Proceedings of the IASTED International Conference
DATABASES AND APPLICATIONS
February 17-19, 2004, Innsbruck, Austria

A QUERY LANGUAGE SOLUTION FOR FASTEST FLIGHT CONNECTIONS

J.A. Bakker and J.H. ter Bekke
Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
E-mail: {J.A.Bakker, J.H.terBekke}@ewi.tudelft.nl

ABSTRACT

This paper discusses how to extend the applicability of the recursive *cascade* update command of the Xplain query language. This command guarantees termination because cycle detection is included, which is in favor of end user computing. The execution of this command is based on graph reduction; therefore it can only be applied to acyclic graphs. Because flights constitute cycles between airports, it seemed impossible to specify a query language solution for shortest flight connections between two arbitrary airports. However, on a time scale cyclic connections between successive flights cannot exist at all. Therefore we designed an appropriate semantic data model for feasible connections between successive flights. Using this extended data model, the *cascade* command still enables end users to specify recursive queries for the fastest series of flights between any pair of airports.

KEY WORDS
shortest path, recursive query, semantic model, transitive closure

1. Introduction

Earlier solutions for recursive problems such as transitive closure are described in papers and textbooks [1, 2, 7, 8, 14, 22]. They often create successor lists much larger than the original data set, which leads to time-consuming duplicate elimination and cycle detection [9]. Moreover, these solutions are specified in languages (often using nested control statements) that are difficult to understand by end users. Another problem is associated with recursive views in SQL3. The syntax of this language [22] still allows for procedural operations as the join operation. Therefore SQL3 cannot guarantee termination [3].

Contrary to these earlier approaches, we designed an algorithm using graph reduction [3, 19], thus avoiding data expansion. It is the basis for recursive data processing as can be specified by the *cascade* update command in the language of the Xplain-DBMS [12, 15]. As far as we know this is a new approach with some important consequences for end user computing:

1. Irrespective the problem at hand, graph reduction processes all data defining a directed graph, even in the case of finding a shortest path, where normally not all node and arc data are involved. This seems to be inefficient, but it also offers an analogue solution for critical path problems. In procedural approaches to these problems a programmer must choose start and finish nodes, which is not always possible because there can be many candidates for start and finish. Xplain releases end users of this problem by determining start and finish itself.
2. Graph reduction either ends with an empty graph or with a graph containing one or more cycles [5]. Only for a directed acyclic graph, reduction produces a well-ordered list of arc data, suitable for further serial calculations.
3. The algorithm guarantees termination because cycle detection is part of the arc ordering process. This is essential for open systems - accessible to millions of unknown users - that cannot be protected by authorization tables [6].
4. The worst case complexity of the arc ordering process is $O(N^2)$, where N is the number of arcs. This ordering avoids the data expansion associated with other approaches [4, 10, 13, 14]. Using the list of ordered arc data, further calculations take $O(N)$ time.

The applicability of the new query language approach to recursive operations in databases has been demonstrated for acyclic directed graphs related to project planning [17] and production planning [18], but also for operations in recursive structures such as finding the ancestors of a person in a family tree [20, 21]. This approach requires that the involved data set does not constitute a cyclic graph. Because series of flights can follow geometric cycles, the initial feeling was that graph reduction could not be used for finding fastest air connections.

Section 2 presents a further discussion on this subject and offers an appropriate data structure supporting directed time graphs not containing any cycle between flights. Using this data model, section 3 describes a query for the fastest series of flights between two airports, adjacent or not.

2. Data definition

If we want to apply the *cascade* update command to fastest flight connection problems, then we have to design such a data model that the essential data define a directed acyclic graph. Therefore we designed a more appropriate data model, which allows us to consider flight as nodes, whereas the possible connections between successive

flights can be presented as arcs. In figure 1, capitals indicate airports: the flights *f3* (BC) and *f2* (BC) start after the arrival of flight *f1* (AB). This figure ignores the duration of flights: only their time ordering is important. Each arc represents the possible connection between an arriving and a departing flight. Capital letters indicate airports. In a geographical sense the flights *f1* and *f6* constitute a cycle (AB; BA), but on a time scale they cannot constitute a cycle. Another geographical cycle is (BC; CE; EB), made by the flights *f2*; *f4*; *f7* and also by the flights *f3*; *f4*; *f7*.

