Thinking in Sets: How to Program in SQL

By

Joe Celko

Copyright 2007

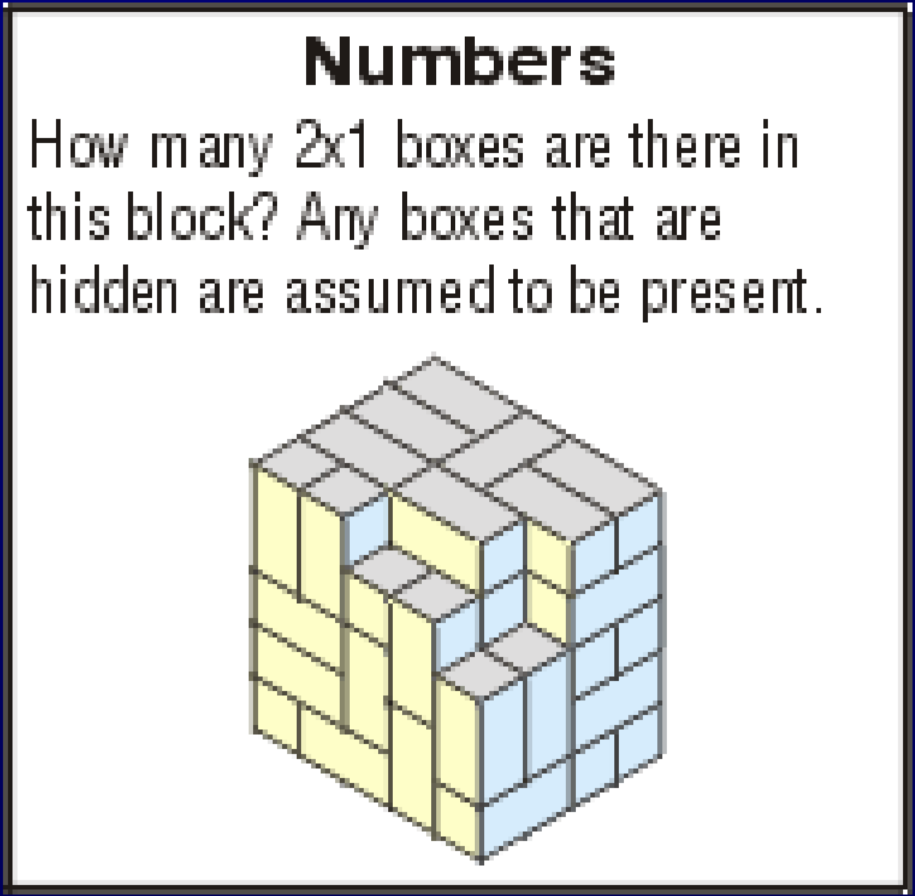
Joe Celko - Articles

- Member of ANSI X3H2
- SQL for Smarties - DBMS Magazine
- Celko on SQL - DBP&D
- SQL Puzzle - Boxes & Arrows
- DBMS/Report - Systems Integration
- WATCOM SQL Column - PBDJ
- Celko on Software - COMPUTING(UK)
- Celko - Intelligent Enterprise
- SELECT FROM Austin - DB/M (Netherlands)
- www.DBAzine.com
- 800+ odd articles

Joe Celko - Books

- JOE CELKO'S SQL FOR SMARTIES
- INSTANT SQL (Wrox Press)
- JOE CELKO'S SQL PUZZLES & ANSWERS
- DATA & DATABASES
- TREES & HIERARCHIES IN SQL
- SQL PROGRAMMING STYLE
- ANALYTICS & OLAP IN SQL
- THINKING IN SETS (2007)

Think in Aggregate



Numbers

How many 2x1 boxes are there in this block? Any boxes that are hidden are assumed to be present.

Joe Celko – Puzzle #1

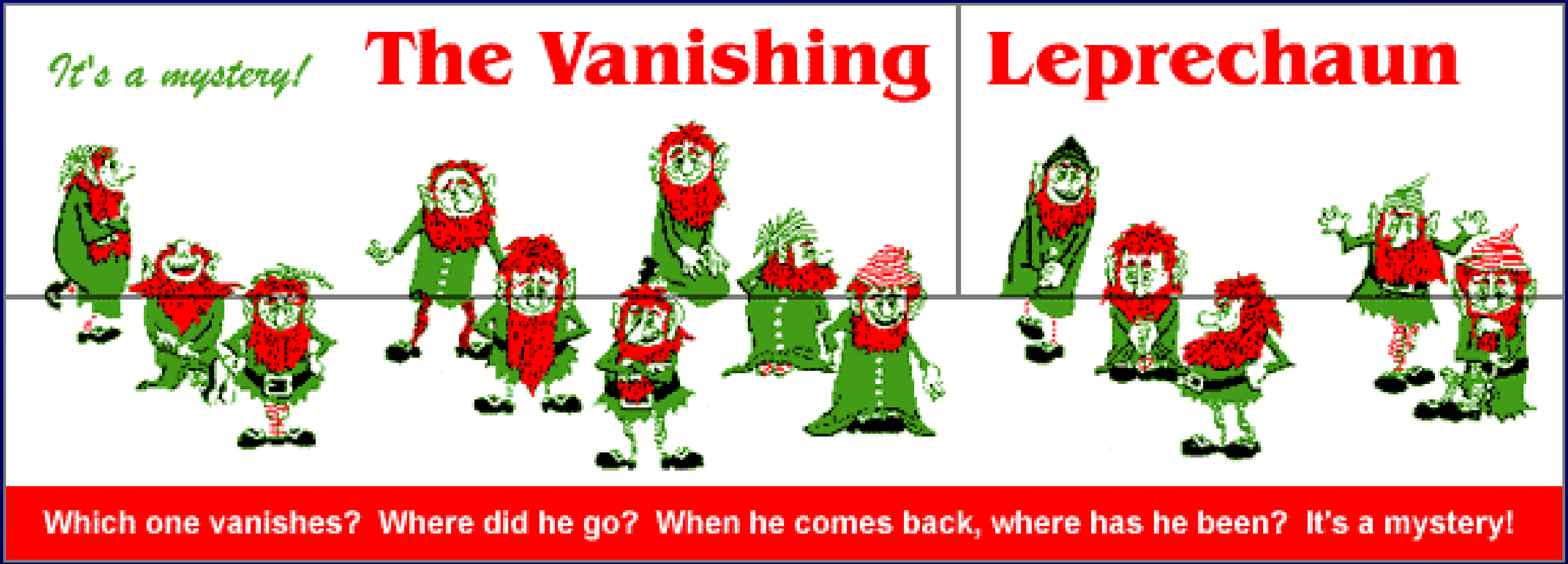
- Do not try to figure out the details ! Look at the whole

