BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

DATA WAREHOUSING SS-G515
LAB-6[MVD Attributes]

DATE: 20/05/2021 **TIME: 02 Hours**

Handling Hierarchies

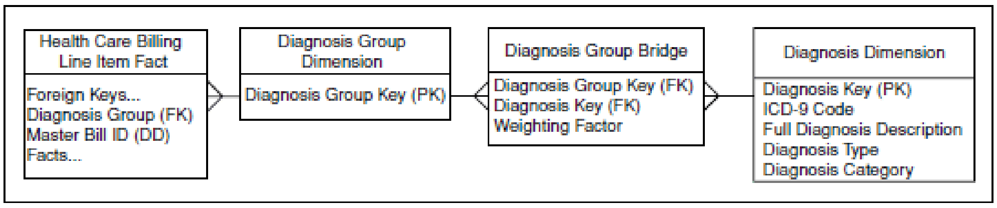
- Hierarchical relationships among dimension attributes are common
- There are various ways to handle hierarchies
  - Store all levels of hierarchy in denormalized dimension table
    - The preferred solution in almost all cases!
  - Create “snowflake” schema with hierarchy captured in separate outrigger table
    - Only recommended for huge dimension tables
    - Storage savings have negligible impact in most cases

The above two ways are for handling fixed depth hierarchies, variable-depth hierarchies can be handled using **Bridge Tables**.

Multivalued Attributes Using Bridge Tables

There will be times when you arrive at a many to many relationship between a fact and a dimension. This many to many relationship is in violation of the dimensional modelling rules. Multi-valued dimensions are more common than most people think and are often “refactored” to a single valued dimension.

If the grain of the fact table is not changed, a multivalued dimension must be linked to the fact table through an associative entity called **a *bridge table***. See Figure below for the health care example.



To avoid a many-to-many join between the bridge table and the fact table, one must create a *group entity* related to the multivalued dimension.

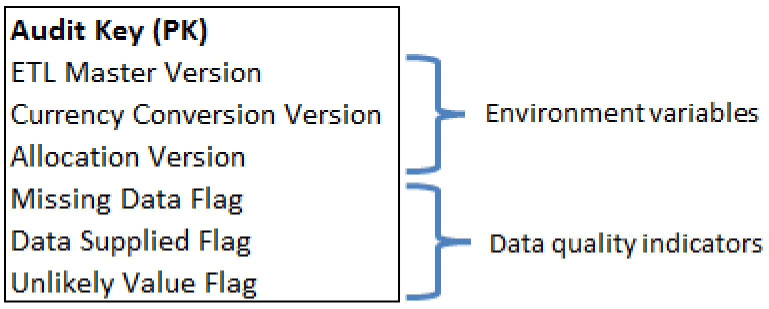
Exercises:

1). Consider a rental fact table, which contains information about rental of DVDs, there is a DVD dimension which contains information about different DVDs like, DVD id, DVD release date, actors in DVD etc. Here actor information is mutivalued attribute. Resolve this using Bridge table and give the result for **'Number of DVDs rented in which actor Y play?'**

2). Consider the scenario of healthcare, a patient can be diagnosed with multiple diseases and one disease can be diagnosed for multiple patients. It is a many-to-many relation. Implement this using bridge tables, also there shouldn't be many-to-many relation between bridge table and fact table. Give the design of system as well as result for query **"How many patients diagnosed with disease X?".**

Audit Dimensions

One of the most effective tools for managing data quality and data governance, as well as giving business users confidence in the data warehouse results, is the audit dimension. We often attach an audit dimension to every fact table so that business users can choose to illuminate the provenance and confidence in their queries and reports. Simply put, the audit dimension elevates metadata to the status of ordinary data and makes this metadata available at the top level of any BI tool user interface.



