Graphs

Graphs are much more challenging than trees. Even something as basic as planar graph drawing is still an open research problem. There are few graph encodings, and all of them are volatile. The set of graph problems is much richer, and those problems are usually hard. Applied to graphs cowboy programmer attitude simply doesn't work.

In this chapter we'll venture into different realms of graph methods: the recursive `with` kingdom, the `connect by` county, and materialized transitive closure province. We'll take a look at typical graph queries, beginning with the number of connected components and ending with aggregated totals.

Schema Design

Being cautioned about complexity of graph problems lets approach the field carefully, starting with the schema design. When designing a schema, the first question to ask is how do we define graphs?

Graphs can be directed and undirected. Our study is devoted almost exclusively to directed graphs; this is why we would normally omit the adjective. Directed graph definition is nearly ubiquitous.

Graph Definition

Graph is a set of *nodes* connected by *edges*. Each edge is an ordered pair of nodes, to which we'll refer to as `head` and `tail`.

It is straightforward to carry over this graph definition to schema design

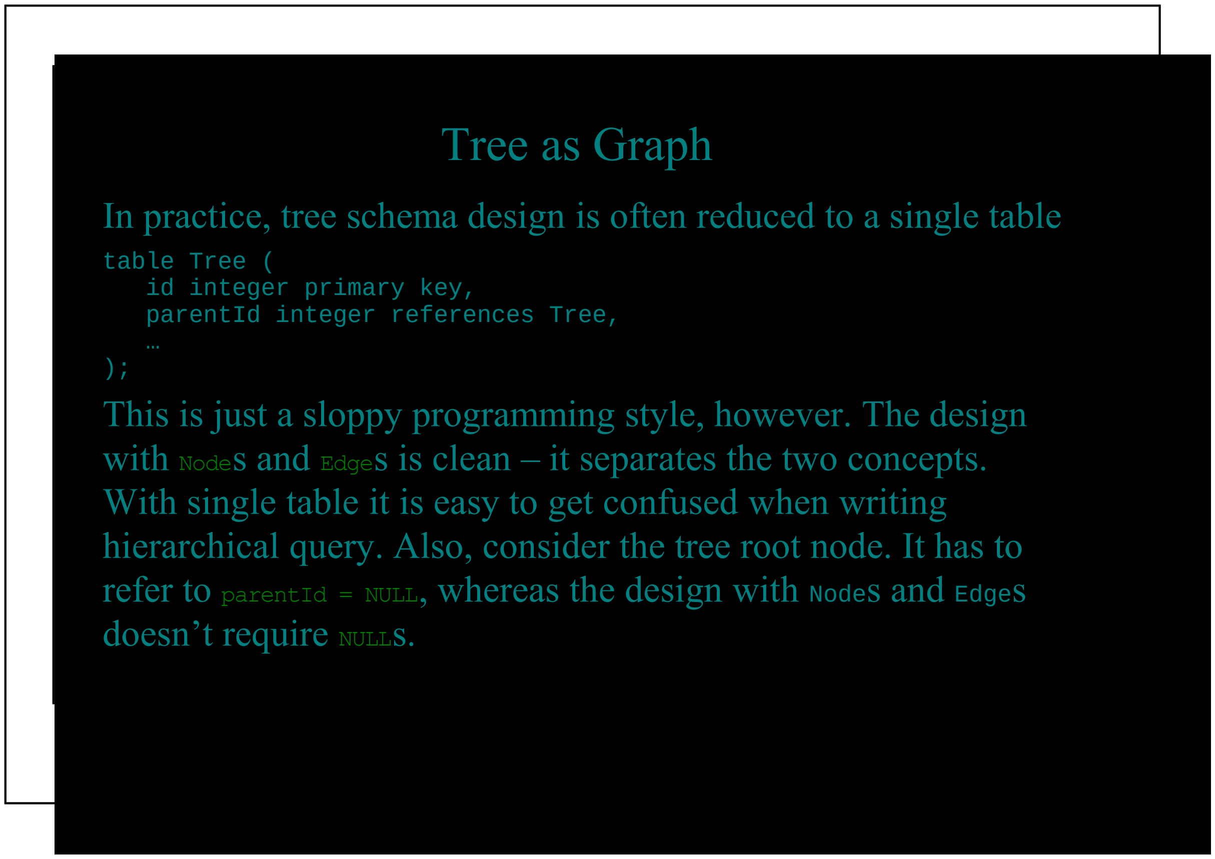
```
table Nodes (
   id integer primary key,
   …
);

table Edges (
   head integer references Nodes,
   tail integer references Nodes,
   …
);
```

Nodes and edges in a graph are normally weighted, hence the ellipsis in schema definition. For example, in a network of cities connected by roads, each edge is labeled with distance.

Tree Constraint

Representing trees as graphs is a very common database application programming practice.



Tree as Graph

In practice, tree schema design is often reduced to a single table

```
table Tree (
   id integer primary key,
   parentId integer references Tree,
   …
);
```

This is just a sloppy programming style, however. The design with `Node`s and `Edge`s is clean – it separates the two concepts. With single table it is easy to get confused when writing hierarchical query. Also, consider the tree root node. It has to refer to `parentId = NULL`, whereas the design with `Nodes` and `Edges` doesn't require `NULL`s.

The graph schema, however, allows graphs which aren't trees. Consider a cycle

head	tail
1	2
2	3
3	1

for example. In order to describe a tree structure in graph terms, we need a couple of auxiliary definitions.

Directed path is a sequence of edges such that the previous edge head is the same as the next edge tail.

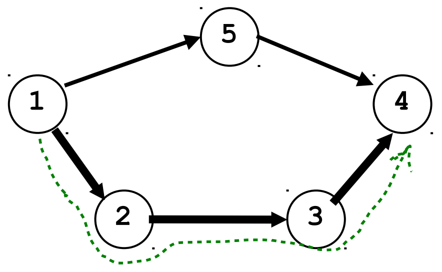


Figure 6.1: Graph with directed path (tail=1,head=2) (tail=2,head=3) (tail=3, head=4) emphasized.

Undirected path is a sequence of edges such that the previous edge head or tail coincides with the next edge head or tail.

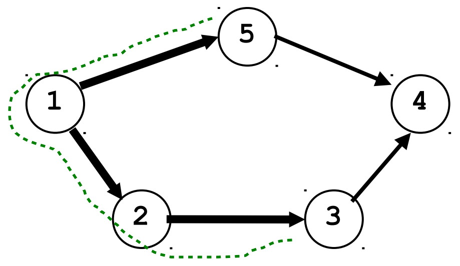
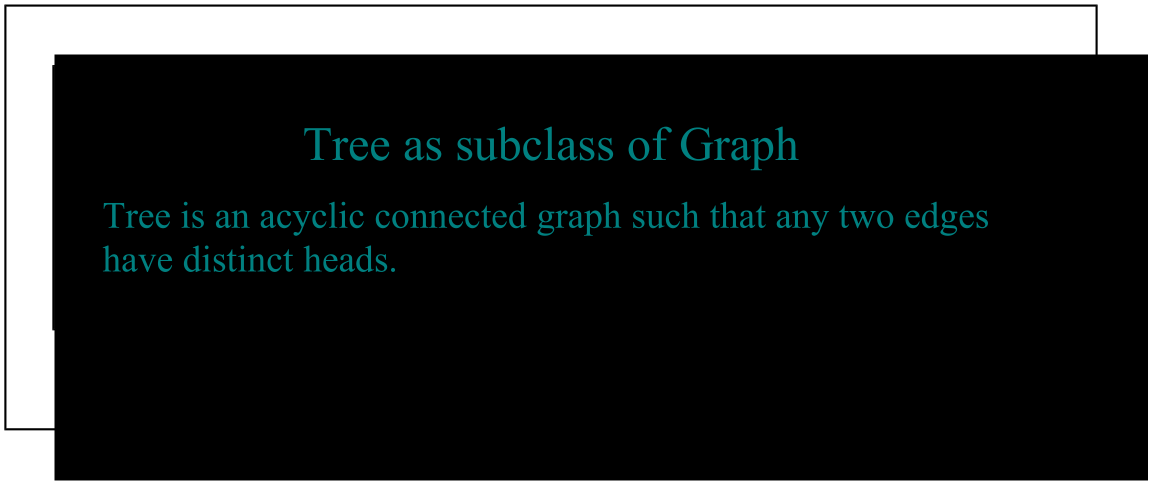


Figure 6.2: Graph with undirected path (tail=1,head=5) (tail=1,head=2) (tail=2,head=3) emphasized.

We need undirected path concept to define **connected graph** – a graph where there is a path between every pair of nodes. Otherwise, by **path** we would understand the directed path. For example, Graph on figure 5.1 is connected.

Cycle is a closed path. In closed path the last edge head is the same as the first edge tail. **Acyclic** graph doesn't contain cycles.

The tree definition, then, is essentially a constrained graph.



Tree as subclass of Graph

Tree is an acyclic connected graph such that any two edges have distinct heads.

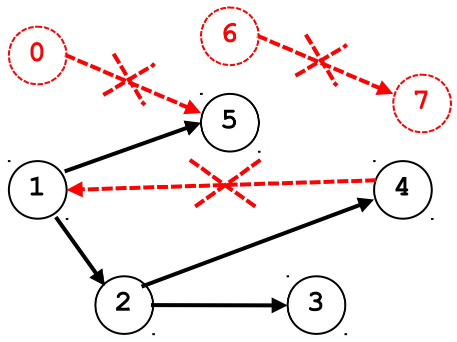


Figure 6.3: Tree constraints. The edge (tail=4,head=1) violates acyclic property. The edge (tail=6,head=7) makes graph disconnected. The edge (tail=0,head=5) has the same head as (tail=1,head=5).

While the condition that any two edges have distinct heads could be declared effortlessly as a unique key constraint on the `head` column, the other two constraints are tough. If we try to leverage the sledgehammer materialized views approach, discussed in chapter 4, then we have to be capable writing one query that finds cycles, and another query that calculates a number of connected components in a graph.

This doesn't look very promising. Therefore, we have to give up on enforcing tree constraint in a graph model. To be fair, though, only a minority of database application programmers would insist that enforcing tree constraint is a matter of life-or-death. More important issues are:

1. How do we query graph model?

2. Are the query methods efficient?

Transitive Closure

There are several key graph concepts that would guide your intuition when writing queries on graphs.

