K. Subieta in (Subieta et al. (1995)). According to this approach, when a class A would have static integer fields labeled x and y, we could request A.(x,y,x+y) and the interpreter would return a three-valued tuple with fields calculated based on the As values of x and y.

Other useful database functions, like count(), average(), distinct() are very similar to those available in MongoDB. However, one of the most interesting functions would be the one corresponding to the FUNNEL clause for the SQL dialect. It is called simply *series* and it results in a tuple of lists of values that match the first condition, the first and the second condition and so on. An example of usage of this operator is shown in the example 6. It is a ChQL version of a query from example 4.

Example 6. Example of the series function usage

```
Cars.car_id.series(mode=’Overlap’, pattern=’A+B+C+’,
  A=(accel > 8 ), B=(speed > 75), C=(accel < -9)).car_id
```

The *series* function accepts arguments passed in positional and keyword mode, as in *Python*. This function has two special named parameters: mode corresponding to FUNNEL modifiers, and pattern corresponding to the PATTERN subclause in SQLs RPR. A, B and C serve as alias names and may be used in search pattern definition as seen in example 7. The limitations on alias names is that only alphanumeric symbols may be used.

Example 7. Another series function usage

```
Alarms.alarm_id.series( A=(alarm=True),
B=(B.alarm = False AND B.time - FIRST(A.time) <= t’5min’),
pattern=’(A+B+){50,}’).
(alarm_id, First(A.time), Last(B.time))
```

The *series* function is dedicated to working with data series, but when we wish to achieve the business funnel functionality, the *pattern* argument should be explicitly set to None. In comparison to SQLs Funnel query, the series function is more complicated but offers much more possibilities. The main difference lies with the definition of patterns. Here the design choices return to the users profile. SQL dialect is intended for people who may not be fluent with programming languages, and as a consequence, may not be experienced with regular patterns. For users more experienced in programming mastering the ChQL would not pose a problem, or, more precisely, would take a similar time as mastering SQLs RPR functionality. Thus, the decision to put easier, but limited, functionality in the SQL dialect, and allow for more robust, more powerful functionalities through a new language.

To summarize, ChQL architectural design was mostly influenced by the SBQL (Subieta et al. (1995); Burzańska and Wiśniewski (2007)), but also by MongoDBs query language, and Python. Each database, collection, projection, and record is an object that inherits its query methods from its generic class. Variables are not assigned a specific type, instead, the type is associated with the data they hold. To distinguish between database objects and user variables ChQL uses SBQLs naming-scoping-binding principle. ChQL offers a limited amount of basic data types and relies mostly on working with objects and their methods. Generic classes offer constructors for new databases and collections. ChQL also offers specific methods for creating and handling new records in a given collection.

5 Conclusions and future work

This paper presented a dual querying approach implemented in a hybrid model ChameleonDB Database Management System. ChameleonDB may serve as a document storage DBMS or a time series focused DBMS. The advantage of this model over popular document databases lies in its capabilities when working with advanced time-related operations and row pattern recognition aspects. On the other hand, the advantage of using ChameleonDB over popular time series DBMSs is the ability to use complex data types as metrics and values.

In addition, two complementary query languages have been introduced. Their design is based on both the intended users profile and the similarity to well-established solutions associated with data mining - SQL and Python. In this approach, some of the more difficult functionalities have been separated and the design of the SQL dialect has been adjusted to fit the intended user profile. At the same, a novel python-like query language supporting the more advanced functionalities has been designed. Among the novel aspects of both the ChQL and the SQL dialect are the support for Row Pattern Recognition functionalities and business funnel construction.

As future work, we plan to design an automated translation tool that would translate SQL queries into ChQL. We also plan to benchmark our concepts against DBMSs like MongoDB and Timescale. Furthermore, there is ongoing development work on the incorporation of elements such as advanced compression and extended indexing in ChameleonDB.

References

Borkar, D., Mayuram, R., Sangudi, G., Carey, M. (2016). Have your data and query it too: From key-value caching to big data management, *Proceedings of the 2016 International Conference on Management of Data*, pp. 239–251.

Burzańska, M., Wiśniewski, P. (2007). L-value and r-value concept-proposition to solve ref & deref chaos in sbql languages family, *Pol. J. Environ. Stud* **18**(3B), 143–151.

Haq, H. B. U., Kayani, H. U. R., Toor, S. K., Zafar, S., Khalid, I. (2020). The popular tools of data sciences: Benefits, challenges and applications, *IJCSNS* **20**(5), 65.

Mazak, A., Wolny, S., Gómez, A., Cabot, J., Wimmer, M., Kappel, G. (2020). Temporal models on time series databases, *target* **1**, 1.

Michels, J., Hare, K., Kulkarni, K., Zuzarte, C., Liu, Z. H., Hammerschmidt, B., Zemke, F. (2018). The new and improved sql: 2016 standard, *ACM SIGMOD Record* **47**(2), 51–60.

Ramesh, D., Sinha, A., Singh, S. (2016). Data modelling for discrete time series data using cassandra and mongodb, *2016 3rd international conference on recent advances in information technology (RAIT)*, IEEE, pp. 598–601.

Rith, J., Lehmayr, P. S., Meyer-Wegener, K. (2014). Speaking in tongues: Sql access to nosql systems, *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 855–857.

Sharma, C. (2020). Flux: From sql to gql query translation tool, *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 1379–1381.

Subieta, K., Beeri, C., Matthes, F., Schmidt, J. W. (1995). A stack-based approach to query languages, *East/West Database Workshop*, Springer, pp. 159–180.

Received July 26, 2021 , revised March 12, 2022, accepted April 12, 2022