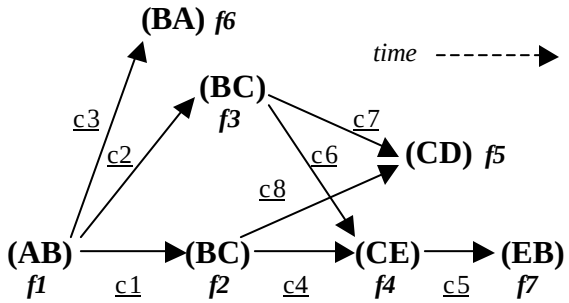
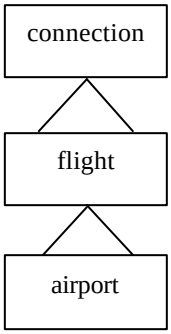


Figure 1. Some connections between flights

Because a connecting flight must start after an arrival, cycles between flights via connections (arcs) cannot occur on a time scale. The semantic data model in figure 2 together with some rules support a correct registration of data on flight connections not having any cycle. We assume that each airport has a certain maximal transfer time for passengers between arrival and departure terminals. Using data on flights and airport transfer times, it is easy to derive the correct instances of 'connection' defining the possible connections between arriving and departing flights.



type connection (i4) = previous_flight (i4),
next_flight (i4), /between_days/ (i2),
/between_time/ (i4), /correct/ (b).
type flight (i4) = carrier (a20), from_airport (a3),
to_airport (a3), duration (i4), arrival_date (d),
arrival_time (i4), departure_date (d),
departure_time (i4).
type airport (a3) = name (a10), transfer_time (i3).

Figure 2. A data model for flights and connections

In the data model 'i4' defines integers of at most 4 digits, 'a3' means a string of at most 3 alphanumeric characters and 'b' indicates the Boolean domain. Time can be expressed through integers of four digits, of which the two most significant digits indicate the hour (0-23); the other two digits indicate minutes (00-59): *hhmm*. Dates are defined in the domain 'd': *yyyymmdd*. Arrival time and departure time are expressed in local time. Flight duration and the between time (available transfer time) of connections is expressed in minutes.

We also have to specify rules, sometimes including a value restriction not expressible in data structure alone. In Xplain they are expressed by assertions specifying a derivable variable. Derived attributes are presented between slashes, but not all of the following derived attributes are shown in figure 2. We assume that a connecting flight departs not later than two days after an arriving flight, which limits the number of connections to be registered.

assert connection *its* between_days (0..2) =
timedif (next_flight *its* departure_date,
previous_flight *its* arrival_date).

The calculation of the time interval between an arrival and a connecting departure requires some preparative calculations:

assert flight *its* departure_hour = departure_time *div* 100.
assert flight *its* departure_minute =
60 * departure_hour + departure_time *mod* 100.
assert flight *its* arrival_hour = arrival_time *div* 100.
assert flight *its* arrival_minute =
60 * arrival_hour + arrival_time *mod* 100.

Now the between time of connections (in minutes) can be calculated:

assert connection *its* between_time =
between_days * 24 * 60 + next_flight *its*
departure_minute – previous_flight *its*
arrival_minute .

Instances of 'connection' must be registered correctly. The following rule assumes that each airport has a certain maximal transfer time for moving passengers between terminals:

assert connection *its* correct (*true*) =
(between_time >= previous_flight *its*
to_airport *its* transfer_time *and* previous_flight *its*
to_airport = next_flight *its* from_airport).

A restriction on the number of days between connecting flights has already been expressed through 'connection *its* between_days (0..2)'. Applying these rules, incorrect connections defining a time cycle of flights cannot be registered.

After designing the data structure (figure 2) and the static restrictions, section 3 proposes a declarative solution for the fastest series of flights between two airports.