1. *Reflexive closure* of a graph is build by adding missing *loops* – edges with the same endpoints. Reflexive closure is expressed easily in SQL:

```
select head, tail from Edges -- original relation
union
select head, head from Edges -- extra loops
union
select tail, tail from Edges -- more loops
```

2. *Symmetric closure* of a (directed) graph is build by adding inversely oriented edge for each edge in the original graph. In SQL:

```
select head, tail from Edges -- original relation
union
select tail, head from Edges -- inverse arrows
```

3. *Transitive closure* of a (directed) graph is generated by connecting edges into paths and creating new edge with the tail being the beginning of the path, and the head being the end. Unlike the previous two cases, transitive closure can't be expressed with bare SQL essentials – the select, project, and join relational algebra operators.

For now, let's assume that we know how to query transitive closure, and demonstrate how armed with these definitions we can approach some problems. Here is a puzzle from the `comp.databases.oracle.misc` forum:

Suppose I have a table that contains Account# & RelatedAccount# (among other things).

How could I use CONNECT BY & START WITH in a query to count relationships or families.

For example, in

```
ACCT     REL_ACCT
Bob      Mary
Bob      Jane
Jane     Bob
Larry    Moe
Curly    Larry
```

there are 2 relationship sets (Bob,Mary,Jane & Larry,Moe,Curly). If I added

```
Curly    Jane
```

then there'd be only 1 larger family. Can I use CONNECT BY & START with to detect such relationships and count them? In my case I'd be willing to go any number of levels deep in the recursion.

Let's ignore oracle specific SQL keywords as inessential to the problem at hand. With proper graph terminology the question can be formulated in just one line:

Find a number of connected components in a graph.

Connected component of a graph is a set of nodes reachable from each other. A node is *reachable* from another node if there is an undirected path between them.

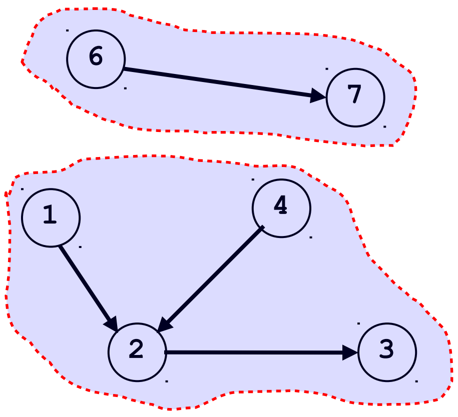


Figure 6.4: A graph with two connected components.

Reachability is an **equivalence** relation: it's reflective, symmetric, and transitive. Given a graph, we formally obtain reachability relation by closing the `Edge`s relation to become reflective, symmetric and, transitive (fig. 6.5).

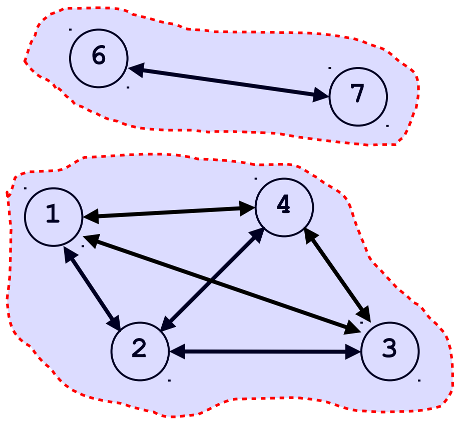


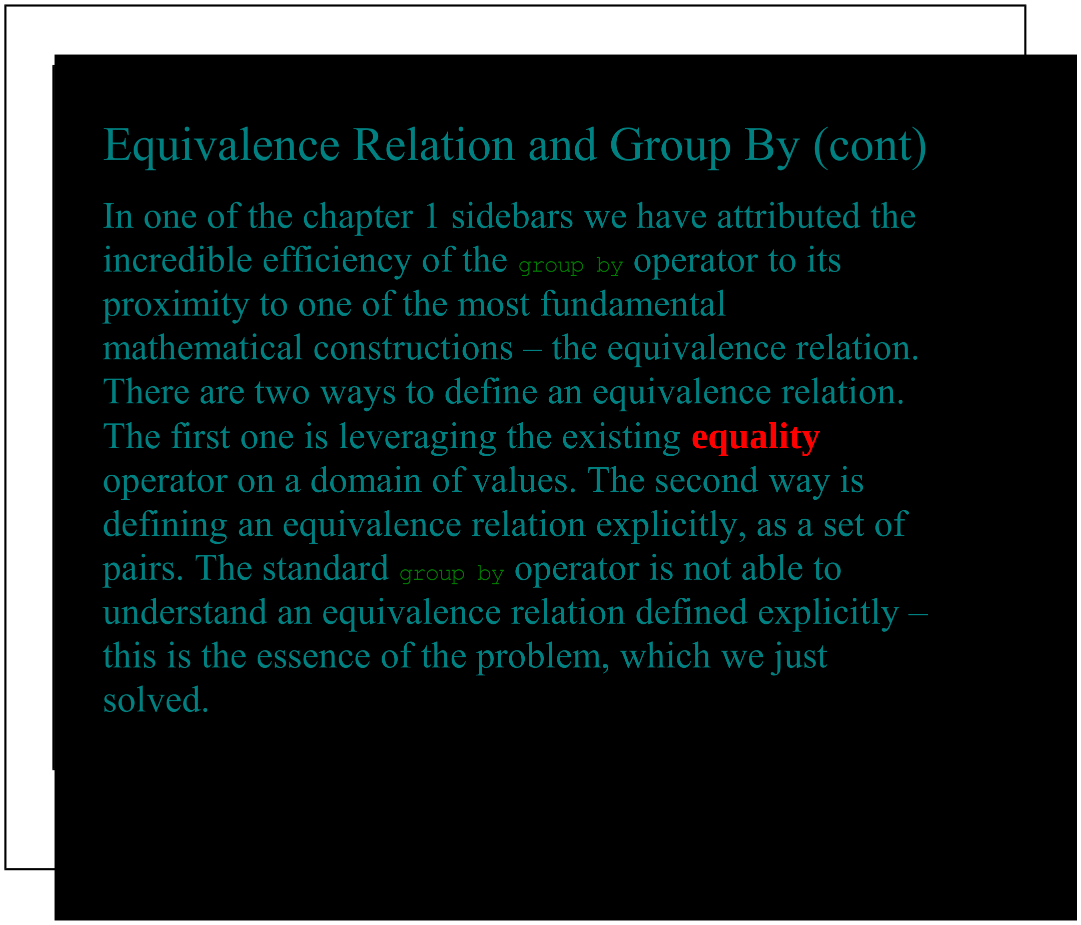
Figure 6.5: Reachability as an equivalence relation: graph from fig. 6.4 symmetrically and transitively closed.

Returning back to the problem of finding the number of connected components, let's assume that we already calculated the reachability relation `EquivalentNodes`s somehow. Then, we just select a smallest node from each component. Informally,

Select node(s) such that there is no node with smaller label reachable from it. Count them.

Formally:

```
select count(distinct tail) from EquivalentNodes e
where not exists (
   select * from EquivalentNodes ee
   where ee.head<e.tail and e.tail=ee.tail
);
```



Equivalence Relation and Group By (cont)

In one of the chapter 1 sidebars we have attributed the incredible efficiency of the `group by` operator to its proximity to one of the most fundamental mathematical constructions – the equivalence relation. There are two ways to define an equivalence relation. The first one is leveraging the existing **equality** operator on a domain of values. The second way is defining an equivalence relation explicitly, as a set of pairs. The standard `group by` operator is not able to understand an equivalence relation defined explicitly – this is the essence of the problem, which we just solved.

Being able to query the number of connected components earned us an unexpected bonus: we can redefine a connected graph as a graph that has a single connected component. Next, a connected graph with `N` nodes and `N-1` edges must be a tree. Thus, counting nodes and edges together with transitive closure is another opportunity to enforce tree constraint.

Now that we established some important graph closure properties, we can move on to transitive closure implementations. Unfortunately, our story has to branch here, since database vendors approached hierarchical query differently.

Recursive SQL

DB2 and SQL Server 2005 support ANSI SQL standard recursive SQL, which renders transitive closure effortlessly

```
with TransClosedEdges (tail, head) as
( select tail, head from Edges
  union all
  select e.tail, ee.head from Edges e, TransClosedEdges ee
  where e.head = ee.tail
)
select distinct * from TransClosedEdges
```

This query looks artificial at first. It requires a certain educational background to fully appreciate it.

Consider *adjacency matrix* of a graph. It's a square matrix with dimensions equal to the number of nodes. It is conventional to enumerate graph nodes with numbers from `1` to `N`, therefore, matching nodes with matrix columns and rows. With this arrangement matrix entry a_{ij} naturally correspond to an edge from node `i` to node `j`. If there is indeed such an edge in a graph, then we define $a_{ij}=1$; otherwise, $a_{ij}=0$.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 6.6: Adjacency matrix for graph at fig 6.1.

For our purposes the powers[1] of adjacency matrix are especially interesting. The entry in column `i` and row `j` of the adjacency matrix raised in the `n`-th power is the number of paths of length `n` from node `i` to node `j`.

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 6.7: Adjacency matrix for graph at fig 6.1 squared. The $a_{13} = 1$ indicates that there is one path of length 2 from node 1 to node 3.

[1] Matrix power A^n is the product of n copies of A: $A^n = A \cdot A \cdot A \cdot \ldots \cdot A$

Although the square of adjacency matrix at fig.6.7 is adjacency matrix of graph shown at fig.6.8, in general, the power of adjacency matrix, doesn't have be an adjacency matrix anymore.

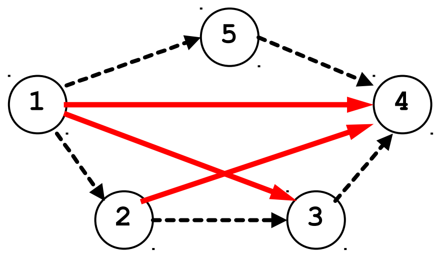


Figure 6.8: Graph corresponding to the adjacency matrix at fig 6.1.

There are various ways to fix this problem, and we'll return to this later. For now, given an adjacency matrix A, consider a formal series

$$T_A = A + A^2 + A^3 + \ldots$$

What kind of matrix T_A adding all the adjacency matrix powers produces? The reader can easily convince himself that an entry in the column `i`, row `j` of the matrix A^n is the number of paths of length `n` from node `i` to node `j` in the original graph. Then, an entry in the matrix T_A is the number of paths [of any length] from node `i` to node `j`.

