Programmatic implementation of the Dijkstra algorithm in the Transact-SQL language using relational algebra

Mikhail Urubkin[1,*], *Vasiliy* Galushka[1], *Vladimir* Fathi[1], *Denis* Fathi[1], and *Sofya* Petrenkova[1]

[1]Don state technical university, chairs Computing systems and information security, 1, Gagarin square, 344000, Rostov-on-Don, Russia

Abstract. The article is devoted to the topical issue of data processing in the database management systems. It presents a solution to the problem of finding paths in a graph using Dijkstra's algorithm, set as a sequence of relational operations and functions of the Transact-SQL language. The efficiency of the known information system architectures and the impact of various ways of distributing functions between system components are reviewed. The article describes features of the relational algebra and the Transact-SQL operations, and provides a brief description of Dijkstra's algorithm. For its programmatic implementation, several stages are defined, for each of which a formal description of the relational operations performed on it is given. The outputs of these operations are shown using the example of the database tables, and the algorithm to find the final path is given. The issues of the proposed method's productivity and security of programmatic implementation of the path search in a graph are discussed separately.

1 Introduction

Contemporary large information systems are built, as a rule, using one of the options of a distributed architecture, the most common of which are client-server and three-tier architectures. Both of them assume the presence of a database (DB) server in the information system structure, which functions are strictly limited to storing data, searching for it, and issuing it on request, as well as an application server in the three-tier architecture or a thick client in the client-server architecture. In any of these cases, there is a need to transfer data over the network between the components of the information system, and along with this, there are increasing requirements for the bandwidth of communication channels and the level of transmission security, which in turn increases the computational cost of performing cryptographic operations. These restrictions are determined by the basic principles underlying all distributed architectures — each part of the application must perform only those functions for which it is intended.

* Corresponding author: mishanya005@ya.ru

However, the specific character of some tasks makes using standard architectures inefficient for the application development. This applies in particular to the tasks that meet the following conditions:

a) large amount of source data,

b) small amount of output data,

c) a relatively simple data processing algorithm.

An example of such a task is to search for paths on a graph. It happens not only when you need to find the route on the map, where, for example, streets are indicated with graph edges and crossroads with graph vertices, but in other cases: building a route to a page on a site, searching for a node in a computer network, etc.

The solution of this task using a classic three-tier architecture will require storing data in the database in form of relational tables, passing them to the application, converting them to adjacency matrices, performing path calculation, and sending the result to the client (Fig. 1). With large amounts of data, these processes will take a long time, which makes this scheme inefficient and leads to the need of using alternative ways to distribute functions between the components of the information system in order to reduce an amount of the data transferred between them and, as a result, increasing the operation speed and security.

This goal can be achieved by combining the functions of the application server and the database server (Fig. 1), which will allow eliminating the need to transfer data between them, perform conversions from the storage format in the database to the presentation format in the program, and will generally simplify the architecture and logic of the information system.

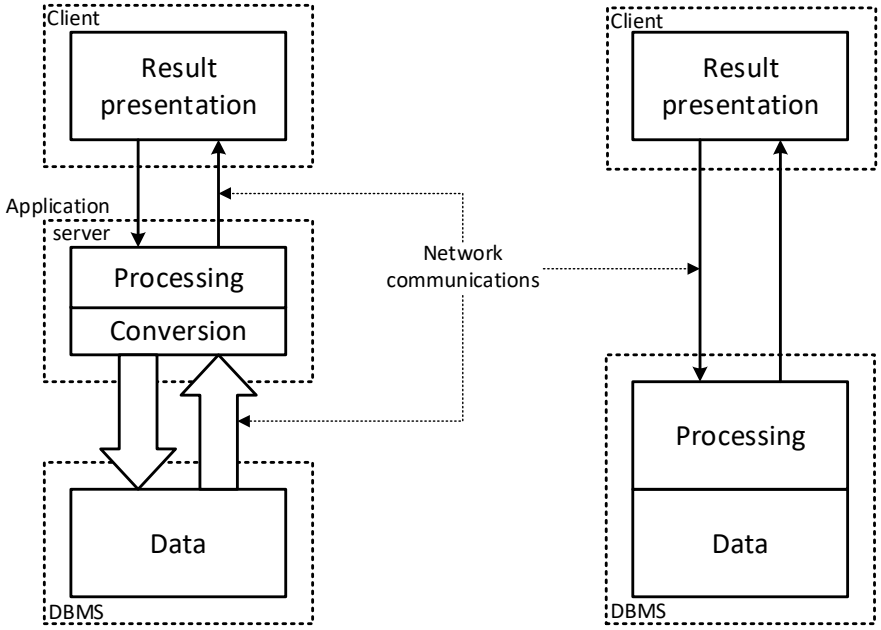

Fig. 1. Processing the Data on Different Tiers of the Application Architecture.