- The completed block is 4 by 5 by 5 units, so its total volume is 100 cubic units.

- It is missing 3 blocks on the corner, which are 2 cubic units each = 6 cubic units
- 100 - 6 = 94 cubic units
- 94/2 = 47 blocks

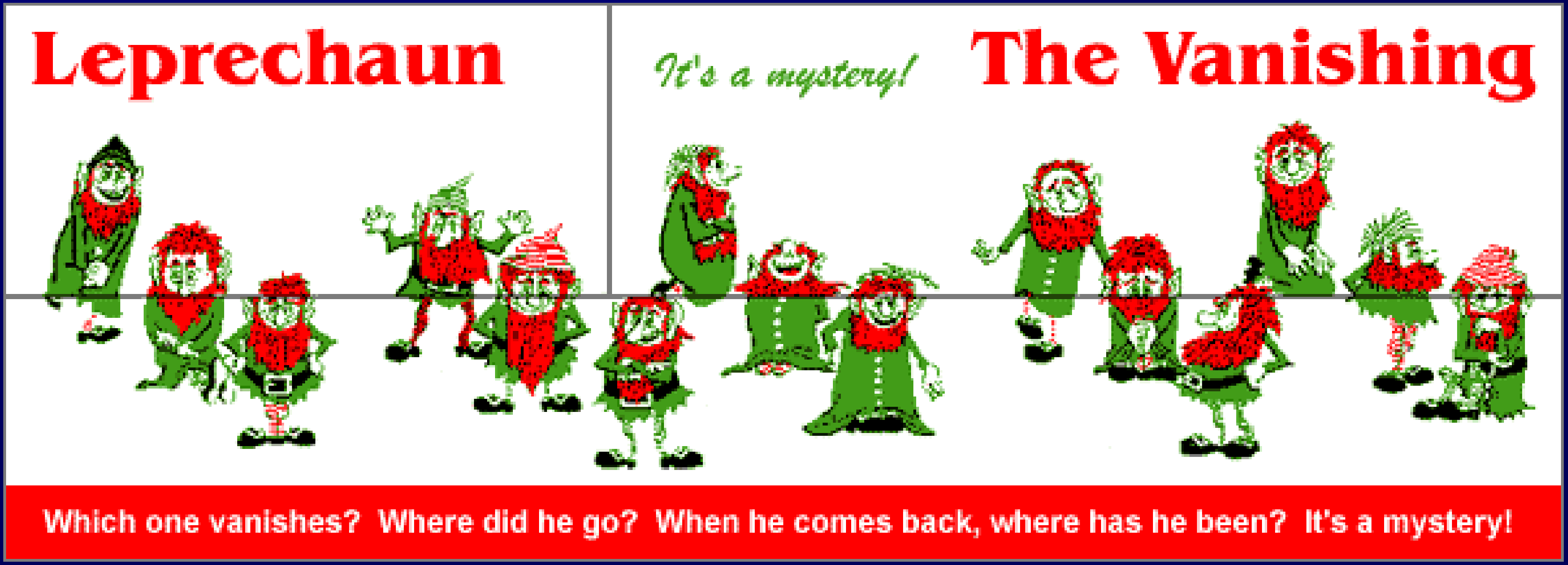
What makes an Entity?

- Puzzle is a jigsaw in three parts and shows 14 leprechauns


It's a mystery!
The Vanishing Leprechaun
Which one vanishes? Where did he go? When he comes back, where has he been? It's a mystery!

