ISO/IEC JTC1/SC21 WG3 DBL MCI-___
X3H2-96-075r1
February 23, 1996
Revised March 6, 1996

I S O
International Organization for Standardization
Organisation Internationale de Normalisation
IEC JTC1/SC21
Information Retrieval, Transfer, and Management for OSI
WG3
Database Languages Rapporteur Group

Source: U.S.A.

Status: SQL3 change proposal

Title: Expressing Recursive Queries in SQL

Authors: Shel J. Finkelstein, Nelson Mattos, Inderpal Mumick, Hamid Pirahesh

Abstract

This paper is a revised version of X3H2-95-457 R1 (DBL LHR-081). Basically, it addresses the issues that were raised during the discussion of X3H2-95-457 and proposes an extended version of the approach described in 95-457 for supporting recursive queries in SQL3. To avoid forcing the reader to look back at the previous papers in this area, this paper integrates most of the text and examples presented in 95-457 (DBL LHR-081), 95-491 (DBL LHR-084), and 95-492 (DBL LHR-086). As we have described in these previous papers, SQL3's recursive union has been (untouched) in the SQL3 document for quite some time. Unfortunately, its specification has a large number of problems. Most of these problems have been caused by the fact that recursive union did not exploit existing SQL constructs to express recursive queries and did not base its definition on the well established concepts of fixpoint semantics and stratification. To address these problems, we are proposing a different and economical way to express recursive queries in SQL. Our approach is exactly the same as the one used in research in the areas of logic programming and deductive databases, i.e., to simply allow a view name or named query expression to be referenced within its own definition. This approach is a lot more powerful than recursive union, requires no additional syntax elements in the language, and does not have any of its problems.

References:

/SQL3/ ISO DBL LHR-004 and ANSI X3H2-95-364, "(*ISO/ANSI Working Draft) Database language SQL3*", Jim Melton (ed), dated October 1995.

/LHR-21/ DBL standing document "Splitting SQL3 into Earlier and Later Progression".

/AhUl79/ "Universality of Data Retrieval Languages", by A.V. Aho, J.D. Ullmann, Proc. 6th ACM Sympos. on Principles of Progr. Languages, San Antonio, Texas, 1979.

/BaRa86/ "An amateur's introduction to recursive query processing", by F. Bancilhon, R.Ramakrishnan, Proc. ACM-SIGMOD Conf., Washington DC, May 1986.

/CGT90/ "Logic Programming and Databases", by S. Ceri, G. Gottlob, L. Tanca, Springer-Verlag, 1990.

/Da87/ "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers", by U. Dayal, VLDB, 1987.

/DaDa92/ "Relational Database Writings 1989-1991", by C.J. Date with H.Darwen, Addison-Wesley, 1992.

/Gal95/ X3H2-95-475, "Bill-of-Materials Example", by Len Gallagher, dated December 1, 1995.

/GaMi78/ “Logic and Databases”, by H. Gallaire, J. Minker (eds.), Plenum Press, 1978.

/GMN81/ “Advances in Database Theory, Vol. I”, by H. Gallaire, J. Minker, J. M. Nicholas, Plenum Press, 1981.

/GMN84a/ “Advances in Database Theory, Vol. II”, by H. Gallaire, J. Minker, J. M. Nicholas, Plenum Press, 1984.

/GMN84b/ “Logic and Databases: A Deductive Approach”, by H. Gallaire, J. Minker, J. M. Nicholas, ACM Computing Surveys, 16(2),1984.

/GaVa89/ “Relational Databases and Knowledge Bases”, by G. Gardarin, P. Valuriez, Addison-Wesley, 1989.

/GW87/ “Optimization of Nested SQL Queries Revisited”, by R. Ganski, H. K. T. Wong, SIGMOD 1987.

/Itoh86/ “Research and Development on Knowledge Base Systems at ICOT”, by H. Itoh, Proc. 12th Intern. Conf. on Very Large Data Bases, Kyoto, 1986.

/KdMS90/ “Making Deductive Database a Practical Technology: a Step Forward”, by G. Kiernan, C. de Maindreville, E. Simon, Proc. SIGMOD 90.

Ki82/ “On Optimizing an SQL-like Nested Query”, by W. Kim, ACM TODS, 1982.

/Ki84/ “SQL-like and quel-like correlation queries with aggregates revisited”, by W. Kiessling, Technical Report UCB/ERL, number = “84/75”, University of California, Berkeley, 1984.

/KRS90/ “Right-, Left- and Multi-Linear Rule Transformations that Maintain Context Information”, by David B. Kemp and Kotagiri Ramamohanarao and Zoltan Somogyi, University of Melbourne, 1990.

/MFPR90a/ “Magic is Relevant”, by I.S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan, Proc. ACM SIGMOD 1990 Int. Conf. on Management of Data, Atlantic City, NJ, May 1990.

/MFPR90b/ “Magic Conditions”, by I.S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan, Proc. ACM PODS 1990, Nashvile, March 1990.

/MPR90/ “The Magic of Duplicates and Aggegrates”, by I.S. Mumick, H. Pirahesh, R. Ramakrishnan, Proc. 16th Conference on Very Large Databases, Brisbane, Australia, 1990.

/MP91/ “Right Linear and Overbound Queries”, by Inderpal Singh Mumick and Hamid Pirahesh, Proc. ACM PODS 1991.

/NRSU89/ “Efficient Evaluation of Right-, Left-, and Multi-Linear Rules”, by Jeffrey F. Naughton and Raghu Ramakrishnan, Yehoshua Sagiv and Jeffrey D. Ullman, Proc. SIGMOD89.

/SPL96/ “Complex Query Decorrelation”, by P. Seshadri and H. Pirahesh and T.Y.C. Leung, Proc. IEEE 12th Int’l Conf. Data Engineering, 1996

/Tar 55/ “A Lattice Theoretical Fixpoint Theorem and its Application”, by Alfred Tarski, in: Pacific Journal of Mathematics, Vol. 5 Issue 2, June 1955, pp. 285-309.

/Ullm85/ “Implementation of Logical Query Languages for Databases”, by J.D. Ullman, ACM Transactions on Database Systems, 10(3), 1985.

1 Introduction

Recursive union is an SQL3 construct that was developed to support the specification of recursive queries in SQL. It has been in the document for quite some time, but, unfortunately, the current SQL3 specification has a large number of severe problems.

This paper proposes a different approach to support recursive queries in SQL3. The proposal is based on the approach presented in X3H2-95-457 (DBL LHR-081), X3H2-95-491 (DBL LHR-084), and X3H2-95-492 (DBL LHR-086), which has been extended to address the concerns raised by some X3H2 members during the discussion of these previous papers.

In large, this proposal integrates the contents of the above 3 papers. Section 2 describes the extensions that were added to address the issues raised at the last X3H2 meeting. Section 3 gives a “tutorial like” introduction on recursive queries. There, we describe the notion of recursion and the basic (but important) concepts related to it: fixpoint semantics, stratification, duplicates and cycles, and forms of recursion. In order to make the reading more “pleasant”, we have included several examples from the area of airline traveling (something that we all should be quite familiar with) to illustrate these concepts. Following, Section 4 gives a description of recursive union and discusses its problems and restrictions. In Section 5, we give an overview of what we are proposing. Finally, Section 6 contains the changes proposed by this paper. This section is followed by an Appendix that contains several queries from the area of Bill of Material (these queries were taken from X3H2-95-491 (DBL LHR-084)).

The paper is heavily based on paper X3H2-95-457 (DBL-LHR-081). We have used change bars to let the reader recognize what have been modified (or added) in this new version of the paper.

Note to the Reader:
We would like to make it clear at this introduction that we are also including as part of our changes the proposal to move non-linear recursion to SQL4. The main reason for that is the fact that non-linear recursion is very rare in real world problems. However, the re-prioritization of non-linear recursion is an orthogonal part of this paper, which can be modified “on the fly” if the committee feels that non-linear recursion should stay in SQL3.

2 Addressing the Issues Raised by ANSI

During the discussion of the several papers on recursion at the last ANSI meeting (i.e., X3H2-95-457 (DBL LHR-081), X3H2-95-491 (DBL LHR-084), X3H2-95-492 (DBL LHR-086), X3H2-95-482 (DBL-083), and X3H2-95-483 (DBL LHR-085)), some concerns have been voiced; most of them address functionality of Recursive Union which was felt to be missing in our proposal, or which was deemed to be provided in a more “user friendly” fashion in the Recursive Union Approach:

1. With the new approach, it is possible that a simple typo may unintentionally cause a query to become recursive (in contrast Recursive Union makes it syntactically explicit that recursion is intended).

2. The new approach offers several inherent ways (e.g., use of UNION rather than UNION ALL) to control unwanted cycles in addition to explicit ways to detect cycle conditions. Nevertheless, acceptance of the new approach could be improved by providing syntactical constructs for detecting cycles (at least in the most simple cases), which are comparable to the Cycle Clause of Recursive Union.

3. Recursive Union provides syntax for tagging the result tuples with information which enables one to subsequently sort the result to reflect depth first or breadth first graph traversal order. Again, this effect can be achieved in the new approach by explicitly maintaining this information; however, there seems to be some demand for syntactic sugar to obtain this effect in an easier fashion.

4. The semantics of Recursive Union is specified in terms of a procedural algorithm, while the new approach is based on fix point semantics. While we are convinced that this approach offers a number of advantages (performance; better integration with existing SQL constructs; compactness; unified treatment of linear and non-linear recursion, direct and mutual recursion; declarativeness; no prejudice for certain implementation policies), we understand why some members of X3H2 would like to have procedural equivalents for the fixpoint based semantics. Although this issue is not concerned with usability, we recognize that it will affect the way the standard is specified.

Below we describe how we propose to address these issues.

Note to the Reader: We assume that the reader is familiar with the approach we are proing. If this is not the case, we recommend the reader to jump to Sections 3 and 4 of this paper and come back to this Section afterwards.

2.1 Addressing unintentional recursion

Consider the following query fragment:

```
WITH B AS
  (SELECT ... FROM A ...
  UNION ALL
  SELECT ... FROM A,V ... )
  select ....
```

By simply mistyping “B” for “V”, this query could unintentionally be turned into a recursive one:

```
WITH B AS
  (SELECT ... FROM A ...
  UNION ALL
  SELECT ... FROM A,B ... )
  select ....
```

As a safeguard against this, a new keyword RECURSIVE is proposed. In the following query,

```
WITH RECURSIVE B AS
  (SELECT ... FROM A ...
  UNION ALL
  SELECT ... FROM A,B ... )
  select ....
```

it is made clear that recursion is really intended. On the contrary, if this keyword is used in a non-recursive context, as in

```
WITH RECURSIVE B AS
  (SELECT ... FROM A ...
  UNION ALL
  SELECT ... FROM A,V ... )
  select ....
```

there are two options: either this is rejected, or the keyword is ignored. We propose the latter approach.

Finally, if the keyword is not used, but the query is recursive, then the query will be rejected, as, for example, in

```
WITH B AS
  (SELECT ... FROM A ...
  UNION ALL
  SELECT ... FROM A,B ... )
  select ....
```

2.2 Supporting a Cycle Clause

Below is an example adapted from X3H2-95-491 (DBL LHR-84), p. 21.

```
with recursive px(major, minor) as
  ( select major, minor from ppa where major = 'P1'
   union all
    select px.major, ppa.minor
    from   px, ppa where ppa.major = px.minor
  )
  select major, minor from px
```

The above query could loop forever if the data has cycles (note that we are using UNION ALL in the query). As shown below, a Cycle Clause can be used to prevent that from happening:

```
with recursive px(major, minor) as
  ( select major, minor
    from   ppa
    where  major = 'P1'
   union all
    select px.major, ppa.minor
    from   px, ppa where ppa.major = px.minor)
   CYCLE MINOR SET CYCLEMARK TO '1' DEFAULT '0' USING PATH
  select major, minor from px
```

Like the Recursive Union Approach, the Cycle Clause of this proposal provides the following information:

- which column of the resulting table is going to record the fact that a cycle has been detected (here, CYCLEMARK).
- which are the values to be used for denoting the cycle detection (here, '1').
- which columns ("navigator columns") are to be used for detecting the cycle condition (here, MINOR).

In addition, the Cycle Clause provides two additional pieces of information:

- the value which goes into the cyclemark column if no cycle has been recorded. In the above example, we have chosen '0'.
- the name of the column of the resulting table (here, PATH) which is to be used to maintain information for detecting a cycle condition. For collecting this information the navigator columns are used to maintain a list of tuples made up of navigator values (in our case, if a tuple of PX is matched by a tuple of PPA, the PATH value of the newly generated tuple is composed of appending ROW (PPA.MINOR)) to PX.PATH.

To capture the semantics of queries like the previous one, we are going to propose Syntax Rules which define equivalent queries which do not contain a Cycle Clause. The equivalent for the previous query is given below. For illustrative purposes the text coming from rewrite is given in capital letters:

```
with recursive px(major, minor, CYCLEMARK, PATH) as
  (  SELECT MAJOR, MINOR, '0', LIST{ROW(MINOR)}
     FROM (select major, minor
           from    ppa where major = 'P1') AS
     CRN1(MAJOR, MINOR)
  union all
     SELECT MAJOR, MINOR,
     CASE WHEN ROW(MINOR) IN (SELECT P.* FROM TABLE (PATH) P)
           THEN '1' ELSE '0' END,
     CONCATENATE(PATH, LIST {ROW(MINOR)})
     FROM (select ppa.major, ppa.minor,PX.CYCLEMARK, PX.PATH
           from    px, ppa
           where   ppa.major = px.minor) AS
     CRN2(MAJOR, MINOR, CYCLEMARK, PATH)
     WHERE CYCLEMARK <> '1'
  )
  select major, minor, cyclemark from px
```

As can be seen, the names of the generated columns (CYCLEMARK, PATH) do not occur in the original <with column list>. This has to be complied when the Cycle Clause is specified.

There is one minor difference between this approach and the approach of <recursive union> in that in <recursive union>, the user is expected to explicitly provide for columns like the cycle mark column. This allows the user for specifying "default values" for the non-cycle case both in the <initial expression> and the <iteration expression>. In doing so, it may happen that a value meant for signalling a cycle condition is also used for initially marking a tuple with the non-cycle condition. In our approach this cannot happen, since the cycle mark column is not available in the query expression on the right hand side of a with list element; instead, Cycle Clause is the only place it is dealt with.

The same approach is taken for the <path column> (in our example: PATH).

2.3 Supporting DEPTH and BREADTH FIRST Search

The approach used to support a search order is essentially the same as the one we used for the Cycle Clause: the DEPTH FIRST and BREADTH FIRST constructs are borrowed from Recursive Union with virtually no changes; using the information provided by the user in the search clause, the query is rewritten into a query without the search clause, thus dealing with the semantics of a search order at the level of Syntax Rules only.

As with our Cycle Clause approach, the proposal, as given below, assumes that users are responsible for providing the names of the columns that are going to contain the BREADTH or DEPTH FIRST ordering information.

2.3.1 BREADTH FIRST

An example of a recursive query with BREADTH FIRST is given below:

```
with recursive px(major, minor) as
  ( select major, minor from ppa where major = 'P1'
   union all
    select ppa.major, ppa.minor
    from   px, ppa where ppa.major = px.minor )
  SEARCH BREADTH FIRST BY MAJOR, MINOR SET ORDERCLMN
  select major, minor from px
  ORDER BY ORDERCLMN
```

Note to the Reader: We are using an ORDER BY clause to illustrate how the resulting table would be sorted. We understand that ORDER BY is only valid at <declare cursor>.

As with Recursive Union, information from MAJOR and MINOR is used to sort new child tuples within their parent tuples. Users need not care about additional information (in effect, the level of recursion) used for that end. The only additional information users need to supply is the name of the column (using the SET ... clause) which is going to receive this information. In our example the name of this column is ORDERCLMN. In addition to the result columns explicitly specified in the <with column list> (here: MAJOR and MINOR) a column with the name specified in the SET clause will be included in the result (here: PX) of the <with list element>.

More details about how this is achieved become apparent after rewrite (again, text generated by the rewrite is given in capital letters):

```
with recursive px(major, minor, ORDERCLMN) as
  ( SELECT MAJOR, MINOR, ROW(0, MAJOR, MINOR)
    FROM (select major, minor
            from ppa where major = 'P1') AS
    CRN1(MAJOR, MINOR)
   union all
    SELECT MAJOR, MINOR, ROW(ORDERCLMN...LEVEL + 1, MAJOR, MINOR)
    FROM (select ppa.major, ppa.minor,
            (SELECT OC.*
             FROM   VALUES PX.ORDERCLMN OC(LEVEL, MAJOR, MINOR))
            from   px, ppa
            where  ppa.major = px.minor)
    CRN2(MAJOR, MINOR, ORDERCLMN)
  )
```

```
select major, minor from PX
order by orderclmn
```

2.3.2 DEPTH FIRST

Different from BREADTH FIRST, a DEPTH FIRST query provides for odering information which enables one to sort a query result in such a way, that for each child tuple within a parent tuple the successor tuples of this child tuple can be ordered before the next sibling tuple and its sucessors. Similar to the BREADTH FIRST example above, the column names MAJOR and MINOR serve the purpose of identifying parent tuples and a search order for the sibling tuples within a parent tuple.

Except for the keyword DEPTH, the next query example is textually equivalent to the previous one:

```
with recursive px(major, minor) as
  (  select major, minor
     from ppa where major = 'P1'
   union all
     select ppa.major, ppa.minor
     from   px, ppa where ppa.major = px.minor)
   SEARCH DEPTH FIRST BY MAJOR, MINOR SET ORDERCLMN
  select * from PX
  order by orderclmn
```

Of course, the different semantics of the DEPTH FIRST query is reflected by the fact that its rewritten format is different from the format of the rewritten BREADTH FIRST query:

```
with recursive px(major, minor, ORDERCLMN) as
  (  SELECT MAJOR, MINOR, LIST{ROW(MAJOR, MINOR)}
     FROM (select major, minor
           from ppa where major = 'P1')
     CRN1(MAJOR, MINOR)
  union all
     SELECT MAJOR, MINOR, CONCATENATE(ORDERCLMN,
                              LIST{ROW( MAJOR,MINOR)})
     FROM (select ppa.major, ppa.minor, PX.ORDERCLMN
           from   px, ppa
           where  ppa.major = px.minor)
     CRN2(MAJOR, MINOR, ORDERCLMN)
  )
  select major, minor from PX
  order by orderclmn
```

2.4 Restrictions on Cycle and Search Order Clauses

It is important to make clear that in order to keep the number of syntax rules small, AND in order to avoid hard problems, we propose to restrict the cycle clause to “simple” recursive queries. That is, we propose to disallow the Cycle and Search Order Clauses for <with list element>s unless they are

- linearly recursive
- not mutually recursive
- simple (i.e., one UNION operator only with operands being <simple table>s). Roughly speaking, a <with list element> is simple if it is of the form (disregarding a Cycle or Search Clause)

  X(...) AS (A UNION ... B)

  such that A does not refer to X, and B is a <query specification> of the form

  SELECT .... FROM X ...

  where X is not referred to anywhere else.

There are several reasons for this proposal:

1. The class of recursive queries carved out by the above restrictions largely covers the same class of linearly recursive queries that is covered by <recursive union>. For this class of queries our proposal provides the same look and feel when it comes to cycle conditions or search order.
2. On the other hand, the specifcation of mutually and non-linearly recursive queries requires more sophisticated users which will not be deterred from explicitly specifying code for detecting cycle conditions or generating ordering information. In these cases, this type of information might be too complex to be mechanically generated in a satisfactorily fashion.

We believe that these restrictions seem reasonable, since they are geared to the class of recursive queries that can be handled by Recursive Union. Users dealing with more complex queries will most likely insist in controlling cycle detection in a more direct fashion without using the above syntactic sugar. Ways for a user to handle cycle detection explicitly are described in Section 3 of this paper.

Please note that the syntactic sugar provided for cycle detection and search order does not have the problems presented by Recursive Union’s cycle clause and search clause (which are described in Section 4 of this paper). As we will described later in this paper, the cycle clause and search clause of Recursive Union are based on a tuple-at-a-time evaluation and rely on the use of sequence numbers. The above approach (although syntactically very close to the search clause and cycle clause of recursive union) does not rely on sequence numbering, but on the set-oriented constructs of SQL. However, both approaches will deliver the same query results (apart from the sequence number).

2.5 Complementary Procedural Specification

As we will show in Section 3 of the paper, there are several algorithms that can be used to evaluate a recursive query in our approach. We will use semi-naive to illustrate one way to evaluate recursive queries. We will also propose the inclusion of the semi-naive algorithm into SQL3.

To explain semi-naive, let us take a look at a simple example that retrieves all cities reachable from Paris:

```
WITH RECURSIVE Reachable_From (Source, Destin) AS
  ( SELECT Source, Destin
    FROM Flights
    WHERE Source = 'Paris'
  UNION
    SELECT in.Source, out.Destin
    FROM Reachable_From in, Flights out
    WHERE in.Destin = out.Source
  )
  SELECT * FROM Reachable_From
```

The simplest[1] implementation of semi-naive would take the above query and "create" pairs of working tables RT and WT for every recursive query specification and every recursive set operator. In the above example, we would need RT1 and WT1 to represent the UNION operator and RT2 and WT2 to represent the recursive query specification of the query expression, i.e.,

```
SELECT in.Source, out.Destin
FROM Reachable_From in, Flights out
WHERE in.Destin = out.Source
```

RT1 and RT2 will be collecting the rows resulting from the several iterations of the recursive query. At each iteration, new tuples will be added to RT1 and RT2. WT1 and WT2 will be used locally to maintain the results of each iteration.

Please note that RT1 will be collecting the tuples resulting from the UNION. Therefore, at the end of the evaluation, RT1 will contain the result of the recursive query.

Given these 4 tables, semi-naive would evaluate the above query in the following way:

---

1 There are possible optimizations to the algorithm given below. Some of the working tables are not needed if their corresponding step does not require elimination of duplicates.

1. *Allocate empty result tables, RT1 and RT2, and empty work tables WT1 and WT2.*

2. *Evaluate the complete query expression once with Reachable_From bound to an empty table and put the rows resulting from the evaluation of the recursive query specification into WT2 and the rows resulting from the evaluation of UNION into WT1.*

   Note that during this first evaluation Reachable_From is empty. Consequently, the recursive part of the query produces no rows (but not always - see Example18 in Section 3 of the paper). Also note that the evaluation of the query expression has to be performed following the semantics of SQL (i.e., UNION will eliminate duplicates during the UNION operation. The same applies for DISTINCT.)

3. *Add the rows of WT1 to RT1 and the rows of WT2 to RT2.*

4. *While both WT1 and WT2 are not empty do*

   (a) *Evaluate the recursive query specification in the query expression (i.e., that referencing Reachable_From) with the recursive table (i.e., Reachable_From) bound to the table WT1 producing a new WT2 and evaluate the recursive set operator, producing a new WT1.*

   Note that the evaluation has to be performed following the semantics of SQL (i.e., DISTINCT will eliminate any duplicates).

   (b) *If the operator of some RTj requires the elimination of duplicates (e.g., UNION in the case of set operators or DISTINCT in the case of query specifications), then*

   (i) *eliminate all duplicates from the corresponding WTj*

   (ii) *eliminate from WTj every row that is already contained in RTj.*

   Note that I am using WTj to mean both WT1 and WT2 and RTj to mean RT1 and RT2.

   (c) *Add the rows of WT2 to RT2 and the rows of WT1 to RT1.*

   End

5. *Set result_table to the contents of RT1.*

Note to the Reader:
RT2 and WT2 are not really needed in this simple example. They have been left in to align this algorithm with the more general version discussed on pages 17 through 20.

