MATEC Web of Conferences 76, 04017 (2016) DOI: 10.1051/matecconf/20167604017
CSCC 2016

Employing finite-state machines in data integrity problems

Andrey Malikov[1,a], Vladimir Voronkin[1] , and Nikolay Shiryaev[1]

[1] *Department of Applied Information Science, North-Caucasus Federal University. 1, Pushkin Street, Stavropol, Russian Federation.*

Abstract. This paper explores the issue of group integrity of tuple subsets regarding corporate integrity constraints in relational databases. A solution may be found by applying the finite-state machine theory to guarantee group integrity of data. We present a practical guide to coding such an automaton. After creating SQL queries to manipulate data and control its integrity for real data domains, we study the issue of query performance, determine the level of transaction isolation, and generate query plans.
Keywords: database, integrity, finite-state machine, finite-state automaton, directed graph, SQL query

1 Problem formulation

The most common definition of a data model includes a combination of three interrelated components [1]: the structural component that defines the structural contents of objects used in the model; the manipulation component, which describes the operators that manipulate the model's objects; the integrity component, which consists of rules that guarantee data integrity.

In context of a relational data model, the structural component defines a data object as an a-nary relation built on a set of domains. Relations contain unorganized sets of tuples, which are identified with a primary key.

The manipulation part of the model consists of relational calculus and relational algebra. The most popular declarative language aid for data access is the Structured Query Language – SQL.

The integrity component is vital for the functioning of informational systems. Despite our inability to fully match all qualities and properties of real-life objects in our model, this component will ensure that these properties are consistent and non-contradictory.

There are three types of integrity rules in relational databases: entity integrity – no primary key attribute of a basic relation should have a NULL value; referential integrity – any value of a foreign key in a relation must match the value of a primary or candidate key in the parent relation or be comprised solely of NULL values and corporate integrity constraints – additional rules for data consistency defined by the data domain.

The first two rules are already implemented in relational DBMS quite effectively and efficiently. Typically, these data integrity mechanisms focus on correctness of singular tuples and data values.

Corporate integrity constraints may demand the tuples be in some group consistency, while remaining individually correct and strictly correspond to the real-life object of the data domain, for example: the overall salary of employees in one department must not exceed some limit and the individual salaries of each employee must be in a certain range; hiring and firing heads of departments should only be performed in conjunction with hiring or firing the inferior staff; managing complex objects (described by more than one tuple) when you need to maintain a chain of tuples, linked by binary relations of order. For example, a table of traffic light colors should be ordered like this: "red" then "yellow" then "green".

There are some special cases when it comes to organizing relations, such as managing temporal data [2], hierarchical data [3], and the combination of the two [4]. The mentioned cases bring structural limitations to the model. That is why it is necessary to maintain group integrity of tuples. Let us examine each case closely.

Constraining temporal data, is when you need to maintain the right chronological order. In this case, every tuple corresponds to a state of an object at a point in time; the subset of these tuples describe the changes that the object undergoes during some period of time. It is necessary, therefore, to make sure that any pair of tuples is not in a temporal contradiction, and that the full subset of tuples creates a logically sound chain like: "state 1" before "state 2" before "state 3" etc. As an example, let us look at the stages a student goes through during their study at a university: "enrolling at the university" then "moving up a year" then "…" and finally "graduating".

Constraining hierarchical data is needed to guarantee correct subordination of all objects. A typical example is the chain of command in a company from "head" to "deputy" to "assistant".

Constraining temporal hierarchical data is simply maintaing the chain of subordination throughout the timeline.

Providing data integrity is relevant not only for relational data models. The currently popular NoSQL DBMS (such as MongoDB [5]) either does not support the ACID transaction management model or supports in only partially

[a] Corresponding author: AMalikov@ncfu.ru

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

[6]. A database designer interacts with simple-structure documents, identifiable by a unique id (key-value model). Interrelations between documents can be created based on their nesting-levels or by using key references, and that is why we are also considering the issue of group integrity of documents.

We can create a binary relation for each of the mentioned cases on the tuple-subset level, i.e. we can compare any pair of tuples from the subset, while the subset itself is ordered. This fact lets us apply automata theory [7] to achieve integrity of any subset of tuples. Introducing new mechanisms of constraining data integrity deviates us from the classical definition of a relation, because tuples stop being completely independent.