3. Finding a fastest connection

As an example we show how the fastest series of flights between two selected airports, starting on a selected date after a selected time, can be determined. Using the desired data on start and finish airport, departure date and time, we can determine the earliest starting time 'est' (in minutes) for each flight having a connection (direct or via other flights) with a suitable flight starting on a selected date and time from the selected airport. This interaction with a user is defined by a series of *input* statements. After that we start with the initialization of the earliest starting time 'est' for all flights to an unreachable value 'inf'. We may assume that 'est' is smaller than 'inf':

```
value start = input (a1)      "Enter the start airport: ".
value selectdate = input (d)
        "Enter departure date (yyyymmdd):".
value selecttime = input (i4)
        "Enter earliest departure time (hhmm): ".
value finish = input (a1)  "Enter the destination airport: ".
value inf = total flight its duration.
extend flight with est = inf.                /* initialization */
update flight its est = 0
     where from_airport = start
     and departure_date = datef (selectdate)
     and departure_time >= selecttime.
cascade flight its est =
    min connection its previous_flight its est +
        previous_flight its duration + between_time
    per next_flight.
```

Data on connections is processed in an order derived by the query interpreter from the query specification [19]. In the previous cascading update the order of assignments is from 'connection *its* previous_flight' to 'connection *its* next_flight'. End users must be aware of this order, but they do not have to specify the ordering process itself, which avoids the risk of specifying an infinite process. Xplain always requires that specifications respect the inherently (through attributes) specified data structure. An example of a correctly specified attribute path is 'connection *its* next_flight *its* from_airport'.

In a similar way we calculate for each flight the earliest arrival time 'eat' possible in a chain of flights between that flight and a flight ending in the destination airport. We assume that the flight arriving at the destination airport may not depart later than two days after the selected starting day:

```
extend flight with eat = inf.
update flight its eat = 0
        where to_airport = finish
        and departure_date >= datef (selectdate)
        and arrival_date <= newdate (selectdate, 2).
cascade flight its eat =
        min connection its next_flight its eat +
            next_flight its duration + between_time
        per previous_flight.
```

Here the destination of assignments by recursive updates is reversed from 'connection *its* next_flight' to 'connection *its* previous_flight'. Now we calculate for each flight the total duration of a series of flights connecting the selected start and finish airport. Then we calculate the minimal value of this chain duration 'cdur':

```
extend flight with cdur = est + eat + duration.
value minimum = min flight its cdur.
```

Not always a series of flights is available between the two selected airports on the selected day. Therefore the following *count* operation determines the number of flights with a usable minimal value for 'flight *its* cdur'. The condition 'est < inf *and* eat < inf' deals with the possibility of a non-connected graph. For example, flights in a chain of connected flights starting in the start airport, but not ending in the finish airport have the property that 'eat = inf'. Similarly, flights in a chain of connected flights ending in the finish airport, but not starting in the start airport have the property: 'est = inf'. The third possibility is that flights belong to a chain neither connected with the start airport nor the finish airport.

```
extend flight with suitable =
        (cdur = minimum and est < inf and eat < inf).
value suitablenumber = count flight where suitable.
value suitablenumber.                  /* a print command*/
value explanation =
        "If suitable number = 0, then there is not any
         suitable connection between the selected
         airports on the chosen day and time: ".
value explanation.
newline.                               /* print commands */
```

More than one chain of connected flights can have the minimum chain duration. Then more than one suitable flight has the same earliest starting time 'est'. A solution for presenting the query results is to calculate the relative position of flights participating in a suitable series of flights starting with the earliest departure time. Then the results can be presented in an order complying with this position:

```
extend flight with position = inf.
update flight its position = 1
        where suitable and est = 0.
cascade flight its position =
    min connection its previous_flight its position + 1
        where previous_flight its suitable
    per next_flight.
value comment =
        /"If more than one flight has a same position,
         then there is more than one connection.".
```

get flight *its* position, from_airport, to_airport,
departure_date, departure_time,
arrival_date, arrival_time, est, eat, duration, cdur
where suitable
per position. /* presentation order */
newline. *value* comment. /*print commands */

4. Test results

For a test the following less interactive initialization is applied to all connections XY: both X and Y can have a value A, B, C, D or E (see figure 1, presenting the items of our test database). So, 25 connections have to be checked:

value start = *input* (a1) "Enter the start airport: ".
value selectdate = *datef* (20040917).
value selecttime = 1015.
value finish = *input* (a1) "Enter the finish airport: ".
value inf = *total* flight *its* duration.