Assuming the following contents for the table flights:

FLIGHTS

Source	Destination	Carrier	Cost
Paris	Detroit	KLM	7
Paris	New York	KLM	6
Paris	Boston	AA	8

Source	Destination	Carrier	Cost
New York	Chicago	AA	2
Boston	Chicago	AA	6
Detroit	San Jose	AA	4
Chicago	San Jose	AA	2

semi-naive would evaluate the query in the following way (we will show only the contents of RT1 and WT1 since RT2 and WT2 are not interesting because their query specification does not require elimination of duplicates (i.e., it does not contain DISTINCT):

1. *Allocate empty result tables, RT1 and RT2, and empty work tables WT1 and WT2.*

RT1

Source	Destin

WT1

Source	Destin

2. *Evaluate the complete query expression once with Reachable_From bound to an empty table and put the rows resulting from the evaluation of the recursive query specification into WT2 and the rows resulting from the evaluation of UNION into WT1.*

RT1

Source	Destin

WT1

Source	Destin
Paris	Detroit
Paris	New York

Source	Destin
Paris	Boston

3. *Add the rows of WT1 to RT1 and the rows of WT2 to RT2.*

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston

WT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston

4. *While both WT1 and WT2 are not empty do*

Iteration 1:

(a) *Evaluate the recursive query specification in the query expression (i.e., that referencing Reachable_From) with the recursive table (i.e., Reachable_From) bound to the table WT1 producing a new WT2 and evaluate the recursive set operator, producing a new WT1.*

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston

WT1

Source	Destin
Paris	San Jose

Source	Destin
Paris	Chicago
Paris	Chicago

(b) If the operator of some RTj requires the elimination of duplicates (e.g., UNION in the case of set operators or DISTINCT in the case of query specifications), then

(i) eliminate any duplicates from the corresponding WTj

(ii) eliminate from WTj every row that is already contained in RTj.

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston

WT1

Source	Destin
Paris	San Jose
Paris	Chicago

(c) Add the rows of WT2 to RT2 and the rows of WT1 to RT1.

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin
Paris	San Jose

Source	Destin
Paris	Chicago

Iteration 2:

(a) *Evaluate the recursive query specification in the query expression (i.e., that referencing Reachable_From) with the recursive table (i.e., Reachable_From) bound to the table WT1 producing a new WT2 and evaluate the recursive set operator, producing a new WT1.*

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin
Paris	San Jose

(b) *If the operator of some RTj requires the elimination of duplicates (e.g., UNION in the case of set operators or DISTINCT in the case of query specifications), then*

(i) *eliminate any duplicates from the corresponding WTj*

(ii) *eliminate from WTj every row that is already contained in RTj.*

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin

(c) Add the rows of WT2 to RT2 and the rows of WT1 to RT1.

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin

Iteration 3:

(a) Evaluate the recursive query specification in the query expression (i.e., that referencing Reachable_From) with the recursive table (i.e., Reachable_From) bound to the table WT1 producing a new WT2 and evaluate the recursive set operator, producing a new WT1.

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin

(b) *If the operator of some RTj requires the elimination of duplicates (e.g., UNION in the case of set operators or DISTINCT in the case of query specifications), then*

(i) *eliminate any duplicates from the corresponding WTj*

(ii) *eliminate from WTj every row that is already contained in RTj.*

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin

(c) *Add the rows of WT2 to RT2 and the rows of WT1 to RT1.*

RT1

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

WT1

Source	Destin

End

5. *Set result_table to the contents of RT1.*

Reachable_From

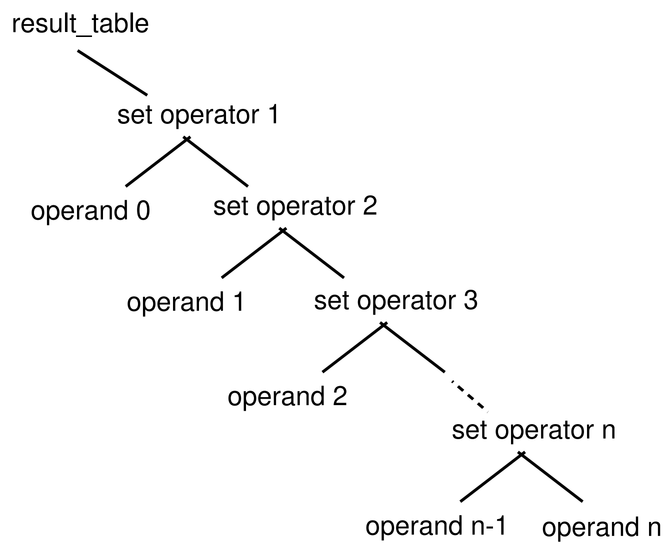
Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	San Jose
Paris	Chicago

Now that we explained a simple query, let us extend the algorithm to handle recursive queries with more than one set operator. Let us assume, for example, the following query

```
WITH RECURSIVE result_table (...) AS
  (
  operand 0
  <set operator 1>
  operand 1
  <set operator 2>
  operand 2
  <set operand 3>
  ...
  operand n-1
  <set operator n>
  operand n
  )
  SELECT ... FROM result_table
```

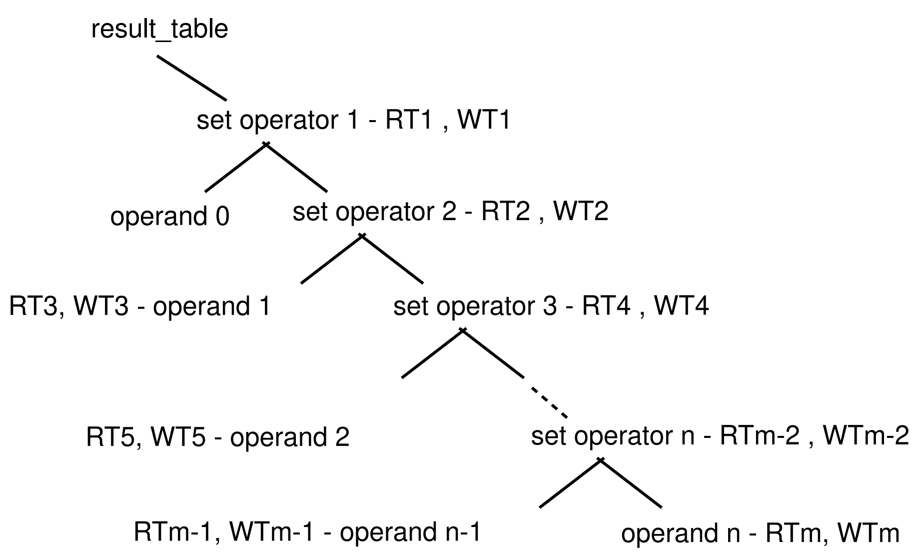
Let us also assume that operand 0 is the non-recursive part of the query.

Queries of this form can be represented graphically as



where the recursive table is an input table of operand 1 to operand n (since operand 1 to operand n are query specifications that reference the result_table).

For a query of this form, semi-naive will need to define several RTs and several WTs; one pair of RT and WT for each recursive query specification (i.e, for each query specification that references result_table) and one pair for each set operator that is recursive (i.e., that has an underlying recursive query specification). Let us call these RTs and WTs, RTj and WTj from j varying from 1 to m where RT1 and WT1 represents the result table. That is, the following RTs and WTs will be used by semi-naive in this example:



Please note that if some set operator does not contain a recursive query specification, then we would not need a corresponding RT and WT. Also note that RT1 will be collecting the rows that will be assigned to result_table at the end of the evaluation. Therefore, in each iteration, the resursive table (i.e., result_table) will be bound to WT1 - the new rows added to RT1).

Following these points, the semi-naive algorithm would be expanded as follows:

1. *Allocate empty result tables, RTj, and empty work tables WTj.*

2. *Evaluate the complete query expression once with the result_table bound to an empty table and put the resulting rows into WT1 to WTn correspondingly.*

   Note that during this first evaluation the result_table i s empty. Consequently, the recursive part of the query is likely to produce no rows (but not always - see Example18 in Section 3 of the paper). Also note that the evaluation of the query expression has to be performed following the semantics of SQL (i.e., UNION will eliminate duplicates during the UNION operation. The same applies for DISTINCT.)

3. *Add the rows of WTj to RTj.*

4. *While some WTj is not empty do*

   (a) *Evaluate every recursive query specification in the query expression (i.e., those that reference result_table) with the recursive table (i.e., result_table) bound to the corresponding table WT1 producing a new WTj and evaluate every recursive set operator producing a new WTj.*

   Note that the evaluation has to be performed following the semantics of SQL (i.e., DISTINCT will eliminate any duplicates).

   (b) *If the operator of RTj requires the elimination of duplicates (e.g., UNION in the case of set operators or DISTINCT in the case of query specifications), then*

   (i) *eliminate any duplicates from WTj*

   (ii) *eliminate from WTj every row that is already contained in RTj.*

   (c) *Add the rows of WTj to RTj.*

   End

5. *Set result_table to the contents of RT1.*

In the above example, all recursive query specifications were referring to result_table. For this reason, in each iteration, the contents of result_table that was used in the recursive query specifications was replaced by the new tuples added to RT1 (which are represented by WT1). If, however, some query specification were refering to another recursive table, then the WT1 for that query specification would be replaced by the new rows added to the RT1 that corresponds to that other recursive table. The following query would be an example:

```
WITH RECURSIVE T1 (...) AS
   (  SELECT ... FROM ...                     operand 0
     UNION                                    set operator 1
       SELECT ... FROM T1, T2 ...             operand 1
     UNION                                    set operator 2
       (WITH RECURSIVE T3 (..) AS
          (  SELECT ... FROM T4               operand 3
            UNION                             set operator 4
              SELECT ... FROM T3, T5          operand 4
            UNION                             set operator 5
              SELECT ... FROM T1, T6          operand 6
          )
       )
   )
```

In the above example, at each iteration, the contents of the recursive table T1 that is used in operand 1 would be bound to the new tuples added to the RT of set operator 1; the contents of the recursive table T3 that is used in operand 4 would be bound to the new tuples added to the RT of set operator 4; the contents of the recursive table T1 that is used in operand 6 would be bound to the new tuples added to the RT of set operator 1.

3 Short Tutorial on Recursion

One of the most important areas of database research in the 1970s and 1980s was that of deductive database systems /GaMi78, GaMi81, GMN84a, GMN84b, Ullm85/ and recursion /Ullm85, BaRa86, GaVa89/.

In this section, we will describe some of the important research results in this area. These results have demonstrated that

1. the semantics of recursive queries is that of fixpoint,
2. some SQL operations are very important in the context of recursive queries and that some others cannot be combined with recursion,
3. data characteristics is an important consideration, and
4. different forms of recursion exist.

3.1 Recursion in deductive database systems

In deductive databases, a distinction is made between extensional and intensional databases. In SQL databases the corresponding features are base tables and views, respectively. It is claimed, however, that the relational view mechanism is inferior to the inferencing rule mechanism of deductive database systems due to the lack of recursion /BaRa86, GaVa89, DaDa92/.

While this observation is valid /AhUl79/, it is based purely on the current syntax of the SQL92 view mechanism. As demonstrated in some research projects (e.g. RDL1 /KdMS90/, Starburst /MFPR90a/, /MFPR90b/, /MPR90/), it has long been recognized that there is nothing that precludes the current view mechanism from being slightly extended to support recursion. The only thing that is needed is the ability to reference the name of the view in the <query expression> of the view. For example, consider the following table describing the flights between different cities:

Table: FLIGHTS

Source	Destination	Carrier	Cost
Paris	Detroit	KLM	7
Paris	New York	KLM	6
Paris	Boston	AA	8
New York	Chicago	AA	2

Source	Destination	Carrier	Cost
Boston	Chicago	AA	6
Detroit	San Jose	AA	4
Chicago	San Jose	AA	2

Based on the above table, a view describing all the cities that can be reached from some other city can be defined as

Example 1:

find all cities that can be reached from some other city

```
CREATE RECURSIVE view Reachable_From (Source, Destin) AS
   SELECT Source, Destin
   FROM Flights
   UNION
   SELECT in.Source, out.Destin
   FROM Reachable_From in, Flights out
   WHERE in.Destin = out.Source
```

As mentioned before, the essential feature in this example is that the name of the view being defined is used on the right hand side of the view definition (in SQL terms within the <query expression> that defines the view).

If one would like to know the cities reachable from Paris, one can use the above view definition to write:

```
SELECT *
   FROM Reachable_From
   WHERE Source = 'Paris'
```

This would return the following answer

Source	Destin
Paris	Detroit
Paris	New York
Paris	Boston
Paris	Chicago
Paris	San Jose

It should be clear that the exploitation of recursion is not restricted to views. In fact, recursion can be expressed in the above form for every query expression (i.e., intermediate table) that has a name. Several syntaxes were developed for this purpose in research, and in SQL3, it can be expressed using the existing <with clause>. That is,

Example 2:

find all cities that can be reached from some other city

```
with RECURSIVE Reachable_From (Source, Destin) AS
   ( SELECT Source, Destin
     FROM Flights
   UNION
     SELECT in.Source, out.Destin
     FROM Reachable_From in, Flights out
     WHERE in.Destin = out.Source
   )
   SELECT * FROM Reachable_From
```

Expressed either as a view or as the above query expression, the meaning of this definition is in simple terms:

> *make Reachable_From such that, when inserted into the query expression defining this named table (i.e., view or table defined with the WITH clause), results in a table which contains the same tuple values as Reachable_From itself.*

Thus, the definition of Reachable_From is rather like a mathematical equation, quite similar to the equation

```
f(x) :: x*x - 2 = x,
```

the solutions of which (x = -1 and x = 2) are called fixpoints of f(x). Similarly, Reachable_From is the fixpoint of the definitions given above.

To put this differently, let HELP (Source, Destin) be a table which is UNION compatible with Reachable_From. Let this table be populated as follows:

```
INSERT INTO HELP (Source, Destin)
 SELECT * FROM Reachable_From
```

Now replace Reachable_From by HELP in the query expression defining Reachable_From. Since the content of the resulting table should be identical to the contents of the table HELP, the query expression

```
TABLE (HELP)
   EXCEPT ALL
   (SELECT Source, Destin
   FROM Flights
```

```
UNION
SELECT in.Source, out.Destin
FROM HELP in, Flights out
WHERE in.Destin = out.Source)
```

should return an empty table. We will refer to queries of this type as fixpoint property tests; clearly, if there is a solution for recursive definitions (this will be addressed in section 3.2) like Reachable_From, it will pass the associated fixpoint test.

3.2 Fixpoint semantics

Fixpoint theory /Tar55/ has been used in computer science as a tool for formally describing the semantics of recursive programs and systems. In particular, denotational semantics are based on fixpoint theory.

Consequently, fixpoint theory has also been used for defining the semantics of recursive constructs in logic programming languages. For an introduction and a literature survey, the reader is referred to /CGT90/.

The use of fixpoint properties for capturing the semantics of recursive tables posed a number of questions that have been worked upon by the DB research community:

1. Does a fixpoint exist which solves the equation defining the view or named query expression definition?
2. Is this solution unique?
3. Are there algorithms to determine the desired fixpoint solution?

It has been shown that fixpoint theory assures that unique solutions really exist under the following:

> *the transformation on the right hand side of a recursive definition must be monotonically increasing.*

That is, adding elements to the input set will not remove or change an element of the result of the transformation.

Fixpoint theory does not only guarantee the existence of unique solutions; rather it also provides algorithms for computing the desired result. One prominent algorithm is Naive Evaluation /BaRa86, GaVa89, DaDa92/. For our recursively defined table Reachable_From, it works like this:

```
begin
  REACHABLE_FROM := FLIGHTS;
  do until REACHABLE_FROM does not change anymore (i.e., fixpoint
```

```
is reached)
REACHABLE_FROM := REACHABLE_FROM UNION (REACHABLE_FROM <join>
FLIGHTS)
end
```

(Note: <join> has been used as a short hand for the join in Example 1, i.e., FROM Reachable_From in, Flights out).

This triggers an interesting question: why not use an algorithmic definition for the semantics of recursive constructs in SQL? As a matter of fact, this approach has been taken by SQL's <recursive union>, using, however, a more elaborate algorithm than Naive Evaluation.

This question deserves several comments:

1. For evaluating recursive queries, numerous algorithms have been developed (for an overview of algorithms and the systems implementing them, the reader is referred to /BaRa86, GaVa89, CGT90/.) Algorithms may be applicable only under certain conditions (e.g., linear vs. nonlinear recursions, see further down), may require a specific structure of the data to be processed (e.g., certain algorithms apply for acyclic data only), or may reflect some particular optimization techniques or make use of data structures being used so that an implementation may have to use several algorithms to cover all cases of recursive queries.
2. If an algorithmic approach were taken, one possible algorithm - let's call it the distinguished algorithm - need to be picked for "effectively" describing the semantics of recursive constructs. Since this algorithm is primarily not intended to be a recipe for implementers, it should be as general and as simple as possible. Implementers may choose alternative algorithms, provided they effectively return the same results as the distinguished algorithm. It is the implementer's privilege to prove this equivalence. This can be a very tedious task, unless the distinguished algorithm and the one used by the implementer both exhibit fixpoint behavior. However, if they do, it is simpler to base the semantic definitions on fixpoints right-away.
3. Most important, the fixpoint approach is a declarative one and it is an extension of the relational calculus; thus it meshes well with the foundation of the Relational Model and of SQL3 (which prefers declarative specifications over computational ones). For a large spectrum of problems, the declarative approach in general and fixpoint based recursive definitions in particular have been proven as powerful mechanisms for mastering complexity.
4. Restrictions on recursive constructs can be motivated as conditions assuring the existence and uniqueness of fixpoints.
5. Finally, conformance tests can be specified and verified more economically when their target constructs are based on easy to understand declaratively defined semantics.

3.3 Stratification (or: Making fixpoint computation work)

3.3.1 Negation

As indicated above, fixpoint requires that the transformations involved in a recursive table definition must be monotonically increasing. In SQL, there are several operations which interfere with this requirement. Negation is an important operation that may conflict with the above requirement. Let's consider an illustrative example; assume a table T(N), where N is a unique column containing, say, integers. Using T, we define a one column view P as follows:

```
CREATE RECURSIVE view P(N) AS
  SELECT N
  FROM T
  WHERE N NOT IN (SELECT N FROM P)
```

This recursive definition involves negation: no element of T must occur in P. In algebraic terms (using " -" to denote set difference) this is captured by the following equation:

```
P = T - P
```

As one can easily see, any solution for P (if there exists one at all) has to meet one of the following conditions:

- P = "empty set"
- P = T
- P proper subset of T
- P subset of T
- P proper superset of T
- P superset of T
- P and T disjoint
- P and T not disjoint (but not in sub/superset relationship either)

Except for T being empty, none of the P's indicated above will solve the equation "P = T - P".

Accordingly, the algorithm we used to compute REACHABLE_FROM will fail here:

```
begin
  P := "empty set"
  do until P does not change anymore
  P := SELECT N FROM T WHERE N NOT IN (SELECT * FROM P)
```

```
end
```

Starting from

```
P0 = "empty set",
```

the first cycle results in

```
P1 = T.
```

Using P1, we obtain

```
P2 = "empty set";
```

that is, by adding new tuples to P (by going from P0 to P1), we are forced to take off tuples (when going to P2). This violation of monotonicity causes the iteration to oscillate for ever between T and "empty set", rather than converging to a final solution (i.e., the fixpoint).

Clearly cases like this one do not have a fixpoint and as such must be prohibited in the language. One may solve this problem by a drastic measure: simply disallow negation, but of course, in SQL such a drastic restriction is not acceptable. Moreover, research in logic programming has found ways to deal with negation. Rather than disallowing negation, it is required only that

> *negation is applied to relations that are completely known (or computed) prior to the application of the negation.*

Thus a recursively defined table (like REACHABLE_FROM) may well be negated, as in

Example 3:

> ***find all cities that can be reached from cities other than Chicago***

```
with RECURSIVE Reachable_From (Source, Destin) AS
  ( SELECT Source, Destin
    FROM Flights
  UNION
    SELECT in.Source, out.Destin
    FROM Reachable_From in, Flights out
    WHERE in.Destin = out.Source
  )
  SELECT * FROM Reachable_From
  EXCEPT
  SELECT * FROM Reachable_From WHERE Source = 'Chicago'
```

or

Example 4:

> ***find all cities that can be reached from cities other than Chicago***

```
with RECURSIVE Reachable_From (Source, Destin) AS
  ( SELECT Source, Destin
   FROM Flights
  UNION
   SELECT in.Source, out.Destin
   FROM Reachable_From in, Flights out
   WHERE in.Destin = out.Source
  )
  SELECT *
  FROM Reachable_From r
  WHERE NOT EXISTS (SELECT *
                    FROM Reachable_From
                    WHERE Source = 'Chicago'
                    AND r.Source = Source
                    AND r.Destin = Destin)
```

In logic programming this type of negation is known as stratified negation; it is supported e.g. in “Datalog with Negations” (see e.g. /GaVa89, CGT90/).

In particular stratified negation is violated if negation is applied to a table which is being defined recursively (this is colloquially alluded to as “negation crosses recursion”). This has been the case with the definition of P(N) above.

3.3.2 Grouping and aggregation

As we have seen, problems are to be expected during computation of recursively defined tables if elements being added to the result affect elements collected so far. In the negation example this had the effect that elements were completely removed. Equally, already adopted elements should not be changed by newly adopted elements. This effect can happen with aggregation. Consider the following view definition:

Example 5:

find the most expensive connections between two cities:

```
CREATE RECURSIVE view Agg_Problem (Source, Destin, Mcost) AS
  SELECT Source, Destin, MAX(Cost)
  FROM FLIGHTS
  GROUP BY Source, Destin
  UNION
  SELECT in.Source, out.Destin, MAX(in.MCost + out.Cost)
  FROM Agg_Problem in, Flights out
  WHERE in.Destin = out.Source
  GROUP BY in.Source, out.Destin
```

In this definition it may happen that a new tuple with a (Source, Destination) pair (e.g. (Paris, San Jose)) is generated which matches an already existing one: in consequence, the existing aggregate value is modified. In fact, this is a very subtle violation of monotonicity. In the negation example removal of tuples from the result has been in conflict with monotonicity; here the transition from result state R1 to result state R2 may simply mean that an aggregation field of a tuple "a" is changed. However, this means that R1 is no longer a subset of R2 (which is part of the semantics of fixpoint).

Stratification also helps in this situation:

> *Aggregation shall be applied only to those tables which are completely known prior to this operation.*

In other words, aggregation shall not cross recursion.

This does not mean that one does not have ways to find out the most expensive connections between two cities. In order to obtain this information, the user should simply move the aggregation "out of the recursion". That is,

```
CREATE view Agg_Problem (Source, Destin, Mcost) AS
  with RECURSIVE Reachable_From (Source, Destin, Cost) AS
  (     SELECT Source, Destin, Cost
        FROM FLIGHTS
     UNION
        SELECT in.Source, out.Destin, in.Cost + out.Cost
        FROM Reachable_From in, Flights out
        WHERE in.Destin = out.Source
  )
  SELECT Source, Destin, MAX (Cost)
  FROM Reachable_From
  GROUP BY in.Source, out.Destin
```

Note that grouping alone may well cross recursion without doing any harm. The reason is that in this case grouping does not violate monotonicity: any newly found tuple matching an established group (representative) will not change it. Thus the previous query would work if restricted to columns Destin and Source, only:

Example 6:

> ***find the connections between two cities that can be reached from each other***

```
CREATE RECURSIVE view Agg_Problem (Source, Destin) AS
  SELECT Source, Destin
  FROM FLIGHTS
  GROUP BY Source, Destin
  UNION
  SELECT in.Source, out.Destin
```

```
FROM Agg_Problem in, Flights out
WHERE in.Destin = out.Source
GROUP BY in.Source, out.Destin
```

Observe that this query is the same as the query in Example 1 with an unnecessary GROUP BY.