Environment variables describing the versions of ETL code used to create the fact rows or the ETL process execution time stamps. These environment variables are especially useful for compliance and auditing purposes because they enable BI tools to drill down to determine which rows were created with what versions of the ETL software.
The data quality indicators are flags that show whether some particular condition was encountered for the specific fact row. If the fact row contained missing or corrupt data (perhaps replaced by null) then the missing data flag would be set to true.
Exercise:
1) Consider order fact table which stores data for each order and their amount. Create an audit dimension for it and and give result for query **"Net Revenue Amount for all the years considering completely valid data only by verifying from Audit Table"**

Recursive Common Table Expression

A CTE can be thought of as a temporary result set and are similar to a derived table in that it is not stored as an object and lasts only for the duration of the query. A CTE is generally considered to be more readable than a derived table and does not require the extra effort of declaring a Temp Table while providing the same benefits to the user. However; a CTE is more powerful than a derived table as it can also be self-referencing, or even referenced multiple times in the same query.

The basic syntax structure for a CTE is shown below:

```
WITH MyCTE
AS ( SELECT EmpID, FirstName, LastName, ManagerID
FROM Employee
WHERE ManagerID IS NULL                                                              )
SELECT *
FROM MyCTE
```

A recursive CTE requires four elements in order to work properly.

1. Anchor query (runs once and the results 'seed' the Recursive query)

2. Recursive query (runs multiple times and is the criteria for the remaining results)

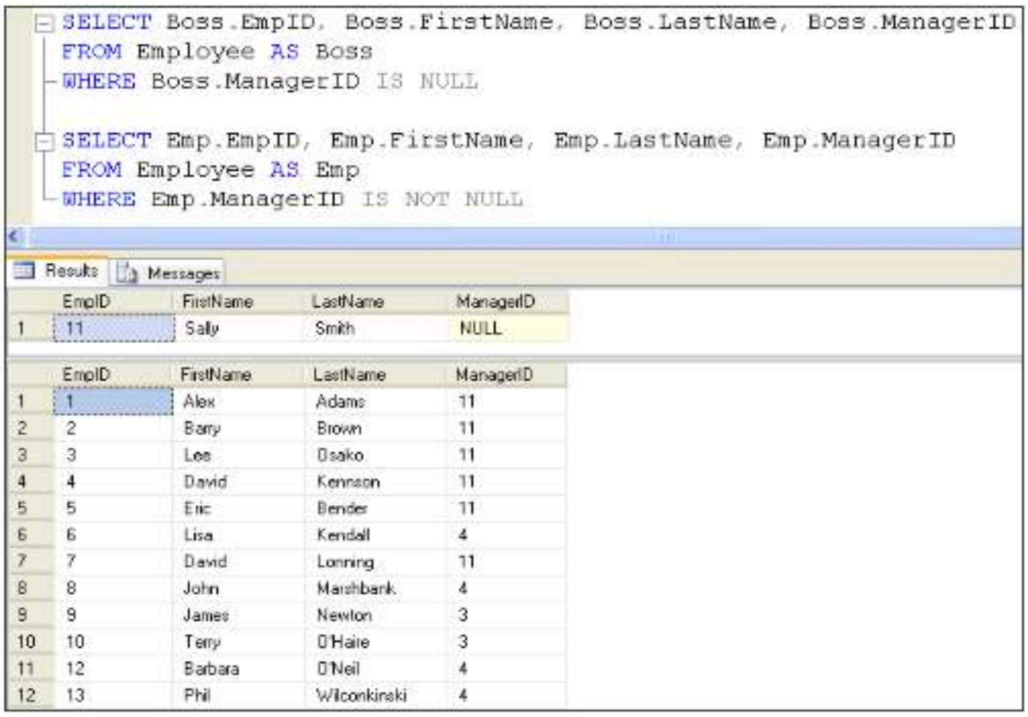
3. UNION ALL statement to bind the Anchor and Recursive queries together.