Typically, when solving such problems, corporate logic is implemented as a servicing application. This spreads the database interaction logic on both the application and the database. With time, such a system becomes more difficult to maintain.

A much simpler way is to implement corporate logic on one level.

The most reasonable way for large systems that operate numerous queries and procedures is to realize the update logic on the database level and put all the rest in the application.

A quite thorough description of finite automata, logics of several orders, and their uses is given in L.Libkin's book [8]. A model of finite automata theory elements in databases is described in V.Vianu's paper [9].

Using finite state machines doesn't just have uses in corporate integrity control. There are applications in the banking sector [10], uses for modelling trading systems [11], and creating recursive queries [12] for tasks like graph processing.

2 Implementing the finite-state machine in a relational database

Let us consider the realization of a finite state machine to constrain group data integrity using the Transact-SQL language in the MS SQL Server DBMS.

The finite state machine is set as $(Q, \sum, \delta, q_0, F)$ [7], [13], where: Q is the finite set of possible states of the machine;

$\sum$ is the alphabet of the automaton;

δ is a transition function, and $\delta : Q \times \sum \rightarrow Q$

q_0 is the start state.

F is the set of possible final states, $F \subseteq Q$

Let us examine the following example (without giving up the generic nature of our model) – a finite state machine of student statuses (fig. 1). The program module that tracks the chronology of students' movement is used in the information and education hub at the North-Caucus Federal University in Stavropol, Russia. Websites [14], [15] provide specialized educational content for students of over 500 schools and universities in the Stavropol region as well as providing consistent data processing of student movement for over 500 000 students.

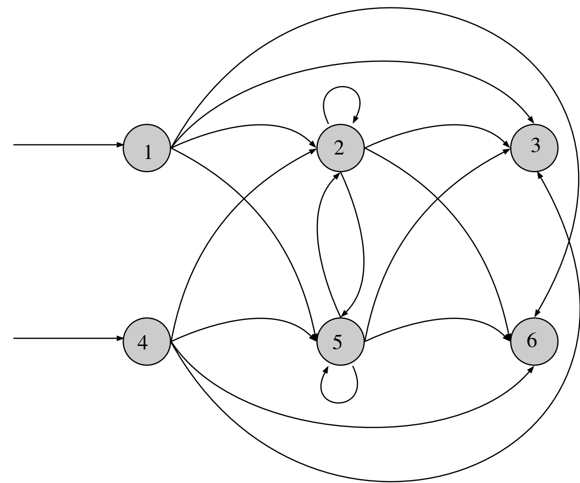

Figure 1. The finite-state machine of student statuses during their education

Table 1. The transition matrix for the machine

Alphabet	Machine state						
	start	1	2	3	4	5	6
1	1						
2		1	1		1		1
3		1	1		1		1
4	1						
5		1	1		1		1
6		1	1		1		1

The set of states consists of six elements $Q = \{1$ – (enrolling); 2 – (moving up a year); 3 – (expulsion due to graduation); 4 – (transfer from another university); 5 – (transfer to another university); 6 – (left to repeat the course)$\}$. (1) and (4) are start states and (3), (5) are final states. The alphabet $\sum$ in our case, is a set of characters that matches the set of possible stages Q. The automaton reads a word $w = (a_1, a_2, \ldots, a_n)$, $a_i \in \sum$ – a time-sorted sequence of states. The machine only accepts the word w, if by the end of starting word the machine ends up in an allowed state.

The transition matrix for the machine is presented in table 1.

To implement a finite-state machine in a relational database, one needs to create special tables, in which the machine's structure is coded by at least one of the following means [16], [3]:

First, the "parent-child" concept for coding the graph's adjacency matrix. The advantage of this method is its versatility in modelling complex graph structures. The weakness is lower query performance compared to the "materialized path" method. Another shortcoming is the difficulty of restoring valid words via a declarative language. However, this method is considered to be universal and can be used to model any arbitrary finite-state machine.