The remaining query is the same as in section 3. The test data are complying with figure 1. Some data such as the flights *f2* and *f3* seem to be incorrect, but are still correct because they pass different time zones. We assumed that all airports have a maximal transfer time between terminals of 100 minutes. Connections are complying with this maximal transfer time.

fli	fr	to	dep_ date	dep_ time	arr_ date	arr_ time	dur
1	A	B	20040917	1020	20040917	1220	120
2	B	C	20040917	1400	20040917	1700	300
3	B	C	20040917	1500	20040917	1805	305
4	C	E	20040917	2000	20040918	0215	615
5	C	D	20040917	1950	20040917	2220	150
6	B	A	20040917	1400	20040917	1550	110
7	E	B	20040918	1100	20040919	0800	900

Table 1. Test data on flights

connection	previous_ flight	next_ flight	between_ time
1	1	2	100
2	1	3	160
3	1	6	100
4	2	4	180
5	4	7	525
6	3	4	115
7	3	5	105
8	2	5	170

Table 2. Test data on connections between flights

Table 3 shows the result of different selections of start and destination airport.

start airport	finish airport	suitable number	flight	position	departure date	dep. time	arrival date	arrival time	est	eat	dur	cdur
A	A	2	1{AB}	1	20040917	1020	20040917	1220	0	210	120	330
			6{BA}	2	20040917	1400	20040917	1550	220	0	110	330
A	B	1	1{AB}	1	20040917	1020	20040917	1220	0	0	120	120
A	C	2	1{AB}	1	20040917	1020	20040917	1220	0	400	120	520
			2{BC}	2	20040917	1400	20040917	1700	220	0	300	520
A	D	4	1{AB}	1	20040917	1020	20040917	1220	0	720	120	840
			2{BC}	2	20040917	1400	20040917	1700	220	320	300	840
			3{BC}	2	20040917	1500	20040917	1805	280	255	305	840
			5{CD}	3	20040917	1950	20040917	2220	690	0	150	840
A	E	4	1{AB}	1	20040917	1020	20040917	1220	0	1195	120	1315
			2{BC}	2	20040917	1400	20040917	1700	220	795	300	1315
			3{BC}	2	20040917	1500	20040917	1805	280	730	305	1315
			4{CE}	3	20040917	2000	20040918	0215	700	0	615	1315
B	A	1	6{BA}	1	20040917	1400	20040917	1550	0	0	110	110
B	B	3	3{BC}	1	20040917	1500	20040917	1805	0	2155	305	2460
			4{CE}	2	20040917	2000	20040918	0215	420	1425	615	2460
			7{EB}	3	20040918	1100	20040919	0800	1560	0	900	2460
B	C	1	2{BC}	1	20040917	1400	20040917	1700	0	0	300	300
B	D	2	3{BC}	1	20040917	1500	20040917	1805	0	255	305	560
			5{CD}	2	20040917	1950	20040917	2220	410	0	150	560
B	E	2	3{BC}	1	20040917	1500	20040917	1805	0	730	305	1035
			4{CE}	2	20040917	2000	20040918	0215	420	0	615	1035
C	A	0										
C	B	2	4{CE}	1	20040917	2000	20040918	0215	0	1425	615	2040
			7{EB}	2	20040918	1100	20040919	0800	1140	0	900	2040
C	D	1	5{CD}	1	20040917	1950	20040917	2220	0	0	150	150
C	E	1	4{CE}	1	20040917	2000	20040918	0215	0	0	615	615

Table 3. Test results

The test results demonstrate that the proposed solution is correct and the found flights satisfy the relationship 'est + eat + duration = cdur'. According to figure 1, the following connections are not available: CA (in table 3) and CC, DA, DB, DC, DD, DE, EA, EC, ED and EE (not shown in table 3). Connection EB is available, but not on the date chosen in the test. In the database there are two return connections B to B, but the query only produces the result *f3; f4; f7* because this series is faster than *f2; f4; f7*. In order to evaluate the usability of the proposed solution, section 5 presents an estimation of its costs in terms of disk-access and time consumption on the basis of some reasonable assumptions.