4. INNER JOIN statement to bind the Recursive query to the results of the CTE.

```
WITH MyCTE
AS ( SELECT EmpID, FirstName, LastName, ManagerID
FROM Employee
WHERE ManagerID IS NULL
UNION ALL
SELECT EmpID, FirstName, LastName, ManagerID
FROM Employee
INNERJOIN MyCTE ON Employee.ManagerID = MyCTE.EmpID
WHERE Employee.ManagerID IS NOTNULL)
SELECT *
FROM MyCTE
```

a) Identify the Anchor and Recursive Query

Anyone who does not have a boss is considered to be at the top level of the company and everyone who does have a boss either works for the person(s) at the top level (upper management), or the people that work for them (mid-management thru base employees). For example, a CEO is at the top level and thus has a ManagerID of null. Likewise, everyone below the CEO will have a ManagerID. This has been demonstrated in the two queries below:



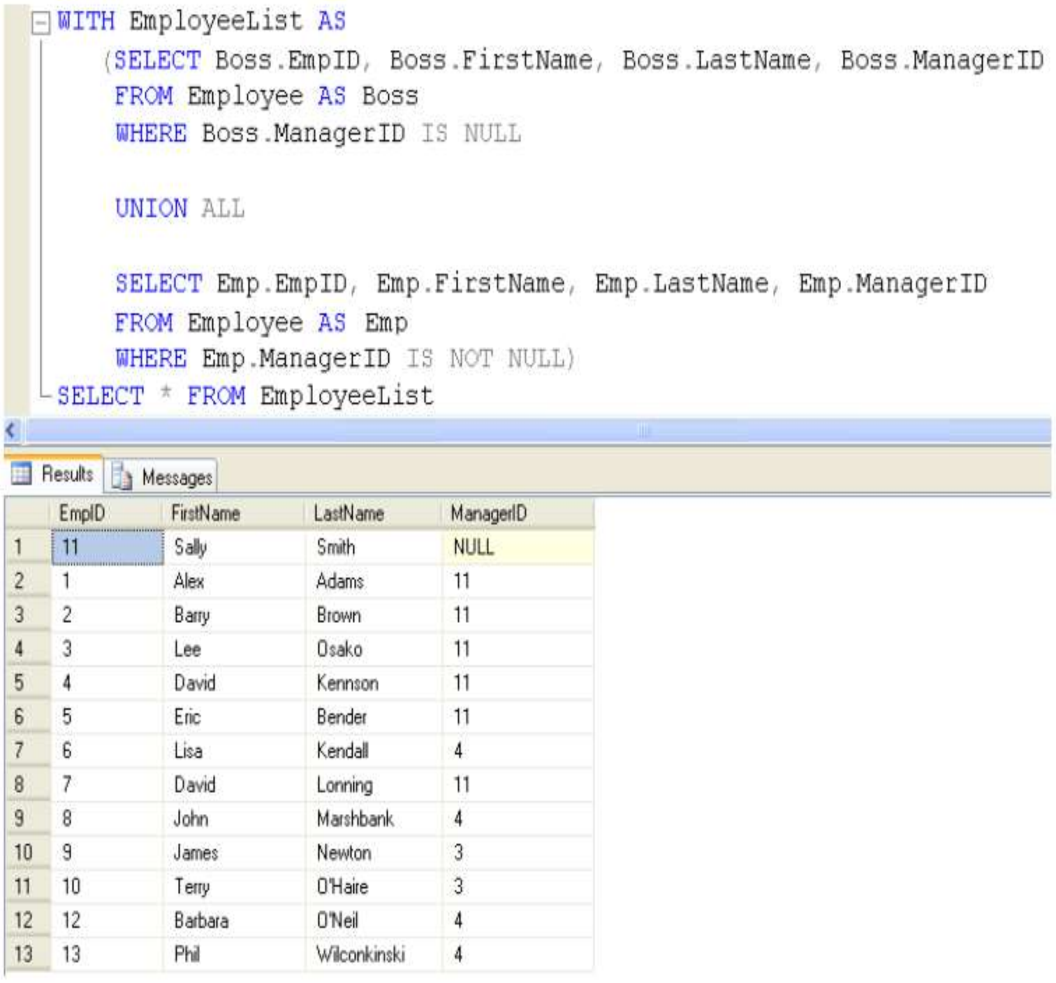
```sql
SELECT Boss.EmpID, Boss.FirstName, Boss.LastName, Boss.ManagerID
FROM Employee AS Boss
WHERE Boss.ManagerID IS NULL

SELECT Emp.EmpID, Emp.FirstName, Emp.LastName, Emp.ManagerID
FROM Employee AS Emp
WHERE Emp.ManagerID IS NOT NULL
```