Table 2. An example of the "*movement*" table

idMov	idSchoolboy	idPr	dateMov
1	1	1	01.09.2014
2	1	2	01.07.2015

Second, the materialized graph path concept. The strengths of this method are data visualization, which makes it easier to comprehend, because each materialized path has a valid word attached to it, and high performance handling graph models when using index structures like B+tree. However, there are two disadvantages to this method. The first is rapid growth of relationship cardinality, which, in the worst case, will grow exponentially, depending on the properties of the machine. The second major disadvantage is the inability to code cycles and cyclical transitions – their presence would lead to a word with infinite length. Nevertheless, this method is the best alternative for modelling large machines with no cycles.

Third, the "ancestor-successor" concept for coding the graph's reachability matrix. This method has all of the pros and cons of previously mentioned methods.

3 Coding the machine in the parent-child model

Let us examine the specificity of using a finite-state machine that will check for data integrity and is coded in the parent-child model. We will require three tables:

First, the index-table of automaton states: *process (idPr int, name varchar(max))*.

Second, a table with the machine itself, two foreign keys of which refer to *process.idPr: automaton (idPar int, idCh int)*. The pair *(idPar int, idCh int)* is the primary key of the "*automaton*" table. According to fig. 1 the "*automaton*" table will have 20 entries; starting states are coded as (0,1) and (0,4), final states – as (3,0) and (5,0). Here are the contents of table "*automaton*" {(0,1), (1,5), (1,3), (1,2), (1,6), (2,2), (2,3), (2,5), (2,6), (0,4), (4,2), (4,6), (4,5), (4,3), (6,2), (6,6), (6,3), (6,5), (3,0), (5,0)}.

Third, a table of student movement. The foreign key will refer to *process.idPr: movement (idMov int, idSchoolboy int, idPr int, dateMov date)*. Here, *idSchoolboy* is the student's id; *dateMov* is the date of movement.

Table 2 contains an example of the "*movement*" table for student *idSchoolboy=1*. Group integrity of tuples is intact, because chronologically, the student was first admitted to the university, and then moved up a year.

According to Table 2, an attempt at adding the row (3, 1, 5, '01.09.2015') will result in success, but trying to add (3, 1, 5, '01.06.2015') will end in failure, because there would be a contradiction, where the final state is between the intervening ones.

Implementing a finite-state machine with this model does not allow us to effectively restore the list of acceptable words via a declarative language. In the worst case, cyclical transitions will make this list infinitely large (refer to fig.1).

We will use two lemmas to judge the acceptability of words, based on their fragments.

Lemma 1 - Word fragment lemma

If $P^n(V', E')$ is a path on graph $G = (V, E)$ of a finite-state machine $V' \subseteq V, E' \subseteq E, v_i' \in V', i = 0..n$; and $P^m(V'', E'')$ is a path on graph $G = (V, E)$ of a finite-state machine $V''' \subseteq V, E''' \subseteq E, v_i''' \in V''', i = 1..k$, so that,

1. $\exists(v_n', v_1'''), (v_n', v_1''') \in E$, or $|V'| = 0$;
2. $\exists(v_k''', v_1''), (v_k''', v_1'') \in E$, or $|V''| = 0$;

then $P = P^n \bigcup P^k \bigcup P^m$ is an acceptable path on graph G.

Consequence of lemma 1: a compound path exists on a graph, if all component paths exist and the condition of chronological order is met.

Lemma 2 – Lemma of the first fragment of an expectable word:

If in lemma 1 $q_0 \in P^n$, then P is the prefix of an expectable word.

According to the lemmas, it is necessary and sufficient to: check whether the new entry is allowed to change from the old state into the new state or if it is placed in the starting position; in order to allow parallel processing of transactions, including simultaneous access to the same word, it is necessary to introduce the following rule: transactions that manipulate data (adding, editing and deleting) should block the range of entry pairs, between which the manipulated entry lies, according to binary ordering relationship.