For adjacency matrices corresponding to directed acyclic graphs the powers would evaluate to 0 for sufficiently large `n`, and the formal series T_A is finite. In other words, all the paths in a directed acyclic graph have their length bounded.

Given transitive closure series matrix T_A, if we just change any nonzero number into `1`, then we'll convert T_A obtain into an adjacency matrix!

$$T_A = \begin{bmatrix} 0 & 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 6.9: Example with graph at fig 6.1 continued: Matrix T_A can be converted to adjacency matrix for transitive closure by changing value 2 into 1. All other matrix entries remain unchanged.

Next, let's arrange the matrix powers sum a little bit differently

$$T_A = A + A\,(A + A^2 + A^3 + \dots)$$

The series in the parenthesis, is again T_A, hence

$$T_A = A + A\,T_A$$

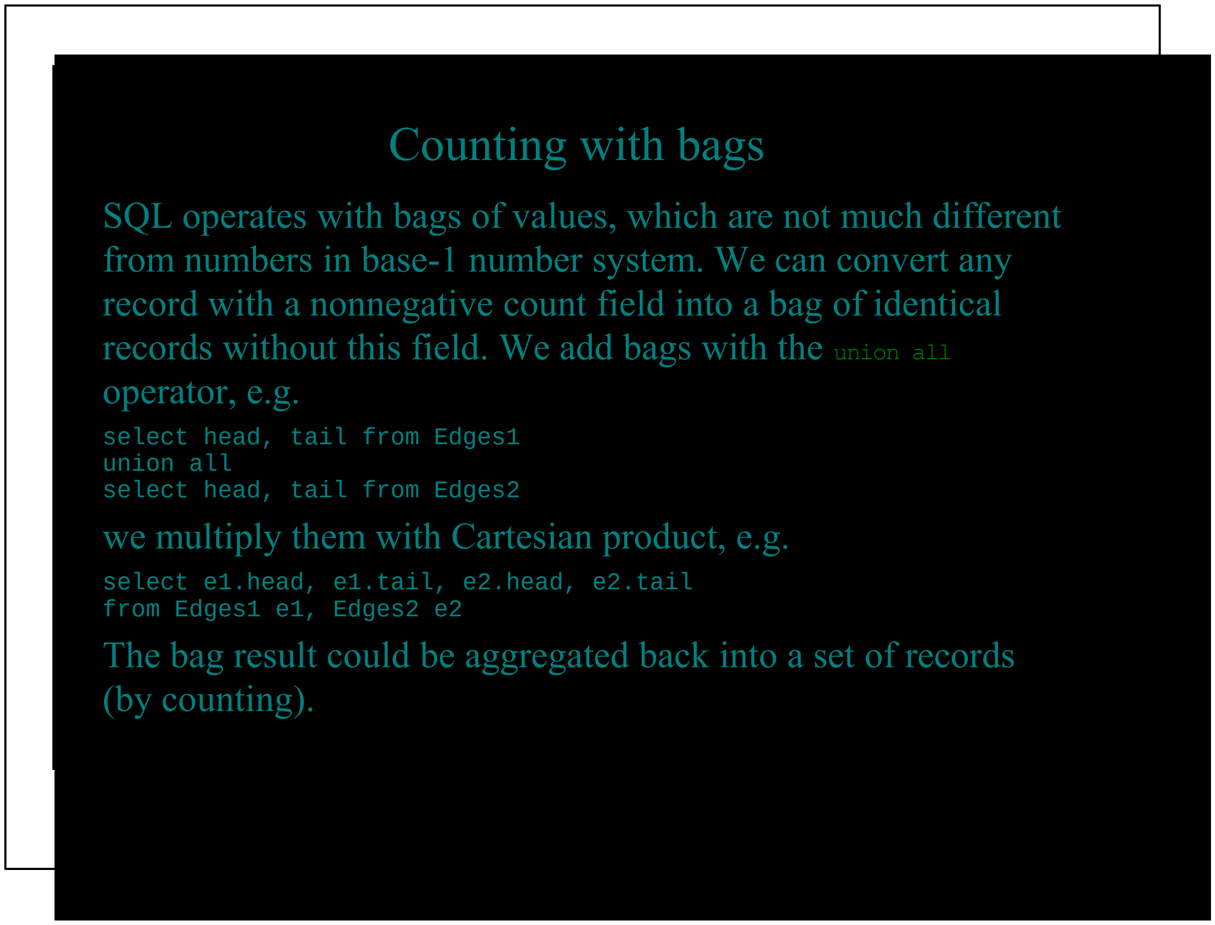
It is this expression, which we compare against the recursive `with` query in the beginning of the section. First, consider the product of matrices A and T_A. Matrix multiplication in SQL can expressed as

```
table A (
   i   integer, -- column i
   j   integer, -- column j
   val number   -- value of A(i,j)
);
table T_A (
   i   integer,
   j   integer,
   val number
);

select A.i AS i, T_A.j AS j, sum(A.val*T_A.val) AS val
from A, T
where A.j=T_A.i
group by A.i, T_A.j
```

The join between the A and T_A looks similar to the join between the `Edges` and `TransClosedEdges` in the recursive SQL query that we introduced in the beginning of the section. It is the aggregation part that appears to make them look alike.

This is not the only way to multiply the matrices in SQL, however. Remember that products and sums of adjacency matrices have nonnegative integer entries. Therefore, instead of having a single row `(i,j,val)`, we can have a bag of `val` identical rows of `(i,j)`. Schema design, once again, is important!



Counting with bags

SQL operates with bags of values, which are not much different from numbers in base-1 number system. We can convert any record with a nonnegative count field into a bag of identical records without this field. We add bags with the `union all` operator, e.g.

```
select head, tail from Edges1
union all
select head, tail from Edges2
```

we multiply them with Cartesian product, e.g.

```
select e1.head, e1.tail, e2.head, e2.tail
from Edges1 e1, Edges2 e2
```

The bag result could be aggregated back into a set of records (by counting).

In a bag design matrix multiplication reduces to

```
table A (
   i   integer, -- column i
   j   integer  -- column j
);
table T_A (
   i   integer,
   j   integer
);

select A.i AS i, T_A.j AS j
from A, T
where A.j=T_A.i
```

and the right side of the equation that defines transitive closure

$$T_A = A + A\ T_A$$

could be written in full in SQL as

```
select i, j
from A
union all
```

```
select A.i AS i, T_A.j AS j
from A, T
where A.j=T_A.i
```

After renaming variables appropriately, it becomes indistinguishable from the recursive view `TransClosedEdges` definition in the query:

```
with TransClosedEdges (tail, head) as
( select tail, head from Edges
  union all
  select e.tail, ee.head from Edges e, TransClosedEdges ee
  where e.head = ee.tail
)
select distinct * from TransClosedEdges
```

The outer query transforms a bag back into a set, because we are interested to know if there are paths from `node` tail to node `head`, rather than their exact number.

Given matrix interpretation of the recursive transitive closure query, we can translate matrix evaluation into SQL execution steps as follows

Step	Matrix	SQL
1: Initialization	$T_A := A$	`TransClosedEdges :=` `select tail, head from Edges`
2: Temporary product	$P := A\ T_A$	`P :=` `select e.tail, ee.head from Edges e, TransClosedEdges ee where e.head = ee.tail`

3: Conditional termination	$P = 0$?
	 return T_A
		`P = ∅ ?`
		 `return TransClosedEdges`

4: Iteration	$T_A := A + P$	`TransClosedEdges :=` `select tail, head from Edges` `union all` `P`

A reader who is already familiar with the recursive SQL from another source might have already learned an entirely different algorithm. SQL, however, is about declarative programming, and not about the algorithms. Any algorithm would do, as long as it produces correct result set. Recursive SQL is known to have many execution strategies: naïve, incremental (semi-naive), etc. Our algorithm could be coined as *matrix evaluation*.

There is a pitfall. The query would never terminate on graphs with cycles. Indeed, a cycle that passes through the node `i` would

eventually produce the edge `(tail=i, head=i)` in the `TransClosedEdges` relation. The `TransClosedEdges` relation never shrinks; therefore this record would remain in the `TransClosedEdges`. Then, the join at step 2 is never going to be empty.
One of the solutions to the cycle problem is suggested in the *DB2 Cookbook* by Graeme Birchall[2]. Recursive `with` construction is very powerful, since we may introduce extra columns, and those columns would be calculated recursively! One such column fits naturally into our query – it's the path expression:

```
with TransClosedEdges (tail, head) as
( select tail, head, tail||'.'||head AS path
  from Edges
  union all
  select e.tail, ee.head, e.tail||'.'||ee.path AS path
  from Edges e, TransClosedEdges ee
  where e.head = ee.tail
)
select distinct * from TransClosedEdges
```

Then Graeme goes on introducing a function called `LOCATE_BLOCK`, which could be used in the `where` clause as an indicator that current node is in the path already:

```
with TransClosedEdges (tail, head) as
( select tail, head, tail||'.'||head AS path
  from Edges
  union all
  select e.tail, ee.head, e.tail||'.'||ee.path AS path
  from Edges e, TransClosedEdges ee
  where e.head = ee.tail
  and LOCATE_BLOCK(e.head, path) = 0
)
select distinct * from TransClosedEdges
```

There is more satisfactory solution to the cycle problem, though. Both DB2 and SQL Server slightly deviated from ANSI SQL standard which demands a **union** in the recursive query definition. With the set semantics the transitive closure relation `TransClosedEdges` can't grow larger than a *complete* graph -- a graph where every pair of nodes is connected.

Let's complete the section with simple wisdom about performance. Clearly, the `Edges.head` column has to be indexed, if you want to scale this query to a hierarchy of any significant size.

[2] A similar solution has been implemented by Serge Rielau. If you are DB2 user, chances are that his CONNECT_BY_NOCYCLE function is a part of the database engine already. Otherwise, download it from IBM developerWorks website.

The same comment applies to Oracle solution that we discuss next.