3.3.3 Intersect and Except

In this section we show examples of queries using intersect and except in recursive queries.

Take Example 1, but we want to qualify the flights with the secure_con provided by companies' security department. The secure_con table contains connections which are confirmed to fly reasonably away from the war zones (e.g., Bosnia). In addition to that, we want to fly with AA and KLM.

Example 7:

```
with RECURSIVE Reachable_From (Source, Destin) AS
  (     SELECT   Source, Destin
        FROM     Flights
        WHERE    carrier in ('AA', 'KLM')
     UNION ALL
        (
           SELECT in.Source, out.Destin
           FROM   Reachable_From in, Flights out
           WHERE  in.Destin = out.Source
           AND  out.carrier in ('AA', 'KLM')
        INTERSECT
           SELECT Source, Destin FROM secure_con
        )
  )
  select * from Reachable_From;
```

IN the above query, we are using UNION ALL because we do not want to eliminate duplicates. We are interested in all connections between two cities in order to find out which ones go through secure zones only.

Suppose we want to ask the same query except that security department provides insecure_con table which is easier to maintain. The insecure_con table contains connections that fly too close to the war zones (e.g., Bosnia).

Example 8:

```
with RECURSIVE Reachable_From(Source, Destin) AS
  (     SELECT   Source, Destin
        FROM     Flights
        WHERE    carrier in ('AA', 'KLM')
```

```
        UNION ALL
           (
              SELECT in.Source, out.Destin
              FROM   Reachable_From in, Flights out
              WHERE  in.Destin = out.Source
              AND   out.carrier in ('AA', 'KLM')
           EXCEPT
              SELECT Source, Destin FROM insecure_con
           )
     )
     select * from Reachable_From;
```

In Section 3.3.1 “Negation”, we have shown that definitions like:

```
P = T - P,
```

violate fixpoint semantics. Consequently, one would expect problems with

```
CREATE RECURSIVE view P(N) AS
   (TABLE T) INTERSECT (TABLE P)
```

However, as we will show, this is not always a problem. First, we observe that the intersection of P and T equals P if P is either T or the empty set; the latter one should be returned as the (minimal) fixpoint solution. Thus, the recursive definition above should be OK. To convince ourselves that it really is, remember that the intersection of sets can be written as a join (modulo null values). Assuming T to be a set, view P could equally well be specified as

```
CREATE RECURSIVE view P(N) AS
   SELECT T.N
   FROM T, P
   WHERE T.N = P.N
```

Starting with an empty P, our fixpoint algorithm will indeed return an empty set as the expected solution for P.

Thus, intersection is acceptable in recursive definitions if sets are involved, as we illustrated in the previous example with INTERSECT. Unfortunately, this is not the case for multiset. If T in our example is a multiset rather than a set, the simple transformation into a recursive definition involving a join is no longer possible, since it does not guarantee the correct multiplicity of duplicates (which, for a given value, is the minimum of the multiplicity in the operands). While this might not be a problem for the example at hand, a fixpoint cannot be guaranteed any longer in general because a tuple added to the result may have to be removed later on to ensure proper multiplicity.

For this reason INTERSECT ALL is not to be allowed in a recursive definition if the table being defined is one of its operands. Likewise, EXCEPT ALL is not allowed. Further, EXCEPT DISTINCT is allowed as long as the recursion is through the left operand. This is the case because EXCEPT is not symmetric as the right operand is a universal quantifier violating stratification if it is part of recursion.

3.3.4 Outer Join

Suppose we have Table risky_cities, which contains all the cities with insecure airports. This table is kept current based on information provided by US state department. We would like to know for each source destination pair in our travel, how many risky cities we pass through in our connections before making our final reservations.

Example 9:

find all connections between two cities and the number of risky hops

```
with RECURSIVE reachable_from (source, destin, risk_count) AS
  (     select source, destin, 0
        from flights
     union all
        select in.source, out.destin,
  -- Add one to the risk count for this flight.
        case when risky_cities.name is null then 0 else 1 end
        from reachable_from in
           inner join (flights out left outer join risky_cities
                          on (out.source = risky_cities.name))
           on (in.destin = out.source)
  )
  select * from reachable_from
```

We join the table Flights with the table risky cities. Of course not every flight goes through a risky city. Therefore, we have to use an outer join instead of a regular join.

As is well known /DaDa92/ outer joins can be rewritten as inner joins. E.g., the expression “A LEFT JOIN B ON C” (where C is a conditional expression) is equivalent to

```
SELECT A.*, B.*
  FROM A, B
  WHERE C
  UNION ALL
  SELECT A.*, NULL, ...., NULL
  FROM A
  WHERE A.* NOT IN (SELECT A.* FROM A, B WHERE C)
```

This looks as if negation crosses recursion. However, this is not necessarily the case. One needs to make sure only that a tuple, once it has been “cross-multiplied” by NULLs, will never need to replace the NULLs since a matching tuple is found in the subsequent course of evaluation. In other words, in order to avoid nonstratified negation, a recursive definition of a table (say T) should contain a left outer join which has T as one of its operands only if T is the operand on the left. That is, the form of this query is roughly as follows:

```
with RECURSIVE REC(....) AS
  ( SELECT ....
  UNION ALL
  SELECT ...
  FROM ( REC LEFT OUTER JOIN NONREC ... ) ...
```

Note that the operands of LEFT OUTER JOIN must not be exchanged:

```
with RECURSIVE REC(....) AS
  ( SELECT ....
  UNION ALL
  SELECT ...
  FROM ( NONREC LEFT OUTER JOIN REC ... ) ...
```

This is not a matter of changing the meaning of the query. In this case, recursion simply will not work since it violates monotonicity. Assume that REC, at some intermediate state, contains no tuple which joins with some tuple TN of NONREC. Consequently, TN would be appropriately extended by NULL and added to the result as TN'. However, if at some later time a tuple enters REC which is able to join with TN, TN' must be retracted from the REC in favor of TN''. This would violate monotonicity and must therefore be disallowed.

By the same token the following pattern is not allowed:

```
with RECURSIVE REC(....) AS
  ( SELECT ....
  UNION ALL
  SELECT ...
  FROM ( REC RIGHT OUTER JOIN NONREC ... ) ...
```

However, the following is allowed for the reasons given above.

```
with RECURSIVE REC(....) AS
  ( SELECT ....
  UNION ALL
  SELECT ...
  FROM ( NONREC RIGHT OUTER JOIN REC ... ) ...
```

Finally, a full outer join must not cross recursion; that is, in a recursive definition, a reference to the table being defined must neither show up as the right nor the left operand of FULL OUTER JOIN. Thus, both patterns below are illegal:

```
with RECURSIVE REC(....) AS
  ( SELECT ....
  UNION ALL
  SELECT ...
  FROM ( REC FULL OUTER JOIN NONREC ... ) ...
```

```
with RECURSIVE REC(....) AS
  ( SELECT ....
  UNION ALL
  SELECT ...
  FROM ( NONREC FULL OUTER JOIN REC ... ) ...
```

3.4 Duplicate Elimination

Traditionally, deductive database systems implicitly assume set without duplicates. In SQL this would not be acceptable since duplicates have been supported right from the beginning. However, research in this area have shown that recursive evaluation can allow duplicates /MPR90/. This is achieved by appropriately “coloring” duplicate rows; as a consequence, fixpoint theory applies both for tables with and without duplicates. It is important to observe that several fixpoint algorithms are available (e.g., the semi-naive evaluation) which are particularly amenable for proper treatment of duplicates.

Consider, for example, the view REACHABLE_FROM of Example 1. According to this view, Chicago is reachable from Paris. Since this view has been defined with the operator UNION, it does not show that table FLIGHTS provides two ways to get to Chicago. If we wanted to reflect this in the view definition, it should be possible to retain duplicates, as in

Example 10:

find all possible connection between two cities

```
CREATE RECURSIVE view All_Reachable_From(Source, Destin) AS
  SELECT Source, Destin
  FROM Flights
  UNION ALL
  SELECT in.Source, out.Destin
  FROM All_Reachable_From in, Flights out
  WHERE in.Destin = out.Source
```

SQL provides a number of facilities to eliminate duplicates, including UNION and INTERSECT operations or the DISTINCT <set quantifier> in <query specification>s.

As shown by the previous view definition and the one in Example 1, when it is desirable to obtain all connections, one should specify UNION ALL. If duplicates are of no interest (as it has been assumed in Example 1), one specifies UNION instead.

How does the presence of the <set quantifier> DISTINCT (or complementarily ALL) interact with set operators (provided they are allowed) in recursive queries. The answer simply is that there is no difference from queries we have known so far. Consider again Example 1. In case of UNION neither DISTINCT nor ALL will make any difference: the result will not contain duplicates in either case. On the other hand, users may fine tune their results by combining e.g., UNION ALL with DISTINCT as follows:

Example 11:

```
with RECURSIVE TEMP (Source, Destin) AS
  (    SELECT   Source, Destin
       FROM ( TABLE Flights
              UNION ALL VALUES(‘Paris’,’Boston’, ‘KLM’, 7) as q
              (Source, Destin, Carrier, cost)
   UNION ALL
       SELECT DISTINCT in.Source, out.Destin
       FROM    Temp in, Flights out
       WHERE   in.Destin = out.Source
  )
  SELECT * FROM TEMP
```

we would retain duplicates for direct connections, but eliminate them for connections requiring multiple hops.

As we will see, <recursive union> does not allow one to exploit existing query facilities in order to specify data requests with such a precision.

3.5 Different forms of joins

As already said in different words, the typical recursion pattern can be paraphrased as such:

A tuple shall be part of the intended result if

- it is contained in the result of some initialization expression (the “seed”), or
- if it can be derived from some tuple already known.

The latter step is usually performed by a join, e.g. during continuation of known paths in graph traversal.

The join pattern we have seen in most of our examples has been the traditional “SELECT ... FROM A, B ...” pattern. However, nothing prevents one from using one of the other facilities of SQL to perform joins. For instance, Example 2 could have been specified as follows,

Example 12:

find all cities that can be reached from some other city

```
with RECURSIVE
  TFlights (Connect, Destin) AS
     (SELECT DISTINCT Source, Destin FROM FLIGHTS),

  Reachable_From (Source, Connect) AS
     ((TABLE TFlights)
        UNION
     SELECT Source, Reachable_From.Connect
     FROM   Reachable_From      JOIN TFlights ON
        Reachable_From.Connect = TFlights.Connect)

  Select Source, Connect AS Destin FROM Reachable_From
```

or alternatively as:

Example 13:

```
with RECURSIVE
  TFlights (Connect, Destin) AS
     (SELECT DISTINCT Source, Destin FROM FLIGHTS),

  Reachable_From (Source, Connect) AS
     ((TABLE TFlights)
        UNION
     SELECT Source, Reachable_From.Connect
     FROM   (Reachable_From JOIN TFlights USING (Connect)) )

  Select Source, Connect AS Destin FROM Reachable_From
```

or finally as:

Example 14:

```
with RECURSIVE
  TFlights (Connect, Destin) AS
     (SELECT DISTINCT Source, Destin FROM FLIGHTS),

  Reachable_From (Source, Connect) AS
     ((TABLE TFlights)
        UNION
     SELECT Source, Connect
     FROM   (Reachable_From NATURAL JOIN TFlights) )

  Select Source, Connect AS Destin FROM Reachable_From
```

Program generators might benefit from these additional degrees of freedom. When being able to use INNER joins for recursion, is it possible to use OUTER joins as well? The answer is yes (some caution is needed to make recursion work properly, but this has been addressed before). For instance, we could modify each connection which cannot be further extended by marking it accordingly. For this purpose, we slightly modify Example 12 above:

Example 15:

```
with RECURSIVE
  TFlights (Connect, Destin) AS
     (SELECT DISTINCT Source, Destin FROM FLIGHTS),

  Reachable_From (Source, Connect) AS
     ((TABLE TFlights)
        UNION
     SELECT Source,
        COALESCE (Connect, CI||' EST FINIS OMNIUM') AS Connect
     FROM  Reachable_From as rf (source, ci)
        LEFT JOIN TFlights as tf (co, connect) ON ci = co)

  Select Source, Connect AS Destin FROM Reachable_From
```

3.6 Exploitation of other SQL constructs

3.6.1 Choice of column identifiers

When deriving all possible connections from table FLIGHTS, we have in essence proceeded as follows:

> *"If you have a connection from A to B, you obtain a new connection from A to C by extending A to B with a direct flight from B to C."*

Instead of this one might equally well choose a slightly different approach:

> *"If you have a connection from A to B and a connection from B to C, you obtain a new connection from A to C by combining the two matching connections."*

This translates into the following recursive query

Example 16:

> ***find the connections between two cities that can be reached from each other as defined in the flights table in addition to the flight ('Paris','Boston', 'KLM', 7)***

```
with RECURSIVE TEMP (Source, Destin) AS
  (  SELECT  Source, Destin
     FROM (Flights UNION ALL VALUES ('Paris','Boston', 'KLM', 7)
            AS q (Source, Destin)
    UNION
     SELECT in.Source, out.Destin
```

```
      FROM    Temp in, Temp out
      WHERE   in.Destin = out.Source
   )
   SELECT * FROM TEMP
```

By the way: the query above is an example of a non-linear recursive query; however this is not our concern right now but will be addressed separately later on. The point we want to make here is that renaming facilities of SQL3 can be helpful in bringing the query text into a format that is easier to understand. For instance, one could use a temporary identifier "HERE" for the end of the incoming path and the beginning of the outgoing path:

Example 17:

```
with RECURSIVE TEMP (Source, Destin) AS
   (  SELECT   Source, Destin
      FROM    (Flights UNION ALL VALUES ('Paris','Boston', 'KLM', 7)
             AS q (Source, Destin))
    UNION
      SELECT in.Source, out.Destin
      FROM    Temp  in (Source, HERE), Temp out (HERE, Destin
      WHERE   in.HERE = out.HERE
   )
   SELECT * FROM TEMP
```

Those who know logic programming languages such as Prolog or Datalog will be reminded to unification by this technique.

It should be noted that this style relies on standard renaming facilities of SQL. In contrast, <recursive union> enforces different renaming facilities: for the <correlation name list> following RECURSIVE UNION, only one common <recursive column list> is possible (either explicit or implied). This in turn does not give any room for the renaming techniques applied in Example 17.

3.6.2 Harvesting without seeding

Our FLIGHTS examples all exhibit a pattern typical for recursive definitions. From FLIGHTS, we take an initial set of tuples to start with; this set is often referred to as "seed". The seed is used to derive further tuples which are something like a second generation seed and so on.

Though reflecting real life situations, there may be cases in which the initial set of tuples (i.e., the seed) is empty, but the final result of the recursion is not. Let's assume the following query that returns cities reachable from other cities, in this case, Kaiserslautern, by either train or plane.

Example 18:

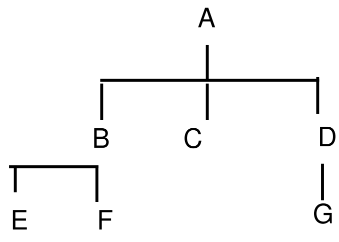
find all cities reachable from Kaiserslautern from either train or plane

```
with RECURSIVE tc (source, destin, carrier) as
  (     SELECT f.source, f.destin, f.carrier
        FROM   flights f
        WHERE  f.source = 'Kaiserslautern'
     UNION ALL
     (
           SELECT r.source, r.destin, 'EuroRail'
           FROM   trains r
           WHERE  r.source = 'Kaiserslautern'
        UNION ALL
           SELECT tcq.source, f.destin, f.carrier
           FROM tc tcq, flights f
           WHERE  tcq.destin=f.source
        UNION ALL
           SELECT tcq.source, r.destin, 'EuroRail'
           FROM tc tcq, trains r
           WHERE  tcq.destin=r.source
     )
  )
  select * from tc;
```

Since Kaiserslautern does not have an airport, it is clear that the first leg of any trip will be done by train. Therefore, the initialization part of the recursive query will be empty, but the final result will certainly not. From Kaiserslautern, one can reach a large number of cities in Europe and in the world.

3.6.3 Depth First And Breadth First data presentation

Often in applications such as bill of material, the result of a graph traversal must be presented in depth first or breadth first. In this section we show examples of such queries. Suppose we have the following graph:



Children of 'A' in breadth first order are: B,C,D,E,F,G.

Children of 'A' in depth first order are: B,E,F,C,D,G

Suppose we have Table PS (parent, child). The following gives the children of A in breadth first

Example 19:

```
with RECURSIVE dt (child, hops) as
  (     select child, 1
        from ps
        where parent= 'A'
     union all
        select ps.child, hops+1
        from dt, ps
        where dt.child = ps.parent
  )
  select child from dt
  order by hops, child;
```

Note that we are not introducing any new SQL constructs. We are exploiting column hops which is naturally part of the query. Hops gives the distance of a node from the root. We simply do an ORDER BY on hops to get the breadth first order. As it will be clear, we will be using hops in several other examples as well.

When written with the “syntactic sugar” described in Section 2.3.1 of this paper, the above query would look like:

```
with RECURSIVE recursive dt (child) as
  (  select child
     from ps
     where parent = 'A'
    union all
     select ps.child
     from   dt, ps
     where dt.child = ps.parent)
    SEARCH BREADTH FIRST BY CHILD SET HOPS
  select child from dt
  ORDER BY hops
```

Note that the column child is not needed in the ORDER BY clause since the hops column that is generated by the syntactic sugar will contain the contents of child as well.

The following gives the children of A in depth first

<u>Example 20:</u>

```
with RECURSIVE dt (child, ak) as
  (     select child, cast (parent||child as varchar(2000))
        from ps where parent= 'A'
     union all
        select ps.child, dt.ak||ps.child
        from dt, ps
        where dt.child = ps.parent
  )
```

```
select child from dt
order by ak, child desc;
```

Column ak for each node gives the concatenation of all the keys from the root up to and including this node. Therefore, a simple ORDER BY will present a query result in DEPTH FIRST order. Certainly, this is just one example of how a query result can be presented in DEPTH FIRST order. SQL already provides other means of producing the same result. A more elegant way of achieving the same result is:

```
with RECURSIVE dt (child, ak) as
  (     select child, list {parent, child}
        from ps where parent= 'A'
     union all
        select ps.child, list_append (dt.ak, ps.child)
        from dt, ps
        where dt.child = ps.parent
  )
  select child from dt
  order by ak
```

Column ak for each node gives a list (as defined in SQL3) of all the keys from the root up to and including this node (the function list_append appends a new element to the list). Therefore, a simple ORDER BY will present a query result in DEPTH FIRST order (remember that SQL3 already specifies how the comparison predicates are appkied to lists).

The above query can also be expressed with the syntatic sugar presented in Section 2.3.2 of this paper:

```
with RECURSIVE recursive dt(child) as
  (  select child
     from ps
     where parent = 'A'
   union all
     select ps.child
     from   dt, ps
     where dt.child = ps.parent)
   SEARCH DEPTH FIRST BY CHILD SET AK
  select child from dt
  ORDER BY AK
```

As it will be clear when we discuss recursive union, an important aspect of the above queries is that they are totally set-oriented, value based, and do not require any new SQL construct to present the result in a specific order.

3.7 Data characteristics and recursion

Recursive table definitions quite often deal with data describing graph structures. The FLIGHTS table used in most of our examples describes a graph, actually a directed acyclic graph. However, in many cases, the tables contain cycles in their graphs and it is important to be able to deal with the existence of such cycles in the data. For such databases, extra steps must be taken to avoid infinite answers to queries (hence making the query safe), and avoid answers that simply ping pong between nodes of the graph.

Let's assume that in our flights table, every flight has a reverse flight. As a result, our database contains a cyclic flight graph. Now we go through several techniques to deal with this issue.

3.7.1 Distinct

Deductive databases predominately rely on distinct to eliminate cycles and ping-pong answers. This is the case for our Example 1. Union distinct eliminates any kind of ping ponging and thus eliminates any infinite loop, making the query safe. Distinct is a simple and powerful construct to handle cycles in large portions of queries.

3.7.2 Bounding the depth of graph traversal

Suppose we are interested in AA flights from Paris and we would like to limit the number of hops to 3.

Example 21:

find all cities reachable from Paris by AA with at most 3 hops

```
with RECURSIVE reachable_from (source, destin, hops) AS
  (  select source, destin, 1
     from flights
     where carrier='AA' and source = 'Paris'
  union all
     select in.source, out.destin, in.hops+1
     from reachable_from in, flights out
     where in.destin = out.source
     and out.carrier = 'AA'
     and in.hops < 3
  )
  select * from reachable_from
```

Obviously, this query is naturally safe. Note that just the use of union distinct instead of union all (i.e., without the additional constraint "in.hops <3") would not make this query safe since the value of hops is increasing as we traverse the flight graph. Therefore, cycles in the flight graph do not create duplicates; hence union distinct cannot be used to stop cycles.

3.7.3 NOT EXISTS Predicate

SQL NOT EXISTS construct is very useful in queries over cyclic graphs. Suppose we are interested in the cheapest flights from Paris to Chicago and from Paris to San Jose.

Example 22:

```
with RECURSIVE Reachable_From (Source, Destin, cost) AS
   (  SELECT   Source, Destin, cost
      FROM     Flights
      WHERE    source = 'Paris'
   UNION ALL
      SELECT in.Source, out.Destin, in.cost+out.cost
      FROM   Reachable_From in, Flights out
      WHERE  in.Destin = out.Source
   )
   SELECT *
   FROM reachable_from q1
   WHERE destin in ('Chicago', 'San Jose')
   AND NOT EXISTS(  SELECT 1
                    FROM reachable_from q2
                    WHERE q2.destin = q1.destin
                    AND q2.cost < q1.cost)
```

The NOT EXISTS predicate specifies that we want the cheapest flights. Observe that this predicate is stratified.

The NOT EXISTS predicate plays a very important role: It makes the query safe and eliminates ping-ponging. Cycles are eliminated since they simply cost more, assuming that cost is never negative.

NOT EXISTS is an elegant and powerful way to handle cyclic data.

3.7.4 Using GROUP BY to eliminate the cycles

Example 23:

find the cheapest flight from Paris to San Jose and from Patir to Chicago

```
with RECURSIVE Reachable_From (Source, Destin, cost) AS
   (
      SELECT   Source, Destin, cost
      FROM     Flights
```

```
   WHERE Source = 'Paris'
UNION ALL
   SELECT in.Source, out.Destin, in.cost+out.cost
   FROM   Reachable_From in, Flights out
   WHERE  in.Destin = out.Source
)
SELECT Source, Destin, min(cost)
FROM reachable_from q1
WHERE Destin in ('Chicago', 'San Jose')
GROUP BY Source, Destin
```

This query can be reformulated with the not exists.

3.7.5 Detecting Cycles

The approaches described above are based on the assumption that users would like to eliminate cycles from the evaluation of their recursive queries. However, there might be also necessary to mark the cycles in the data in addition to (possibly) eliminating them. In section 2.2, we have shown how one can use SQL constructs to mark cycles in the data when writting a recursive query. We have also shown "syntactic sugar" for the definition of queries with such constructs.