Let us create a query to add a single tuple. This will be a successful try (the initial state of the "*movement*" table is presented in Table 2)

Query 1:

```
declare @idSchoolboy int,
        @dateMov dateTime,
        @idPr int
set @idSchoolboy=1
set @dateMov='01.09.2015'
set @idPr=5
set transaction isolation level
    serializable

insert into movement (
        idSchoolboy,
        dateMov,
        idPr)
select @idSchoolboy, @dateMov, @idPr
from automaton previousConnection,
       automaton nextConnection
where previousConnection.idCh=@idPr
    and previousConnection.idPar=
        isnull((select top 1 idPr
    from movement b
    where b.idSchoolboy=@idSchoolboy
        and b.dateMov<=@dateMov
    order by dateMov desc),0) and
  nextConnection.idPar=@idPr and
  nextConnection.idCh=
      isnull((select top 1 idPr
```

```
        from movement b
    where b.idSchoolboy=@idSchoolboy
        and b.dateMov>=@dateMov
    order by dateMov asc),
    case isnull((select top 1 idPr
      from movement b
      where b.idSchoolboy=@idSchoolboy
          and b.dateMov<=@dateMov
      order by dateMov desc),0) when 0
    then (select top 1 idCh
      automaton anyConnection
      where idPar=@idPr)
    else 0 end)
```

And now, and attempt at adding an entry, which will fail the integrity check.

Query 2:

```
declare @idSchoolboy int,
        @dateMov dateTime,
        @idPr int
set @idSchoolboy=1
set @dateMov='01.06.2015'
set @idPr=5
set transaction isolation level
    serializable

insert into movement (
                idSchoolboy,
                dateMov,
                idPr)
select @idSchoolboy, @dateMov, @idPr
from automaton previousConnection,
         automaton nextConnection
where previousConnection.idCh=@idPr and
  previousConnection.idPar=
    isnull((select top 1 idPr
    from movement b
    where b.idSchoolboy=@idSchoolboy
      and b.dateMov<=@dateMov
    order by dateMov desc),0) and
  nextConnection.idPar=@idPr and
  nextConnection.idCh=
    isnull((select top 1 idPr
      from movement b
    where b.idSchoolboy=
      @idSchoolboy and
      b.dateMov>=@dateMov
    order by dateMov asc),
    case isnull((select top 1 idPr
      from movement b
      where b.idSchoolboy=
        @idSchoolboy and
        b.dateMov<=@dateMov
      order by dateMov desc),0) when 0
    then (select top 1 idCh
        from automaton anyConnection
        where idPar=@idPr)
    else 0 end)
```

The main purpose of this SQL-query is to find adjacent entries between which we are adding a new item. If the machine allows the transition from a previous state to the new one and from the new one to the next one, then the operation will be successful. The exceptions are trying to add a starting or a final state. In the first case, we only need to check if it is consistent with the following states, and in the second – we only need to verify consistency with states before it. If the algorithm finds inconsistency, the operation fails.

The query consists of four parts:

1. Preparation of data to be added into the target table:

```
insert into movement (idSchoolboy,
                      dateMov, idPr)
select @idSchoolboy, @dateMov, @idPr
```

2. The table with data about the structure of the automaton is opened under two pseudonyms to search for states before and after the one in question:

```
from automaton previousConnection,
       automaton nextConnection
```

3. Verifying that the previous sates are acceptable or that the entry is in a starting state:

```
where previousConnection.idCh=@idPr and
  previousConnection.idPar=
    isnull((select top 1 idPr
    from movement b
    where
      b.idSchoolboy=@idSchoolboy and
      b.dateMov<=@dateMov
    order by dateMov desc),0) and
```

4. Verifying that the following sates are acceptable or that the entry is in a final state. Also, performing the consistency check for a case when the entry we are adding is the very first one in the group:

```
  nextConnection.idPar=@idPr and
  nextConnection.idCh=
    isnull((select top 1 idPr
    from movement b
    where b.idSchoolboy=@idSchoolboy
    and b.dateMov>=@dateMov
    order by dateMov asc),
    case isnull((select top 1 idPr
      from movement b
      where b.idSchoolboy=
        @idSchoolboy and
        b.dateMov<=@dateMov
      order by dateMov desc),0) when 0
    then (select top 1 idCh
      from automaton anyConnection
      where idPar=@idPr)
    else 0 end)
```