	EmpID	FirstName	LastName	ManagerID
1	11	Sally	Smith	NULL

	EmpID	FirstName	LastName	ManagerID
1	1	Alex	Adams	11
2	2	Barry	Brown	11
3	3	Lee	Osako	11
4	4	David	Kennson	11
5	5	Eric	Bender	11
6	6	Lisa	Kendall	4
7	7	David	Lonning	11
8	8	John	Marshbank	4
9	9	James	Newton	3
10	10	Terry	O'Haire	3
11	12	Barbara	O'Neil	4
12	13	Phil	Wilconkinski	4

The first SELECT statement will become the Anchor query as it will find the employee that has a ManagerID of null (representing Level 1 of the organization). The second SELECT statement will become your Recursive query and it will find all employees that do have a ManagerID (representing Level 2-3 of this organization).
As we can see from the results so far, these queries are unable to give hierarchical data on which level each employee is at within the organization.

b) Add the Anchor and Recursive query to a CTE
Begin transforming this entire query into a CTE by placing a UNION ALL statement between the Anchor and Recursive queries. Now add parentheses around the entire query, indenting it, moving it down, and adding the declaration WITH EmployeeList AS before the open parenthesis, and then add SELECT * FROM EmployeeList on the next line after the close parenthesis.
Your query would look something like:



```
WITH EmployeeList AS
    (SELECT Boss.EmpID, Boss.FirstName, Boss.LastName, Boss.ManagerID
     FROM Employee AS Boss
     WHERE Boss.ManagerID IS NULL

     UNION ALL

     SELECT Emp.EmpID, Emp.FirstName, Emp.LastName, Emp.ManagerID
     FROM Employee AS Emp
     WHERE Emp.ManagerID IS NOT NULL)
SELECT * FROM EmployeeList
```

Results | Messages

	EmpID	FirstName	LastName	ManagerID
1	11	Sally	Smith	NULL
2	1	Alex	Adams	11
3	2	Barry	Brown	11
4	3	Lee	Osako	11
5	4	David	Kennson	11
6	5	Eric	Bender	11
7	6	Lisa	Kendall	4
8	7	David	Lonning	11
9	8	John	Marshbank	4
10	9	James	Newton	3
11	10	Terry	O'Haire	3
12	12	Barbara	O'Neil	4
13	13	Phil	Wilconkinski	4