3.8 Different forms of recursion

Another important result from the research in deductive databases is a classification of the existing forms of recursion and its relative importance to support real application problems. One can basically define recursive queries under 2 categories:

- directness
  - mutual recursion (or indirect)
  - direct recursion
- linearity
  - linear recursion
  - non-linear recursion

3.8.1 Mutual recursion

In SQL today, when creating a view, say A, it is required that the definition of A only uses base tables, or views which in turn do not rely on A. We extended create view to allow recursive references to the view being defined. This type of recursion is called direct recursion.

Indirect recursion or mutual recursion occurs if for instance a view A depends on a view B which in turn depends on A. This type of recursion is well-known from functions for processing recursive data structures.

For example, lets define two views EVEN(N) and ODD(M) containing even and odd numbers:

Example 24:

```
CREATE RECURSIVE view EVEN(N) AS
   (VALUES(0) UNION SELECT M + 1 FROM ODD)

CREATE RECURSIVE view ODD(M) AS
   (SELECT N + 1 FROM EVEN )
```

These views could be used e.g. in

```
SELECT * FROM EVEN WHERE N < 12
```

which should contain the numbers 0, 2, ..., 10.

Clearly, supporting mutual recursion through views in stand-alone DDL would be a problem as it would be a more drastic extension of the view definition facilities than just direct recursive view definitions. We need to note, however, that it may be possible to support them within schema definitions, but we are not proposing this.

It should be clear that the WITH clause provides a more attractive way to support mutual recursion in SQL today. The above example can be written with the WITH clause as:

Example 25:

```
with RECURSIVE
   EVEN (N) AS
      (VALUES(0) UNION SELECT M + 1 FROM ODD),

   ODD(M) AS
      (SELECT N + 1 FROM EVEN)

   SELECT * FROM EVEN WHERE N < 12
```

Note that mutual recursion is very important to express queries in real world applications. For this reason, it is important that mutual recursion is expressible in recursive queries.

Let's look into a more conventional example. Assume two tables

```
EMP (EmpId, EmpName, DeptId),
DEPT (DeptId, MgrId)
```

The way the tables are defined, one needs to look up the department of an employee to find out the Id of the manager he or she reports to. To extend the reporting line, it would be convenient to have a table BELONGS_TO, which, for a given manager, contains all the departments he or she is responsible for. To derive this table from DEPT, a table REPORTS_TO (which for a given employee, returns all the managers in the reporting line). So lets define these two tables recursively.

Note that we need both of these tables to compute the recursion, even if the final query were not interested in both. Here it goes:

Example 26:

```
with RECURSIVE
  REPORTS_TO (EmpId, EmpName, MgrId) AS
     ( SELECT EmpId, EmpName, MgrId
       FROM   DEPT, EMP
       WHERE  DEPT.DeptId = EMP.DeptId
      UNION
       SELECT e.EmpId, e.EmpName, b.MgrId
       FROM   EMP e, BELONGS_TO b
       WHERE  e.DeptId = b.DeptId
     ),
  BELONGS_TO(DeptId, MgrId) AS
     ( SELECT d.DeptId, r.MgrId
       FROM   DEPT d, REPORTS_TO r
       WHERE d.MgrId = r.EmpId
     )
  SELECT * FROM REPORTS_TO UNION JOIN BELONGS_TO
```

3.8.2 Linear recursion

The examples we have seen so far are all linearly recursive; these examples do have one characteristic in common:

> *In every <joined table> the table being defined recursively is referenced once at most (except for union join).*

In addition linear recursion will not allow that a table being defined recursively is referenced both in the FROM clause of query specification and in a subquery of this very query specification, or more than once in the FROM clause of a queri specification. Similar rules apply for mututally defined recursive tables.

If one of the above conditions is violated, then it is a non-linear recursion (see next subsection). Note that multiple references to a recursive table does not necessarily mean that a recursion is non-linear. For instance, Example 18 is linear even though it has two recursive references to the recursive table tc.

3.8.3 Non-linear recursion

An example of non-linear recursion is given below:

Example 27:

```
CREATE RECURSIVE view Fibonacci (N, F) AS
  VALUES(0,0), (1,1)
  UNION
  SELECT N + 1, p.F + pp.F
  FROM Fibonacci p, Fibonacci pp
  WHERE (p.N - 1) = pp.N
```

This definition is non-linear since “Fibonacci” is joined with itself.

Obviously, “Fibonacci” could equally well be defined in a linear fashion:

Example 28:

```
CREATE RECURSIVE view Fibonacci2(N1, F1, N2, F2) AS
  VALUES(0,0, 1,1)
  UNION
  SELECT N1 + 1, F1 + F2, N1 +2, F1 + 2 * F2
  FROM Fibonacci2 p
```

As described in /RaBa86/ most interesting recursive queries can be specified in a linear fashion. Non-linear recursive queries are rare. (Examples can be found in /MFPR90a/ and /MFPR90b/.) Indeed it is not easy to give examples for recursive queries which cannot be specified in a linear way (and this is the main reason we are proposing to move non-linear recursion to SQL4).

Another important aspect of non-linear recursion is that linear recursion is a special case of non-linear recursion. This is easily demonstrated by the fact that linear recursion restricts the number of joins in which the recursive table is involved whereas non-linear recursion does not. Consequently, a recursive query expressed in linear or non-linear form (as in examples 27 and 28) must always return the same result.

3.8.4 Relationship between linearity and directness

The above two categories are completely independent from each other. A recursive query can be mutually recursive and at the same time be either linear or non-linear. The fact that recursion is expressed directly or indirectly has nothing to do with the number of joins in which the recursive table is involved.

3.9 Conclusion

In this Section, we have described some of the important work from research in databases to support recursive queries:

1. fixpoint

   We have shown that the semantics of recursion is that of fixpoint.

2. stratification

   There are certain SQL operations that violate the principle that recursive tables must be monotonically increasing and, as such, must be prohibited in a recursive query.

3. Duplicates and cycles

   It is important to be able to deal with duplicates and with cycles in the data.

4. Linear and non-linear recursion

   Linear recursion is a special case of non-linear recursion and also the kind of recursion found most commonly in real world applications.

5. Mutual recursion

   Mutual recursive queries (which may be linear or not) are an important form of recursive queries found in real world problems.

Moreover, we have illustrated the use of recursion with several existing SQL constructs, such as, different forms of join, UNION and DISTINCT, NOT EXISTS, etc.

4 SQL3's recursive union

It should be clear to a reader familiar with SQL3's <recursive union> that (unfortunately) SQL3 decided NOT to make use of the results provided by the database research community. Instead, SQL3 invented its own form of expressing recursion. SQL3 did not exploit the minor syntactic extensions described in Section 3 of this paper. Rather than relying on existing SQL syntactic constructs and their semantics, a specialized one, the <recursive union>, has been invented.

As we will see in Section 4.1, <recursive union> has some interesting aspects, but at the same time, has several severe problems:

1. Recursive union does not benefit from fixpoint semantics.
2. The search clause, cycle clause, and the limit clause have several ambiguities and may return non-deterministic query results. Moreover, they force a procedural semantics that will make optimization and parallel execution extremely difficult.
3. Mutual recursion is not permitted, but is very important in practice.
4. The very important controlled duplicate elimination feature of SQL is not extended to recursion.
5. Linear recursion is not a special case of non-linear recursion.
6. Different forms of joins as well as other important SQL constructs cannot be combined with recursive union.

Before we go over these problems in more detail, let us describe SQL3's recursive union.

4.1 An Overview of recursive Union

<recursive union> is a construct of SQL3 to solve transitive closure problems like graph traversal (see the FLIGHTS examples), parts explosion, etc. It maps one or more input tables into a single output table. The construct supports both linear and nonlinear recursion.

For instance, Example 10 would read this way:

Example 29:

```
CREATE view All_Reachable_From(Source, Destin) AS
  (  SELECT Source, Destin
     FROM Flights
    RECURSIVE UNION in (Source, Destin)
     SELECT in.Source, out.Destin
     FROM Flights out
```

```
        WHERE in.Destin = out.Source
    )
```

Most important, <recursive union> is part of <query expression>; thus it can be used at places where other <query expression>s may be used.

4.1.1 Short description of its evaluation

The RECURSIVE UNION operator, which is associated with one (linear recursion) or more (non-linear recursion) correlation variables, connects two query expressions (or input tables), one of which is the initial expression and the other the iteration expression, generating a single output table.

Evaluation is defined in a procedural way. It starts by putting the result of the initial expression on a stack. In case of linear recursion elements are picked from this stack and bound to the correlation variable; then the iteration expression is evaluated with one element bound to the correlation variable. While the resulting new tuples are added to the stack, the tuple that had been picked from the stack is added to the result. Evaluation ends when the intermediate stack gets empty.

The description given above is of course an oversimplification. It does not reflect aspects like

- cycle detection
- maintaining of a tuple count field (sequence number)
- facilities to control the query evaluation

4.1.2 Search strategies in <recursive union>

The <search order> facility of <recursive union> enables one to traverse a graph structure in depth first, breadth first, or an implementation defined order to generate the result table. With table FLIGHTS, e.g., breadth first would first visit the connections from Paris to the directly connecting airports Boston, Detroit etc., before descending to the connections to Chicago and San Jose; with depth first, one path (e.g., Paris -> Boston -> Chicago -> San Jose) would be completed before visiting the next one.

It is important to note that with either option (DEPTH FIRST, BREADTH FIRST, PREORDER) the result of <recursive union> is unordered and always the same since the order of rows in any query result (or <query expression>), except when an ORDER BY clause is specified, is implementation-defined. Therefore, the search order has meaning only within the evaluation of the <recursive union>. For this reason, any desire to enforce the delivery of result tuples, for example, in DEPTH FIRST order, must be achieved by a subsequent query with appropriate ordering specifications.

Whenever a search order is specified, an integer number in ascending order, representing a tuple field (we have also used the term tuple count or sequence number), gets generated during the evaluation of a recursive union. This number is inserted as the value of a column of the resulting recursive table and reflects the order in which each tuple in the result was visited during the recursive evaluation. Therefore, when used in conjunction with other clauses like <limit clause> or <cycle clause>, the search order does influence the results.

In this case, depending on the way a graph is traversed, the same node will be associated with a different sequence number; and if the limit clause limits the query result to a maximum number of output tuples, the subgraph effectively visited depends on the <search order> that has been specified.

4.1.3 <limit clause>

The <limit clause> is used to limit the number of tuples resulting from a recursive union. For example, if one is interested only on 4 cities that are reachable from Paris, one can limit the resulting table as in

```
CREATE view All_Reachable_From(Source, Destin) AS
  (     SELECT Source, Destin
        FROM Flights
     RECURSIVE UNION in (Source, Destin)
        SELECT in.Source, out.Destin
        FROM Flights out
        WHERE in.Destin = out.Source
     RETURN LIMIT (4)
  )
```

The limit clause is beneficial in cases where it is known in advance that a limited number of result tuples is needed; in this case evaluating the complete collection of result tuples would be a needless waste of resources.

4.1.4 <cycle clause>

The <cycle clause>, like the limit clause, is used as a way to stop the recursive evaluation. In this case, however, the evaluation would stop when a cycle is reached. A cycle is defined based on the values of some columns of the tables being processed. That is, given columns A, B, C (specified by the user), the evaluation of the iteration part will not further consider a new tuple T, if the values of A, B, and C are equal to the values of these same columns in a tuple that has been visited previously, and this tuple (either directly or indirectly) caused the generation of T.

4.2 Problems with recursive union

As mentioned previously, <recursive union> is a new operator in SQL that does not rely on existing syntax and semantics to provide recursion. It invented a new syntax plus semantics to several mechanisms that are already part of the SQL language rather than exploiting them. As we will show in this section, the decision:

- made impossible to use recursive union with some other SQL constructs, violating the principle of good language design and orthogonality
- caused the introduction of new mechanisms to handle “things” for which mechanisms already existed in SQL, inserting redundancy in the language
- introduced a procedural evaluation of <query expression>s, compromising important aspects like query optimization and parallelism.

4.2.1 Procedural (tuple-oriented evaluation)

It must be clear that mechanisms like search order and limit clause that are based on the generation of sequence numbers, although advantageous in certain situations, are a drastic deviation from SQL’s approach of providing declarative rather than procedural query facilities. In no other place in SQL (take referential constraints as an example), it is specified in which order the tuples of the input tables for a given operation are processed. However, if one specifies, for example, a DEPTH-FIRST order, the tuples of the input tables must be “visited” in a very specific order.

For example, given the table:

Table DF

PX	PY	
P1	P2	a
P1	P5	b
P2	P3	c
P5	P6	d
P3	P4	e
P6	P7	f

The following recursive union query should return the tuples as traversed in DEPTH FIRST order.[2]

[2] Assuming that the bug (that we will described later) is fixed in the SQL3 document.

Example 30:

```
SELECT 0, T0.PX, T0.PY FROM T0 WHERE PX = 'P1'
  RECURSIVE UNION POP(E_NUMBER, PX,PY)
  SELECT 0, T0.PX, T0.PY FROM T0 WHERE T0.PX = POP.PY
  SEARCH DEPTH FIRST BY PY SET E_NUMBER
  )
```

That is, it would return.

Table result

E_NUMBER	PX	PY	
1	P1	P2	a
2	P2	P3	c
3	P3	P4	e
4	P1	P5	b
5	P5	P6	d
6	P6	P7	f

As described previously, recursion is evaluated (by the large majority of algorithms) by a sequence of joins, and such sequence of joins is likely to return:

Table result

PX	PY	
P1	P2	a
P1	p5	b
P2	P3	c
P5	P6	d
P3	P4	e
P6	P7	f

Consequently, joins could not be used to implement recursive union queries with a given search order because recursive union would force a procedural evaluation so that tuples can be delivered in the specified order.

Declarativeness has been considered one of the most salient virtues of SQL; besides easing the use of query facilities - this is not our concern right now - declarativeness leaves it open to the system to apply the evaluation algorithm which is most appropriate for the problem at hand. Search order undoubtedly restricts this freedom.

The set of potential evaluations strategies for recursive queries is not only restricted by the <search order> facility, but also by the general approach itself. As we have seen in Section 4.1, the evaluation of <recursive union> queries follows a “ tuple oriented join” philosophy. To recall it: for each tuple picked from the intermediate stack, a single join (i.e., with a single tuple), as specified in the <iteration expression>, is performed. In other words, the <iteration expression> is evaluated once for each tuple picked from the stack.

Note that a tuple oriented join MUST be executed because otherwise the tuple field (E_NUMBER in the above result table for the DEPTH FIRST evaluation) and the limit clause cannot be supported in a deterministic way. A set-oriented join would (in each iteration) provide a set of tuples (the union of the set of tuples for each tuple in the input table of the join) whose order will depend on the join algorithm (nested loop, sort merge, hash join, etc) that is being used.

Consequently, a query optimizer cannot choose the best join algorithm for a given recursive query. It is forced to execute it in a tuple-oriented fashion (and either with a DEPTH-FIRST or BREADTH-FIRST search order) in order to be able to generate the tuple field and to apply the limit clause in the resulting table.

As mentioned above, SQL3 is executing the iterative part of the recursive union for each tuple of the recursive union correlation variable. This is particularly undesirable in parallel query processing environments (see problems associated with parallel execution of correlated expressions in Jim Gray’s talk at SIGMOD 95).

An additional problem is caused by the fact that the tuple field, which represents a sequence number, has a dense value (i.e., 1. 2, 3, 4, ..., N) which are generated in order in a tuple-oriented fashion. This makes parallel execution of a recursive union impossible because parallel subtasks independently work on partitions of the data, and as such cannot generate a dense value for the tuple field.

Because the SQL3 sequence numbering is forcing us to traverse the graph in a certain way, it also makes it unnecessarily difficult for an implementation to traverse the graph in some other order. As a result, the SQL3 specification is in conflict with the following popular query optimization mechanism for recursive queries. During optimization, often, it is very important to choose between two alternatives of searching the graph: top down or bottom up. The alternative is chosen based on the predicates specified in the queries over query expressions or views containing recursion.

For example, if one is looking for cities that are reachable from Paris, the implementation is likely to evaluate the query top down since there is a single starting node. On the other hand, if one is looking for all cities from which one can reach San Jose, the implementation is likely to evaluate the query bottom up since, in this case, there is a single ending node. (For further information on this important optimization see /KRS90/, /NRSU89/, and /MP91/.)

In addition to the <search order>, the tuple field, and the tuple-oriented join approach, <recursive union> has another construct, the limit clause, that imposes further restrictions on query optimization. The limit clause directly conflicts with one of the most important optimization techniques for recursion (or for queries in general), i.e., that to push predicates down into recursion (e.g., see /MFPR90a/, /MFPR90b/,/ KRS90/, /MP91/, /NRS89/). Such optimization techniques often result in orders of magnitude in performance improvement. If predicates are pushed down into the recursive evaluation, then the number of tuples generated will be different, causing a different sequence number to be generated and, consequently, making the query produce different results. Again, here, the limit clause is in conflict with the declarativeness of SQL.

Note, however, that limiting the final number of tuples returned by a query (instead of limiting the number of tuples in an intermediate result of the query) does not conflict with the optimization. The reason is that, in this case, the limit is restricting what is shown to the user, not what an optimizer can do to produce the results. For example, if a limit clause appears in a view definition that uses recursive union, then an optimizer will never be able to push predicates down when that same view is used in a SQL query. However, if the top level query over the recursive view specifies a limit clause, then the optimizer can push predicates of the query down into the view. Since we believe that a large number of users will exploit recursion by defining views (as recursive queries are quite difficult to write and understand), predicate push down is a very important optimization technique. Therefore, the limit clause imposes a major problem for the performance of recursion.

We recognize that the need for a facility to deliver only a certain number of tuples in a query result has been voiced several times independently of recursive query facilities. However, if ever added to the SQL language it should be applied to the final query result and not to intermediate results to avoid restricting important optimization techniques.

Considering all the above restrictions on the implementation of recursive union by an SQL engine and remembering that evaluation of recursive queries is, in general, very expensive, we wonder about the performance of recursive queries expressed by recursive union. As we showed, its procedural semantics makes optimization and parallel execution extremely difficult and optimization/parallelization, are critical (especially in this context) to achieve acceptable performance.

4.2.2 No guaranteed support of fixpoint semantics

If restricted to properly stratified recursion a variety of algorithms are available to evaluate recursive queries, guaranteeing query results with fixpoint properties. Therefore, it is important that any SQL extension to provide recursion reflects the semantics of fixpoint and restricts negation to those that are stratified, otherwise very few (if any) efficient algorithms can be used to evaluate recursion in SQL.

Due to the lack of generality in some of the constructs of <recursive union>, certain kinds of non-stratified queries (e.g., ”P = T - P”) cannot be formulated. Unfortunately, however, the remaining non-stratified queries can be written in <recursive union>.

As discussed before, basing the SQL recursion on least fix point allows us to benefit from the wealth of previous work in this area. It also allows SQL to be in par with the state of the art. We will give further examples of where we benefit from the set oriented least fixed point semantics, namely in handling of distinct and mutual recursion. Further we benefit from the wealth of optimization techniques developed for fixpoint, which are crucial for success of support for recursive queries.

4.2.3 Lack of support for mutual recursion

As we have shown in Section 3.8.1, often it is desirable to define two recursive tables using each other. For example, an oddLengthPath may be defined in terms of an evenLengthPath, and at the same time, an evenLenthPath may be defined in terms of an oddLengthPath. Such a definition is intuitive, has a well defined semantics, and can be implemented efficiently.

However, we see no way of writing mutually recursive queries/views in the SQL3 recursive union proposal. We do not see how Examples 25 and 26 can be expressed using recursive union.

This is an example of the case where set oriented least fix point semantics is important. Essentially, if two tables S and R mutually depend on each other, we are solving a set of multi-variable equations.

S= X0;

S= f(R)

R= g(S)

It is not clear how the tuple-at-time/correlation semantics of SQL3’s recursive union can be extended when we have to iterate over multiple tables. Fortunately, least fix point semantics already has addressed this issue. Leading researches and experts in this area who have examined SQL3 have pointed out that lack of acceptance of the established theory of recursion by SQL is quite troublesome, leading to more problems down the road.

4.2.4 Lack of support for both set and multiset semantics

There is no recursive union analog that removes duplicates. However, as already mentioned before, logic database programming has traditionally been defined with sets. Also in SQL, elimination of duplicates is an essential part of the language. We have several constructs to eliminate duplicates producing sets as final or intermediate results of queries: SELECT DISTINCT, UNION, etc. As shown previously, duplicate elimination within the recursive evaluation can be naturally taken care of by union distinct.

In contrast, <recursive union> always follows a multiset approach. This is due to the philosophy that processing goes through iteration cycles to acquire new tuples. Each such tuples is used to fire the iteration query within recursive union. Of course, the iteration query may use the DISTINCT option to suppress duplicates. However, this does not prevent an individual iteration query from adding duplicates to the stack of tuples awaiting their turn to fire new iterative queries. Consequently, <recursive union> cannot produce a recursive table without duplicates[3]. Therefore, the majority of the examples in Section 3 are not expressible with recursive union.

In contrast to this approach, the recursive construct described in Section 3 is open to both set and multisets because it is based on existing SQL constructs that either eliminate duplicates, producing sets, or keep them in the result, producing multisets.

4.2.5 Use of SELECT DISTINCT makes no sense

If DISTINCT is used in a SELECT statement, a SQL engine will do the so-called tuple collapsing. That is, two or more tuples that have the same values are “collapsed” as a single tuple in the result.

A problem with the tuple-at-time/correlation semantics of SQL3 is that it renders the SELECT DISTINCT quite useless. However, SQL’s semantics of SELECT DISTINCT is that the result will not have any duplicates. This semantics is defined irrespective of how the query execution works-- tuple at a time, set a time, etc. The tuple-at-time/correlation semantics of recursive union only guarantees that no duplicates exists within each iteration, but not across iterations. Set oriented least fix point gives the desirable semantics, naturally extending the existing semantics of SQL for non-recursive queries to recursive queries.

The tuple-at-time/correlation semantics also introduces another interesting situation whenever a search order is specified. In this case, a tuple count (or sequencing) is added to every tuple. Since all tuples will have a different sequence number, this number will cause tuples to always be different making the specification of DISTINCT useless.

[3] Unless it happens that no duplicates exist.

4.2.6 Lack of support for Outer join and other forms of join

One of the important contributions of the SQL92 standard was the definition of several forms of joins that can be specified in queries: left, right, and full outer join, inner join, union join, cross join, and natural join. These different forms of join were introduced to attend important user requirements. Unfortunately, none of these joins can be used in conjunction with recursive union. This is certainly a major problem. Again, this problem is caused by the fact that recursion was introduced to SQL by inventing a new construct instead of extending existing ones so that the orthogonality provided by SQL92 did not come for free. Although, as described previously, recursive union makes use of a tuple-oriented join between the tuple represented by the correlation name and the tuples generated by the query expression in the iteration clause, this join operation is defined implicitly. Consequently, there is no language construct that would allow a user to specify which of the several forms of joins should be applied.

Note that recursive union had to make use of an "implicit join" operation because if it were to use an explicit one, it would have to choose one of the following approaches to specify the kind of join operation (neither of which are acceptable, in our opinion):

1. use the existing SQL syntax and semantics for an explicit join and remove the specification of search orders, tuple counts, limit clause, and cycle clause since, as we described previously, cannot be combined with a set-oriented join operation as defined in SQL92.

2. use the existing SQL syntax for an explicit join operation, but define a new, different semantics for this syntax (that would apply when it is used within recursive union), making the same language element "behave" differently depending on the context in which it is used.