Connect By

There are several ways to query transitive closure in Oracle, beginning with parsing `sys_connect_by_path` pseudo column, and ending with firing corellated `connect by` subquery:

```
select a.tail, b.head from Edges a, Edges b
where b.head in (
      select head from Edges
      connect by prior head = tail
      start with tail = a.tail
)
```

Without question the most satisfactory method (both aesthetically and performance-wise) is:

```
select connect_by_root(tail), tail
from Edges
connect by prior head = tail
```

This transitive closure query is succinct, but still the syntax could be improved. Apparently, the designers were preoccupied with tree problems, hence the `connect_by_root` pseudo column. There is no concept of the root node anywhere in the transitive closure problem scope. Likewise, the concept of the `prior` edge belongs to the solution space rather than is inherent to the transitive closure problem. Overall, transitive closure is defined **symmetrically** in terms of the `head` and `tail`, but the query above is skewed.

The cycle issue is no-brainer, as there is a dedicated keyword to take care of it

```
select connect_by_root(tail), tail
from Edges
connect by nocycle prior head = tail
```

Incremental Evaluation

Transitive closure enjoyed a lot of attention in the database research community. Proving impossibility of doing things in a certain way is one of the favorite theoretical topics. Not surprisingly, it was established very early that transitive closure can't be expressed by simple means of relational algebra (even enhanced with aggregation and grouping).

We have already seen the power of incremental evaluation idea in the database implementation world. Indexes and materialized views are the most familiar incremental evaluation structures. Dong et al[3] proved that transitive closure can be efficiently maintained via incremental evaluation system. In this section we explore Dong's approach, although with some deviation that would simplify the matter.

Let's revisit the transitive closure expression in terms of adjacency matrix

$$T = A + A^2 + A^3 + \ldots$$

From mathematical perspective this is quite a handsome series. It could be made even prettier if we consider the *identity matrix* – the matrix with ones on the main diagonal and zeros elsewhere. A common notation for the identity matrix is symbol 1. Perhaps the most important equality involving the identity matrix is

$$1 = A^0$$

There we see that the identity matrix literally begs to be added in front of the series for transitive closure

$$T = 1 + A + A^2 + A^3 + \ldots$$

Speaking graph language, what we did was including the loops at each node. Formally, T is now both reflexive and transitive closure.

Let see if can get any immediate benefits. Let's multiply[4] T by 1 – A

$$T(1 - A) = 1 + A + A^2 + A^3 + \ldots - (A + A^2 + A^3 + \ldots)$$

where all the (nonzero) powers of A at the right side cancel out

$$T(1 - A) = 1$$

Multiplying both sides by the inverse of 1 – A gives an explicit formula for transitive closure

$$T = (1 - A)^{-1}$$

[3] G. Dong, L. Libkin, J. Su and L. Wong. Maintaining the transitive closure of graphs in SQL. http://citeseer.ist.psu.edu/dong99maintaining.html

[4] Usually, when speaking matrix multiplication we have to be careful, since matrix product is not commutative, and clarify if we meant the left or right multiplication. In this case it doesn't matter, so we can afford a sloppy language.

Therefore, we might be able to calculate transitive closure (of directed acyclic graphs, at least), if we know how to invert matrices in SQL! Unfortunately, inverting matrices in SQL is difficult. The matrix approach, however, still shows its practical merit in the scope of incremental evaluation system.

In the incremental evaluation method we store the original graph together with another graph -- its transitive closure. Every time the original graph is updated (that is new edge is inserted or deleted) the transitive closure graph has to be changed as well, so that the information in both structures remains consistent. The problem of *transitive closure graph maintenance* essentially is finding an efficient SQL expression for the insertions and deletions in the transitive closure graph.

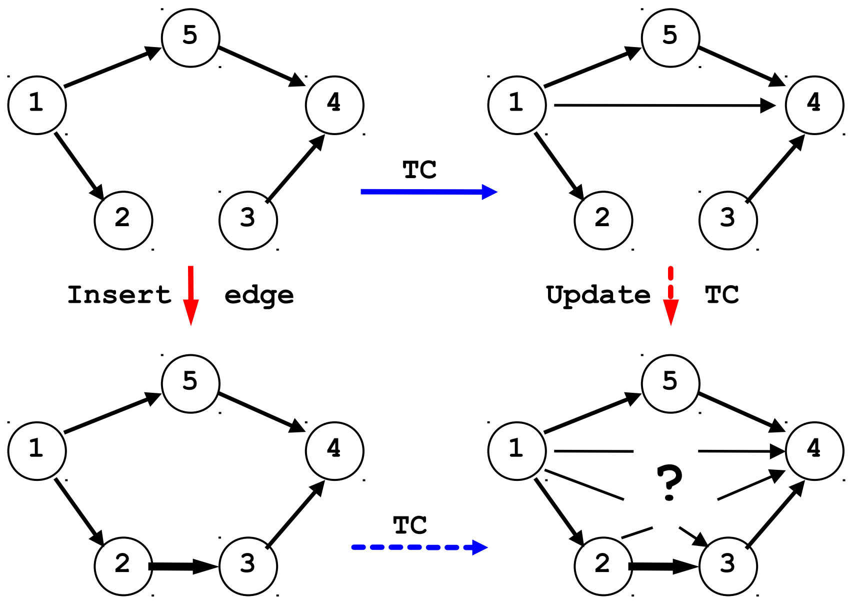


Figure 6.10: Transitive closure graph maintenance problem: Inserting the edge *(2,3)* into the graph at the top left, is expected to update the transitive closure structure at the top right accordingly.

Let's continue pursuing adjacency matrix approach. Inserting an edge `(x,y)` into the original graph produces a new graph with the adjacency matrix

$$A + S$$

that is the original matrix A incremented by S, the latter being a matrix with a single nonzero entry in the column `x`, row `y`. How would this increment of adjacency matrix affect transitive closure matrix? Well, let's calculate. The new transitive closure matrix is

$$1 + A + S + (A + S)^2 + (A + S)^3 + \ldots$$

Expanding powers into polynomials[5] we get

$$1 + (A + S) + (A^2 + A S + S A + S^2) +$$
$$+ (A^3 + A^2 S + A S A + A S^2 + S A^2 + S A S + S^2 A + S^3) + \ldots$$

Let's rearrange terms more suggestively

$$1 + A + A^2 + A^3 + \ldots + (S + A S + S A + A^2 S + A S A + S A^2 + \ldots) +$$
$$+ (S^2 + A S^2 + S A S + S^2 A + S^3 + \ldots)$$

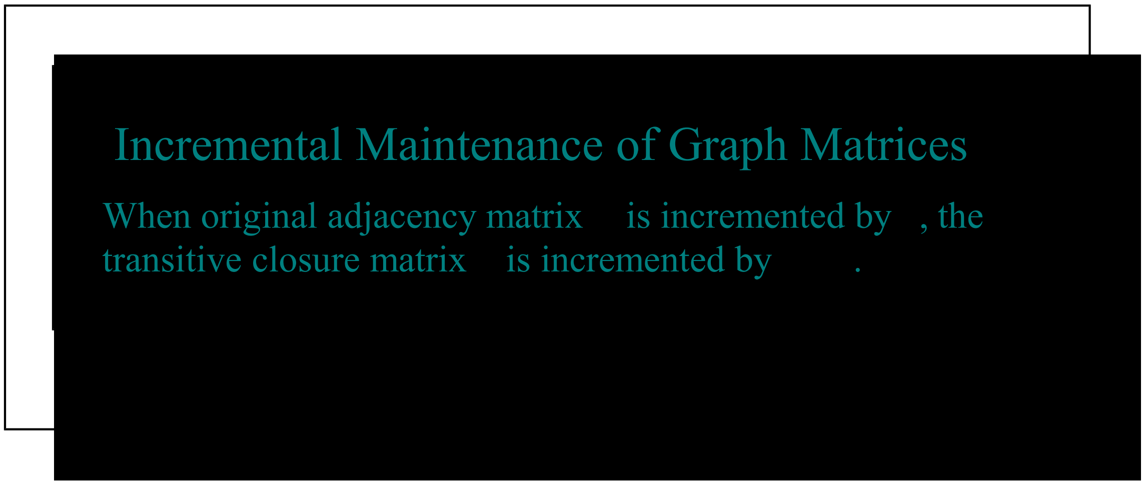
The first series is the familiar transitive closure matrix T of the original (not incremented) adjacency matrix A. The second series could be factored into

$$(1 + A + A^2 + A^3 + \ldots)\, S\, (1 + A + A^2 + A^3 + \ldots)$$

which reduces to T S T. The last series vanishes, if we limit the scope to directed acyclic graphs. Indeed, each term in that series is multiplied by S at least twice. In other words, each term in the series corresponds to a path in the graph that goes through the edge `(x,y)` twice, which implies a cycle.

Summarizing, the new transitive closure matrix reduces to

$$T + T S T$$



Incremental Maintenance of Graph Matrices

When original adjacency matrix   is incremented by , the transitive closure matrix   is incremented by   .

[5] Careful with matrix multiplication being non-commutative!

This is a very succinct result. Let's double check by comparing it with the query computing the transitive closure increment in Dong's paper:

```
SELECT *
FROM (
      SELECT VALUES (a, b)
    UNION
      SELECT Start = TC.Start, End = b
      FROM TC
      WHERE TC.End = a
    UNION
      SELECT Start = a, End = TC.End
      FROM TC
      WHERE b = TC.Start
    UNION
      SELECT Start = TC1.Start, End = TC2.End
      FROM TC AS TC1, TC AS TC2
      WHERE TC1.End = a AND TC2.Start = b
) AS T;

SELECT *
FROM TCNEW AS T
WHERE NOT EXISTS (SELECT *
                  FROM TC
                  WHERE TC.Start=T.Start AND TC.End=T.End)
INTO TEMP DELTA;

INSERT INTO TC
SELECT *
FROM DELTA;
```

The first part of the query, which is a union of 4 blocks, is supposed to correspond to our transitive closure matrix increment T S T. It appears that they are very dissimilar. How this can be?

Remember, however, that we conveniently decided to operate reflexive transitive closure matrices. If go back and represent T as 1+T', where T' is not reflexive anymore, then the increment matrix has to be written as (1+T') S (1+T') which can be expanded into

$$S + T'S + S\,T' + T'S\,T'$$

with 4 terms as in Dong's method.