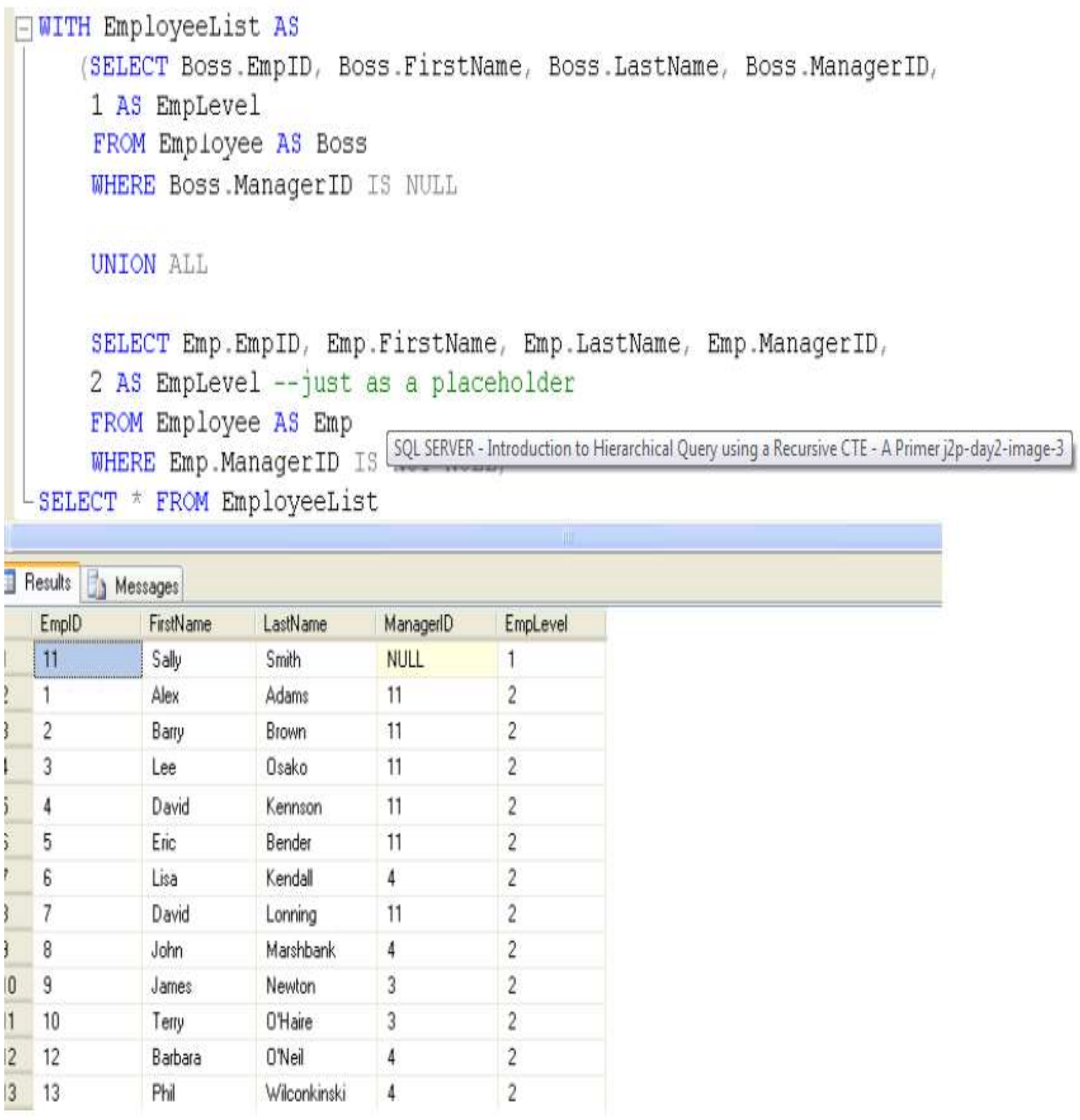
The results from your CTE are exactly the same as the results returned from running the anchor and Recursive queries simultaneously in the previous example.

c) Add an expression to track hierarchical level

The Anchor query (aliased as 'Boss') inside the CTE represents everyone at Level 1 (i.e. Sally Smith). The Recursive query (aliased as 'Emp') represents everyone at Levels 2 and 3. In order to visualize each level in a result set, you will need to add an expression field to each query.

Add the expression "1 AS EmpLevel" to the Anchor query and the expression "2 AS EmpLevel" to the Recursive query. Before executing the entire query, look closely at the expression field. The EmpLevel expressions in the Anchor query will hard-code the numeral 1 (for Sally Smith's level), while the EmpLevel expressions in the Recursive query will hard-code the numeral 2 for everyone else.