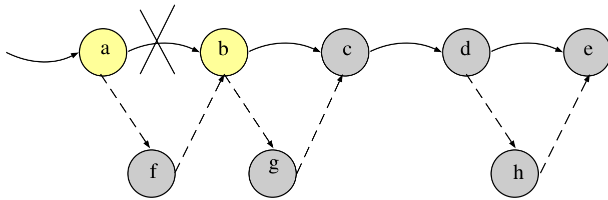

Figure 2. Parallel execution of transactions that alter the same word

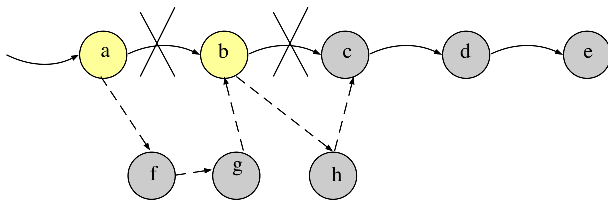

Figure 3. Batch loading of several symbols into a word

4 The issue of query performance and parallel transaction processing

Let us examine the issue of paralleling transactions that alter the same word simultaneously and the issue of implementing batch data loading transactions.

When dealing with parallel transaction processing, it is necessary to establish blocking rules, which would deny access to joint characters while the transaction of adding new characters is in processing. Fig. 2. presents a case of parallel processing of three transactions: tr1, tr2, and tr3, each one trying to add one symbol to alter the word 'abcde'. Transaction tr1 destroys the fragment 'ab' by creating a new one – 'afb'. This leads to a blocking of a range of entries with symbols 'a' and 'b'. That is why any transaction that alters the fragment 'ab' cannot be processed simultaneously with tr1. Transaction tr2 must run after tr1 because it will wait for the entry with 'b' to be unblocked. Transactions tr1 and tr3 can be processed at the same time, because they do not block shared entries.

In the case of batch loading of data, where one transaction attempts to add several symbols to a word, it is important to note that new word fragments are created with already existing symbols as well as new ones. Fig. 3. presents a situation, where tr1 blocks the range of entries with symbols 'a', 'b', and 'c' by destroying fragment 'abc' and creating a new one – 'afgbhc'.

To make sure that all transactions are executed correctly, we will need a SERIALIZABLE level of transaction isolation. It would block a range of keys that meet the following conditions [17]: instructions cannot read data that was just altered by another transaction but not yet formalized; other transactions cannot alter data that is being read by the current transaction until it finishes; other transactions cannot create new rows with key values that are in the range of keys being read by the instruction of the current transaction until it finishes.

Analysis of queries (1, 2) shows ways of improving performance of data processing operations. Taking into account query predicates and chosen fields, we will need to create the following B+tree type indexes:

– A clustered composite index with uniqueness support – *automaton.IX_idCh_idPar(idCh, idPar)*. It will be used to search for previous states when we know current ones and will ensure that the composite primary key {idCh, idPar} is unique.

– A non-clustered composite index *automaton.IX_idPar_idCh (idPar, idCh)*. It will be used to search for following states when we know the current ones.

– A non-clustered compound index *movement.IX_idSchoolBoy_dateMov_idPr(idSchoolBoy, dateMov)* with included column *ipPr*. It will be used to search data by students and their movement dates.

The relevant part of a query plan is shown in Fig. 4. It uses index searching to access data. Analysis operators that check whether a new tuple is consistent with previous sates are in box number one, the ones checking consistency with the next states are in box number two, and the ones checking the newly added starting and final states are in box three.

Now let us create a query to add a set of tuples containing symbols, which will alter the initial word (the initial state of the "*movement*" table is presented in Table 2)

Query 3:

```
declare @movement_new table (idMov int,
  idSchoolboy int, dateMov dateTime,
  idPr int)
insert into @movement_new (idMov,
  idSchoolboy, dateMov, idPr)
    values (-1, 1, '01.10.2014', 1)
insert into @movement_new (idMov,
  idSchoolboy,  dateMov, idPr)
    values (-2, 1, '01.10.2015', 2)
insert into @movement_new (idMov,
  idSchoolboy,  dateMov, idPr)
    values (-3, 1, '01.11.2015', 3)
set transaction isolation level
serializable;

begin transaction;
with movement_tmp (idMov,
  idSchoolboy, dateMov, idPr, isNew)
as
(select idMov, idSchoolboy,
        dateMov, idPr, 1
 from @movement_new
    union all
 select idMov, idSchoolboy,
        dateMov, idPr, 0
 from movement
    where idSchoolboy in
      (select distinct idSchoolboy
       from @movement_new))
insert into movement (idSchoolboy,
```