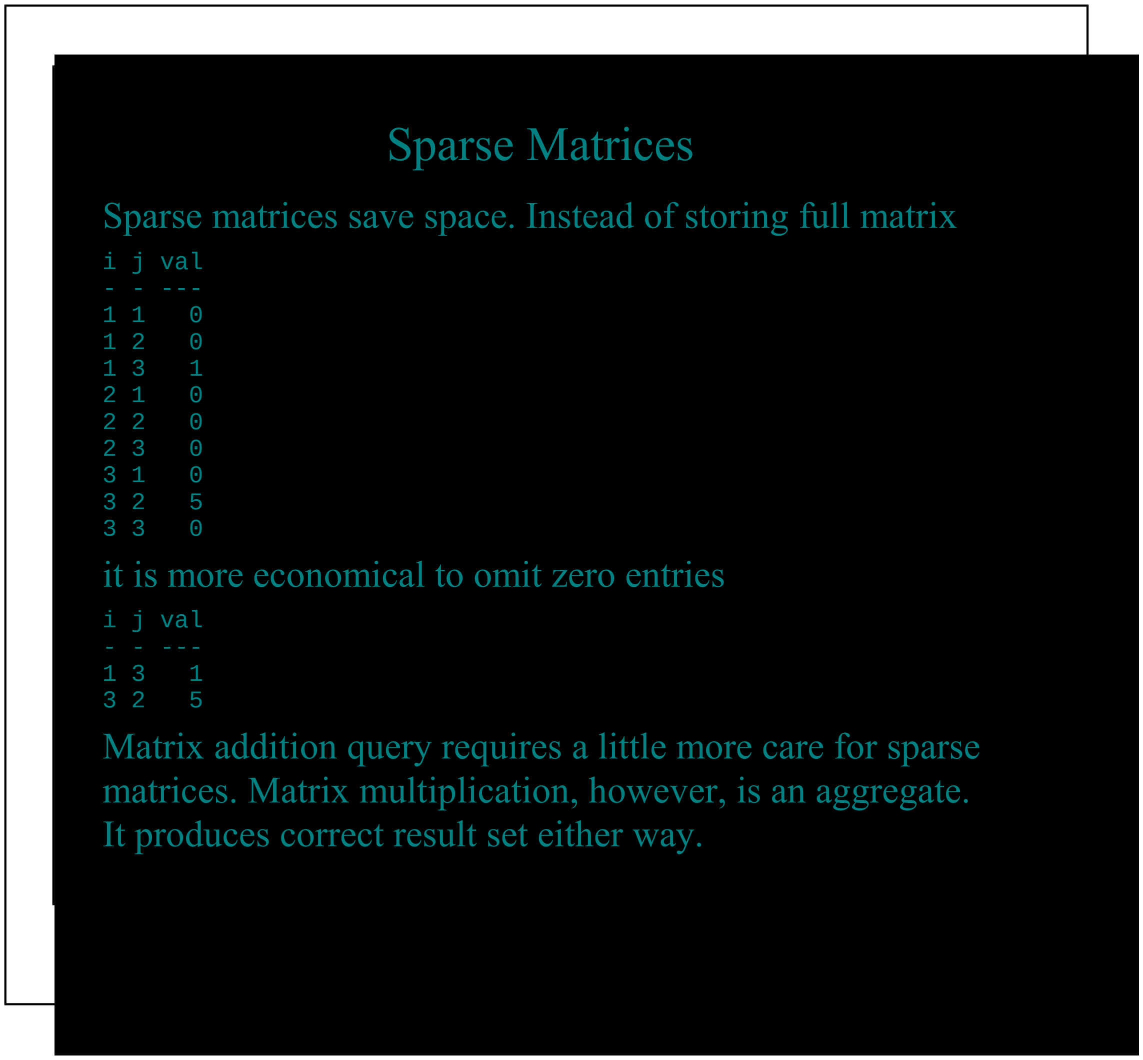
Now that we are confident with matrix solution, let's finalize it in terms of SQL. We have already seen that operating matrices of integers that count paths in the (directed acyclic) graph offers an exceptional clarity. Therefore, let's represent transitive closure relation directly after transitive closure matrix that we have investigated early

```
table TRC (      -- transitive, reflexive closure
   i   integer, -- tail
```

```
  j   integer, -- head
  val integer  -- weight
)
```

Once again, it's the reflexive property and additional information in the `val` column that distinguishes our method formally from Dong's.

There is an ambiguity. What about zero matrix entries? We have an option either to store it as `(i,j,0)`, or ignore such rows. The first option simplifies SQL that maintains table `TRC`. The second option provides natural means for compressing **sparse** matrices.



Sparse Matrices

Sparse matrices save space. Instead of storing full matrix

```
i j val
- - ---
1 1   0
1 2   0
1 3   1
2 1   0
2 2   0
2 3   0
3 1   0
3 2   5
3 3   0
```

it is more economical to omit zero entries

```
i j val
- - ---
1 3   1
3 2   5
```

Matrix addition query requires a little more care for sparse matrices. Matrix multiplication, however, is an aggregate. It produces correct result set either way.

Now all the ground work for transitive closure maintenance in SQL is complete. Inserting an edge `(x,y)` into adjacency graph has to trigger a conforming change in the transitive closure table `TRC`. The values in the `TRC.val` column should be incremented accordingly by the entries of the product of three matrices T S T.

We have already discussed how to write a product of two arbitrary matrices in SQL. A product of three matrices -- A B C -- can be written as a composition of two binary product operations (A B) C. Alternatively, we sum matrix elements at once

$$\sum_l \left(\sum_k a_{ik} b_{kl} \right) c_{lj} = \sum_{k,l} a_{ik} b_{kl} c_{lj}$$

which is easy to translate into SQL

```
select A.i AS i, C.j AS j, sum(A.val*B.val*C.val) AS val
from A, B, C
where A.j=B.i and B.j=C.i
group by A.i, C.j
```

Please note how naturally matrix associativity property goes along with relational join associativity.

Actually, we are after a simpler matrix product -- T S T. There no challenge adapting general case to our needs

```
select t1.i AS i, t2.j AS j, sum(t1.val*t2.val) AS val
from TRC t1, TRC t2
where t1.j = :x and t2.i = :y
group by t1.i, t2.j
```

If we have chosen the option of storing zero matrix entries, then the above query fits naturally into an update statement to the `TRC` table

```
update TRC
set val = (
   select val + sum(t1.val*t2.val)
   from TRC t1, TRC t2
   where t1.j = :x and t2.i = :y
   and t1.i = trc.i, t2.j = trc.j
   group by t1.i, t2.j
)
```

It is fascinating that the transitive closure table maintenance is solvable with a single update, but this answer is unrealistic for two reasons. First, the `TRC` table has to grow at some moment, and this issue is left out of scope. Second, from performance perspective updating (or trying to update) all the rows in the `TRC` table smells a disaster. We'd better have a good idea which rows require update, and formalize it within the `where` clause (which is entirely missing).

Therefore, let's materialize updates to the TRC table in a designated table `TRCDelta`

```
insert into TRCDelta
select t1.i AS i, t2.j AS j, sum(t1.val*t2.val) AS val
from TRC t1, TRC t2
where t1.j = :x and t2.i = :y
group by t1.i, t2.j
```

As we want to keep this table small, we can't afford to store matrix entries with zero values any longer. On the other hand, by just storing a sparse matrix stored in the `TRC` table, we automatically have the `TRCDelta` calculated as a sparse matrix either.

It is the `TRC` table update step that has to be adjusted. There are two cases:

1. All the rows in `TRCDelta` that don't match `TRC` rows have to be inserted

```
insert into TRC
select * from TRCDelta
where (i,j) not in (select i,j from TRC)
```

2. The rows that match have to increment their values

```
update TRC
set val = val + (select val from TRCDelta td
                 where td.i = trc.i and td.j = trc.j)
where (i,j) in (select i,j from TRCDelta)
```

This completes our program in the case when an edge is inserted. What about deletion? From matrix perspective the cases of inserting an edge and deleting it are **symmetric**. Following our earlier development we could use the same matrix S that corresponds to a single edge `(x,y)`, except that we have to subtract it everywhere. The final result in matrix form is transitive closure matrix T decremented by T S T.

In the first naïve update solution for the `TRC` table all that we have to do is reversing the sign

```
update TRC
set val = (
   select val – sum(t1.val*t2.val)
   from TRC t1, TRC t2
   where t1.j = :x and t2.i = :y
   and t1.i = trc.i, t2.j = trc.j
   group by t1.i, t2.j
)
```

Please note, that this symmetry is made possible because of the extra information that we store in the `TRC` table. Since we know how many paths are going from node `i` to node `j`, we can just subtract all the paths that are affected by the deletion of an edge `(x,y)`. In Dong's approach maintenance under deletion is more complicated.

Carrying over the solution to sparse matrices requires little insight. The `TRCDelta` table that stores the T S T matrix is calculated the same way as in the edge insertion scenario. Then, subtracting the T S T from T brings up the two possibilities:

1. An entry in the transitive closure matrix T is the same as corresponding entry in the T S T.
2. An entry in the transitive closure matrix T is bigger than corresponding entry in the T S T.

In the first case the entry in the difference matrix T – T S T is `0`. Therefore, all these entries have to be deleted

```
delete from TRC
where (i,j, val) in (select i,j, val from TRCDelta)
```

All the other entries have to be adjusted

```
update TRC
set val = val - (select val from TRCDelta td
                 where td.i = trc.i and td.j = trc.j)
where (i,j) in (select i,j from TRCDelta)
```

Now that we have several methods for transitive closure calculation/ maintenance, let's again return to applications. Perhaps the most significant problem that can be expressed in terms of transitive closure is aggregation on graphs.

Hierarchical Weighted Total

Before exploring aggregation on graphs, let's have a quick look to aggregation on trees. Aggregation on trees is much simpler and has a designated name: *hierarchical total*. A typical query is

Find combined salary of all the employees under (direct and indirect) supervision of King.

This query, however, has no inherent complexity, and effectively reduces to familiar task of finding a set all node's descendants.

In graph context, the hierarchical total query is commonly referred to as *bill of materials* (BOM). Consider bicycle parts assembly

Part	SubPart	Quantity
Bicycle	Wheel	2
Wheel	Reflector	2
Bicycle	Frame	1
Frame	Reflector[6]	2

Parts with the same name are assumed to be identical. They are modeled as nodes in an (acyclic directed) graph. The edges specify the construction flow:

1. Assemble each wheel from the required parts (including reflectors).
2. Assemble frame from the required components (including reflectors).
3. Assemble bicycle from the required parts (including frame and wheels).

Suppose we want to order the total list of parts for bike assembly. How do we calculate each part quantity? Unlike hierarchical total for a tree we can't just add the quantities, as they multiply along each path.