3. introduce a new syntax for every form of join defined in SQL92 so that the semantics for this new explicit join operation could be defined independently from the SQL92 joins, introducing even more redundancy to the SQL language.

It should be clear that all the above solutions are unacceptable. However, we strongly believe that equally unacceptable is the restriction that recursive union cannot be combined with the several forms of joins provided by SQL92. Thus, we see this as a major problem with recursive union.

Assuming the Example 9, if one wants to know through how many risky cities his/her next travel arrangements will go through, he/she can write

Example 31:

find all connections between two cities and the number of risky hops

```
with RECURSIVE reachable_from (source, destin, risk_count) AS
  (     select source, destin, 0
        from flights
```

```
          union all
             select in.source, out.destin,
       -- Add one to the risk count for this flight.
             case when risky_cities.name is null then 0 else 1 end
             from reachable_from in
                inner join (flights out left outer join risky_cities
                                on (out.source = risky_cities.name))
                on (in.destin = out.source)
       )
       select * from reachable_from
```

Because the above query cannot use a regular join, it cannot be expressed with recursive union.

4.2.7 Cannot handle recursion with an "empty" initialization

In Section 3.6.2, we have shown that some recursive queries may have an initialization part that is empty and still provide a non-empty result. Example 18 contained a query looking for cities that were reachable from Kaiserslautern either by plane or train. When translated to recursive union this query would look like:

Example 32:

```
( SELECT f.source, f.destin, f.carrier
  FROM   flights f
  WHERE  f.source = 'Kaiserslautern')
RECURSIVE UNION tcq (src, dest, carrier, hops)
(
     SELECT r.source, r.destin, 'EuroRail'
     FROM   trains r
     WHERE  r.source = 'Kaiserslautern'
  UNION ALL
     SELECT tcq.source, f.destin, f.carrier
     FROM flights f
     WHERE  tcq.destin=f.source
  UNION ALL
     SELECT tcq.source, r.destin, 'EuroRail'
     FROM trains r
     WHERE  tcq.destin=r.source
)
```

As we illustrated, the resulting query is not empty because, although Kaiserslautern does not have an airport, the first leg of any trip out of this city can be done by train.

However, the transformation of this query to recursive union has a drastic effect. Its answer is empty! The reason is that the initialization is empty. SQL3 thinks of tcq as correlation variable, and thinks of the iteration like a correlated expression. Since we did not have any instances for tcq from the initialization side, then the iteration part is not executed at all, resulting in an empty set!

This is again a major problem. Users must understand the procedural semantics of recursive union and remember that their initialization part can never be empty. The question is how can they be sure about this?

Fixpoint semantics handles this correctly and uniformly without any exceptions.

4.2.8 Recursion with operations other than UNION

SQL3's recursive union (possibly because of its name) does not allow other SQL constructs, such as intersect (or other set operations), to be used to build recursion.

We are confident that as DBMS products introduce recursion, there will be demand for incorporation of other SQL constructs in recursion, namely other set operations and group by, with well defined semantics. The SQL3 recursive union has no room for such extensions. Several of such extensions have been proposed in /MPR90/ where it introduces group by and aggregation in recursion.

4.2.9 Non-deterministic query results with limit clause

In addition to the problems that the limit clause will create for query optimization and parallelization as described previously, the limit clause is ambiguous in several ways and, as such, may cause recursive union to produce non-deterministic results.

The first problem is that the limit clause works in conjunction with the search order and that the search order is not always required. Once the search order has been specified, it defines the order in which the tuples in the iteration part will be visited and the limit clause defines when a certain number of tuples have been visited so that the evaluation can stop. However, the specification of the search order is optional (which certainly makes a lot of sense as we do not want to force users to always have to specify whether their query should be evaluated in a DEPTH FIRST or BREADTH FIRST manner). It should be obvious that if the search order is not specified, then the answer is ambiguous since the limit may be reached in some arbitrary order.

The second problem is that even when the search order has been specified it may still be ambiguous because the search order cannot guarantee a global ordering for the tuples. By a global ordering, we mean a search order where there is no "ties" between two tuples. As the search order is defined using some columns of the tuple, it is always possible to have two or more tuples with the same values in these columns and in this case the search order is not sufficient to break the "ties" between them (i.e., cannot define the ordering between these tuples). This is especially the case if no unique keys are involved in defining the search order or if the search order refers to only some columns, and other columns are used as part of the predicates in the iteration clause. Whenever the search order does not provide a global ordering, the results are not deterministic. Let's illustrate why this is so. If the search order is partial, when we move from a "parent" tuple to its N "children" (by applying the iteration part to this tuple), a complete order among these children cannot be found. Since the algorithm continues by visiting each one of these children to generate the children of these children, the children of these children are generated in arbitrary order. Since the limit may be reached before all children have been processed, the execution will stop in some arbitrary place, resulting in an arbitrary (or non-deterministic) answer.

As we have described previously, a much better specification is to restrict not the number of rows, but the number of levels visited by the recursive evaluation or to use a clause that limits the number of rows produced by the final query result. If such a clause is used, it should not be part of recursive union, but at the top level of a query, just like ORDER BY. However, we must note that even outside the query such a feature is not unproblematic since it may cause the final result to be non-deterministic, unless if combined with an ORDER BY that produces a global ordering for the tuples in the final result.

4.2.10 Restricted cycle detection

As we described, cycle detection is very important. In Section 3.7 we have shown examples that use NOT EXISTS or non-equality (e.g., LESS THAN) to stop the recursive evaluation when a cycle has been reached.

In recursive union, cycle detection is specified not with existing SQL constructs (like NOT EXISTS or LESS THAN), but with a new construct, the <cycle clause>. It should be clear to any reader familiar with SQL that the use of NOT EXISTS or non-equality is a lot more powerful than the use of the <cycle clause> which is based purely on equality of certain columns.

4.2.11 Non-deterministic query results with cycle clause

In addition of being not as powerful as existing SQL constructs, the cycle clause presents the same problems of producing non-deterministic query results that were described for the limit clause. If the search order is not specified or if the search order is specified, but does not provide a global ordering for the tuples, then the cycle clause is ambiguous and will produce non-deterministic results. In order to eliminate this ambiguity, the union of the columns specified in the cycle clause with the columns specified in the search clause must cover all key columns of the tuple. Since base tables usually have a large number of columns, most of which are not relevant to a recursive query, this is unlikely to be the case.

4.2.12 Linear recursion is not special case of non-linear recursion

As mentioned before, <recursive union> supports both linear and nonlinear recursion. Since linear recursion is a special case of nonlinear recursion, just one set of General Rules would sufficient to capture the semantics of both cases. As a matter of fact this approach has not been taken; instead one set of GRs is provided for the linear case, a second one for the nonlinear case.

This approach is quite risky since it puts on the language specification the burden of proving that the linear rule set is indeed a special case of the nonlinear rule set. We should be reminded that least fix point semantics deals with linear and non-linear recursion in a unified way. For this purpose, let's use table "G0" below for a paper and pencil evaluation, using GR2) of subclause 7.15 <recursive union>.

Table G0 represents a DAG with three nodes (P1, P2, P3); two arcs go from P1 to P2, one from P2 to P3 (rows have been marked for reference purposes).

Table GO

PX	PY	
P1	P2	a
P1	P2	b
P2	P3	c

The following query should return the tuples as traversed in DEPTH FIRST order.

Example 33:

```
SELECT 0, G0.PX, G0.PY FROM G0 WHERE PX = 'P1'
  RECURSIVE UNION POP(E_NUMBER, PX,PY)
  SELECT 0, G0.PX, G0.PY FROM G0 WHERE G0.PX = POP.PY
  SEARCH DEPTH FIRST BY PY SET E_NUMBER
  )
```

This is indeed the case when following GR1); the result is:

E_NUMBER	PX	PY
1	P1	P2
2	P2	P3
3	P1	P2
4	P2	P3

With GR2), however, evaluation proceeds as follows:

1. Obtain initial expression IT as the rows a,b.

2. Put rows a and b on the empty stack S, such that row b is the first one,row a the last one; mark both entries as as R-entries (result rows).

3. Put rows a and b on empty work list WL, with row a first, row b last; mark both entries as NO (not output).

4. Take the last element of stack S (i.e. row a); perform NODE(a) and OUTPUT(a), i.e.:

   find the corresponding entry in WL (assume row a is picked), mark it as OUtput, and enter sequence number 1; enter sequence number 1 also in row a and put this row into overall result; generate a combination row aCR from row a and append it to stack S.

5. In a new iteration cycle, take the last row of S (which was the combination row aCR) and perform COMBINATION(aCR), i.e.: scan WL for an OU entry which has the same (PX, PY) value as aCR (this is row a); copy sequence number (which remains 1); now take the first result constituent of aCR (which is actually aCR itself), find a NO entry in WL with matching (PX, PX) (this is row b), and set the sequence number both in aCR and row b entry to 2; find descendants of aCR (this is row c) and put them in set IT; since IT is not empty, perform OUTPUT(result constituent of aCR). ERROR: a row (2, P1, P2) is added to the result|

We stop the paper and pencil evaluation of GR2) at this point since we have come to an error: sequence number 2 should go to (P2, P3) rather than to (P1, P2) (i.e., the result obtained from GR1)).

We feel very uncomfortable observing that the general rules of recursive union would produce different results for the same query depending on whether it is evaluated as linear or non-linear recursion. Due to the procedural nature of the recursive union specification, we also believe that we will never be 100% sure that GR2) always behaves like GR1 (in case of linear recursion). We would like to emphasize again that such a problem would not occur if recursion was defined using existing SQL constructs that had always relied on a declarative specification of its semantics.

4.2.13 Other problems

4.2.13.1 New column renaming clause

It is not clear to us, why recursive union has also to invent its own column renaming syntax. SQL already has a construct to define column names, as for example in view definitions, and for correlation variables and it is not clear why recursive union did not adopt the same construct instead of inventing a new one.

4.2.13.2 Correlation names with their own column names

In SQL whenever one specifies a correlation name, one can also specify a <derived column list> to define the names of the columns associated with that correlation name. SQL does not restrict, for example, that if two or more correlation names refer to the same table, they must have the same <derived column list>. For example, one is allowed to write

```
select ...
  from T as q1 (a, b, c), T as q2 (x, y, z)
  where ...
```

Unfortunately, the same flexibility is not supported by recursive union. There all correlation names will have the same <recursive column list> so that all column names will be the same for all correlation names. This is just weird. Why were the same features we have for non-recursive correlation names not applied here?

The reason it was specified this way was probably because recursive union does not refer to the recursive table explicitly at the natural place where it is used (i.e., the FROM clause of the iteration part). Remember that, as we described previously, recursion has the semantics of a join, but in recursive union this join operation is defined implicitly. Consequently, one cannot make use of the rich set of SQL constructs that are available for the FROM clause.

In the syntax presented in chapter 2, we are using SQL constructs to express recursion and as such the recursive table can be named in the FROM clause (or anywhere a table can be used). For each reference, column renaming can optionally be provided as business as usual.

4.2.13.3 Strange DEPTH FIRST behavior

Consider the following table DF, representing a tree rooted in P1; the rows have been labelled for reference purposes

Table DF

PX	PY	
P1	P2	a
P1	P4	b
P2	P3	c

The following query should return the tuples as traversed in DEPTH FIRST order.

Example 34:

```
SELECT 0, T0.PX, T0.PY FROM T0 WHERE PX = 'P1'
  RECURSIVE UNION POP(E_NUMBER, PX,PY)
  SELECT 0, T0.PX, T0.PY FROM T0 WHERE T0.PX = POP.PY
  SEARCH DEPTH FIRST BY PY SET E_NUMBER
  )
```

However, instead of the expected result, i.e.

E_NUMBER	PX	PY
1	P1	P2
2	P2	P3
3	P1	P4

The first row (i.e. a) is followed by b, as can be seen from the sequence of stack states given below:

1. push a first, then b:

   b; a

2. pop a, mark it with 1 and put it into result; push child c of a:

   c; b

3. pop b: ERROR, this is NOT the item to be marked as the second one.

As a matter of fact, the result would look like a BREADTH FIRST traversal.

The reason for this, as we assume, is an error in the definition of "pushing" and "popping". In order to be able to reconstruct the sequence of "stack" states described in these papers, we think that we have to make the following changes:

1. The (iteration step) result "IT" is always APPENDED to "stack" S. "IT" has to be sorted as follows before appending:

   (a) PREORDER: implementation defined
   (b) DEPTH FIRST: descending order
   (c) BREADTH FIRST: ascending order

2. Pop the last element from the "stack" S with DEPTH FIRST.

3. Pop the first element from the “stack” S with BREADTH FIRST.

Note to the Reader: This is assumed in the pencil and paper exercises with <recursive union> contained in this proposal.

4.2.13.4 Missing <search clause>

A <search clause> is not a mandatory part of <recursive union>. However, there is no default specified for the case of a missing <search clause>, nor do the GRs prescribe any appropriate action in that case.

4.2.13.5 Minor problems in subclause 7.15 <recursive union>

1. In the definition of “predecessor” (GR1)h)vi)), the last sentence should start as follows: “A row R1 ..”.

2. GR2)h) introduces an OUTPUT-STATUS with two values (both underlined) output and not output. Is there any reason that the first one is italicized (in all occurrences), while the second has only the “not” italicized?

3. In GR2)j)i) “natural search order” and “specified search order” appear (without any obvious reason) in parens.

4. GR2)h) introduces “entry for RR in WL”; however, this notion is repeatedly used with the “in” replaced by “on”, as in GR2)m)i)1)and2), GR2)m)iii)1), GR2)n)vi)

5. In GR2) the iteration result IT is not defined when the WHILE loop GR2)j) is left by performing NODE(R) (i.e., when the WHILE loop is left thru GR2)j)v)1))

4.3 Conclusions

When we started looking at the specification of <recursive union> our original goal was to make sure that the current SQL3 specification was clean and sound and was powerful enough to satisfy the users, particularly the ones that are familiar with state of the art recursion support in logic. Unfortunately, the more we looked into it the more convinced we became that its specification, although providing some interesting features (i.e., search order, limit clause, cycle detection, etc), was a mixed blessing as it was a major deviation from both established concepts of SQL and mainstream logic database programming. As we have demonstrated here, this affects issues like

1. declarativeness

2. intuitivity and language design

3. orthogonality

4. abstraction
5. theoretical foundations
6. query optimization and parallelism
7. design economy

Moreover, as we have shown, we discovered several severe bugs in the specification of recursive union as well as major shortcomings, most of which were caused by the fact that recursion is expressed by a new SQL construct, recursive union, that does not make use of an explicit join between the input tables and the table being recursively defined.

1. It does not benefit from the extensive and established body of work in recursion in research and in industry. This leads to considerable error prone and unneeded reinventions.
2. The semantics defined in SQL3 are slightly to drastically different from the established body of work in this area, which is rooted in the fixpoint semantics and stratification. Thus, employment of crucial established optimization techniques and parallel execution strategies is next to impossible.
3. It syntactically prohibits the use of many valuable SQL constructs in recursion, namely, outer join, natural join, union join, and intersect. We believe that as recursion is increasingly used, there will be demand for supporting other SQL constructs in recursion, namely group by, with well defined semantics. There is no room for such extensions in SQL3's recursive union.
4. The very important controlled duplicate elimination feature of SQL is not extended to recursion.
5. Mutual recursion is missing. Mutual recursion is important in practice. This does not have to be the case, as it is demonstrated in the established least fixed point work where semantics of direct and mutual recursions are unified.
6. A major part of SQL3 rightfully deals with cycles in the data. Unfortunately, this has introduced a dangerous ambiguity in query semantics, leading to ambiguous query results. This is a major drawback and must be dealt with. Again, this does not have to be the case. As we have shown, SQL already has constructs that can be used to solve this problem.
7. SQL3 recursive union rightfully deals with the useful feature of limiting the number of tuples in the answer set. This feature must not be limited to recursive union, as it is very useful for any query. Unfortunately, incorporation of the limit clause in recursive union leads to not well defined semantics and ambiguity in query results. Further, it limits the crucial predicate pushdown optimization. Again, this does not have to be the case. This feature must be moved out of the recursive union to somewhere else in SQL.

8. Linear and non-linear recursion is treated differently. As it is demonstrated in the established least fixed point work, semantics of linear and non-linear recursion can be unified.

9. An empty initialization query will produce an empty result although the result may not be empty

10. Depth first produces a breadth first ordering.

Finally, we were horrified in discovering the restrictions imposed by recursive union in important and well established concepts of query optimization and parallelism, which are crucial to achieve acceptable performance and make this part of SQL3 successful.

We then started to ask ourselves why recursion in SQL was not introduced along the well established body of work in this area: by extending the view mechanism to allow the view name to be referenced within the view definition and by extending the notion of a named query expression to allow the name of the query to be referenced within the query definition. Maybe when recursive union was proposed to SQL, the research results were not yet widely available. Maybe the notion of fixpoint and the importance of stratification were not yet so clear. Maybe ....
Unfortunately, we do not really know since we were not involved in the committee at that time.

In any case, we concluded that there is just one way we could modify the definition of recursive union to solve all the above problems. We have to eliminate the search order, limit clause, and cycle clause and to allow an explicit specification of the recursive table in the FROM clause of the recursive query. The first would eliminate the problems with non-deterministic results, optimization, etc, and the latter would allow us exploit all existing SQL constructs, eliminating the remaining problems. However, such a solution would require the use of a view name or a name for a query expression that is then used respectively within the view or query definition so that the join operation can be expressed explicitly. This would bring us to the same solution that is well established in logic programming and deductive databases and that was illustrated in chapter 3. And this is exactly what is being proposed by this paper.

5 A Different approach

Though quite different in format, logic data programming and relational database systems have much in common. They are both rooted in First Order Logic, and they both share the principle of specifying their search requests in a declarative fashion. These commonalities have been leveraged in numerous logic database programming systems which were on RDBMs to which they were coupled in a more or less tightly manner.

When adding new query facilities to SQL, every effort should be made to further evolve these commonalities. It has been shown in detail that <recursive union> fails to do so. It does not follow well established principles (like fixpoint semantics and stratification). In consequence, it both complicates the specification of its semantics and restricts the possibility of reusing algorithms from logic programming for its implementation. Furthermore, its syntactic format, its procedural approach, and its limitations in expressive power cause severe problems. Therefore, it should not be surprising that recursive union received so much criticism from experts in recursion, query evaluation, and logic programming.

By the same token, an alternative approach should avoid these problems. We believe that the recursive query facilities being proposed in this paper meet these criteria. They are based on the principle described in Section 3, i.e., to allow a view or named query expression to be referenced within its definition. It follows exactly the same semantics given in Section 3. Therefore, it is based on fixpoint semantics and on existing SQL constructs. For this reason, our approach not only requires NO additional format rules to the language, but it also drastically reduces the number of General Rules capturing their semantics. As one may see in the next Section, the main effort will go into some additional Syntax Rules to cover aspects like stratification or linearity (i.e., remember that we are moving non-linear recursion to SQL4).

It is important to note that our approach for supporting recursive queries is not restricted to views. In fact, it can be used with any "named derived table" by referencing the table name within its definition, exploiting the existing SQL3 <WITH CLAUSE> feature.

A remark should be made here: because recursion is not restricted to view definitions, when recursion is needed, one does not necessarily have to pay the costs of a permanent schema object in order to express the recursion.

Most important, <with clause> allows the definition of more than one <query name>; that is, this feature resembles a set of "equations" (rather than a single "equation" in case of view definitions). Thus, it has everything one needs to support mutual recursion, an important requirement of recursive query capabilities, as illustrated in Examples 25 and 26.

In Section 3, we illustrated both possible syntaxes for recursion that we are proposing: within views and by means of the WITH clause. We also described the semantics of these two SQL extensions and showed the necessary restrictions to guarantee stratification. For this reason, we will not repeat these points here. We refer the reader back to Section 3 to review these points. The rest of this Section describes the advantages of this approach and its limitations.

5.1 Advantages of the approach

As already mentioned, <recursive union> has a number of problems which are avoided by our approach. We take a few to contrast our approach with the approach of <recursive union>.

5.1.1 Fixpoint and Stratification

In particular, fixpoint semantics cannot be guaranteed if certain operations do cross recursion. This is avoided if recursive queries are appropriately stratified as described in Section 3.3.

5.1.2 Expressive power

To address this topic, let us pick one example: set versus multiset semantics.

<recursive union> supports multiset semantics only. E.g., Example 29 would return a multiset of connections (e.g. three possible connections from Paris to San Jose). In our approach, this would be expressed as

Example 35:

```
with RECURSIVE Reachable_From (Source, Destin) AS
  (     SELECT Source, Destin
        FROM Flights
      UNION ALL
        SELECT in.Source, out.Destin
        FROM Reachable_From in, Flights out
        WHERE in.Destin = out.Source
  )
  SELECT * FROM Reachable_From
```

By simply omitting “ALL”, one can switch from multiset semantics to set semantics:

Example 36:

```
with RECURSIVE Reachable_From (Source, Destin) AS
  (     SELECT Source, Destin
        FROM Flights
```

```
        UNION
            SELECT in.Source, out.Destin
            FROM Reachable_From in, Flights out
            WHERE in.Destin = out.Source
    )
    SELECT * FROM Reachable_From
```

This fits beautifully into existing SQL3 concepts in a consistent and orthogonal fashion, which is not the case for recursive union.

5.1.3 Mutual recursion

<recursive union> does not support mutual recursion; the only type it supports is called “single rule recursion”. Our approach supports mutual recursion (see Example 25 and 26), thus making another feature of logic programming expressible in SQL.

5.1.4 Support of Outer Join and other forms of join

Because our approach is based on explicit join operations between the input tables and the table being recursively defined, one can make use of any form of joins that exists in SQL today to express the recursive table (except for cases in which the join operation would make the recursive query non-stratified). As we have shown in Section 4, recursive union makes use of an implicit join operation, therefore not being able to be combined with all the several forms of joins that are available in SQL today.

5.1.5 Support for cycle detections and search strategies

Our approach relies on the use of regular SQL constructs to detect cycles and order the query results in depth or breadth first order. Consequently, we do not have the problems presented by Recursive Union that were described in Section 4. On the other hand, as we illustrate in Section 3 of the paper, our approach is extensible to provide syntactic sugar for an easy specification of the cycle clause and search clause.

5.2 “Restrictions”

Though we believe that our approach is superior to <recursive union> since it supports recursion in a way which is consistent with the results that have evolved from research in logic programming, and at the same time offers facilities that are not available in <recursive union>, there are some <recursive union> features which do not have direct counterparts in our approach. These are

1. Sequence number

2. Limiting the number of tuples

As we have made it clear in Section 4, these constructs have a large number of problems and should not, for these reasons, be supported at all. However, one may still believe that they are nice features that should be supported one way or the other. We will try to address this in the sections below.

5.2.1 Sequence number

Recursive Union provides a way to generate a sequence number for recording when a result tuple has been generated. We do NOT want to support a sequence number in the Amended New Approach for the several reasons given previously. Basically, this feature is in conflict with fix point semantics and implementers will be unduly restricted in their options of evaluating recursive queries in an efficient way.

5.2.2 Limit Clause

The <limit clause> of <recursive union> is used to limit the number of tuples produced by the recursive evaluation. It may be used in two situations:

1. when it is known in advance that a query result is not needed in its entirety
2. when the programmer wants to avoid endless loops

Let's take the first case first.