As illustrated in Fig. 1 and the above descriptions, the proposed approach requires the capability to implement data processing by means of a database management system (DBMS). The SQL language is responsible for processing data in the DBMS, namely its DML (Data Manipulation Language) subset, however, some DBMSs have its extended versions, in particular Transact-SQL in Microsoft SQL Server, which rely on standard relational algebra operations and supplement them with the structural programming

capabilities. The features of this language allow you solving a wide range of the data processing tasks of almost any complexity. In this paper, we will consider the programmatic implementation of the path search algorithms in a graph in the Transact-SQL language, using the example of Dijkstra's algorithm.

2 Relational Algebra and Transact-SQL Language

Most contemporary database management systems are based on relational theory. This is reflected both in the structure of databases consisting of tables, which formal description is given through the concept of relation, and in the data manipulation tools based on the relational algebra operations. Support for these operations is available in all relational DBMSs and is described in ANSI SQL-92 standard.

Relational operations can be divided into set-theoretic and special. Set-theoretic operations include operations known from the set theory, such as union, intersection, difference, and Cartesian product. The second group includes operations that apply only to relations: selection, projection, and join.

Selection is an operation that selects a set of rows in a table that meet specified conditions. Any logical expression can be a condition.

Projection is an operation in which only attributes from the specified domains are selected from the relation, that is, only the necessary columns are selected from the table, and if several identical tuples are obtained, then only one instance of such a tuple remains in the resulting relation.

The join operation is the reverse of the projection operation and creates a new relation from two already existing relations. The new relation is obtained by concatenating tuples of the first and second relations, while concatenating only relations in which the values of the specified attributes match.

These operations allow performing rather complex data manipulations, but the Transact-SQL language has additional special commands corresponding to the structural programming operators that allow controlling the flow of script execution, interrupting it or directing it to the desired branch.

- Block of grouping (BEGIN … END) is a structure that combines a list of expressions into a single logical block.
- Block of condition (IF … ELSE) is a structure that checks whether a certain condition is met.
- Block of cycle (WHILE … BREAK … CONTINUE) is a structure that organizes the repetition of a logical block execution.
- Transition (GOTO) is a command that performs transition of the script execution flow to the specified label.

Similar to other programming languages, Transact-SQL has variables for temporarily storing intermediate results. However, it does not contain arrays in the usual sense of the term. Nevertheless, given that Transact-SQL operates with databases, it uses tables instead of arrays. They are created either by means of standard DDL (Data Definition Language) commands, or by means of proper operators that allow declaring table variables or creating local temporary tables.

The Transact-SQL language also has several options for implementing subprograms, namely stored procedures, scalar and tabular user functions.

Taking into account peculiarities of the described programming language and relational algebra operations when we implement the path search algorithms in a graph, it is necessary to modify an ingenious method (in this case, Dijkstra’s algorithm), while preserving its general essence and idea.

3 Dijkstra's Algorithm

Dijkstra's algorithm is designed to find the shortest distance from one of the vertices of the graph to all the others. It works well only on graphs without negative weight edges. This algorithm iterates through all vertices of a graph gradually and assigns them labels that are a known minimum distance from the source vertex to the specific vertex. When assigning labels and iterating over vertices, you should follow the rules below:

1) at each subsequent iteration, the vertex with the smallest known distance is selected;

2) for the selected vertex, the distance from the source vertex to all its neighboring vertices is calculated;

3) if at any stage of the algorithm, the distance to the vertex was found to be less than the minimum known distance, then the minimum known distance is replaced with the found distance value.