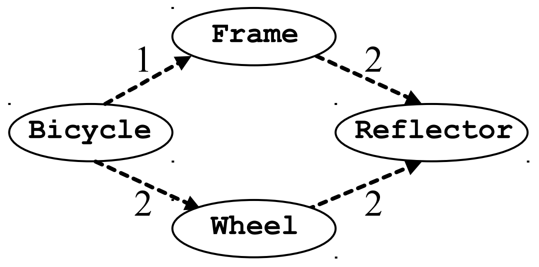
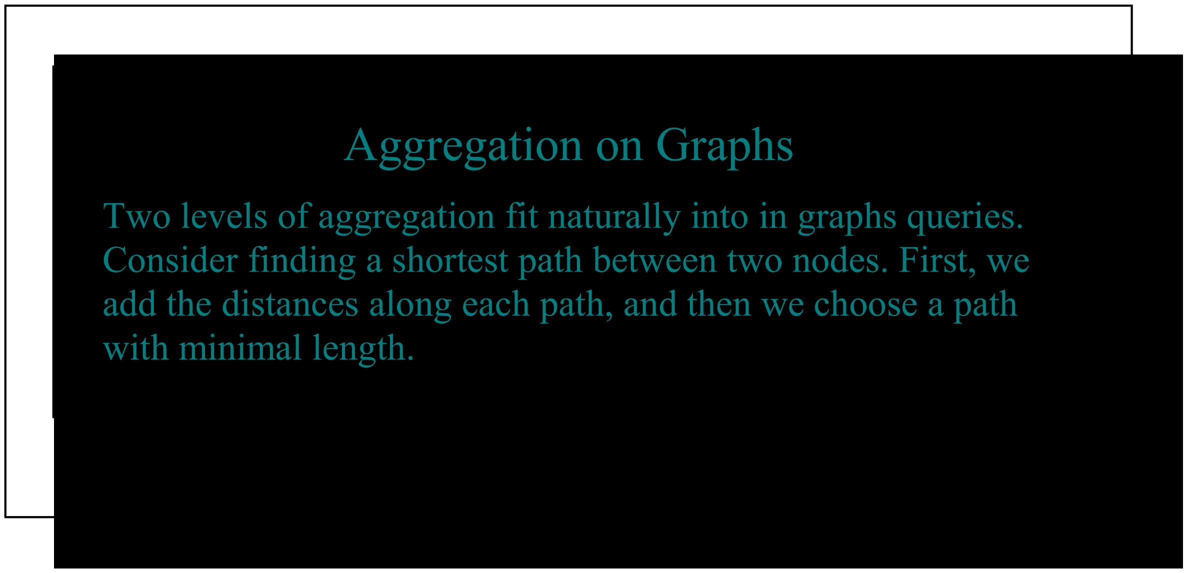


Figure 6.11: Bicycle assembly graph. The total number of reflector parts should be calculated as the sum of parts aggregated along each path. The path *Bicycle/Wheel/Reflector* contributes to 2*2=4

[6] Actually, both frame reflectors have different colors (red and silver). Also, they are different in shape and color from the reflectors that are mounted on wheels. To make example more interesting we assume all of them to be identical.

parts: bicycle has 2 wheels, each wheel has 2 reflectors. Likewise, the path */Bicycle/Frame/Reflector* contribute to 1*2=2 more parts.

Therefore, there are two levels of aggregation here, multiplication of the quantities along each path, and summation along each alternative path.



Aggregation on Graphs

Two levels of aggregation fit naturally into in graphs queries. Consider finding a shortest path between two nodes. First, we add the distances along each path, and then we choose a path with minimal length.

Double aggregation is not something unique to graph queries, however. Consider

Find the sum of the salaries grouped by department. Select maximum of them.

When expressing such a query in SQL, we accommodate the first sentence as an inner subquery inside the outer query corresponding to the second sentence

```
select max(salaryExpences) from (
   select deptno, sum(sal) salaryExpenses
   from Emp
   group by dept
)
```

Hierarchical weighted total query has the same structure. The first level of aggregation where we join edges into paths is analogous to the `group by` subquery from the salary expense query. Formally, it is a transitive closure, which is enhanced with additional aggregates, or *generalized transitive closure*.

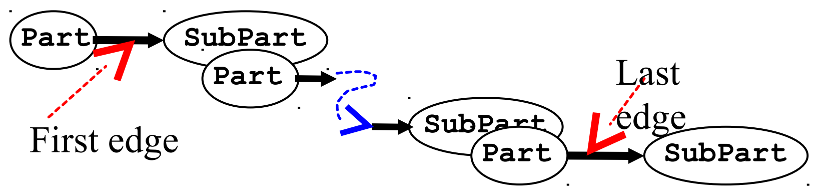


Figure 6.12: Generalized transitive closure. There are several aggregates naturally associated with each path: the first edge, the last edge, the path length, or any aggregate on the edge weights.

Let's suggest some **hypothetical** SQL syntax for generalized transitive closure

```
select distinct first(Part), last(SubPart), product (Quantity)
from AssemblyEdges
connect by prior Part = later SubPart
```

Unfamiliar syntax requires clarification:

- The `product` is a non-standard aggregate from chapter 4.
- The `first` and `last` refer to the first and last edges in the path, correspondingly. These aggregates are unique to **ordered** structures such as (directed) graphs.
- Although we didn't use it in the example, concatenating edge labels into a string is one more natural aggregation function, which is unique to graphs. The `list` aggregate function is a standard way to accommodates it.
- The `later` keyword is just a syntactic sugar fixing apparent asymmetry caused by the `prior` keyword.
- There is no `start with` clause, which is, in fact, redundant. It is an outer query where we'll restrict paths to those originated in the `'Bicycle'` node.

The generalized transitive closure query is enveloped with second level of aggregation, which is accomplished by standard means

```
select leaf, sum(factoredQuantity) from (
   select product(Quantity) factoredQuantity,
          first(Part) root, last(SubPart) leaf
   from AssemblyEdges
   connect by prior Part = later SubPart
) where root = 'Bicycle'
group by leaf
```

Enough theory, what do we do in real world to implement an aggregated weighted total query? Let's start with Oracle, because the proposed syntax for generalized transitive closure resembles Oracle `connect by`.

First, we have to be able to refer to the first and the last edges in the path

```
select connect_by_root(Part) root, SubPart leaf
from AssemblyEdges
connect by prior Part = SubPart
```

Unlike our fictional syntax, Oracle treats edges in the path asymmetrically. Any column from the `AssemblyEdges` table is assumed to (implicitly) refer to the last edge. This design dates back to version 7. The `connect_by_root` function referring to the first edge has been added in version 10. It is remarkable how this, essentially ad-hock design proved to be successful in practice.

Next, Oracle syntax has several more path aggregate functions:

1. `level` – the length of the path.
2. `sys_connect_by_path` – which is essentially the `list` aggregate in our fictitious syntax.
3. `connect_by_is_leaf` – an indicator if there is no paths which contain the current path as a prefix.
4. `connect_by_is_cycle` – an indicator if the first edge is adjacent to the last.

Unfortunately, we need an aggregate which is a product[7] of edge weights, and not a string which is a concatenation of weights produced by `sys_connect_by_path`. You might be tempted to hack a function which accepts a string of the edge weights, parses it, and returns the required aggregate value.

Nah, too easy! Can we solve the problem without coding a function (even a simple one)? Yes we can, although this solution would hardly be more efficient. The critical idea is representing any path in the graph as a concatenation of three paths:

- a prefix path from the start node to some intermediate node `i`

[7] Or sum. In general, any multiplicative problem can be converted into addition by applying logarithm function.

- a path consisting of a single weighted edge `(i,j)`
- a postfix path from the node `i` to the end node

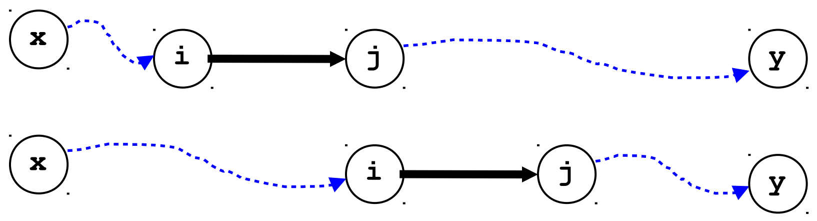


Figure 6.13: A path from node *x* to node *y* is a composition of the path from node *x* to node *i*, the edge *(i, j)*, and the path from *j* to *y*. The edge (i, j) can be positioned anywhere along the path from *x* to *y*.

Then, we freeze the path from `x` to `y`, while interpreting the edge `(i,j)` as a **variable**. All we need to do is aggregating the weights of all those edges.

Let's assemble the solution piece by piece. First, all the paths in the graphs are expressed as

```
with TClosure as (
   select distinct connect_by_root(Part) x, SubPart y,
       sys_connect_by_path('['||Part||','||SubPart||'>',' ') path
   from AssemblyEdges
   connect by nocycle Part=prior SubPart
   union
   select Part, Part, '' from AssemblyEdges
   union
   select SubPart, SubPart, '' from AssemblyEdges
) …
```

This is essentially reflexive transitive closure relation enhanced with the path column. Paths are strings of concatenated edges; each edge is sugarcoated into `'['||Part||','||SubPart||'>'`, which helps visual perception, but is inessential for the solution.

Next, we join two paths and intermediate edge together, and group by paths

```
… , PathQuantities as (
   select t1.x, t2.y,
       t1.p||' ['||e.Part||','||e.SubPart||'>'||t2.p,
       product(Quantity) Quantity
   from TClosure t1, AssemblyEdges e, TClosure t2
   where t1.y = e.Part and e.SubPart = t2.x
   group by t1.x, t2.y, t1.p||' ['||e.Part||','||e.SubPart||'>'||t2.p
) …
```

There we have all the paths with quantities aggregated along them. Let's group the paths by the first and last node in the path while adding the quantities

```
select x, y, sum(Quantity)
from PathQuantities
group by x, y
```

This query is almost final, as it needs only a minor touch: restricting the node `x` to `'Bicycle'` and interpreting the `y` column as a `Part` in the assembly

```
select y Part, sum(Quantity)
from PathQuantities
where x = 'Bicycle'
group by x, y
```

Let's move on to recursive SQL solution. It turns out to be quite satisfactory

```
with TCAssembly as (
   select Part, SubPart, Quantity AS factoredQuantity
   from AssemblyEdges
   where Part = 'Bicycle'
   union all
   select te.Part, e.SubPart, e.Quantity * te.factoredQuantity
   from TCAssembly te, AssemblyEdges e
   where te.SubPart = e.Part
) select SubPart, sum(Quantity) from TCAssembly
group by SubPart
```

The most important, it accommodated the inner aggregation with non-standard aggregate effortlessly! Second, the cycle detection issue that plagued recursive SQL in the section on transitive closure is not a problem for directed acyclic graphs.