If the associated query defines the content of a cursor, this effect can be easily achieved by issuing the appropriate number of fetches. Of course, this does not help if the query expressions that we want to limit the number of tuples is part of another expression. If this is the case (or if one needs to limit the "size" of the resulting table), one can use the approach described in Section 3.7.2, i.e., that of bounding the size of the resulting table to the depth of the graph traversal. However, it is important to note that this problem is not a unique problem of recursive union. Several SQL operations (e.g., UNION) may produce a large table and, if there is a need to restrict the size of the resulting table, this should be addressed independently of recursion. That is, we need a more general solution than the one which just holds for <recursive union>.

On the other hand, the second aspect, is unique to recursive union: that of possible endless loops. If the limiting feature is meant to help the programmer to avoid recursion with endless loops, there is several easy solutions at hand, and even better, all of them are covered by fixpoint semantics. These solutions were described in Section 3.7. Here, let us just repeat one of them. Rather than limiting the number of tuples, it would be sufficient to limit the number of iterative cycles (i.e., hops) as in:

Example 37:

```
CREATE RECURSIVE view Reachable_From (hops, Source, Destin) AS
    SELECT 1, Source, Destin -- mark seed tuples by zero
    FROM Flights
  UNION ALL
    SELECT hops + 1 AS Itnam, in.Source, out.Destin
    FROM Reachable_From in, Flights out
    WHERE in.Destin = out.Source
    AND hops < 2
```

The effect of this definitions is that no connections are returned which require more than 2 hops between the initial source and the final destination. That is,

Reachable_From		
Itnum	**Source**	**Destin**
1	Paris	Detroit
1	Paris	New York
1	Paris	Boston
2	Paris	Chicago
2	Paris	Chicago
2	Paris	San Jose
1	New York	Chicago
2	New York	San Jose
1	Boston	Chicago
2	Boston	San Jose
1	Detroit	San Jose
1	Chicago	San Jose

It should be noted in passing that this table would pass the fixpoint test.

5.3 Possible Extensions

In this section we give an example of a query which is useful and has an intuitive meaning but is not supported in the SQL3 recursive union and our proposal. We are confident that as recursion is used more widely, users will request support for more of such queries.

Note that our approach syntactically supports such queries, however it disallows them semantically due to the use of set functions (i.e., aggregation as described in Section 3.3) in the HAVING clause of a GROUP BY that is part of recursion. At some later time, one can change the rules to allow such queries, without changing the syntax.

Here is the example that is taken from /MPR90/. We are given Table owns (CM, CS, S) with the interpretation that company CM directly owns S percent of the stock of company CS.

We want to write a query which handles the following case. A company CM is said to have bought another company CS if CM controls more than 50 percent of the stock of CS. CM controls the stock it directly owns. CM also controls stock controlled by any other company CM has bought.

Example 38:

find the companies that have bought other companies directly or indirectly

```
with RECURSIVE

  controls (cm, cs, s) AS
     (     SELECT cm, cs, s
           FROM owns
        UNION
           SELECT bought.cm, controls.cs, controls.s
           FROM bought , controls
           WHERE bought.cs = controls.cm
           AND bought.cm != controls.cm
     ),

  bought (cm, cs) as
     (     SELECT cm, cs
           FROM controls
           GROUP BY cm, cs
           HAVING sum(s) > 50
     )

  SELECT * FROM bought
```

This query contains GROUP BY as part of recursion with set function SUM in the HAVING clause, which is clearly disallowed in our current proposal and in the SQL3 recursive union.

6 Proposed Changes

Note: When proposing changes to existing material, we have used the convention of **highlighting new material** and ~~striking out material that should be deleted~~.

6.1 Changes for the new approach

6.1.1 Subclause 7.14 <query expression>

6.1.1.1 Change The format of <with clause> to:

<with clause> ::= WITH **[RECURSIVE]** <with list>

6.1.1.2 Change The format of <with list element> to:

<with list element> ::=

<query name>

[<left paren> <with column list> <right paren>]

AS <left paren> <query expression> <right paren>

[search clause]

[cycle clause]

6.1.1.3 Add new productions following production of <with list element>

<search clause> ::=

SEARCH <search order> SET <sequence column>

<search order> ::=

DEPTH FIRST BY <sort specification list>

BREADTH FIRST BY <sort specification list>

<sequence column> ::= <column name>

<cycle clause> ::=

CYCLE <cycle column list>

SET <cycle mark column> TO <cycle mark value>

DEFAULT <non-cycle mark value>

USING <path column>

<cycle column list> ::=

<cycle column> { <comma> <cycle column> }...

<cycle column> ::= <column name>

<cycle mark column> ::= <column name>

<path column> ::= <column name>

<cycle mark value> ::= <value expression>

<non-cycle mark value> ::= <value expression>

6.1.1.4 Add new Subrule before SR1 a (“Let n be the number ...”):

a’) If a <with clause> immediately contains RECURSIVE, this <with clause> and its <with list element>s are said to be potentially recursive. Otherwise they are called non-recursive.

6.1.1.5 Change the last sentence of Syntax Rule 1 a (“Let n be the number ...”) as follows:

If WLEi is not potentially recursive, it shall not immediately ...

Note to the Reader: If a <with clause> is not potentially recursive, a <with list element> shall only refer to <query name>s defined by preceeding <with list element>s.

6.1.1.6 Replace SR1)b) (“ For all i ..”) with the following Subrules:

b) If the <with clause> is non-recursive, then for all i between 1 and n, the scope of the <query name> WQN immediately contained in WLEi is the <query expression> immediately contained in every <with list element> WLEk, where k ranges from i+1 to n, and the <query expression body> immediately contained in <query expression>. A <table or query name> contained in this scope that immediately contains WQN is a query name in scope.

b’) If the <with clause> is potentially recursive, then for all i between 1 and n, the scope of the <query name> WQN immediately contained in WLEi is the <query expression> immediately contained in every <with list element> WLEk, where k ranges from 1 to n, and the <query expression body> immediately contained in <query expression>. A <table or query name> contained in this scope that immediately contains WQN is a query name in scope.

6.1.1.7 In SR1)c) (“For every .. WLE, let WQE be the ...”) modify subsubrule i) (“If any two columns ..”) as:

i) If any two columns of WQT have the same name, **or if WLE is potentially recursive,** then WLE shall specify a <with column list>. If WLE specifies a <with column list> WCL, then:

1) ...

2) ...

6.1.1.8 Add the following Subrules after SR1)c) (“For every .. WLE, let WQE be the ...”):

(a) ...

(b) ...

(c) ...

(d) A query name dependency graph QNDG of a potentially recursive <with list> WL is a directed graph such that:

  (i) Each node represents a query name WQNi immediately contained in a <with list element> WLEi of WL.

  (ii) Each arc from a node WQNi to a WQNj represents the fact that WQNj is contained in the <query expression> immediately contained in WLEi. WQNi is said to depend immediately on WQNj.

(e) For a potentially recursive <with list> WL with n elements, and for i ranging from 1 to n, let
WLEi be a <with list element> of WL,
WQNi be the <query name> immediately contained in WLEi,
WQEi be the <query expression> immediately contained in WLEi,
WQTi be the table defined by WQEi,
QNDG be the query name dependency graph of WL.

  (i) WL is said to be recursive if QNDG contains at least one cycle.
  Case:

    (a) If QDNG contains an arc from WQNi to itelf, then WLEi, WQNi, and WQTi are said to be recursive. WQNi is said to belong to the stratum of WQEi.

    (b) If QDNG contains a cycle comprising WQNi1, ..., WQNim with m > 1, then it is said that
    WQNi1, ..., WQNim are recursive and mutually recursive to each other,
    WQTi1, ..., WQTim are recursive and mutually recursive to each other,
    WLEi1, ..., WLEim are recursive and mutually recursive to each other.

    For each k ranging from i1 to im, WQNk belongs to the stratum of WQEi1,..., and WQEim.

    (c) Among the WQEi1, ... WQEim of a given stratum, there shall be at least one <query expression>, say WQEj, such that
    1) WQEj is a <non-join query expression> which immediately contains UNION.
    2) WQEj has one operand which does not contain WQNi1, ..., WQNim. This operand is said to be the non-recursive operand of WQEj.
    3) WQEj is said to be an anchor expression, and WQNj an anchor name.
    4) Let CCCG be the subgraph of QNDG which contains no nodes other than WQNi1, ..., WQNim. For any anchor name WQNj remove the arcs to those query names WQNl which are contained in WQEj. The remaining graph SCCGP shall not contain any cycle.

Note to the Reader: The above rules are primarily intended for mutual recursion. They help to determine the rowtypes of the resulting recursively defined tables. With anchor expressions, the data types of the resulting tables are determined by the non-recursive operands of UNION operations. The types of the remaining tables are determined in a bottom-up fashion (similar to evaluating a set of equations with a triangular coefficient matrix); for instance, the type of B is determined from the (known) type of, say, A, then the type of C is determined from the known types of A and B, and so on. Strictly speaking, the above rules could be simplified by forcing all WQEi to be anchor expressions. This could easily be achieved by transcribing a WQEi into

SELECT ... FROM VALUES (....) .. WHERE 1=0 UNION WQEi

where the left hand side does not add data to the result but has the only purpose to indicate the desired type of the resulting table. However, this would force one to write unnecessarily unelegant recursive queries.

(ii) If WLEi is recursive, then:

(a) If WQEi contains at most one WQNk that belongs to the stratum of WQEi, then WLEi is linearly recursive.

Note to the Reader: This rule ensures that WQNk is only recursively dependent on itself, and there is no join of WQNk with itself involved.

(b) Otherwise, let WQEi contain any two <query names> WQNk and WQNl both of which belong to the stratum of WQEi.
Case:

a) WLEi is linearly recursive if each of the following conditions is satisfied:

i) WQEi does not contain a <table reference list> that contains both WQNk and WQNl.

ii) WQEi does not contain a <joined table> such that TR1 and TR2 are the first and second <table reference>s, respectively, and TR1 and TR2 contain WQNk and WQNl, respectively, except for union join.

Note to the Reader: The 2 rules above make sure that there is no join between WQNk and WQNl involved.

iii) WQEi does not contain a <table expression> that immediately contains a <from clause> that contains WQNk, and immediately contains a <where clause> containing a <subquery> that contains WQNl.

Note to the Reader: To understand the possibility of nonlinearities resulting from this query pattern, remember that certain queries with subqueries in the where clause can be transformed into joins.

b) Otherwise, WLEi is said to be non-linearly recursive.

(iii) For each WLEi, for i ranging from 1 to n, and for each WQN that belongs to the stratum of WQEi:

(a) WQEi shall not contain a <non-join query expression> that contains WQN and immediately contains EXCEPT where the right operand of EXCEPT contains WQN.

Note to the Reader: This is a stratification rule which makes sure that EXCEPT does not cross recursion. For details consult section 3.3.1.

(b) WQEi shall not contain a <routine invocation> with an <argument list> that contains one or more <argument>s that immediately contain a <value expression> that contains WQN.

Note to the Reader: The motivation for this rule can be found e.g. in LHR-086, p.25.

(c) WQEi shall not contain a <table subquery> TSQ that contains WQN, unless TSQ is immediately contained in a <table reference> which
is immediately contained in a <table expression> which

is immediately contained in a <query specification> which
is immediately contained in a <non-join query expression> which
is WQEi.

Note to the Reader: As already outlined above, subqueries in where clauses potentially make recursive queries non-linear. Furthermore this query pattern may be in conflict with stratification, e.g. if universally quantified predicates are involved. This rule is a straightforward approach to prohibit this from happening. In essence it accepts only subqueries in patterns like ... UNION ... SELECT ... FROM (SELECT ... FROM WQN ...) ...

(d) WQEi shall not contain a <query specification> QS such that:

a) QS immediately contains a <table expression> TE that contains WQN and

b) QS immediately contains a <select list> SL or TE immediately contains <having clause> HC and SL or TE contain a <set function specification>.

Note to the Reader: These stratification rules make sure that aggregation does not cross recusion. For detail see section 3.3.2.

(e) WQEi shall not contain a <non-join query term> that contains WQN and immediately contains INTERSECT ALL or EXCEPT ALL.

Note to the Reader: For a motivation of this rule see section 3.3.1 and 3.3.3.

(f) WQEi shall not contain a <qualified join> QJ with:

a) QJ immediately contains a <join type> FULL and a <table reference> that contains WQN.

b) QJ immediately contains a <join type> LEFT and a <table reference> following the <join type> that contains WQN.

c) QJ immediately contains a <join type> RIGHT and a <table reference> preceding the <join type> that contains WQN.

Note to the Reader: These rules have been motivated in section 3.3.4.

SQL3 only (not SQL4)-ANSI and ISO

(iv) If WLEi is recursive, then WLEi shall be linearly recursive.

(v) WLEi is said to be expandable if all of the following is true:

(a) WLEi is recursive.

(b) WLEi is linearly recursive.

(c) WQEi is a <non-join query expression> which immediately contains UNION or UNION ALL. Let QELi and QTRi be the <query expression> and the <query term> immediately contained in WQEi. WQNi shall not be contained in QELi, and QTRi shall be is a <query specification>.

(d) WQNi is not mutually recursive.

Note to the Reader: These restrictions have been addressed in section 2.4. They help in

reducing the number of query patterns that need to be considered during the rewrite process dealing with the Search and Cycle Clauses.

(f) If a <with list element> WLE is not expandable, it shall not immediately contain a <cycle clause> or a <search clause>.

(g) Let WLEC be an expandable <with list element> immediately containing a <cycle clause> or a <search clause>. Let
WQN be the <query name>,
WCL the <with column list>, and
WQE the <query expression> immediately contained in WLEC. Let OP be the set operator immediately contained in WQE, and TLO and TRO be the first and the second operand of OP, respectively.

(a) Let TROSL be the <select list> of TRO. Let WQNTR be the <table reference> immediately contained in the <from clause> immediately contained in the <table expression> TROTE immediately contained in TRO such that WQNTR immediately contains WQN.

Note to the Reader: The definitions above identify portions of a "simple query"(as discussed in section 2.4) which are relevant for the rewrite process detailed below.

Case:

1) If WQNTR immediately contains a <correlation name>, then let WQNCRN be this correlation name.

2) Otherwise, let WQNCRN be WQN.

(b) Case:

1) If WLEC contains a <search clause> SC, let SQC the <sequence column> and SO the <search order> immediately contained in SC. Let SPL be the <sort specification list> immediately contained in SO.

a) SQC shall not be contained in WCL.

b) Every <column name> of SPL shall also be contained in WCL. No <column name> shall be specified more than once in SPL.

c) Case:

i) If SO immediately contains DEPTH, let SCEX1 be WQNCRN.SQC.
Let SCEX2 be
CONCATENATE(SQC, LIST {ROW(SPL)})
Let SCIN be
LIST {ROW(SPL)}

Note to the Reader: SCEX1 and SCEX2 are used to generate sequence column values for tuples generated in the iterative part TRO of the expandable with list element WLEC. SCIN is used to expand the select list of the initial part TLO. It might be helpful to compare these definitions with the examples in section 2.3.2.

ii) If SO immediately contains BREADTH, let SCEX1 be

(SELECT OC.*

FROM VALUES(WQNCRN.SQC)

OC(LEVEL, SPL))

Let SCEX2 be

ROW(SQC...LEVEL + 1, SPL)

Let SCIN be

ROW(0, SPL)

Note to the Reader: SCEX1 and SCEX2 are used to generate sequence column values for tuples generated in the iterative part TRO of the expandable with list element WLEC. SCIN is used to expand the select list of the initial part TLO. It might be helpful to compare these definitions with the examples in section 2.3.1.

2) If WLEC contains a <cycle clause> CC, let

CCL be the <cycle column list>,

CMC the <cycle mark column>,

CMV the <cycle mark value>,

CMD the <non-cycle mark value>, and

CPA the <path column> immediately contained in CC.

a) Every <column name> of CCL shall be contained in WCL. No <column name> shall be specified more than once in CCL.

b) CMC and CPA shall be different from each other and from any <column name> of WCL.

c) The data type of CMV and CMD shall be character string of length 1. CMV and CMD shall be literals and must be different from each other.

d) Let CCEX1 be

WQNCRN.CMC, WQNCRN.CPA

Let CCEX2 be

CASE WHEN ROW(CCL) IN (SELECT P.* FROM TABLE(CPA) P)
THEN CMV ELSE CMD END,

CONCATENATE(CPA, LIST {ROW(CCL)})

Let CCIN be

CMD, LIST {ROW(CCL)}

Let NCCON1 be

CMC <> CMV

Note to the Reader: CCEX1 and CCEX2 are used to generate sequence column values for tuples generated in the iterative part TRO of the expandable with list element WLEC. CCIN is used to expand the select list of the initial part TLO. It might be helpful to compare these definitions with the examples in section 2.2.

3) Case:

a) If WLEC contains only a <search clause>,

let EWCL be

WCL,SQC

Let ETLOSL be

WCL, SCIN

Let ETROSL be

WCL, SCEX2

Let ETROSL1 be

TROSL, SCEX1

Let NCCON be

TRUE

b) If WLEC contains only a <cycle clause>, let EWCL be

WCL, CMC, CPA

Let ETLOSL be

WCL, CCIN

Let ETROSL be

WCL, CCEX2

Let ETROSL1 be

TROSL, CCEX1

Let NCCON be

NCCON1

c) If WLEC contains both a <search clause> and a <cycle clause> CC, then:

i) The <column name>s SQC, CMC and CPA must be different from each other.

ii) Let EWCL be

WCL, SQC, CMC, CPA

Let ETLOSL be

WCL, SCIN, CCIN

Let ETROSL be

WCL, SCEX2, CCEX2

Let ETROSL1 be

TROSL, SCEX1, CCEX1

Let NCCON be

NCCON1

(c) WLEC is equivalent to the expanded with list element

```
WQN(EWCL) AS
(SELECT ETLOSL FROM (TLO) TLOCRN(WCL)
        OP
SELECT ETROSL
FROM (SELECT ETROSL1 TROTE) TROCRN(EWCL)
WHERE NCCON
)
```

Note to the Reader: ETLOSL, ETROSL and ETROSL1 are select lists. TROTE is a table expression of the form “FROM ..., ... WQNCRN WHERE ..”. TLOCRN and TROCRN are correlation names.

6.1.1.9 Replace Syntax rule 16 (“Case:”), Subrule b (“Otherwise”), item ii (“The data type and null class of the i-th column...”) with:

16) Case:

a) If a ..

b) Otherwise,

i) Case:

1) ...

2) ...

ii) **If TR is not an anchor expression, then the** data type and null class of the i-th column of TR is determined by applying the Syntax Rules of Subclause 9.3, “Set operation result data types and nullabilities”, to the data types of the i-th column of T1 and the i-th column of T2. Case:

1) ...

2) ...

iii) If TR is an anchor expression, then:

1) If the i-th column of TR is not recursively referred to, then the data type and null class of the i-th column of TR is determined by applying Subclause 9.3, “Set operation result data types and nullabilities”, to the data types of the i-th column of T1 and the i-th column of T2. If the i-th columns of either T1 or T2 are known not nullable, then the i-th column of TR is known not nullable; otherwise, the i-th column of TR is possibly nullable.

2) If the i-th column of TR is recursively referred to then
a) The i-th column of TR is possibly nullable.
b) Either T1 or T2 shall not be part of recursion.
c) CASE:

1) If T1 is not part of this recursion, then if the i-th column of TR is recursively referred to then the data type of the i-th column of TR is the same as the data type of the i-th column of T1.

2) If T2 is not part of this recursion, then if the i-th column of TR is recursively referred to then the data type of the i-th column of TR is the same as the data type of the i-th column of T2.

6.1.1.10 Change GR1) (“If <with clause> ..”) as follows:

1) If a **non-recursive** <with clause> is specified, then:

a) ...

b) ...

Add a new GR1') between GR1) ("If ...") and GR2) ("Case: ...") as follows:

1') If a potentially recursive <with clause> WC is specified, then:

(a) Let n be the number of <with list element>s WLEi of the <with list> WL immediately contained in WC. For i ranging from 1 to n, let
WQNi and WQEi be the <query name>s
and the <query expression>s immediately contained in WLEi, and WQTi be the table resulting from the evalutation of WLEi.
Let WLPj be the elements of a partitioning of WL such that each WLPj contains all WLEi which belong to one stratum, and let m be the number of partitions. Each partition of WL shall be disjoint from any other partition of WL.
Let the partition dependency graph PDG of WL be a directed graph such that:
   (i) Each node of PDG represents a partition WLPj of WL.
   (ii) Each arc from a node WLPj to a node WLPk represents the fact that WLPj contains at least one WLEi in WLPj and at least one WLEh in WLPk such WQEi contains the <query name> WQNh.

(b) While the set of nodes of PDG is not empty do:
   (i) Evaluate the partitions of PDG which have no outgoing arc.

Note to the Reader: The evaluation of a partition is defined in subrules d) and e).

   (ii) Remove the partitions and their incoming arcs from PDG.

(c) Let LIP be some linear partition of WL. Let m be the number of <with list element>s in LIP, and for i ranging from 1 to m, let WLEi be a <with list element> of LIP, WQNi and WQEi the <query name> and <query expression> immediately contained in WLEi.
   (i) Any <query expression> contained in any WQEi which contains a recursive query name in scope shall be marked as recursive.
   (ii) Any <query expression> AQE contained in any WQEi shall be associated with a specific intermediate result table RT and a working table WT. The types of RT and WT shall be the types of AQE. If AQE is immediately contained in a recursive <with list element> AWLE, then RT and WT associated to AQE are also associated to the recursive <query name> AWQN immediately contained in AWLE.

Note to the Reader: The above comment on AWLE are in particular true for the WLEi of the linear partition LIP. It also covers the case the a WLEi contains a query expression with a with clause.

   (iii) If a <query expression> AQE not marked as recursive is immediately contained in a <non-join query expression> which is marked as recursive and which specifies UNION, then AQE is marked as iteration ignorable.

Note to the Reader: Iteration ignorable tables will be treated as empty tables after the initialization steps which follow below.

   (iv) Allocate empty tables for the RTs and WTs of all <query expression>s AQE in LIP. If an AQE is not marked as recursive, evaluate that AQE and put the resulting rows into the associated RT and WT.
   (v) Bind empty table to all recursive query names in scope. Evaluate all WQEi. For every <query expression> contained in any WQEi, put the resulting rows into the associated WT and RT.
   (vi) For every <query expression> AQE of LIP which is marked as iteration ignorable, remove the tupels from the associated RT and bind AQE to RT. For every other <query expression> AQE which is not marked as recursive bind AQE to RT.

Note to the Reader: This ends the initialization phase.

(vii) While some WT of LIP is not empty do:

(a) Bind every recursive query name in scope to its associated WT.

(b) If WQEi is lineraly recursive, then evaluate WQEi with the associated WT. Otherwise, WQEi is the result of a UNION ALL between the result of the evaluation of WQEi with the associated WT and the following:
(1) Let k be the number of recursive query names RQN in scope.
(2) For p from 1 to k, evaluate WQEi with RQNp bound to its associated WTp and all the other query names RQNq (for q from 1 to p and from p+1 to k) bound to the result of RTq EXCEPT ALL WTq.

(c) For every <query expression> AQE contained in WQEi which is marked as recursive replace the content of the associated WT with the rows resulting from AQE. If this AQE is supposed to produce no duplicates then
(1) Eliminate the duplicates from this WT.
(2) Eliminate from WT all rows which are contained in the RT associated to AQE.

(d) Add the rows of this WT to this RT.

(viii)Any reference to WQNi shall identify the table RT associated to WQNi.

6.1.2 Subclause 11.26 <view definition>

6.1.2.1 Change the format rule of <view definition> to:

<view definition> ::=

CREATE **[RECURSIVE]** VIEW <table name> [<left paren> <view column list> <right paren>]

AS <query expression>

[WITH [<levels clause>] CHECK OPTION]

6.1.2.2 Delete "the viewed table defined by <view definition>" from SR4 ("No <table reference> generally contained ...").