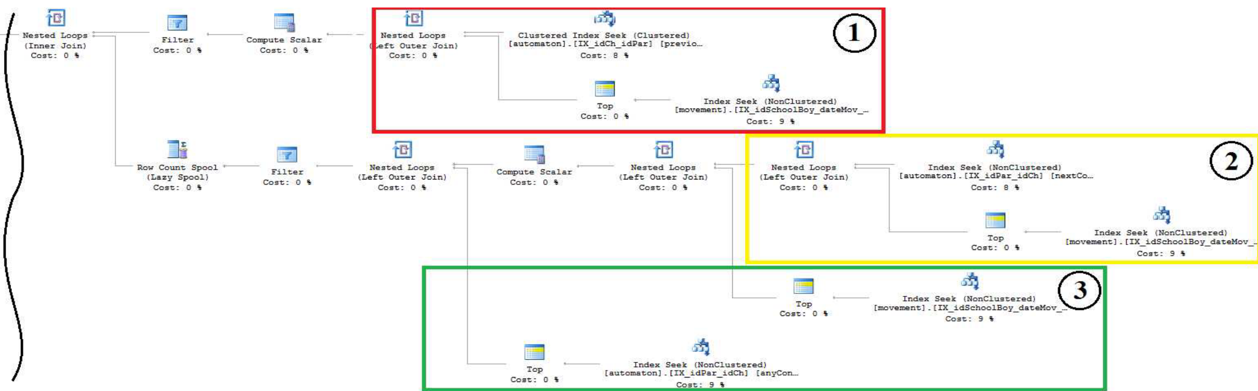

Figure 4. The relevant part of query plan to add a single tuple

```
                    dateMov, idPr)

select movement_tmp.idSchoolboy,
  movement_tmp.dateMov,
  movement_tmp.idPr
from automaton previousConnection,
  automaton nextConnection,
  movement_tmp
where movement_tmp.isNew=1 and
  previousConnection.idCh=
    movement_tmp.idPr and
  previousConnection.idPar=
    isnull((select top 1 idPr
      from movement_tmp b
      where b.idSchoolboy=
        movement_tmp.idSchoolboy
        and b.dateMov<=
          movement_tmp.dateMov and
        b.idMov<>movement_tmp.idMov
      order by dateMov desc),0) and
  nextConnection.idPar=
    movement_tmp.idPr and
  nextConnection.idCh=
    isnull((select top 1 idPr
      from movement_tmp b
      where b.idSchoolboy=
        movement_tmp.idSchoolboy and
        b.dateMov>=
          movement_tmp.dateMov
        and b.idMov<>
          movement_tmp.idMov
      order by dateMov asc),
    case isnull((select top 1 idPr
      from movement_tmp b
      where b.idSchoolboy=
        movement_tmp.idSchoolboy and
          b.dateMov<=
            movement_tmp.dateMov
          and b.idMov<>
            movement_tmp.idMov
      order by dateMov desc),0) when 0
    then (select top 1 idCh
          from automaton anyConnection
```

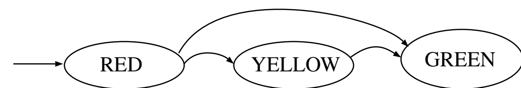

Figure 5. A finite-state machine of traffic light configurations

```
          where idPar=
            movement_tmp.idPr)
      else 0 end)
commit transaction;
```

This SQL-query is very similar to the one we examined previously. There are several differences between the query to add a single tuple and this one. Firstly, we declare a *table* type variable to which we add tuples with candidate-symbols to be added to a word. The generalized table expression *movement_tmp* contains a predicted result of the operation, which is further specified by the insert command. We use *movement_tmp.isNew* as an attribute for identifying candidate-symbols and already existing characters. The result of *insert* is a new word in the movement table – a word that was altered only with new characters from *@movement_new* which passed the automaton.

Operations of deletion and updating are implemented similarly.