4 Initialization Phase

The source data for the algorithm is a list of edges stored in the database as a relation:

$$S_E = \{\underline{id}, in, out, len\}, \quad (1)$$

where *id* is the surrogate primary key of the relation,

in is the number of one of the vertices (the conditional start of the edge),

out is the number of the vertex with which vertex *in* is associated (the conditional end of the edge),

len is the length of the edge connecting vertices *in* and *out*.

In fact, this table already stores elementary paths, i.e. paths that are one edge long.

According to Dijkstra's algorithm, you need to keep a list of visited vertices. However, taking into account that all the vertices will be added to the algorithm during its implementation, as well as the need to store additional information about the vertices and peculiarities of processing tables in the databases when it is more beneficial in terms of computation to perform one mass operation than a set of single operations, we will add to the vertex table the information about all the vertices at once. This table is a representation of Relation *V* with the following scheme:

$$S_V = \{v, status, distance, prev\}, \quad (2)$$

where *v* is the primary key of the relation specifying the vertex number,

status is the current state of the vertex at the specific algorithm iteration (0 — not visited, 1 — visited, 2 — being reviewed during this iteration), at the beginning of the algorithm, the vertex from which paths are searched for is assigned "*status* 2";

distance is the current computed shortest distance from the source vertex to this specific vertex;

prev is the number (*id*) of the previous vertex which the path passes through with the minimum distance.

Table *V* is populated based on Relation *R* using the following formula:

$$V = \Pi_A(E), \quad (3)$$

where Π is a special relational operation of projection on a set of attributes *A*.

The value of each attribute is selected according to the following rules:

1) the value of *id* is taken without changes from the source table *E*;

2) 2 if *id* is the same as the source vertex number, 0 – in all other cases;

3) 0 if *id* is the same as the source vertex number, ∞ – in all other cases (instead of ∞ you should calculate the maximum possible value for the data type used in this attribute, for example, for integers using the built-in function POWER(2.0, 31) – 1);

4) it takes the value of *NULL* for all the rows, reflects the fact that the previous vertices were not set before starting the graph traversal and they will be specified later.

5 Vertex Update Phase

The next step starts the direct calculation of paths. According to Dijkstra's algorithm, during each iteration, there are considered paths from the vertex, which is current at that moment.

Their length is calculated using the formula:

$$T = \Pi_{E.out,0,V.distance+E.len,E.in}(\sigma_{status=2}(V \rhd\lhd E)), \quad (4)$$

where $\sigma_{status=2}$ is the relational operation of selection according to the condition of "*status* = 2";

$\rhd\lhd$ is the relational operation of natural join of relations, i.e. the join by common attributes (for *V* and *E* such an attribute is *id*);

Π is the projection of the output of previous operations on a set of attributes.

We can see that the scheme of the received Relation *T* matches scheme *V*: its first attribute is the vertex number, the second is the status, the third is the distance, and the fourth is the vertex number too. Fig. 2 illustrates the example of the table-type representation of all the mentioned relations and performance of the operations over them.

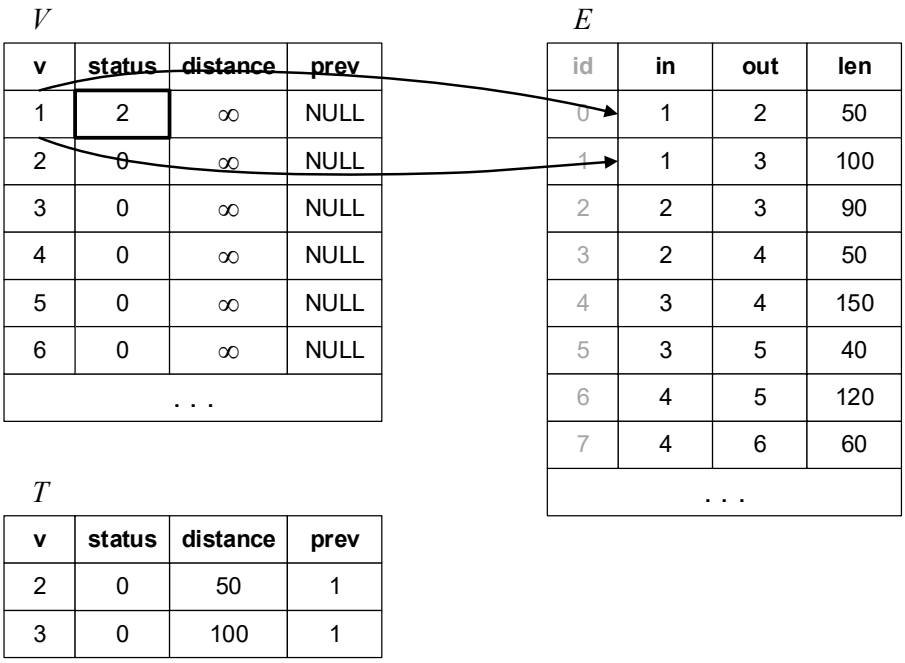

Fig. 2. Merging Tables at the First Algorithm Iteration.