What makes an Entity? -2

- Swap the top two pieces and you have 15 leprechauns

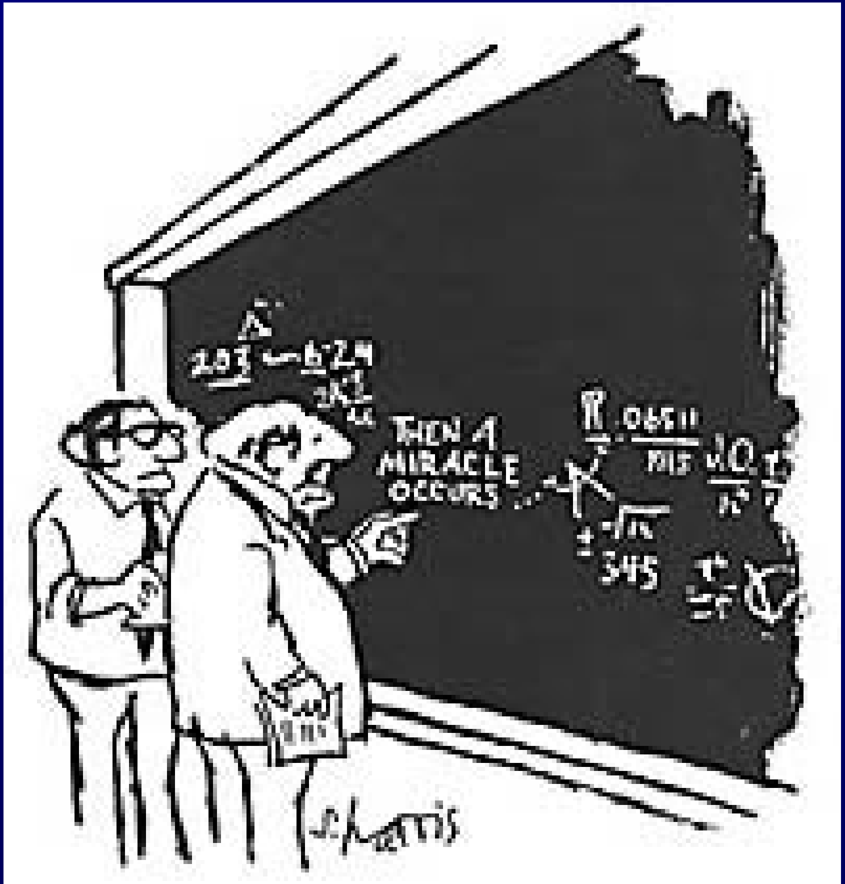

Leprechaun
It's a mystery!
The Vanishing
Which one vanishes? Where did he go? When he comes back, where has he been? It's a mystery!

What makes an Entity? -3

- This is a false question
  - Each set of Leprechauns is a totally different aggregation of “leprechaun parts” .
  - It depends on not having a clear rule for knowing what makes a leprechaun.
  - Break a piece of chalk in half and you have two pieces of chalk!

- In SQL, a table models one and only one kind of thing
- Each kind of thing has one and only one table

Don’t Sweat Details -1



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Don’t Sweat Details -2

- Try to start with a high level solution to your problem
- Once you know the approach is going to work, then add the details
- But when you get to the details, be exact
  - I have a whole lecture on scales, measurements and the design of encoding schemes

SQL is Not Procedural

- SQL is NOT a procedural language
  - All computable problems can be solved in a non-procedural language like SQL - big fat hairy deal!
- SQL works best with whole tables, not with single rows
  - Tables have no order
  - All relations are shown as scalar values within a column
- You tell it what you want; it figures out how to get it
- Good specifications are hard to write!

SQL is Not Computational

- SQL is NOT a computational language
- Standard SQL has only four function math; everything else was a vendor extension until SQL-92
- Rounding and truncation are implementation defined
- You really ought to pass the data to a report writer or statistical tool for any fancy calculations.

Principles

- Think in sets, not single values and rows
- A good data model will save you a lot of pain.
- Much procedural code can be moved inside a query with the CASE expression and COALESCE() function
- GROUP BY is very useful
- WHERE and HAVING are not the same thing
- Algebra is important!
- Logic is very important!

Mother Celko’s Heuristics - 1

- Do not draw boxes and arrows
  - the arrows imply a flow of something
  - flow means process
  - process means procedures

- Draw circles -- set diagrams
  - Sets can be nested, disjoint, overlapping, etc.
  - These are relationships

- Test for empty sets, NULLs and special values

- Develop with small sets, but test with a large sets

Mother Celko's Heuristics - 2

- Do not use temp tables
  - they usually hold steps in a process
  - process means procedure
  - You are mimicking magnetic tape files

- Do use derived tables
  - They are part of the query and the optimizer can get to them

- Nesting of functions is good and you can do it more than **you think**
  - **Talk to a LISP programmer**

Sequence -1

- There are no FOR-NEXT loops in SQL
- Instead of doing things one at a time, you have to do them all at once, in a set, in parallel
- To get a subset of integers, first you need to have a set of integers
- Build an auxiliary table of sequential numbers from 1 to some value (n)

Sequence -2

- Sequence tables should start with one and not with zero – it’s important for use with COUNT(*)

- Other columns in the table can be
  - –Random numbers
  - –Number words (ordinal and cardinal)
  - –Complicated functions

Sequence -3

- Sequence can be used as a loop replacement
- Example: given a string with a comma separated list, cut it into integers:

‘12,345,99,765’ becomes a column in a table

- Procedural approach:
  - parse the string left to right until you hit a comma
  - slice off the substring to the left of the comma
  - cast that substring as an integer
  - parse past comma
  - loop and lop until the string is empty

Sequence -4

- Non-procedural approach
  - find all the commas at once
  - find all the digits bracketed by pairs of sequential commas, adding one on each end
  - convert that set of substrings into integers as a set

- Hint: first find al the commas

```
SELECT  I1.keycol, S1.seq
  FROM InputStrings AS I1,
            Sequence AS S1
 WHERE SUBSTRING (‘,’ || instring || ‘,’ FROM S1.seq FOR
   1) = ‘,’;
```