5 Coding a finite-state machine using materialized paths

Let us look closely at using a finite-state machine, a model of which utilizes materialized paths of a graph in relational system. This case demands there be no cycles, like in Fig. 5 – a model of traffic light configurations (either with two or three lights)

We will need three tables in order to code this machine in a relational database:

1. An index table of colors named *color (idC int, name varchar(max))*. The table will contain { (1, red), (2, yellow), (3, green) }.

2. A table with the machine itself, done in the materialized path model. The key of the path *graphKey* is primary

Table 3. the *trafficLight* table

idTL	idC
1	1
1	3

for the relation. The clustered B+tree type index is built on this key. Every byte of it contains a value for each individual color of the traffic light, coded from left to right: *(idC int, graphKey varchar(max))*. According to Fig. 5. the automaion table will have four entries: {(1,'1'), (2,'12'), (3,'13'), (3,'123')}. Materialized path values in final states are valid words; and values in interim stages are word prefixes.

3. A traffic light table with a foreign key referring to *color.idC: trafficLight (idTL int, idC int)*. In our case, idTL is an id of a traffic light and (idTL, idC) is the primary key.

An example of the *trafficLight* is in table 3.

According to Table 3, attempting to add (1, 2) will be successful, because it will create an acceptable three-coloured traffic light *idTL = 1* out of an acceptable two-coloured light. Trying to add (1, 3) will fail, because the system will realize that we are trying to make a traffic light with two red lights.

In this case, we already know all valid words and prefixes. The integrity check will consist of just creating a word and checking if it exists in the *automaton* table. Adding new entries will be done through a table type variable *@light_new* that holds candidate-entries to be added to the *trafficLight* table. Note that the system will react differently to adding new entries than our previous example. The parent-child method assumes that we can add individual symbols that pass the machine, whereas materialized paths allow characters to be added and edited in groups – if all new symbols meet the machines restrictions, then they all get in the word, otherwise, if at least one character does not satisfy the machine's criteria, the entire operation is declined.

Let us create a query to add a set of tuples as a batch. These tuples contain characters that would modify the existing word. Initial state of the *trafficLight* table is given in Table 3.

Query 4:

```
declare @light_new table (idTL int,
                          idC int)
insert into @light_new (idTL, idC)
  values (1, 2)
insert into @light_new (idTL, idC)
  values (2, 1)
insert into @light_new (idTL, idC)
  values (2, 3)
set transaction isolation level
  serializable;

begin transaction;
with trafficLight_tmp (idTL, idC)
as
(select idTL, idC
  from @light_new
union all
select a.idTL, a.idC
  from trafficLight a
  where a.idTL in (select
                  distinct idTL
                  from @light_new))

insert trafficLight
select tl_new.idTL, tl_new.idC
from @light_new tl_new,
     (select distinct idTL
      from trafficLight_tmp) tl,
        automaton au
where au.graphKey=
  (select top 1
    (select convert(char(1),b.idC)
      from trafficLight_tmp b
        where b.idC<=a.idC and
          a.idTL=b.idTL
        order by 1 FOR xml path(''))
        from trafficLight_tmp a
        where a.idC=(select max(idC)
          from trafficLight_tmp b
          where a.idTL=b.idTL) and
            a.idTL=tl.idTL) and
         tl_new.idTL=tl.idTL
commit transaction;
```

In this case, an attempt at adding two traffic lights will be successful, because we are creating a valid tree-colored traffic light idTL = 1 out of a valid two-colored light and adding a new two-colored light idTL=2 with an acceptable set of colors.

The main purpose of this SQL-query is to form new words and prefixes out of candidate-symbols and already existing characters, then search for this word or prefix in the machine's table. If it finds a match, then it lets the symbols through; if not – ignores them. The query consists of four parts:

1. Preparing data to be added into the target table.

```
declare @light_new table (idTL int,
                          idC int)
insert into @light_new (idTL, idC)
  values (1, 2)
insert into @light_new (idTL, idC)
  values (2, 1)
insert into @light_new (idTL, idC)
  values (2, 3)
```

2. Forming a generalized table expression with an expected outcome of the operation.

```
with trafficLight_tmp (idTL, idC)
as (select idTL, idC
  from @light_new
union all select a.idTL, a.idC
  from trafficLight a where a.idTL in
    (select distinct idTL
     from @light_new))
```