6.1.2.3 Add the following Syntax Rules after new SR 4.

1. If a <table reference> generally contained in the <query expression> identifies the viewed table defined by <view definition>, then this view definition is said to be recursive.

2. If <view definition> is recursive, then

(a) <view column list> shall be specified.

(b) RECURSIVE shall be specified.

(c) The <view definition> is equivalent to
CREATE VIEW <table name> AS
WITH RECURSIVE <table name> (<column name list>) AS (<query expression>)
[WITH [<levels clause>] CHECK OPTION]

6.2 Move Non-linear recursion to SQL4

6.2.1 Mark non-linear recursion in /LHR-21/ as SQL4.

6.3 Removal of recursive union

6.3.1 In “Foreword” (ANSI ONLY part), in the itemized list “Significant new features are:”, change item 5) (“A recursive union...”) as follows:

5) Support for recursive specification of derived tables.

6.3.2 Delete Subclause “4.22.11 Functional dependencies in a <recursive union>”

6.3.3 In Subclause “5.4 Names and identifiers”, change SR 21) (“An <identifier> that is a <correlation name> is .. “) as follows:

21) An <identifier> that is a <correlation name> is associated with ... <query specification> (see Subclause 6.7, “<table reference>”), ~~or is a <query term> that contains a <recursive union> (see Subclause 7.15, “<recursive union>”)~~, or is a <trigger definition> (see Subclause 11.45, “<trigger definition>”)....

6.3.4 In Subclause “6.3 <item reference>”, change SR5)a)ii) (“If IR is contained in a <table expression> TE ... <recursive union> ...”) as follows:

ii) If IR is contained in a <table expression> TE or a <quantified predicate> TE and the scope of IQ is some <SQL procedure statement>,~~ <recursive union>~~, <trigger definition>, <quantified predicate>, or <table reference> that contains TE, then ...

6.3.5 In subclause “6.4 <row reference>”, change SR2) (“ If Q is contained in a ..”) as follows:

2) If Q is contained in a <table expression> TE or a <quantified predicate> TE and the scope clause of Q is some <SQL procedure statement>, ~~<recursive union>,~~ <trigger definition>, <quantified predicate>, or <table reference> that contains TE, ... outer reference to the table associated with Q.

6.3.6 Subclause “7.14 <query expression>”

6.3.6.1 Delete “<recursive union> from the format rule of “<non-join query term>”.

6.3.6.2 In SR14) (“Case: “) remove subrule b) (“If the <non-join query term> is a <recursive union>, then ...”).

6.3.6.3 In SR24) (“The row type RT of a <non-join query term> NJQT ..”), remove subrule b (“If NJQT immediately contains <recursive union> RU, then the row type RT of NJQT is that of RU.”).

6.3.6.4 In the Levelling Rules, replace LR 2a) (“A <non-join query term> shall contain no <recursive union>.”) by:

a) A <query expression> shall not specify a <with clause>.

6.3.7 Delete Subclause 7.15 <recursive union>.

6.3.8 In Subclause “18.2.35 ENABLED_APPLICABLE_ROLES view”, change the view definition to the following:

```
CREATE RECURSIVE VIEW ENABLED_APPLICABLE_ROLES ( ROLE_NAME ) AS
        SELECT ENABLED_ROLE
        FROM USER_SESSION_ENABLED_ROLE
    UNION ALL
        SELECT RAD.ROLE_NAME
        FROM ROLE_AUTHORIZATION_DESCRIPTORS RAD,
             ENABLED_APPLICABLE_ROLES R
        WHERE RAD.GRANTEE = R.ROLE_NAME
```

6.3.9 In Subclause “18.2.36 ROLE_TABLE_GRANTS view”, change the view definition to the following:

```
CREATE VIEW ROLE_TABLE_GRANTS AS
    SELECT GRANTOR, GRANTEE, TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
        PRIVILEGE_TYPE
    FROM DEFINITION_SCHEMA.TABLE_PRIVILEGES
    WHERE GRANTEE IN
        (WITH RECURSIVE R (ROLE_NAME) AS
            (       SELECT ENABLED_ROLE
                    FROM    USER_SESSION_ENABLED_ ROLE
              UNION ALL
                    SELECT RAD.ROLE_NAME
                    FROM DEFINITION_SCHEMA.ROLE_AUTHORIZATION_DESCRIPTORS RAD, R
                    WHERE RAD.GRANTEE = R.ROLE_NAME
            )
        SELECT * FROM R
        )
    AND TABLE_CATALOG = ( SELECT CATALOG_NAME
                          FROM    INFORMATION_SCHEMA_CATALOG_NAME )
```

6.3.10 In Subclause “18.2.37 ROLE_COLUMN_GRANTS view”, change view definition to the following:

```
CREATE  VIEW ROLE_COLUMN_GRANTS AS
    SELECT GRANTOR, GRANTEE, TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
            COLUMN_NAME, PRIVILEGE_TYPE
    FROM   DEFINITION_SCHEMA.COLUMN_PRIVILEGES
    WHERE GRANTEE IN
        (WITH RECURSIVE R (ROLE_NAME) AS
            (       SELECT ENABLED_ROLE
                    FROM USER_SESSION_ENABLED_ROLE
              UNION ALL
                    SELECT RAD.ROLE_NAME
                    FROM DEFINITION_SCHEMA.ROLE_AUTHORIZATION_DESCRIPTORS RAD, R
                    WHERE RAD.GRANTEE = R.ROLE_NAME
            )
        SELECT * FROM R
        )
```

```
AND TABLE_CATALOG = (SELECT CATALOG_NAME
                       FROM INFORMATION_SCHEMA_CATALOG_NAME)
```

6.3.11 In Subclause “18.2.38 APPLICABLE_ROLES view”, change the view definition to the following:

```
CREATE RECURSIVE VIEW APPLICABLE_ROLES(GRANTEE, ROLE_NAME, IS_GRANTABLE) AS
  (   SELECT GRANTEE, ROLE_NAME, IS_GRANTABLE
      FROM DEFINITION_SCHEMA.ROLE_AUTHORIZATION_DESCRIPTORS
      WHERE GRANTEE IN ( USER, 'PUBLIC' )
  UNION
      SELECT RAD.GRANTEE, RAD.ROLE_NAME, RAD.IS_GRANTABLE
      FROM DEFINITION_SCHEMA.ROLE_AUTHORIZATION_DESCRIPTORS RAD,
                APPLICABLE_ROLES R
      WHERE RAD.GRANTEE = R.ROLE_NAME )
```

Note to the Reader: In the current specification, the SELECT list following “RECURSIVE UNION R” seems to need a third element. This has been fixed in the above query.

6.3.12 Subclause “A.2 Full SQL Specifications”

6.3.12.1 Replace item 22 (“Subclause 7.14, <query expression>” with:

22) Subclause 7.14, “<query expression>”:

a) A <query expression> shall not specify a <with clause>.

6.3.12.2 Remove the following item 23 (“Subclause 7.15, “<recursive union>”).

6.3.13 In “Implementation defined elements”, remove the following items 66) and 67):

66) Subclause 7.15, “<recursive union>”: The implicit limit value in a <limit clause> is an implementation-defined positive integer.

67) Subclause 7.15, “<recursive union>”: The order of rows returned when PREORDER is specified is implementation-defined.

7 Appendix

In this section, we would like to illustrate how the approach for recursion in SQL can be easily used to express important queries from the area of bill of material. We will pursue this by addressing the queries described in X3H2-95-482 followed by the query described in X3H2-95-475. After these examples, the section will conclude with some performance considerations.

8 Queries from X3H2-95-482

In this section, we will show how the queries described in X3H2-95-482 can be expressed with the approach described in 95-457.

For each query, we will show the original query as presented in 95-482 and then describe how it can be expressed in our approach. We will also show the query results to avoid the reader having to flip-flop between multiple papers.

Like in 95-482, we will be using an ORDER BY clause although we realize that it should be part of a cursor definition that uses the given queries.

The following tables are taken from 95-482 and they are used in the queries.

Part-Part		
MAJOR	**MINOR**	**QTY**
P1	P2	2
P1	P3	3
P1	P4	2
P2	P3	3
P2	P5	2
P3	P5	4
P3	P6	2
P4	P3	5
P4	P6	2
P7	P8	3
P7	P9	2

PARTMASTER			
PNO	PNAME	LEVELCODE	ISPHANTOM
P1	MAIN	root	no
P2	RACK		no
P3	FITTING		yes
P4	COVER		yes
P5	BRACE	leaf	no
P6	ANGLE	leaf	no

8.1 Parts Explosion, depth-first (Section 3.2 of 95-482)

Recursive Union

```
SELECT *
   FROM (SELECT 0, 0, MAJOR, MINOR FROM PP WHERE PP.MAJOR=P1
      RECURSIVE UNION PX(NODE, PARENT, MAJOR, MINOR)
           SELECT 0, PX.NODE, MAJOR, MINOR
           FROM PP WHERE PP.MAJOR=PX.MINOR
      SEARCH DEPTH FIRST BY MINOR SET NODE)
   ORDER BY NODE;
```

New Approach

```
with RECURSIVE px (major, minor, ak) as
   (    select major, minor, list {major, minor} from pp where pp.major = 'P1'
      union all
        select pp.major, pp.minor, list_append (ak, pp.minor)
        from px, pp where pp.major = px.minor
   )
   select major, minor from px
   order by ak
```

Result

MAJOR	MINOR
P1	P2
P2	P3
P3	P5

MAJOR	MINOR
P3	P6
P2	P5
P1	P3
P3	P5
P3	P6
P1	P4
P4	P3
P3	P5
P3	P6
P4	P6

Note that the above result is the same as the one given in Section 3.2.2 of 95-482 except for the columns node and parent. As we show in the appendix of this paper (with copies of reports from an existing product for bill of material), these columns are meaningless to the user. They are a "side effect" of recursive union that should not be in the final result. However, this is not to say that our approach cannot generate them. As we have shown in other examples there are several ways to produce this kind of information if really required.

8.2 Parts Explosion Breadth-first (Section 3.3 of 95-482)

Recursive Union

```
SELECT *
   FROM (SELECT 0, 0, MAJOR, MINOR FROM PP WHERE PP.MAJOR=P1
      RECURSIVE UNION PX(NODE, PARENT, MAJOR, MINOR)
          SELECT 0, PX.NODE, MAJOR, MINOR
          FROM PP WHERE PP.MAJOR=PX.MINOR)
      SEARCH BREADTH FIRST BY MINOR SET NODE);
```

New Approach

```
with RECURSIVE px (major, minor, level) as
   (   select major, minor, 1 from PP where pp.major = 'P1'
      union
       select pp.major, pp.minor, level+1
       from px, pp where pp.major = px.minor
   )
   select major, minor, level from px
```

```
order by level, major, minor;
```

Result

MAJOR	MINOR	LEVEL
P1	P2	1
P1	P3	1
P1	P4	1
P2	P3	2
P2	P5	2
P3	P5	2
P3	P6	2
P4	P3	2
P4	P6	2
P3	P5	3
P3	P6	3

In this example, we intentionally showed the depth level of nodes of the graph. (The use of this column becomes clear shortly.)

The result is the same the one in Section 3.3.1 of 95-482, except that at the depth level 3, we show p3 and its children only once, where as recursive union shows it twice. The reason is that recursive union cannot deal with duplicates. p3 at depth level 3 is reachable from two parents-- P2 and P4. Therefore, the graph is not a tree and recursive union would need additional mechanisms to eliminate the duplicates (rather than just using union distinct). Our proposal can handle both.

The following version of this query keeps the duplicates, producing the same result as recursive union.

```
with RECURSIVE px (major, minor, level, unique) as
   (   select major, minor, 1, unigen() from PP where pp.major = 'P1'
      union all
        select pp.major, pp.minor, level+1, unigen()
        from px, pp where pp.major = px.minor
   )
   select major, minor, level from px
   order by level, major, unique, minor;
```

Result

MAJOR	MINOR	LEVEL
P1	P2	1
P1	P3	1
P1	P4	1
P2	P3	2
P2	P5	2
P3	P5	2
P3	P6	2
P4	P3	2
P4	P6	2
P3	P5	3
P3	P6	3
P3	P5	3
P3	P6	3

The function unique returns an arbitrary unique value. We simply attach a unique value to each node so that we can distinguish the duplicates at the same depth level. Note that there is no requirement for unique to be a dense sequential value so that none of the problems with recursive union's sequence number (or tuple count) that we described in Section 3.2 of 95-457 are problems here. Note also that the use of a function is just one of the several possible ways to address this.

8.3 Parts Explosion, PRE-ORDER (Section 3.4 of 95-482)

SQL3 recursive union defines PRE-ORDER as implementation dependent. It states that the implementation will substitute preorder with either depth-first or breadth-first (or some other implementation dependent ordering). We presented examples of both of these presentation orders in the previous sections. For this reason, we do not see the need to repeat them here.

8.4 Depth first explosion with level numbers (Section 3.5.1 of 95-482)

Recursive Union

```
SELECT *
   FROM (SELECT 0, 0, 1, MAJOR, MINOR FROM PP W HERE PP.MAJOR=P1
        RECURSIVE UNION PX(NODE, PARENT, LEVEL, MAJOR, MINOR)
           SELECT 0, PX.NODE, PX.LEVEL+1, MAJOR, MINOR
           FROM PP WHERE PP.MAJOR=PX.MINOR)
        SEARCH DEPTH FIRST BY MINOR SET NODE)
   ORDER BY NODE;
```

New Approach

```
with RECURSIVE px (major, minor, level, ak) as
  (    select major, minor, 1, list{major,minor} from PP where pp.major = 'P1'
     union all
       select pp.major, pp.minor, px.level+1, list_append (ak, pp.minor)
       from px, pp where pp.major = px.minor
  )
  select level, major, minor from px
  order by ak
```

Result

LEVEL	MAJOR	MINOR
1	P1	P2
2	P2	P3
3	P3	P5
3	P3	P6
2	P2	P5
1	P1	P3
2	P3	P5
2	P3	P6
1	P1	P4
2	P4	P3
3	P3	P5
3	P3	P6
2	P4	P6

The result is the same as the one presented in Section 3.5.2 of 95-482.

8.5 Breadth First Explosion with level numbers (Section 3.5.3 of 95-482)

Recursive Union

```
SELECT *
   FROM (SELECT 0, 0, 1, MAJOR, MINOR FROM PP WHERE PP.MAJOR=P1
        RECURSIVE UNION PX(NODE, PARENT, LEVEL, MAJOR, MINOR)
        SELECT 0, PX.NODE, PX.LEVEL+1, MAJOR, MINOR
           FROM PP WHERE PP.MAJOR=PX.MINOR)
        SEARCH BREADTH FIRST BY MINOR SET NODE)
   ORDER BY NODE;
```

New Approach

This example is the same as the Example in Section 2.2 (Breadth-first order), except that the level number also appears in the select list. Hence, the following query just adds the level number to the select list. Recall that we gave two versions of this query, depending on the definition of the breadth first for non-tree graphs. Please see Section 2.2 for further details. Here are the two versions:

This version eliminates duplicates (using union instead of union all).

```
with RECURSIVE px (major, minor, level) as
   (    select major, minor, 1 from PP where pp.major = 'P1'
      union
        select pp.major, pp.minor, level+1
        from px, pp where pp.major = px.minor
   )
   select major, minor, level from px
   order by level, major, minor;
```

Result

MAJOR	MINOR	LEVEL
P1	P2	1
P1	P3	1
P1	P4	1
P2	P3	2
P2	P5	2
P3	P5	2
P3	P6	2
P4	P3	2

MAJOR	MINOR	LEVEL
P4	P6	2
P3	P5	3
P3	P6	3

This version preserves duplicates:

```
with RECURSIVE px (major, minor, level, unique) as
   (   select major, minor, 1, unigen() from PP where pp.major = 'P1'
      union all
       select pp.major, pp.minor, level+1, unigen()
       from px, pp where pp.major = px.minor
   )
   select level, major, minor from px
   order by level, major, unique, minor;
```

Result

LEVEL	MAJOR	MINOR
1	P1	P2
1	P1	P3
1	P1	P4
2	P2	P3
2	P2	P5
2	P3	P5
2	P3	P6
2	P4	P3
2	P4	P6
3	P3	P5
3	P3	P5
3	P3	P6
3	P3	P6

The result is the same as the one presented in Section 3.5.4 of 95-482.

8.6 Pre-Order Explosion with level numbers (Section 3.5.5 of 95-482)

This is defined to be either depth-first or breadth-first (or some other ordering), depending on the implementation. Both of these presentation orders were covered in previous examples.

8.7 Single-Level Explosion (Section 3.6 of 95-482)

Recursive Union

```
CREATE VIEW V1 AS
   SELECT DISTINCT MAJOR
   FROM (SELECT MAJOR, MINOR FROM PP WHERE PP.MAJOR=P1
        RECURSIVE UNION PX(MAJOR, MINOR)
           SELECT MAJOR, MINOR FROM PP WHERE PP.MAJOR=PX.MINOR);


SELECT * FROM PP, PARTMASTER
   WHERE EXISTS (SELECT * FROM V1 WHERE V1.MAJOR=PP.MAJOR)
   ORDER BY MAJOR, MINOR
```

New Approach

```
with RECURSIVE px (major, minor, qty) as
   (    select major, minor, qty from PP where pp.major = 'P1'
      union
        select pp.major, pp.minor, pp.qty
        from px, pp where pp.major = px.minor
   )
   select major, minor, qty from px
   order by  major, minor;
```

Result

MAJOR	MINOR	QTY
P1	P2	2
P1	P3	3
P1	P4	2
P2	P3	3
P2	P5	2
P3	P5	4
P3	P6	2

MAJOR	MINOR	QTY
P4	P3	5
P4	P6	2

The result is the same as the one presented in Section 3.6 of 95-482.

Note that we simply used union (i.e., with duplicate elimination) to eliminate duplicates, instead of creating a view as suggested in 95-482. With our approach, everything can be done within a single query. (We could have expressed this query with a view, but it is not necessary.)

The following is the second part of the query, which also gives the part names. (Again here we are not making use of an extra view.)

Recursive Union

```
SELECT MAJORPM.PNAME,PP.MAJOR, MINORPM.PNAME, PP.MINOR, PP.QTY
   FROM PP, PARTMASTER MAJORPM, PARTMASTERM MINORPM
   WHERE PP.MAJOR=MAJORPM.PNO AND PP.MINOR=MINORPM.PNO
   AND EXISTS SELECT * FROM V1 WHERE V1.MAJOR=PP.MAJOR
   ORDER BY  MAJOR, MINOR;
```

New Approach

```
with RECURSIVE px (major, minor, qty) as
   (   select major, minor, qty from PP where pp.major = 'P1'
      union
       select pp.major, pp.minor, pp.qty
       from px, pp where pp.major = px.minor
   )
   select major.pname, px.major, minor.pname, px.minor, qty
   from px, partmaster major, partmaster minor
   where px.major = major.pno and px.minor=minor.pno
   order by  major, minor;
```

Result

PNAME	MAJOR	PNAME	MINOR	QTY
MAIN	P1	RACK	P2	2
MAIN	P1	FITTING	P3	3
MAIN	P1	COVER	P4	2
RACK	P2	FITTING	P3	3
RACK	P2	BRACE	P5	2

PNAME	MAJOR	PNAME	MINOR	QTY
FITTING	P3	BRACE	P5	4
FITTING	P3	ANGLE	P6	2
COVER	P4	FITTING	P3	5
COVER	P4	ANGLE	P6	2

The result is the same as the one presented in Section 3.6 of 95-482.

The same query with order by major.pname, minor.pname creates the following result, which is the same as that of X3H2-95-482.

PNAME	MAJOR	PNAME	MINOR	QTY
COVER	P4	ANGLE	P6	2
COVER	P4	FITTING	P3	5
FITTING	P3	ANGLE	P6	2
FITTING	P3	BRACE	P5	4
MAIN	P1	COVER	P4	2
MAIN	P1	FITTING	P3	3
MAIN	P1	RACK	P2	2
RACK	P2	BRACE	P5	2
RACK	P2	FITTING	P3	3

Finally, the last query

Recursive Union

```
SELECT 0, MAJORPM.PNAME,PP.MAJOR,
          MINORPM.PNAME, PP.MINOR, PP.QTY
    FROM PP, PARTMASTER MAJORPM, PARTMASTERM MINORPM
    WHERE PP.MAJOR=MAJORPM.PNO AND PP.MINOR=MINORPM.PNO
    AND    PP.MAJOR=P1
  UNION
  SELECT 1, MAJORPM.PNAME,PP.MAJOR,
          MINORPM.PNAME, PP.MINOR, PP.QTY
    FROM PP, PARTMASTER MAJORPM, PARTMASTERM MINORPM
    WHERE PP.MAJOR=MAJORPM.PNO AND PP.MINOR=MINORPM.PNO
    AND EXISTS SELECT * FROM V1 WHERE V1.MAJOR=PP.MAJOR
```

```
    AND    PP.MAJOR <> P1
  ORDER BY 1, 2, 4
```

New Approach

In this query, part MAIN must appear first in the result. To produce this result, similar to X3H2-95-482, we mark MAIN with integer 0, and its children with 1, and we add this mark to the order by clause.

```
with RECURSIVE px (major, minor, qty, root) as
   (   select major, minor, qty, 0 from PP where pp.major = 'P1'
     union
       select pp.major, pp.minor, pp.qty, 1
       from px, pp where pp.major = px.minor
   )
   select major.pname, px.major, minor.pname, px.minor, qty
   from px, partmaster major, partmaster minor
   where px.major = major.pno and px.minor=minor.pno
   order by  root, major, minor;
```

Result

PNAME	MAJOR	PNAME	MINOR	QTY
MAIN	P1	COVER	P4	2
MAIN	P1	FITTING	P3	3
MAIN	P1	RACK	P2	2
COVER	P4	ANGLE	P6	2
COVER	P4	FITTING	P3	5
FITTING	P3	ANGLE	P6	2
FITTING	P3	BRACE	P5	4
RACK	P2	BRACE	P5	2
RACK	P2	FITTING	P3	3

Observe that we are not showing the column used to mark the tuples in the result as it is not relevant to the user.

Again, we do not need to use views or any additional query expression on top of recursive union to eliminate duplicates!

Like in X3H2-95-482, we could also use case expression to achieve the same result.

8.8 Preliminary Quantities Calculation (Section 3.7.1 of 95-482)

Recursive Union

```
SELECT *
   FROM (SELECT * FROM PP WHERE PP.MAJOR = P1
        RECURSIVE UNION PX (MAJOR, MINOR, QTY)
           SELECT MAJOR, MINOR, PX.QTY*PP.QTY FROM PP
                WHERE PP.MAJOR=PX.MINOR )
```

New Approach

```
with RECURSIVE px (major, minor, qty) as
   (    select major, minor, qty from PP where pp.major = 'P1'
      union all
        select pp.major, pp.minor, px.qty*pp.qty
        from px, pp where pp.major = px.minor
   )
   select major, minor, qty from px;
```

Result

MAJOR	MINOR	QTY
P1	P2	2
P1	P3	3
P1	P4	2
P2	P3	6
P2	P5	4
P3	P5	24
P3	P6	12
P4	P3	10
P4	P6	4
P3	P5	12
P3	P6	6
P3	P5	40
P3	P6	20

The result is the same as the one presented in Section 3.7.2 of 95-482. (The order of the results could be different since no order by is specified.)