Hierarchical total turns out to be surprisingly useful in the following two sections. Selecting all the subsets of items satisfying some aggregate criteria – *Basket generation* -- appears to have little with hierarchical total, at first. The other, rather unexpected application is *Comparing hierarchies*.

Generating Baskets

Given a set of items

```
select * from items;
```

NAME	PRICE
monitor	400
printer	200

notebook	800
camera	300
router	30
microwave	80

how much, say, $500 would buy? Even though the problem may sound unbelievably simple, please note that an answer involves finding all sets of items, not just item records that cost below $500. However, what those sets have to do with graph problems that we are focusing on in this chapter? Well, we don't have any method how to generate sets of items, but even if we had it, the second challenge would be aggregating price on these sets[8].

Let's reformulate the problem in terms of graphs with the very suggestive figure 6.15 as the basic idea.

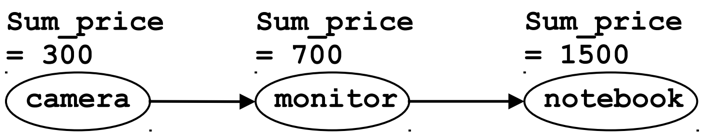
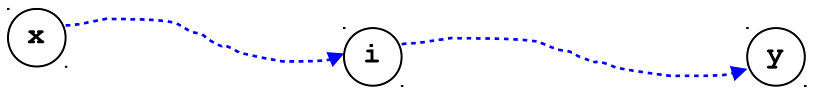


Figure 6.15: Price aggregation on the set *{camera, monitor, notebook}* can be viewed as a hierarchical total on a graph.

What criteria connect the items in the graph? Anything that doesn't list an item twice would do. For our purposes, we connect any two items if the first precedes lexicographically to the second one. Then, as in the hierarchical total section, our solution had to branch in order to accommodate differences between various platforms.

In oracle we adopt the idea from figure 6.13. Unlike previous section, however, where we aggregated values on edges, here we add together values on nodes. The path is decomposed into the three components as shown on figure 6.15.



[8] A familiar `group by` clause does aggregate on sets, but those sets are required to be **disjoint**.

Figure 6.15: A path from node *x* to node *y* is a composition of the path from node *x* to node *i*, the node *i*, and the path from *i* to *y*. Summation of the weights assigned to all such nodes *i* produces the aggregate weight of the path .

The transitive closure relation together with aggregation query formally is

```
with Sets as (
   select distinct connect_by_root(name) x, name y,
      case when connect_by_root(name) = name then
         ''
      else
         substr(sys_connect_by_path(name,','),
          instr(sys_connect_by_path(name,','),',',2))
      end p
   from items
   connect by nocycle name > prior name
), SetTotals as (
   select t1.x||t1.p||t2.p,
   sum(price) price_sum
   from Sets t1, Items i, Sets t2
   where t1.y = i.name and i.name = t2.x
   group by t1.x||t1.p||t2.p
)
```

Most of the attention here has been focused on making the path string to look right. Without this twisted `case` condition joining the paths according to the figure 6.15 would produce strings with duplicate items, e.g. `camera, camera, monitor`.

Now that we have the sets of items, we are one step from the final query

```
select * from SetTotals
where price_sum < 500
```

T1.X||T1.P||T2.P	PRICE SUM
microwave	80
microwave,printer,router	310
microwave,monitor	480
camera	300
microwave,printer	280
printer	200
camera,microwave,router	410
monitor,router	430
printer,router	230
microwave,router	110
camera,microwave	380
camera,router	330
monitor	400
router	30

Now that we examined the `connect by` based solution, let's see what recursive SQL can offer. The solution is embarrassingly simple:

```
with Sets (maxName, itemSet, sumPrice) as (
select name, name, price
union
select name, itemSet || ',' name, sumPrice + price
from Sets, T
where name > maxName
and sumPrice + price < 500
) select itemSet, sumPrice from Sets
```

Not only it is clearer, but it is also more efficient. Imagine a store with 50K items. Calculating all the subsets of the items is not a proposition that is expected to be completed in the remaining lifetime of the universe. The number of baskets with the aggregated cost limited by some modest amount, however, is reasonably small. Once again, applying predicates early is a good idea in general.

Comparing Hierarchies

Being able to navigate graph structures is only a part of the story. Comparing hierarchies is more challenging. First of all, by what criteria do we consider two hierarchical structures as equivalent? In particular, is the order of siblings important?

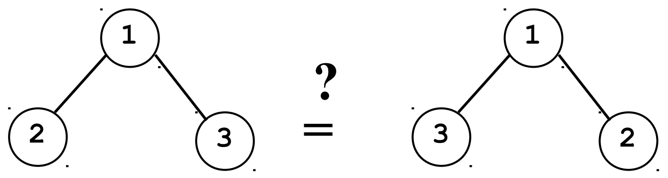


Figure 6.15: The first dilemma of tree comparison. Is reorganizing a tree by reordering siblings allowed?

Depending on the context the reader may lean to one or the other answer.

Likewise, can children become grandchildren?

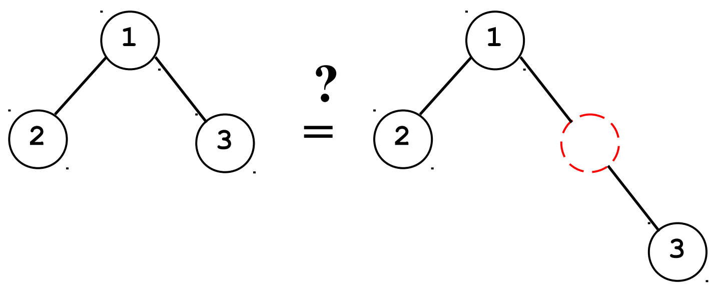


Figure 6.16: The second dilemma of tree comparison. Can a tree be restructured with an intermediate node in the chain of ancestors introduced?

Even though a reader may lean to conclusion that dummy intermediate nodes aren't allowed, it is easy to provide a counter example when ignoring certain types of intermediate nodes is imperative.

This ambiguity is discouraging. Yet, let's ignore it for a while and try to develop some basic understanding how trees can be compared. Looking desperately for a bright idea, I pulled the following entry from `thesaurus.com`

Main Entry:	compare
Synonyms:	analyze, approach, balance, bracket, collate, confront, consider, contemplate, correlate, divide, equal, examine, hang, inspect, juxtapose, match, match up, measure, observe, oppose, parallel, ponder, rival, scan, scrutinize, segregate, separate, set against, size up, study, touch, **weigh**

The last synonym – weigh – sounds promising. When we compare two things in the physical world, we measure them on some scale, or weigh them. Perhaps we can compare tree weights somehow?

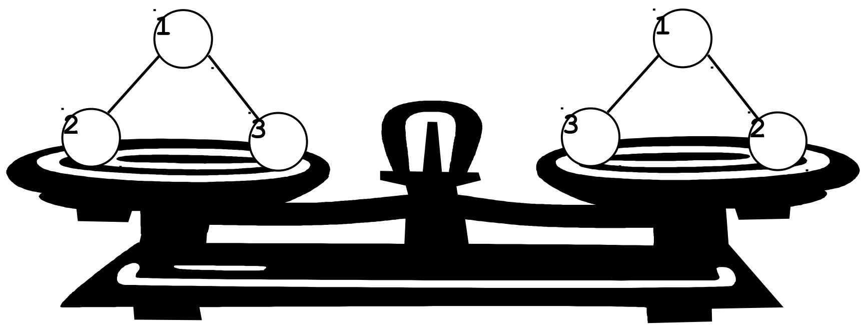


Figure 6.17: Weighting trees.

Let's revisit the idea of hierarchical total. When calculating total a raw node weight is augmented by the (augmented) weights of its children. Starting from the root node we descend recursively down to the leaves. The trick is to guess the right kind of aggregation, which accommodates both the tree structure and (unaugmented) node weights. Then, for all comparison purposes a tree can be identified with the augmented weight of its root node.

The easiest aggregation to try is just the sum of all tree node weights. This naïve method, however, fails to take into account differences in the tree structure.

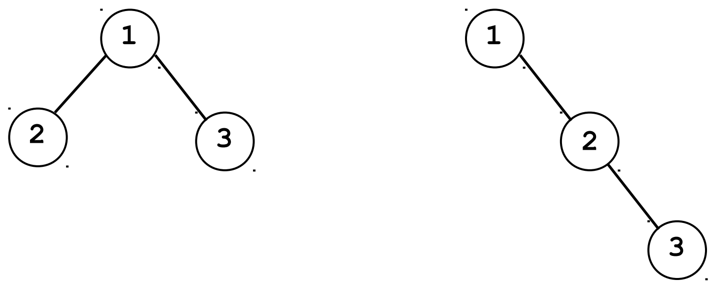


Figure 6.18: Adding node weights fails to distinguish tree structure. Both the tree on the left, and the tree on the right have the same weight 1+2+3 = 6

Joe Celko noticed that even though the weights at the root nodes are the same, the weights at the node `2` are different. What is needed is comparing the two sets of nodes element-by-element with their respective weights. Easy enough, store the weights in the `Table1`

AggregatedWeight
3
5
6

and `Table2`

AggregatedWeight
3
2

```
6
```

Then, check if the symmetric difference

```
select AggregatedWeight from Table1
minus
select AggregatedWeight from Table2
union
select AggregatedWeight from Table2
minus
select AggregatedWeight from Table1
```

is empty.

Still, the idea of comparing scalar values rather than sets of values is very appealing. Sets of integers are well known to be mapped bijectively to ordinary integers. This may be not a practical solution, but, at least, it supports our intuition that a scalar based tree comparison is possible.

Why the method of mapping integer sets into integers is unpractical? First, tree nodes can be labeled with values of datatypes other than integers. This is easily fixable, since the only important property of the label for tree comparison purposes is its identity. We can map any value of any datatype to integer, and in fact such a map is commonly known as a **hash function**. Second, much more difficult problem is that the mapping of sets integers to integers grows very fast. The range of computer integer numbers overflows pretty easily even for sets of moderate size. Hash function, however, provides quite satisfactory solution to range overflowing problem as well.