- Now find the pairs of Commas

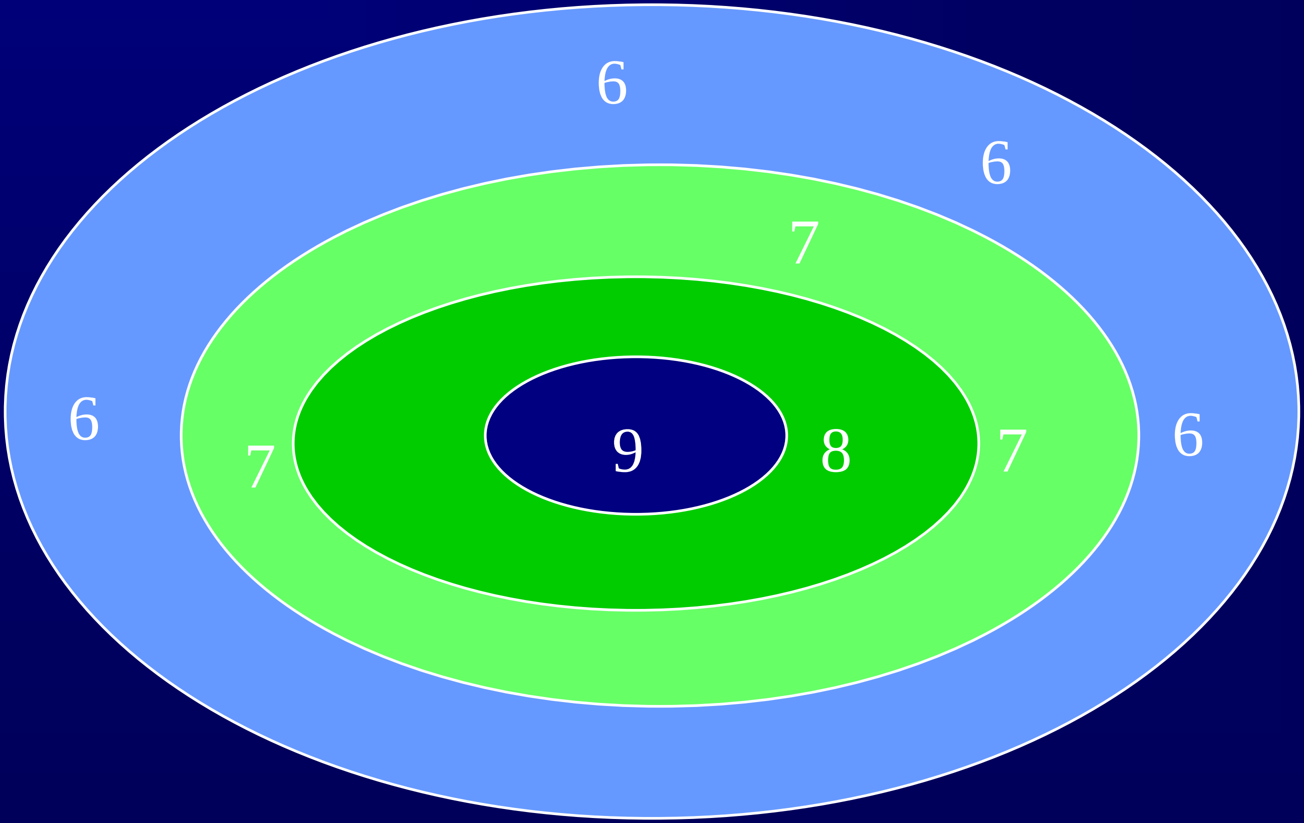
Sequence -5

```
●   SELECT I1.keycol,
      CAST (SUBSTRING (‘,’ || instring || ‘,’ FROM S1.seq +1
                        FOR S2.seq - S1.seq -1) AS INTEGER)
 FROM InputStrings AS I1,
        Sequence AS S1,  Sequence AS S2
WHERE SUBSTRING (‘,’ || instring || ‘,’ FROM S1.seq FOR 1)
  = ‘,’
 AND SUBSTRING (‘,’ || instring || ‘,’ FROM S2.seq FOR 1)
  = ‘,’
 AND S1.seq < S2.seq
 AND S2.seq = (SELECT MIN(S3.seq)
                  FROM  Sequence AS  S3
                 WHERE S1.seq < S3.seq);
```

Subsets - 1

- Not everything works on equality
- Less than, greater than and BETWEEN define subsets nested within a larger set
- This sort of query usually involves a self-join where one copy of the tables defines the elements of the subset and the other defines the boundary of the subset

Subsets - 2



Top(n) Values

- One version exists in Microsoft ACCESS and SQL Server
  - Those implementations use an ORDER BY clause and hide file sorting under the covers
- This is best done procedurally with the Partition routine from QuickSort
- The MAX() and MIN() are okay because they return a scalar value
- TOP(n) returns a set of rows, so it is not a function

Top (n) Values -2

- Procedural approach:
  - Sort the file in descending order
  - return the top (n) of them with a loop
- Problems: the spec is bad
  - How do you handle multiple copies of a value?
  - How do you handle exactly (n) values?
  - How do you handle less than (n) values?

Top (n) Values - 3

- Subset approach:
  - Decide if ties count or not; this is the pure set model versus SQL’s multi-set model
  - Find the subset with (n) or fewer members whose values are equal to the (n) highest values in the entire set
  - Use one copy of the table as the elements of the subset and one to establish the boundary of it.