3. Forming a word out of the expected result for each object

```
(select top 1
  (select convert(char(1),b.idC)
   from trafficLight_tmp b
     where b.idC<=a.idC and
       a.idTL=b.idTL
     order by 1 FOR xml path(''))
       from trafficLight_tmp a
       where a.idC=(select max(idC)
         from trafficLight_tmp b
         where a.idTL=b.idTL) and
           a.idTL=tl.idTL)
```

4. If this word is one of the values of *automaton.graphKey*, then all new symbols are added into the target table

```
insert trafficLight
select tl_new.idTL, tl_new.idC
from @light_new tl_new,
  (select distinct idTL
   from trafficLight_tmp) tl,
     automaton au
```

Operations of deletion and updating are implemented similarly.

6 Conclusion

This paper tackles the issue of providing data integrity. Corporate integrity constraints often demand tuples of certain relationships be consistent and non-contradictory as a group. This paper presents a new method of achieving integrity in logically connected subsets of data in relational databases using finite-state machines.

Implementation the automaton in flat tables can be achieved using several models: 'parent-child', 'ancestor-successor' or with materialized paths on a graph. The 'parent-child' model supports cycles in graphs but requires recursive restoration of valid words of the machine. This model works well when we need to check, whether a word fragment is valid for a given machine. In addition, it lets us add only those symbols that are acceptable to the machine. The 'materialized paths on a graph' model lets us store and process valid words and prefixes more clearly but does not support cycles. This model is preferred for cases where symbols are added as a group and when we need to verify that all new symbols are acceptable.

We provide examples of data management queries and integrity check queries using the Transact SQL language in the MS SQL Server DBMS. We looked at query performance, stated index requirements, and determined the necessary level of transaction isolation.

All models, data structures, and code presented in this paper are used successfully in a number of informational and educational outlets of the North-Caucasus Federal University and schools in the Stavropol Region, Russia (such as http://eCampus.ncfu.ru, http://olymp.ncfu.ru).

References

[1] J.D. Christopher, *Relational Theory for Computer Professionals (Theory in Practice)* (O'Reilly Media, 2013)

[2] C. Date, H. Darwen, *Time and Relational Theory: Temporal Databases in the Relational Model and SQL* (Morgan Kaufmann Publishers Inc., 2014)

[3] J. Celko, *Joe Celko's Trees and Hierarchies in SQL for Smarties* (Morgan Kaufmann Publishers Inc, 2004)

[4] A. Malikov, M. Sugakov, D. Parkhomenko, Y. Gulevskiy, *Temporal tree and its use in the conceptual modeling of databases* (2010), Vol. 2, pp. 99–104

[5] *Mongodb*, https://www.mongodb.org/

[6] B.G. Tudorica, C. Bucur, *A comparison between several NoSQL databases with comments and notes*, in *Roedunet International Conference (RoEduNet), 2011 10th* (2011), pp. 1–5

[7] T. Kam, T. Villa, R.K. Brayton, A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Functional Optimization* (Kluwer Academic Publishers, 2013)

[8] L. Libkin, *Elements of Finite Model Theory* (Springer-Verlag Berlin Heidelberg, 2004), ISBN 978-3-662-07003-1

[9] V. Vianu, *Databases and Finite-Model Theory*, in *In descriptive complixity and finite models* (American Mathematical Society, 1997), pp. 97–148

[10] I. Ion, C. Cristian, P. Sorin, *Collaborative Systems - Finite State Machines* (2011), Vol. 15

[11] M. Dippery, *Modeling states and transitions in relational databases* (2014)

[12] N. Deininger, *Finite-State Machines with recursive SQL* (2013), https://ef.gy/fsm-recursive-sql

[13] H. Straubing, *Finite Automata, Formal Logic, and Circuit Complexity* (Birkhauser Verlag, Basel, Switzerland, Switzerland, 1994)

[14] *Educational and informational hub of the north-caucasus federal university*, http://eCampus.ncfu.ru

[15] *Hub for university and school student competitions*, http://olymp.ncfu.ru

[16] V. Tropashko, *Nested Intervals with Farey Fractions* (2004), http://arxiv.org/html/cs/0401014

[17] J. Celko, *Joe Celko's SQL for Smarties: Advanced SQL Programming* (Morgan Kaufmann Publishers Inc., 2010)