The basic premise of any hash-based method is that (unlikely) hash code collisions are tolerable. Given an object `a`, the chance that there exists another object `x≠a` such that `hash(a)=hash(x)` is considered as negligible. Likewise, for any objects `a`, `b`, `x` and `y` satisfying the equation `hash(a)+hash(b)=hash(x)+hash(y)` it must either follow that `x=a`, `y=b` or `x=b`, `y=a`. This could be contrasted to the ordinary addition where the equation `a+b=x+y` is too ambiguous to determine `x` and `y`.

Let's define hierarchical total query recursively. At each node labeled `p`, which has children c_1, c_2, …, c_n, we aggregate the following value

$$total(p) = hash(p+total(c_1)+total(c_2)+\ldots+total(c_n))$$

Recursive definition is the easiest to implement with – you guess it – recursive SQL. Consider employee hierarchy

```
table Employees ( -- tree nodes
   id   integer primary key,
   sal  integer
);

table Subordination ( -- tree edges
   mgr integer references Employees,
   emp integer references Employees
);
```

Then, hierarchical total query starts with the leaf nodes

```
select id, hash(id) AS total
from Employees
where id not in (select mgr from Subordination)
```

The recursive step mimics the recurrence definition

```
with NodeWeights as (
   select id, hash(id) AS total
   from Employees
   where id not in (select mgr from Subordination)
    union all
   select e.id, e.id+sum(hash(total))
   from Subordination s, Employees e, NodeWeights nw
   where s.mgr = e.id and s.emp = nw.id
   group by e.id
) select weight from NodeWeights
where id not in (select emp from Subordination)
```

After all hierarchy nodes are weighted, the outermost query selects the weight at the root node.

We have already have experienced difficulty fitting a recursive idea into `connect by` framework. Likewise, when studied nested sets and intervals we ignored recursive ideas altogether. Could hash based tree comparison method adapted to these contexts as well?

Let's invoke our familiar path encoding – after all, it reflects a tree structure. Admittedly, I don't know how to aggregate paths in a subtree of descendants, but I can suggest adding their hash values instead! More specifically, we can multiply a hash value of each node by a hash value of the node path, and sum them up the hierarchy. This definition is free of recursion and, therefore, is exactly what required for the hierarchical total method to work.

Now that we have methods for basic tree comparison, we can amend it to meet the exact tree equality specification. In the beginning of the section there were two tree equality dilemmas that we left hanging. In the first dilemma if reordering of siblings is allowed, then we do node summation exactly as we described. Otherwise, the node weights should be multiplied by the order numbers. For example, for the left tree in figure 6.14 the (augmented) weight of the root node has to be recalculated to become `1*1 + 2*1 + 3*2 = 7`. If the leaves `2` and `3` are swapped, then the weight at the root node has to change to `1*1 + 3*1 + 2*2 = 8`. A similar idea applies to the hash-based comparison method, where the hierarchical total formula becomes

$$\text{total}(p) = \text{hash}(p+1\cdot\text{total}(c_1)+ 2\cdot\text{total}(c_2)+\ldots+ n\cdot\text{total}(c_n))$$

Likewise, in the second dilemma if intermediate empty nodes in the tree structure are allowed, then Celko's method works as it is. Otherwise, we multiply each (unaugmented) node weight by the level. For example, for the left tree in figure 6.15 the (augmented) weight of the root node has to be recalculated to become `1*1 + 2*2 + 2*3 = 11`. For the right tree it is `1*1 + 2*2 + 2*0 + 3*3 = 14`. By contrast, the hash-based tree comparison method considers the tree structures in figure 6.15 as different.

Summary

- Learning transitive closure is essential for mastering queries on graphs.
- If you are implementing incrementally maintained transitive closure, then consider matrix method.
- Hierarchical weighted total query has two levels of aggregation.
- Hierarchical total can be leveraged for tree comparison.

Adjacency relation (tree edges; standalone,	Nested Sets	Materialized Path	Nested Intervals via Matrix encoding

or combined with the tree nodes)			
Have to use **proprietory** SQL extensions for finding ancestors and descendants; although the queries are efficient	Standard SQL	Standard SQL	Standard SQL
Finding descendants is relatively efficient (i.e. proportional to the size of the subtree)	Finding descendants is easy and relatively efficient (i.e. proportional to the size of the subtree)	Finding descendants is easy and relatively efficient (i.e. proportional to the size of the subtree)	Same as MP: Finding descendants is easy and relatively efficient
Finding ancestors is efficient	Finding ancestors is easy but inefficient	Finding ancestors is tricky but efficient	Same as MP: Finding ancestors is tricky but efficient
Finding node's children is trivial	Finding node's children as all the descendants restricted to the next level is inefficient (e.g. consider root node)	Finding node's children as descendants on next level is inefficient	Same as AR: Finding node's children is trivial
Finding node's parent is trivial	Finding node's parent as ancestor on the previous level Is inefficient due to inefficiency of ancestors search	Finding node's parent as ancestor on the previous level is efficient	Same as AR: Finding node's parent is trivial
Aggregate queries are relatively efficient (i.e. proportional to the size of the subtree)	Aggregate queries are relatively efficient (except counting, which is super fast)!	Aggregate queries are relatively efficient (i.e. proportional to the size of the subtree)	Aggregate queries are relatively efficient (i.e. proportional to the size of the subtree)
Tree reorganization is very simple	Tree reorganization is hard	Tree reorganization is easy	Tree reorganization is easy (but not as simple as in AR)

Figure 6.18: Feature matrix of different hierarchy methods

Exercises

1. For many practical problems the *forest* data structure is more natural than tree. A forest is a set of disjoint trees, and it's a simpler concept than tree, because it requires only two constraints. Which ones?

2. In graph theory tree is defined as a connected acyclic graph. Both concepts – tree and graph – are different from what we used in this chapter. In graph theory graphs are undirected, and trees are not rooted. Does graph theory perspective makes defining tree constraints easier?

3. Adapt Graeme's cycle detection function `LOCATE_BLOCK` implementation to handle transitive closure for graph with cycles.

4. Write matrix summation query `A+B` for sparse matrices `A` and `B`.

5. Some RDBMS offer a syntactic shortcut for combining `insert` and `update` into a single SQL command – `merge`. Combine the the `TRC` table maintenance operations together. Does the solution feel natural?

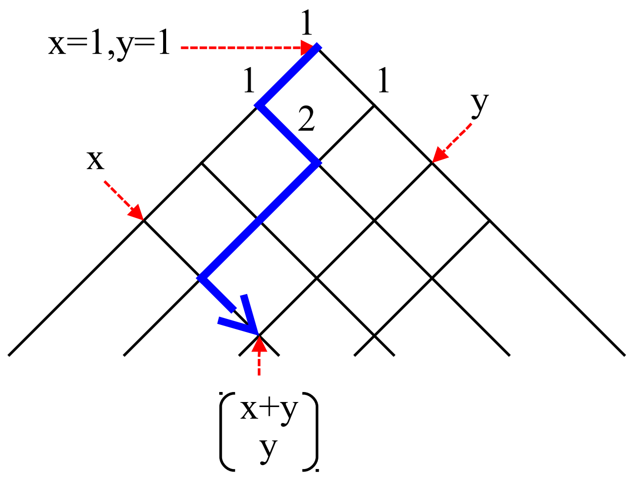
6. In the transitive closure maintenance under deletion we could have updated the `TRC` table first, and then wipe out all zero entries. Write SQL commands implementing this idea.

7. Implement a function that accepts a string of numbers separated with comma and returns the product of them. Write the hierarchical weighted total query that leverages it.

8. Write a query that finds the longest path between two nodes in a graph.

9. (Binomial coefficients continued). Any `x+y choose y` entry in the Pascal triangle is a number of paths from the node `x=1,y=1` to the node `x,y` (see the lattice picture below). Write a

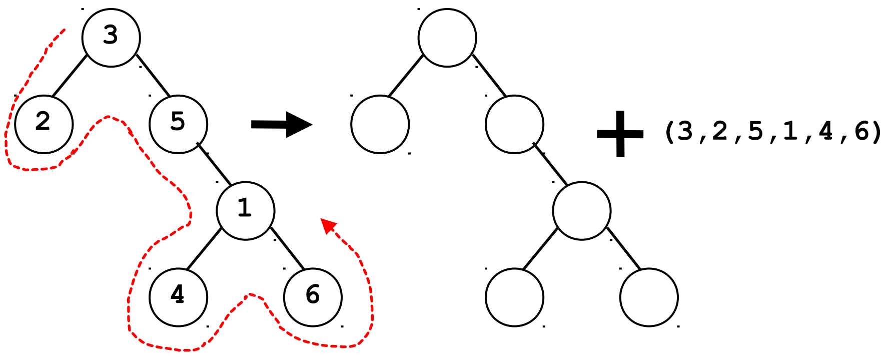
hierarchical total query that calculates binomial coefficient. Hint: build a lattice graph from the two sets of integers.



10. Applying predicates early is usually a good idea. In many cases no matter how we write the query, optimizer would be able to push the predicate. A query block with the `connect by` clause is an exception. Reengineer the `connect by` solution for the hierarchical total query so that the root node is restricted to `'Bicycle'` as early as possible.

11. Implement the hash based path encoded tree comparison idea.

12. A labeled tree is determined by an unlabeled tree ("the shape")



and by the sequence of node labels where nodes are traversed in some fixed order (say, preorder). An unlabeled tree is just a system of nested parenthesis, e.g. `(()((()())))`. Develop this fuzzy idea into a complete tree comparison method.

13. *Topological sort* of a DAG is a linear ordering of its nodes where each `head` node comes before all `tail` nodes. This definition is nondeterministic: for the graph at figure 6.1 both `1,2,3,5,4` and `1,2,5,3,4` are valid topological sorts. It is impossible to express nondeterministic queries in SQL. Yet, it is easy to make the problem deterministic, requiring to find lexicographically smallest sequence among all possible topological sorts. Write topological sort in SQL.

    Hint: It is quite challenging to follow the above definition of topological sort, since it involves a set of all legitimate topological sorts. Instead, approach the problem recursively. Find all the `head` nodes that are not listed as `tail` nodes. Enumerate them in the increasing order. Reduce the problem to smaller problem of finding topological sort of the graph with all the edges that exclude the `head` nodes, which were considered at the last recursion step.