Then, from table *T*, you should select vertices for which the calculated path value is less than the known path stored in table *V*.

$$T = (T \rhd\lhd_{\beta} V), \tag{5}$$

where $\rhd\lhd_{\beta}$ is the relational operation of θ-connection using predicate β.

$$\beta = \text{T.v=V.v} \wedge \text{T.distance} < \text{V.distance}, \tag{6}$$

where $\wedge$ is the operator of logical AND.

For the example shown in Fig. 2, all the rows in table *T* will be selected, because at the first iteration of the algorithm, any path found will be considered the shortest.

The next step is to replace the old entries in table *V* with the new ones, specifying shorter paths found. To do this, you need to perform two relational operations: difference and union:

$$V = (V - T) \cup T. \tag{7}$$

After that, the vertex that was selected as the current one should be marked as visited by setting "*status* = 1", and you should move to the definition of the new current vertex. To do this, from table *V* the one that has the minimum value of the *distance* attribute is selected and its *status* is changed to "2", after that the path calculation is repeated again. The criterion for stopping the algorithm is the absence in table *V* of vertices with the value of "*status* = 0".

Fig. 3 shows the result of making one more step of the algorithm. At that step, unlike the previous one, the route is found to the vertex under number 3, through vertex 2 having the length of 140; however, it will not replace already existing in *V* the route to vertex 3 through 1 of the shorter length.

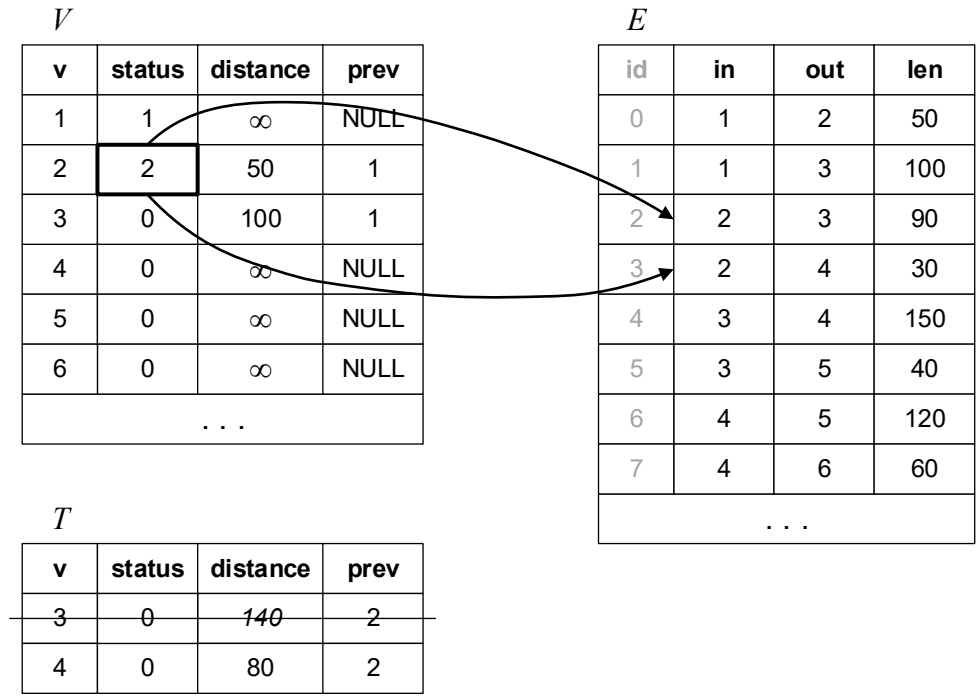

Fig. 3. Merging Tables at the Next Algorithm Iteration.