5. Cost estimations

We assume that flight data is available from all airports over the world because airports need such data for daily operations. Of course each airport database contains data on flights arriving or leaving that airport. Our approach requires that also data on the possible connections (transfers) is stored, but these data can be derived from the local flight data.

We have to transfer these local data to a central flight information system. We assume that for each individual fastest connection problem, we only have to import flight data related to a selected period of three days. Using these assumptions we can calculate the number of flight and connection data for the calculation of a fastest series of flights between a start and finish airport on a selected date and time.

We estimate the number of major civil airports all over the world is 1000. We assume an average of 30 arrivals and also 30 departures per hour at each airport during 10 hours per day. This means 300 arrivals and 300 departures per day on each airport, so the number of flight records that has to be imported over a period of three days is approximately 1800 per airport, thus 1.800.000 over the world in three days. However, a starting flight is also an arrival for the destination airport, so the number of flight data in three days is 900.000. If the format of each flight record is 200 bytes, then approximately 180 Mbytes of data has to be sent to the flight information system.

We assume that the number of possible connections after each arrival flight is 100 per day, thus 300 per three days per airport. In the case of 300 arrivals per day we have to register 300 * 100 = 30.000 connections for each airport: 1000 * 30.000 = 30.000.000 connections over the world in three days. If a connection record takes 100 bytes, this means that an additional 3000 Mbytes must be transported. Including flight records a total of 3180 Mbytes must be sent to the flight information system. If the transfer rate is 1 Mbytes/s, this means a total transfer time of 53 minutes.

However, it is not necessary to transfer all data on connections; they can be derived from data on flights and the transfer time within airports. Probably, depending on the chosen file organization (applying hashing or indices), this is less time consuming.

Another problem is the number of disk-accesses required to store 30.000.000 connection records and 900.000 flight records. Each connection record has a format of 100 bytes, so we need minimally one disk-access per ten records: in total 3000.000 disk-accesses. If a disk-access takes 10 ms, this means a total of 30.000 seconds or 500 minutes for storing connection records. Similarly we can estimate that storing the flight records takes 180.000 disk-accesses or 30 minutes. Then storing all these data takes 530 minutes or almost 9 hours. Data can be imported during nightly hours and because of the many information requests, all data on future flights and connections must be locally stored for a period of several months.

The time complexity of the graph reduction algorithm is $O(\boldsymbol{dN})$, where $\boldsymbol{N}$ is the number of arcs (connections) and $\boldsymbol{d}$ is the depth of the graph (probably smaller than $\boldsymbol{N}$). Considering the large number of connection and flight records needed for processing one information request using the *cascade* command, it would take 53 minutes (3180 Mbytes, 1 Mbytes per sec.) to read the disk data needed to answer a request about a fastest connection within three days. A solution is to store all data permanently in main memory. This would reduce the time duration with at least a factor 5000 to approximately 0.7 second.

6. Discussion and conclusion

The present paper shows some important results:

- The declarative Xplain language enables end users to specify recursive operations. We think that earlier procedural solutions, such as Dijkstra's shortest path algorithm [1], are less suitable for end user programming because they require the application of control statements with nested loops.
- The underlying algorithm itself is fast and guarantees termination, which is essential in open systems with many unknown users: it is not possible to protect such systems by authorization tables. This contrasts with the application of recursive views in SQL3 [22], a language with a syntax that cannot guarantee termination [3]. Relational languages allow us to specify procedural operations such as Cartesian product and join that do not require respecting referential integrity. In Xplain only the attribute path's existing in a data model may be used.
- The proposed semantic data model enables us to transform a cyclic geometric directed graph for airports and flights into an acyclic directed graph for flights and connections. Data on feasible connections can be derived from flight and airport data.
- The time complexity of the described solution is high because of the large size of the required data set; still it can be executed in polynomial time.

Future work has to reveal whether the proposed solution can also be applied to more complex transitive closure problems.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, Reading (MA), p. 208 (1974).

[2] D.A. Bailey, Java structures: Data structures in JAVA for the principled programmer, MacGraw-Hill, Boston, USA (1999).

[3] J.A. Bakker and J.H. ter Bekke, A Query Language Solution for Shortest Path Problems in Cyclic Geometries, *Proceedings International Conference on Database Applications (DBA 2004)*, Innsbruck, Austria 2004, ACTA Press, Calgary (2004).