8.9 Summarized Quantities Calculation (Section 3.7.3 of 95-482)

Recursive Union

```
SELECT MINOR, SUM(QTY)
   FROM (SELECT * FROM PP WHERE PP.MAJOR = P1
        RECURSIVE UNION PX (MAJOR, MINOR, QTY)
           SELECT MAJOR, MINOR, PX.QTY*PP.QTY FROM PP
                WHERE PP.MAJOR=PX.MINOR)
   GROUP BY MINOR
```

New Approach

```
with RECURSIVE px (major, minor, qty) as
   (    select major, minor, qty from PP where pp.major = 'P1'
      union all
        select pp.major, pp.minor, px.qty*pp.qty
        from px, pp where pp.major = px.minor
   )
   select minor, sum(qty) from px
   group by minor;
```

Result

MINOR	SUM (QTY)
P2	2
P3	19
P4	2
P5	80
P6	42

The result is the same as the one presented in Section 3.7.4 in 95-482.

8.10 Eliminating 'phantom' Parts (Section 3.8 of 95-482)

Recursive Union

```
CREATE VIEW PPX AS SELECT *
   FROM ( SELECT 0, 0, 1, MAJOR, MINOR, QTY FROM PP WHERE PP.MAJOR=P1
        RECURSIVE UNION PX(NODE, PARENT, LEVEL, MAJOR, MINOR, QTY)
        SELECT 0, PX.NODE, PX.LEVEL+1, MAJOR, MINOR, PX.QTY*PP.QTY
            FROM PP WHERE PP.MAJOR=PX.MINOR
   SEARCH DEPTH FIRST BY MINOR SET NODE)
```

```
CREATE VIEW PPXMM
   (NODE, PARENT, LEVEL, MAJOR, MINOR, QTY, MAJORIS, MINORIS)
   AS SELECT PPX.*, MAJORPM.ISPHANTOM, MINORPM.ISPHANTOM
   FROM PPX, PARTMASTER MAJORPM, PARTMASTER MINORPM
   WHERE PPX.MAJOR=MAJORPM.PNO AND PPX.MINOR=MINORPM.PNO

CREATE VIEW SPANPHANTOMS AS SELECT * FROM
   FROM (SELECT * FROM PPXMM
        RECURSIVE UNION R
        SELECT R.NODE, R.PARENT, R.LEVEL, R.MAJOR, R2.MINOR,
           R.QTY*R2.QTY, R.ISPHANTOM, R2.ISPHANTOM
        FROM PPXMM R2
        WHERE R.MINORIS=yes AND R.MINOR=R2.MAJOR
        AND R.NODE=R2.PARENT);

CREATE VIEW NOPHANTOMS AS
   SELECT NODE, PARENT, LEVEL, MAJOR, MINOR, QTY
   FROM SPANPHANTOMS
   WHERE MAJORIS='no' AND MINORIS='no';
```

New Approach

In our formulation for this example, we choose to do the whole work with one recursion (instead of double recursion as used in X3H2-95-482). We use the CASE expressions to skip over phantoms. Further, unlike X3H2-95-482, we avoid creating views. In practice, often predicates such as major='P1' are coded as major=:hv. Since host variable are not allowed in view definitions, views cannot always be used.

```
 with RECURSIVE ppx (level, major, majoris,  minor, qty, ak) as
   (    select 1,major, partmaster.isphantom, minor, qty, list {major, minor}
        from PP, partmaster
        where pp.major = 'P1' and pp.major = partmaster.pno
        and partmaster.isphantom = 'NO'
      union all
        select case partmaster.isphantom
                   when 'YES' then ppx.level else ppx.level+1 end,
               case partmaster.isphantom
                   when 'YES' then ppx.major else pp.major end,
               partmaster.isphantom,pp.minor,
               case partmaster.isphantom
                   when 'YES' then ppx.qty*pp.qty else pp.qty end,
               case partmaster.isphantom
                   when 'YES' then ak else list_append (ak, pp.minor) end
        from ppx, pp, partmaster
        where pp.major = ppx.minor and pp.major = partmaster.pno
   )
   select ppx.level, ppx.major, ppx.minor, ppx.qty from ppx, partmaster
   where ppx.minor = partmaster.pno and partmaster.isphantom = 'NO'
   order by ak
```

Result

LEVEL	MAJOR	MINOR	QTY
1	P1	P2	2
2	P2	P5	12
2	P2	P6	6
2	P2	P5	2
1	P1	P5	12
1	P1	P6	6
1	P1	P5	40
1	P1	P6	20
1	P1	P6	4

The result is the same as the one presented in Section 3.8.1 of 95-482.

8.11 WHERE-USED (Section 3.9 of 95-482)

Recursive Union

```
SELECT MAJOR, MINOR
   FROM (SELECT * FROM PP WHERE PP.MINOR='P5'
        RECURSIVE UNION WU
           SELECT * FROM PP WHERE PP.MINOR = WU.MAJOR)
```

New Approach

```
with RECURSIVE px (major, minor) as
   (select major, minor from PP where pp.minor = 'P5'
   union all
   select pp.major, pp.minor from px, pp where pp.minor = px.major
   )
   select major, minor from px;
```

Result

MAJOR	MINOR
P2	P5
P3	P5
P1	P2

MAJOR	MINOR
P1	P3
P2	P3
P4	P3
P1	P2
P1	P4

The result is the same as the one presented in Section 3.9.1 of 95-482.

To produce the distinct version of the answer, we simply use union instead of union all. In X3H2-95-482, distinct must be introduced in another query expression since recursive union cannot handle distinct.

Recursive Union

```
SELECT DISTINCT MAJOR
   FROM (SELECT * FROM PP WHERE PP.MINOR='P5'
        RECURSIVE UNION WU
           SELECT * FROM PP WHERE PP.MINOR = WU.MAJOR)
```

New Approach

```
with RECURSIVE px (major, minor) as
   (    select major, minor from PP where pp.minor = 'P5'
      union
        select pp.major, pp.minor from px, pp where pp.minor = px.major
   )
   select major, minor from px;
```

8.12 Testing For Leaf Nodes In A Parts Explosion (Section 3.10 of 95-482)

Recursive Union

```
SELECT *
   FROM (SELECT 0, 0, MAJOR, MINOR, PARTMASTER.PNAME, LEVELCODE
           FROM PP,PARTMASTER
           WHERE PP.MAJOR=P1 AND PP.MINOR=PARTMASTER.PNO
   RECURSIVE UNION PX (NODE, PARENT, MAJOR, MINOR, PNAME, LEVELCODE)
           SELECT MAJOR, MINOR, TAG, PX.NODE, LEVELCODE
           FROM PP, PARTMASTER
           WHERE PP.MAJOR=PX.MINOR  AND PP.MINOR=PARTMASTER.PNO
           AND PX.LEVELCODE <> 'leaf'
   SEARCH PRE ORDER SET NODE)
```

New Approach

```
with RECURSIVE px (major, minor, pname, levelcode) as
   (   select major, minor, partmaster.pname, partmaster.levelcode
       from PP, partmaster
       where pp.major = 'P1' and pp.minor = partmaster.pno
     union all
       select pp.major, pp.minor, partmaster.pname, partmaster.levelcode
       from px, pp, partmaster
       where pp.major = px.minor and pp.minor = partmaster.pno
       and px.levelcode <> 'LEAF'
   )
   select major, minor, pname, levelcode from px;
```

Result

MAJOR	MINOR	PNAME	LEVEL CODE
P1	P2	RACK	
P1	P3	FITTING	
P1	P4	COVER	
P2	P3	FITTING	
P2	P5	BRACE	LEAF
P3	P5	BRACE	LEAF
P3	P6	ANGLE	LEAF
P4	P3	FITTING	
P4	P6	ANGLE	LEAF
P3	P5	BRACE	LEAF
P3	P6	ANGLE	LEAF
P3	P5	ANGLE	LEAF
P3	P6	ANGLE	LEAF

8.13 Cycles (Section 4 of X3H2-95-482)

Part-Part Table PPA

PPA	
MAJOR	**MINOR**
P1	P2
P1	P3
P1	P4
P3	P5
P3	P6
P6	P2
P6	P3
P6	P4

Recursive Union

```
SELECT  *
   FROM (SELECT 0, 0, MAJOR, MINOR FROM PPA WHERE PPA.MAJOR='P1'
        RECURSIVE UNION PX(NODE, PARENT, MAJOR, MINOR)
           SELECT 0, PX.NODE, MAJOR, MINOR
                FROM PPA WHERE PPA.MAJOR=PX.MINOR
        SEARCH DEPTH FIRST BY MINOR SET NODE)
   ORDER BY PARENT ASC, NODE DESC;
```

New Approach

The query formulated in section 4.1 of X3H2-95-482 has an infinite loop. This problem can be easily solved in our approach by using union instead of union al.

```
with RECURSIVE px (major, minor) as
   (    select major, minor from PPA where ppa.major = 'P1'
      union
        select ppa.major, ppa.minor from px, ppa where ppa.major = px.minor
   )
   select major, minor  from px
   order by major asc, minor desc;
```

Result

MAJOR	MINOR
P1	P2
P1	P3
P1	P4
P3	P5
P3p	P6
P6	P2
P6	P3
P6	P4

If we use union all, then the query has an infinite loop, just like the one presented in X3H2-95-482.

8.14 Cycles - explosion with CYCLE clause (Section 4.3 of 95-482)

```
SELECT *
   FROM (SELECT 0, 0, MAJOR, MINOR, 0 FROM PPA WHERE PPA.MAJOR='P1'
        RECURSIVE UNION PX(NODE, PARENT, MAJOR, MINOR, CYCLEMARK)
           SELECT 0, PX.NODE, MAJOR, MINOR, 0
              FROM PPA WHERE PPA.MAJOR=PX.MINOR
        SEARCH DEPTH FIRST BY MINOR SET NODE
        CYCLE SET CYCLEMARK TO 1)
   ORDER BY PARENT ASC, NODE DESC;
```

New Approach

```
with RECURSIVE px (major, minor, cyclemark, ak) as
   (    select major, minor, '0', list {major, minor}
        from PPA where ppa.major = 'P1'
      union all
        select ppa.major, ppa.minor,
           case when ppa.minor in
                (select * from table (list_append {ak, ppa.minor}))
                then '1' else '0' end, list_append {ak, ppa.minor}
        from px, ppa where ppa.major = px.minor and px.cyclemark = '0'
   )
   select major, minor, cyclemark  from px
   order by major, minor desc;
```

Result

MAJOR	MINOR	CYCLE MARK
P1	P4	0
P1	P3	0
P1	P2	0
P3	P6	0
P3	P5	0
P6	P4	0
P6	P3	1
P6	P2	0

The result is the same as the one presented in Section 4.3 of X3H2-95-482.

8.15 Result of the explosion in NODE order (Section 4.4)

This is the same as the query of Section 4.2, except that the result is presented in depth-first order. Note that the list column ak is very handy to produce this result order, as explained earlier.

```
with RECURSIVE px (major, minor, cyclemark, ak) as
  (   select major, minor, '0', list {major, minor}
      from PPA where ppa.major = 'P1'
     union all
      select ppa.major, ppa.minor,
        case when ppa.minor in
           (select * from table (list_append {ak, ppa.minor}))
           then '1' else '0' end, list_append {ak, ppa.minor}
      from px, ppa where ppa.major = px.minor and px.cyclemark = '0'
  )
  select major, minor, cyclemark  from px
  order by ak;
```

Result

MAJOR	MINOR	CYCLE MARK
P1	P2	0
P1	P3	0

MAJOR	MINOR	CYCLE MARK
P3	P5	0
P3	P6	0
P6	P2	0
P6	P3	1
P6	P4	0
P1	P4	0

Again this is the same result as the one given in Section 4.4 of 95-482.

8.16 Subcomponents (Section 4.5 of 95-482)

Table PPB

PPB	
MAJOR	**MINOR**
P1	P2
P1	P3
P1	P4
P3	P5
P3	P6
P6	P2
P6	P3
P6	P4
P2	P10
P2	P11
P2	P12
P11	P13
P11	P14

Recursive Union

```
SELECT *
   FROM (SELECT 0, 0, MAJOR, MINOR, 0 FROM PPB WHERE PPB.MAJOR='P1'
      RECURSIVE UNION PX(NODE, PARENT, MAJOR, MINOR, CYCLEMARK)
           SELECT 0, PX.NODE, MAJOR, MINOR, 0
                FROM PPB WHERE PPB.MAJOR=PX.MINOR
                AND PX.MAJOR<>'P2'
        SEARCH DEPTH FIRST BY MINOR SET NODE
        CYCLE SET CYCLEMARK TO 1)
   ORDER BY PARENT ASC, NODE DESC;
```

It is important to make a couple remarks here: it is surprising to observe that in recursive union once we added the predicate: PX.MAJOR <> 'p2', our cycle detection changes as a side-effect, resulting in a major change in the answer set. Addition of this predicate causes a new reference (PX.MAJOR) to the recursive table. As a side-effect, this column is also compared to detect cycles! Before this predicate was added, we were only looking to see if we have an ancestor with the value for PX.MINOR. Now we are looking for an ancestor having the same value for both PX.MAJOR and PX.MINOR.

Further, the user will be surprised seeing that by adding a new predicate ANDed with the other predicates, we go from 8 tuples in Section 4.3 to 13 tuples in Section 4.5! We naturally expected to see less tuples!

We think semantics with such side-effects is not desirable.

To show that our proposal can also handle this example we formulated the following query. Fortunately, using the SQL3 LIST and row expression constructor, we can easily modify our earlier query to get this result.

New Approach

```
with RECURSIVE px (major, minor, cyclemark, ak) as
   (    select major, minor, '0', list {(major, minor)}
        from PPA where ppa.major = 'P1'
      union all
        select ppa.major, ppa.minor, case when (ppa.major,ppa.minor) in
           (select * from table (list_append {ak, (ppa.major,ppa.minor)}))
           then '1' else '0' end, list_append {ak, (ppa.major, ppa.minor)}
        from px, ppa
        where ppa.major = px.minor and px.major <> 'P2' and px.cyclemark = '0'
   )
   select major, minor, cyclemark, ak from px
   order by major, minor desc;
```

Result

MAJOR	MINOR	CYCLE MARK	AK
P1	P4	0	{(P1, P4)}
P1	P3	0	{(P1, P3)}
P1	P2	0	{(P1, P2)}
P3	P6	0	{(P1, P3),(P3, P6)}
P3	P6	1	{(P1, P3),(P3, P6),(P6, P3),(P3, P6)}
P3	P5	0	{(P1, P3),(P3, P5)}
P3	P5	0	{(P1, P3),(P3, P6),(P6, P3),(P3, P5)}
P6	P4	0	{(P1, P3),(P3, P6),(P6, P4)}
P6	P3	0	{(P1, P3),(P3, P6),(P6, P3)}
P6	P2	0	{(P1, P3),(P3, P6),(P6, P2)}

Unfortunately, the answer presented in Section 4.6 of X3H2-95-482 is not correct. To demonstrate this, we have Column ak in our output List, which captures all (major, minor) of each node and its ancestors. Tuple (P3, P6) has (P3, P6) in one of its ancestor tuples. Therefore, this must be marked with CYCLEMARK = '1'. In X3H2-95-482, this tuple is marked as '0'. After correcting this, the last 3 tuples in Section 4.6 of X3H2-95-482 must be deleted since the cycle occurs earlier, and does not generate any children.

8.17 Non-Linear Recursive Union (Section 5 of 95-482)

Recursive Union

```
(SELECT* FROM PERSON WHERE ID IN (A, B, C, D)
   RECURSIVE UNION M, F (ID, MOTHER, FATHER)
   SELECT * FROM PERSON C
      WHERE C.MOTHER = M.ID AND    C.FATHER = F.ID) )
```

New Approach

```
with RECURSIVE descendents (id, mother, father) as
   (   SELECT * from person where id in ('A', 'B','C','D')
      UNION
       SELECT c.* FROM descendents m, descendents f, person c
       WHERE c.mother = m.id and c.father = f.id
   )
```

```
SELECT * FROM descendents;
```

8.18 Non-Linear: Breadth-first and Depth First (Section 6.1 and 6.2 of 95-482)

The definition of breadth-first and depth-first came from graph theory. The graph traversal queries are linear recursive queries. Therefore, it is not clear to what extent it makes sense to talk about depth-first and breadth first ordering for non-linear recursive queries. Even if one extends such definitions, it is not clear that there is a well accepted definition. For this reason, we could write a query that produces the same result as recursive union, but it is unclear that this is the result that the hypothetical users of non-linear queries would expect. As mentioned in 95-457, non-linear queries are currently rare and it is prudent to wait until we better understand the application of recursive queries before we choose the semantics of breadth first and depth first presentation.

The nice thing about our proposal is that the traversal is defined by the query. Consequently, the SQL3 standard does not need to be bound to any definition, avoiding this problem and making future changes possible. In our proposal, we reuse existing SQL3 constructs (such as lists) which sufficiently provide the primitives for the users to choose the semantics they want.

8.19 The Brothers example (Section 6.3 of 95-482)

Recursive Union

```
CREATE VIEW BROTHER
   AS SELECT * FROM HBROTHER
   UNION SELECT ID AS BRID, BRID AS ID FROM HBROTHER
   SELECT ID
   FROM (SELECT * FROM PERSON WHERE P.ID IN (1, 4, 5, 7)
   RECURSIVE UNION YB, OB (ID, AGE)
   SELECT PERSON.*
   FROM PERSON P, BROTHER POB, BROTHER PYB
   WHERE P.ID=POB.ID AND P.ID=PYB.ID
   AND PYB.BRID=YB.ID AND POB.BRID=OB.ID
   AND P.AGE < OB.AGE AND P.AGE > YB.AGE)
```

New Approach

```
with RECURSIVE recbro (id, age) as
   (    SELECT id, age FROM person WHERE id in (1,4,5,7)
      UNION ALL
        SELECT p.id, p.age
```

```
     FROM recbro yb, recbro ob,person p, brother pob, brother pyb
     WHERE p.id = pob.id AND p.id = pyb.id
     AND pyb.brid = yb.id AND pob.brid = ob.id
     AND p.age < ob.age AND p.age > yb.age
)
SELECT * FROM recbro;
```

Note to the Reader: I'm not reproducing the result in here. It is too much work!

9 Query from X3H2-95-475

The following query was taken from X3H2-95-475.

Note that we have modified Len's query a little because it was not a valid SQL query in its original form:

1. null cannot be in the select list

2. some columns had to be fully qualified since it appears in multiple tables

Here is the modified version of Len's query:

Recursive Union

```
    SELECT 0, Part_Name, Part_Id,
            cast(null as char(5)), 1,
          cast(null as char(8)) ,
          cast(null as integer), cast(null as varchar(8))
    FROM Components
    WHERE Part_Name = 'Wheel_Assembly'
RECURSIVE UNION
     Rslt (Seq_Nbr, Part_Name, Part_Id, Used_In, Quantity,
     Availability, Cost, Cycle_Mark)
    SELECT 0, Components.Part_Name, Structure.Part_Id,
          Structure.Used_In, StructureQuantity,
          Inventory.Availability, Inventory.Cost,
          '<space>'
    FROM (Structure NATURAL LEFT OUTER JOIN Inventory)
        INNER JOIN Components
        USING (Part_Id)
    WHERE Structure.Used_In = Rslt.Part_Id
    AND NOT (Rslt.Availability = 'stocked')
SEARCH DEPTH FIRST BY Part_Name SET Seq_Nbr
CYCLE Part_Id SET Cycle_Mark TO '*'
RETURN LIMIT (100)
```

New Approach

```
with RECURSIVE Rslt (Part_Name, Part_Id, Used_In, Quantity, Availability,
   Cost, Cycle_Mark, ak) AS
   (    SELECT  Part_Name, Part_Id, cast (null as char(5)), 1,
           cast (null as char(8)), cast (null as integer),
           cast (null as varchar(8)), list{part_id}
        FROM Components
        WHERE Part_Name = 'Wheel_Assembly'
      UNION all
        SELECT  Components.Part_Name, Structure.Part_Id, Structure.Used_In,
           Structure.Quantity, Inventory.Availability, Inventory.Cost,
           case when Structure.Part_Id in
             (select * from table (list_append(ak, Structure .Part_Id) )
             then '*' else '<space>' end, list_append(ak, Structure.Part_Id)
        FROM Rslt, ( Structure NATURAL LEFT OUTER JOIN Inventory)
             INNER JOIN Components USING (Part_Id)
        WHERE Rslt.Cycle_Mark != '*' AND Structure.Used_In = Rslt.Part_Id
        AND NOT (Rslt.Availability = 'stocked')
   )
   select *  from Rslt  order by ak;
```

Note that the only aspect that we are not addressing is the limit clause. It could, however, as described in Section 3.2.9 of X3H2-95-457, be added to SQL3 as an extension at the same level as ORDER BY as in SELECT * FROM Rslt ORDER BY ak LIMIT (100).

10 Performance considerations

Users of SQL databases rely on SQL DBMSs for efficient execution of queries formulated in the SQL high level language. As a result, SQL DBMSs apply sophisticated techniques to query optimization and execution. Inadequacies in this area often leads to orders of magnitude performance degradation, resulting in unacceptable response time and resource consumption. Handling of recursive queries is no exception in this regard. Hence, it is crucial to verify that proposed extensions to SQL can indeed be implemented efficiently.

The execution model for our proposal is mainly based on semi-naive evaluation. Semi-naive evaluation is a well accepted technique for execution of recursive queries. For the purpose of our discussion, think of semi-naive as breadth first evaluation. For a set of tuples (called parent tuples) we generate a new set of tuples (called children). Then we join the new set with whatever else we have in the recursive part of the query to generate yet another new set (called children of children), and so on. For the join involving the new set, the optimizer has the freedom to choose the least cost alternative from the alternatives associated with a set of join methods (e.g., nested loop join, hash join, merge join, etc.) and join orders (e.g., in our depth-first query, the optimizer can choose between (px, pp) or (pp, px) join orders). Set oriented processing naturally fits parallelism. Note that set orientedness of SQL is the main reason that parallel relational DBMSs are so successful. Parallel processing is crucial for processing of large queries. Since join is an important and fundamental operation in SQL, relational DBMSs invest heavily in optimization and efficient execution of joins. Such investment includes sophisticated join methods, join order optimization, and parallelization. The semi-naive evaluation, which is based on joins, benefits from such investment directly.

Recursive union is heavily biased toward one at the time correlation. This combined with the procedural stack model, biases the recursive union toward non-set oriented execution. Its execution is strongly biased toward execution of correlated subqueries, instead of set oriented joins. This puts recursive union at considerable disadvantage. As Jim Gray presented in SIGMOD 95, parallel execution of correlated subqueries is one of the major challenges of relational DBMSs today. The performance of such query forms does not nearly match the performance of joins. Considerable effort has been put in converting correlated subqueries to join, exactly for this reason. After nearly a decade of effort, only a very small subset of such queries can be converted /Ki82/, /Ki84/, /Da87/, /GW87/, /SPL96/. These approaches only work in certain cases.

From the query optimization and execution viewpoints, it begs the following question. If we are forced to convert recursive union where possible (with considerable effort), then why SQL3 did not adopt the established model of recursion from deductive database world, where the queries are expressed in the form of joins? (Of course, our proposal follows the join model, avoiding such problems).

There has been tremendous work on query optimization for recursive queries. Few examples of such optimization techniques include left linear to right linear transformation and magic set transformation /MFPR90a/. Such techniques are designed for the established recursive query models. It is our assessment that it is difficult and it is unnecessary to adapt such query transformations to recursive union, as we explained earlier.

END OF PAPER