6 Route Recovery

The described algorithm determines the shortest distance from the source vertex to all the others, but in practice, the most important thing is not the length of the route, but the points or vertices that it covers. To find them, you need to perform the route recovery procedure, which is performed according to table *V* as follows:

$$p_i = v_k.prev, \tag{8}$$

where p_i is the *i*th point of the route,
v_k is the *k*th row of table *V*.

$$k = \Pi_{id}(\sigma_{id=rk.prev}(V)). \tag{9}$$

In this case, the condition of the completion will be the absence of the *prev* value in the *k*th row of table v_k, i.e. $v_k.prev$ = NULL. The example of the route recovery is shown in Fig. 4.

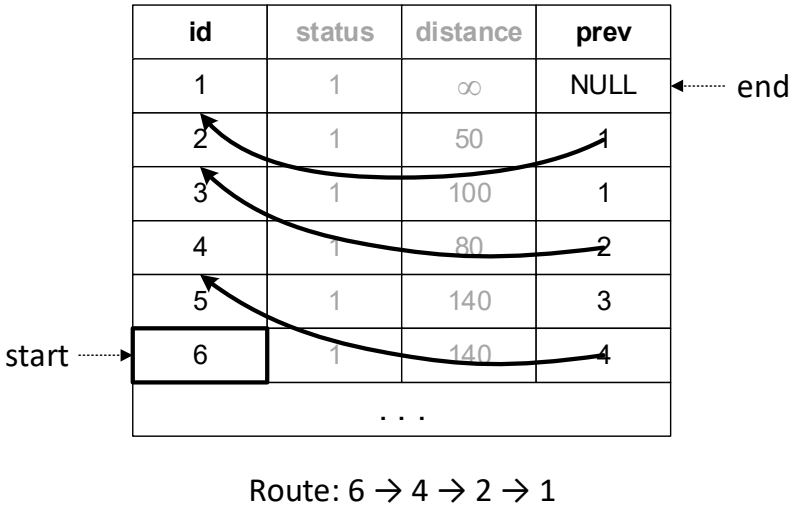

Fig. 4. Route Recovery.

7 Conclusions

The described method of implementing Dijkstra's algorithm allows using tools of Microsoft SQL Server DBMS to perform the data processing functions, namely, finding the shortest path in a graph. This approach reduces the load on client computers or application servers, significantly reduces network traffic between components of the information system, and, as a result, increases the information security. It allows shifting a significant part of the computing operations to the database server, the processing power of which is often not fully used.

The disadvantage of this method is lower performance compared to the traditional algorithmic programming languages. However, when developing a complex information system, it is required to take into account the specifics of the task for which it is designed and to consider the application not separately but together with the other parts of the information system given the need for their integration. These performance losses are compensated with no need for the data conversion from the format of storing in the

database to the format of representing in the application, and with the absence of time delays for transferring them between the database and application.

The performed programmatic implementation of the method using the Transact-SQL language shows the opportunity for further implementation of other graph algorithms in the same way and development of a generalized approach to implementing algorithms on graphs by means of relational algebra.

References

1. A. Vasconcelos, P. Sousa, J. Tribolet, The Electronic Journal Information Systems Evaluation **10(1)**, 91-122 (2007) ISSN 1566-6379
2. J. Sowa, J. Zachman, IBM Systems Journal **31** (1992) https://doi.org/10.1147/sj.313.0590
3. A.-W. Scheer, *Architecture of Integrated Information Systems* (Springer, Heidelberg, 1992) https://doi.org/10.1007/3-540-26661-5_25
4. D. Harabor, A. Grastien, AAAI Conference on Artificial Intelligence, 1114-1119 (2011) https://doi.org/10.5555/2900423.2900600
5. A. Mendelzon, P. Wood, SIAM J. Comput. **24(6)**, 1235–1258 (1995) https://doi.org/10.1137/S009753979122370X
6. I. Düntsch, Artificial Intelligence Review **23**, 315-357 (2005) https://doi.org/10.1007/s10462-004-5899-8
7. P. Höfner, G. Struth, International Joint Conference on Automated Reasoning **5195** (2008) https://doi.org/10.1007/978-3-540-71070-7_5
8. E. Dijkstra, Numer. Math **1** (1959) https://doi.org/10.1007/BF01386390