[4] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies, *Proceedings 1986 ACM SIGMOD International Conference on Management of Data*, Washington D.C., USA (1986), pp. 16-52.

[5] J. Bang-Jensen and G. Gutin, Digraphs: Theory, Algorithms and Applications, Springer-Verlag, London, UK (2001).

[6] Bert Bakker and Johan ter Bekke, Fool Proof Query Access to Search Engines, *Proceedings Third International Conference on Information Integration and Web-based Applications & Services (IIWAS 2001)*, Linz, Austria, W. Winiwarter, S. Bressan and I. K. Ibrahim (Eds.), Österreichisches Computer Gesellschaft (2001), pp. 389-394.

[7] T. Conally, C. Begg, A. Strachan, Database systems: A practical approach to design, implementation and management, Addison-Wesley, Reading (MA) (1995).

[8] S. Dar, R. Ramakrishnan, A performance study of transitive closure algorithms, ACM SIGMOD Record, 23 (1994), pp. 454-465.

[9] A. Karayiannis and G. Loizou, Cycle detection in critical path networks, Information Processing Letters, 7 (1978), pp. 15-19.

[10] J.C. Molluzzo, A first course in discrete mathematics, Wadsforth, Belmont CA, (1986).

[11] C.H. Papadimitriou, Database Metatheory: Asking the Big Queries, *Proceedings ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, CA, USA (1995), pp. 1-10.

[12] F. Rolland, The essence of databases, Prentice Hall, Hemel Hempstead (1998).

[13] A. Rosenthal, S. Heder, U. Dayal and F. Manola, Traversal Recursion: A Practical Approach to Supporting Recursive Applications, *Proceedings ACM SIGMOD International Conference on Management of Data*, Washington D.C., USA (1986), pp. 166-176.

[14] D. Suciu and J. Paredaens, Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure, *Proceedings ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Minneapolis, Minnesota, USA (1994), pp. 201-209.

[15] J.H. ter Bekke, Semantic Data Modeling, Prentice Hall, Hemel Hempstead (1992).

[16] J.H. ter Bekke, Advantages of a compact semantic meta model, *Proceedings 2nd IEEE Metadata Conference*, Silver Spring, USA (1997), http://www.computer.org/conferen/proceed/meta97/papers/jterbekke/jterbekke.html.

[17] J.H. ter Bekke and J.A. Bakker, Content-driven specifications for recursive project planning applications, *Proceedings International Conference on Applied Informatics (AI 2002)*, Innsbruck, Austria, M.H. Hamza (Ed.), ACTA Press, Calgary (2002), pp. 448-452.

[18] J.H. ter Bekke and J.A. Bakker, Recursive queries in product databases, *Proceedings 5th International Conference on Flexible Query Answering Systems (FQAS 2002)*, Copenhagen, Denmark (2002), Lecture Notes in Computer Science (Subseries LNAI) Volume 2522, T. Andreasen, A. Motro, H. Christiansen and H. Legind Larsen (Eds.), Springer-Verlag, Berlin (2002), pp. 44-55.

[19] J.H. ter Bekke and J.A. Bakker, Fast Recursive Data Processing in Graphs Using Reduction, *Proceedings International Conference on Applied Informatics (AI 2003)*, Innsbruck, Austria, M.H. Hamza (Ed.), ACTA Press, Calgary (2003), pp. 490-494.

[20] J.H ter Bekke and J.A. Bakker, Modeling and Querying Recursive Data Structures I: Introduction, *Proceedings International Conference on Artificial Intelligence and Soft Computing (ASC 2003)*, Banff, Canada, H. Leung (Ed.), ACTA Press, Calgary (2003), pp. 278-282.

[21] J.H ter Bekke and J.A. Bakker, Modeling and Querying Recursive Data Structures II: A Semantic Approach, *Proceedings International Conference on Artificial Intelligence and Soft Computing (ASC 2003)*, Banff, Canada, H. Leung (Ed.), ACTA Press, Calgary (2003), pp. 283-289.

[22] J.D. Ullman and J. Widom, A First Course in Database Systems, Prentice Hall, Hemel Hempstead (1997).