Top (n) Values - 4

-
  ```
  SELECT DISTINCT E1.salary
   FROM Employees AS E1  -- elements
  WHERE :n -- n is parameter
         > (SELECT COUNT(*)
              FROM Employees AS E2 -- boundary
            WHERE E1.salary > E2.salary);
  ```
- Use > or >= , depending on where you put the boundary in relation to the elements.
- Use SELECT or SELECT DISTINCT, depending on how you want to count elements
- Use COUINT(*) or COUNT(DISTINCT <col>), depending on how you want to count NULL elements

Top (n) Values - 5

- An equivalent version can also be done with a self-join and a GROUP BY clause
- SELECT E1.salary

```
    FROM Personnel AS E1, Personnel AS E2
 WHERE E1.salary < E2.salary
  GROUP BY E1.salary  -- boundary value
HAVING COUNT(DISTINCT E2.salary) < :n;
```

- The same possible versions of the query exist here

Top (n) Values - 6

- You can also use the SQL-99 OLAP functions

```
SELECT *
  FROM (SELECT Foo.*,
                        ROW_NUMBER()
                        OVER(ORDER BY foo_key) AS row_nbr
             FROM Foo) AS F
 WHERE row_nbr < (:n);
```

- You can also use the RANK() and DENSE_RANK() functions to handle duplicates.

Relational Division - 1

- Relational division is easier to explain with an example. We have a table of pilots and the planes they can fly (dividend); we have a table of planes in the hanger (divisor); we want the names of the pilots who can fly every plane (quotient) in the hanger.

- CREATE TABLE PilotSkills

```
(pilot CHAR(15) NOT NULL,
plane CHAR(15) NOT NULL);
```

- CREATE TABLE Hanger(plane CHAR(15));

Relational Division -2

- The standard solution (i.e. Chris Date) is to find the pilots for whom there does not exist a plane in the hanger for which they have no skills.

-
  ```
  SELECT DISTINCT pilot FROM PilotSkills AS PS1
   WHERE NOT EXISTS
               (SELECT * FROM Hanger
                WHERE NOT EXISTS
                          (SELECT * FROM PilotSkills AS PS2
                           WHERE (PS1.pilot = PS2.pilot)
                             AND (PS2.plane = Hanger.plane)));
  ```

Relational Division - 3

- Imagine that each pilot gets a set of stickers that he pastes to each plane in the hanger he can fly. If the number of planes in the hanger is the same as the number of stickers he used, then he can fly all the planes in the hanger.

-
  ```
  SELECT Pilot
      FROM PilotSkills AS PS1, Hanger AS H1
    WHERE PS1.plane = H1.plane
     GROUP BY PS1.pilot
  HAVING COUNT(PS1.plane)
                = (SELECT COUNT(*) FROM Hanger)
  ```

Relational Division - 4

- The SQL-92 set difference operator, EXCEPT, can be used to write a version of relational division.

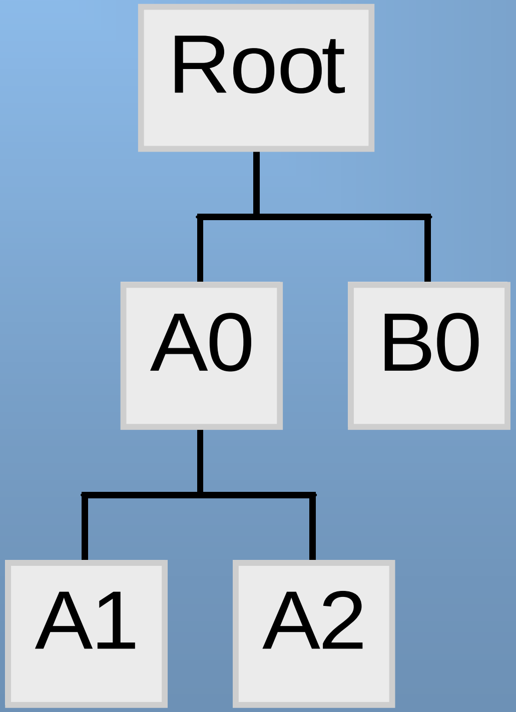
- SELECT Pilot

```
 SELECT Pilot
   FROM PilotSkills AS P1
WHERE NOT EXISTS
          (SELECT plane FROM Hanger
            EXCEPT
           SELECT plane FROM PilotSkills AS P2
          WHERE P1.pilot = P2.pilot);
```

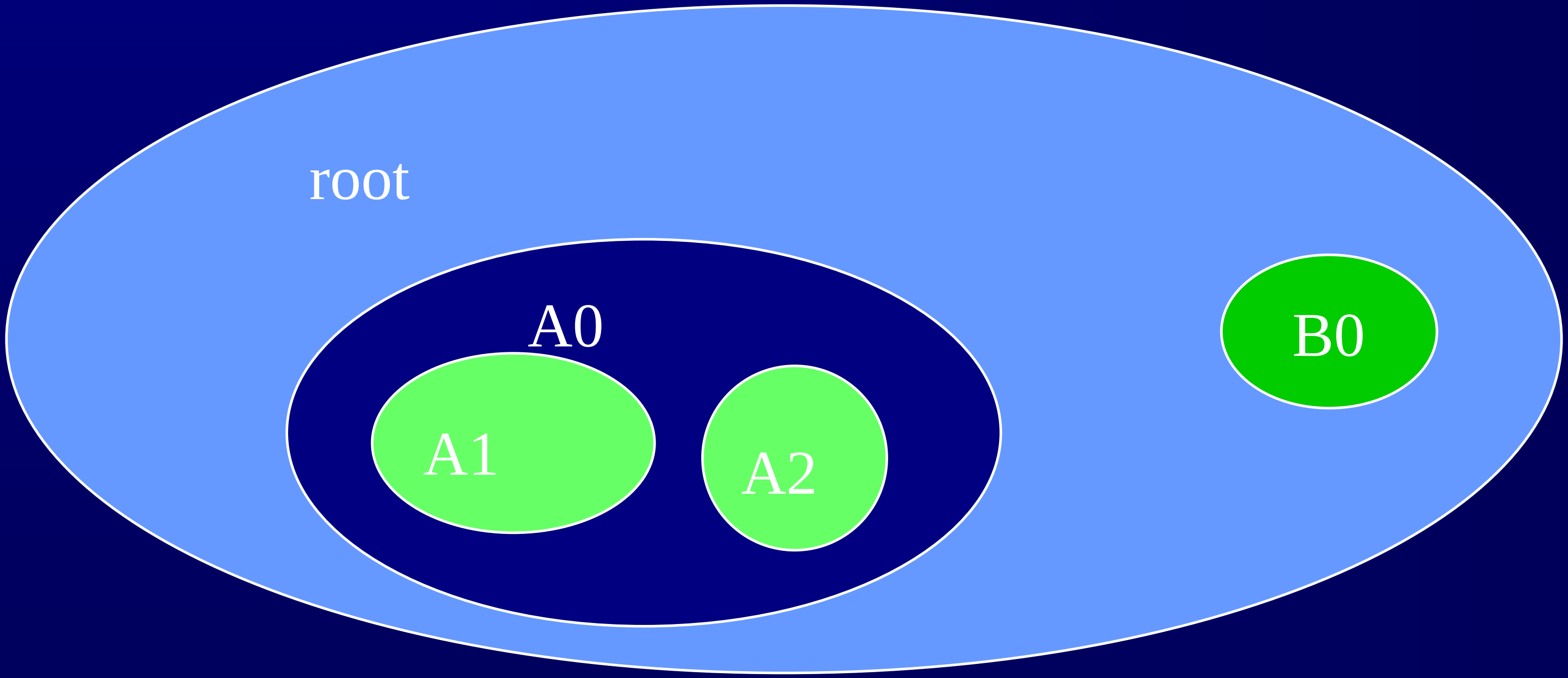
Trees in SQL

- Trees are graph structures used to represent
  - Hierarchies
  - Parts explosions
  - Organizational charts
- Three methods in SQL
  - Adjacency list model
  - Nested set model (and variations)
  - Transitive closure list

Tree as Graph



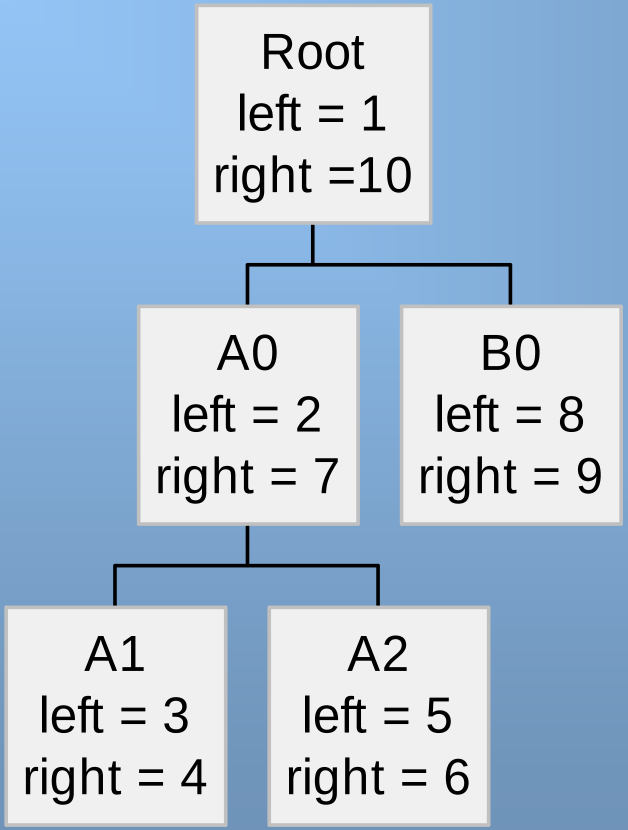
Tree as Nested Sets



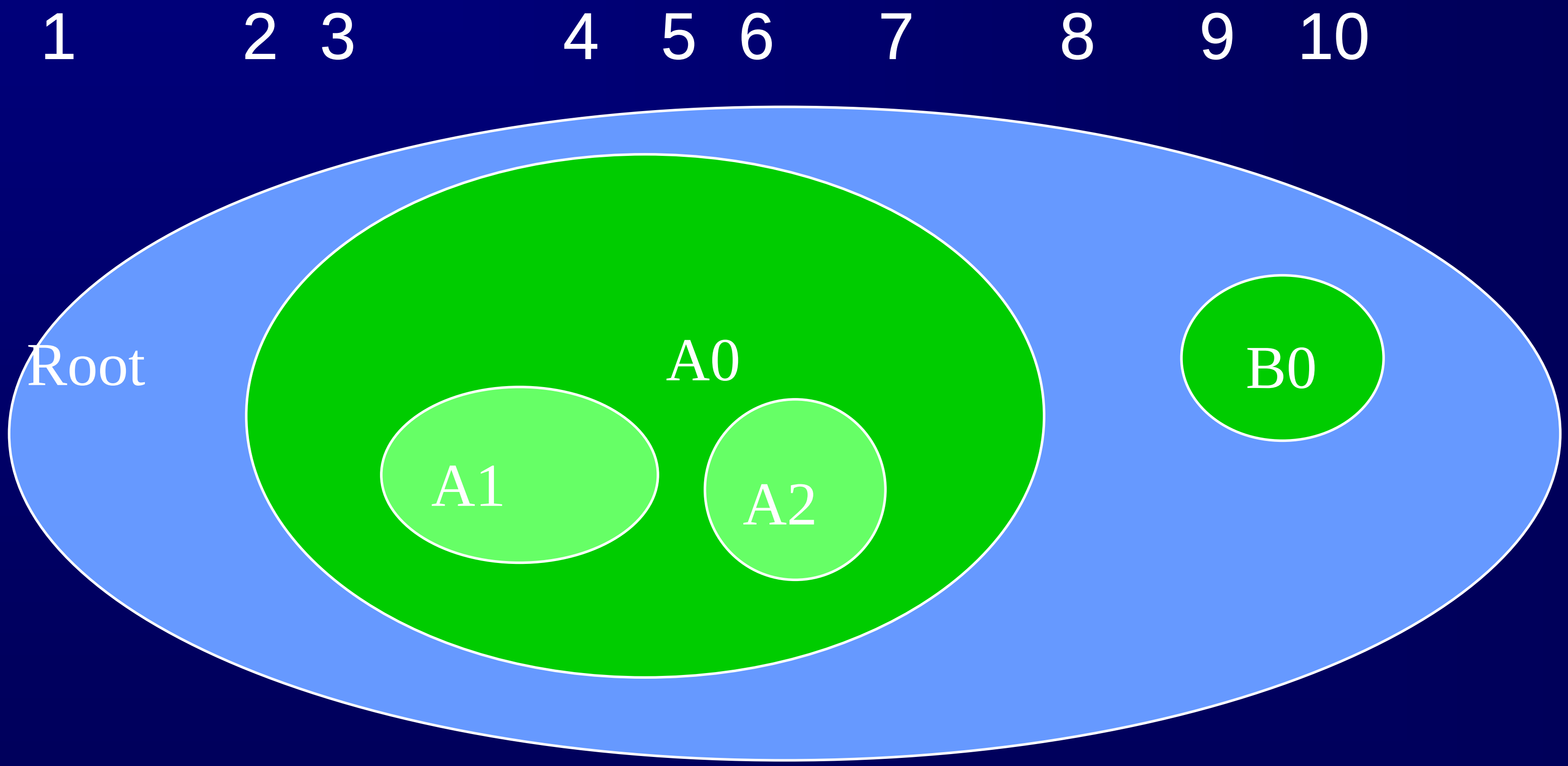
Graph as Table

node	parent
Root	NULL
A0	Root
A1	A0
A2	A0
B0	Root

Graph with Traversal



Nested Sets with Numbers



Nested Sets as Numbers

```
 Node      lft     rgt
============
Root        1      10
A0          2        7
A1          3        4
A2          5        6
B0          8        9
```

Problems with Adjacency list -1

- Not normalized - change A0 and see that it changes in many places
- You have to use cursors or self-joins to traverse the tree
- Cursors are not a table -- their order has meaning -- Closure violation!
- Cursors take MUCH longer than queries
- Ten level self-joins are worse than cursors

Problems with Adjacency list -2

- Often mix structure (organizational chart, edges) with elements (personnel, nodes)
- These are different kinds of things and should be in separate tables
- Another advantage of separating them is that you can have multiple hierarchies on one set of nodes

Example of Self-Join

- Find great grandchildren of X

-
  ```
  SELECT T1.node, T2.node, T3.node, T4.node
    FROM Tree AS T1, Tree AS T2,
                Tree AS T3, Tree AS T4
  WHERE T1.node = ‘X’
        AND T1.node = T2.parent
        AND T2.node = T3.parent,
        AND T3.node = T4.parent;
  ```

Find Root of Tree

-
  ```
  SELECT *    -- adjacency list
    FROM Tree
  WHERE parent IS NULL;
  ```

-
  ```
  SELECT *   -- nested sets
   FROM Tree
  WHERE lft = 1;  -- index the lft column
  ```

- Note that you can tell the size of a subtree rooted at a given node with the simple formula

  ((rgt - lft +1) /2)

Find All Leaf Nodes

-
  ```
  SELECT *   -- adjacency list
  FROM Tree
  WHERE node NOT IN
     (SELECT parent FROM Tree);
  ```

-
  ```
  SELECT *   --nested sets
  FROM Tree
  WHERE lft = rgt -1; -- index on lft
  ```

Find Superiors of X

- Traversal up tree via procedure or N-way self-join

- SELECT Super.*
   FROM Tree AS T1, Tree AS Supers
 WHERE node = ‘X’
     AND T1.lft BETWEEN Supers.lft AND Supers.rgt;

Find Subordinates of X

- Traversal down tree via cursors or N-way self-join

- SELECT Subordinates.*

```
 FROM Tree AS T1,
              Tree AS Subordinates
 WHERE T1.node = ‘X’
AND Subordinates.lft BETWEEN
              T1.lft AND T1.rgt;
```

Totals by Level in Tree

- In Adjacency model you put traversal results in a temp table, then group and total

-
  ```
  SELECT T1.node, SUM(C1.cost)
    FROM Tree AS T1, Tree AS T2,  Costs AS C1
  WHERE C1.node = T2.node
      AND T2.lft BETWEEN T1.lft AND T1.rgt
  GROUP BY T1.node;
  ```

The Median

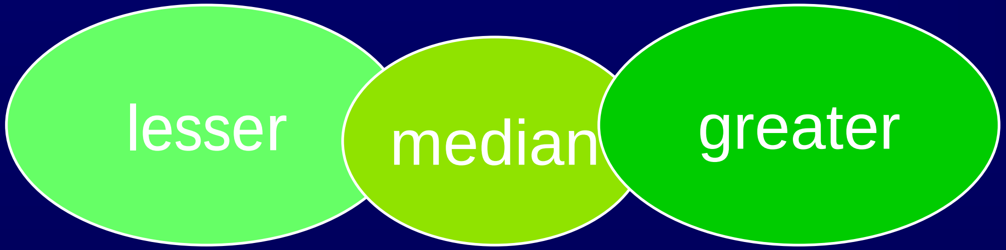
- The median is a statistic that measures central tendency in a set of values
- The median is the value such that there are as many cases below the median value as there are above it
- If the number of elements is odd, no problem
- If the number of elements is even, then average the middle values

Procedural Way

- Sort the values
- Count the size of the set (n)
- If n is odd then read (n/2) records
- Print the next record
- If n is even then read (n/2) and (n/2)+1 records
- Average them and print results

Think in Sets

- Do not ask for values, but for a set of values
- The median is the average of the subset of values which “sit in the middle”
- A middle implies something on either side of it
- The subset of greater values has the same cardinality as the subset of lesser values



Median by Partition -1

- Now the question is how to define a median in terms of the partitions.
- Clearly, the definition of a median means that if (lesser = greater) then the value in the middle is the median.
- Let’s use Chris Date’s Parts table and find the Median wgt of the Parts.

Median by Partition -2

- If there are more greater values than half the size of the table, then wgt cannot be a median.
- If there are more lesser values than half the size of the table, then the middle value(s) cannot be a median.
- If (lesser + equal) = greater, then the middle value(s) is a left hand median.
- If (greater + equal) = lesser, then the middle value(s) is a right hand median.
- If the middle value(s) is the median, then both lesser and greater have to have tallies less than half the size of the table.

Median by Partition -3

- Instead of a WHERE clause operating on the columns of the derived table, why not perform the same test as a HAVING clause on the inner query which derives Partitions?

-
  ```
  SELECT AVG(DISTINCT wgt)
   FROM (SELECT P1.weight
                 FROM Parts AS P1, Parts AS P2
               GROUP BY P1.pno, P1.weight
              HAVING SUM(CASE WHEN P2.weight = P1.weight
                                    THEN 1 ELSE 0 END)
                   >= ABS(SUM(CASE WHEN P2.weight < P1.weight
                                         THEN 1
                                         WHEN P2.weight > P1.weight
                                          THEN -1   ELSE 0 END)))
  AS Partitions;
  ```

Median by Sequence - 1

- Let’s use the SQL-99 OLAP operators to get a weighted median using a sequence operator

- Moral to the story – sequences are still useful

-
```
SELECT AVG(DISTINCT wgt)
   FROM (SELECT P1.wgt,
ROW_NUMBER() OVER(ORDER BY  wgt ASC) AS low,
ROW_NUMBER() OVER(ORDER BY  wgt DESC) AS high
                    FROM Parts) AS Sides
  WHERE low IN (high , high-1,  high+1);
```

Characteristic functions -1

- A characteristic function returns a one or a zero if a predicate is TRUE or FALSE
- We can write it with a CASE expression in SQL-92
- You can use it inside aggregate functions to get descriptions of subsets
- It gives you set properties

Characteristic functions -2

- Example: find the number of men and women in the company in each department

- SELECT department,

```
SELECT  department,
            SUM(CASE WHEN sex = ‘m’
                  THEN 1 ELSE 0 END)  AS men,
            SUM(CASE WHEN sex = ‘f’
                   THEN 1 ELSE 0 END) AS women
 FROM Personnel
GROUP BY department;
```

CASE Expressions -1

- Use in place of procedural code
- Example: raise price of cheap books by 10%, and reduce expensive books by 15%; Use $25 as break point
- First attempt:

```
 BEGIN
UPDATE Books SET price = price *1.10 WHERE price <= $25.00;
UPDATE Books SET price = price *0.85 WHERE price > $25.00;
END;
```

- Look at what happens to a book priced $25.00

CASE Expressions -2

- Second attempt: use a cursor

- Third attempt: procedural code

```
BEGIN
IF (SELECT price FROM Books WHERE isbn = :my_book)
     <= $25.00
THEN  UPDATE Books
               SET price = price *1.10
          WHERE isbn = :my_book
ELSE  UPDATE Books
               SET price = price *0.85
          WHERE isbn = :my_book
END;
```

CASE Expressions -3

- Use the CASE expression inside the UPDATE statement
- UPDATE Books

```
        SET price = CASE WHEN  price <= $25.00
                             THEN price *1.10
                            WHEN price > $25.00
                            THEN price *0.85
                             ELSE price  END;
```

- The ELSE clause says “leave it alone” as a safety precaution

Questions & Answers

?