Your query would look like:



```
WITH EmployeeList AS
    (SELECT Boss.EmpID, Boss.FirstName, Boss.LastName, Boss.ManagerID,
     1 AS EmpLevel
     FROM Employee AS Boss
     WHERE Boss.ManagerID IS NULL

     UNION ALL

     SELECT Emp.EmpID, Emp.FirstName, Emp.LastName, Emp.ManagerID,
     2 AS EmpLevel --just as a placeholder
     FROM Employee AS Emp
     WHERE Emp.ManagerID IS
SELECT * FROM EmployeeList
```

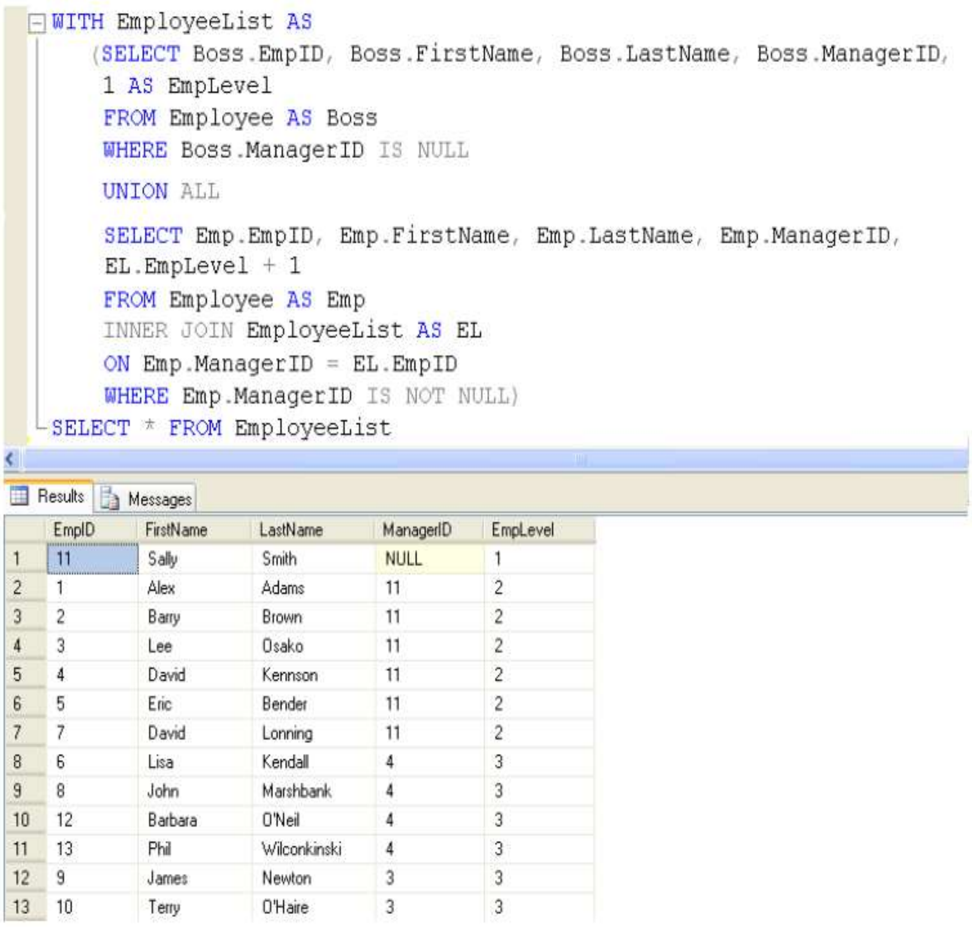
SQL SERVER - Introduction to Hierarchical Query using a Recursive CTE - A Primer j2p-day2-image-3

EmpID	FirstName	LastName	ManagerID	EmpLevel
11	Sally	Smith	NULL	1
1	Alex	Adams	11	2
2	Barry	Brown	11	2
3	Lee	Osako	11	2
4	David	Kennson	11	2
5	Eric	Bender	11	2
6	Lisa	Kendall	4	2
7	David	Lonning	11	2
8	John	Marshbank	4	2
9	James	Newton	3	2
10	Terry	O'Haire	3	2
12	Barbara	O'Neil	4	2
13	Phil	Wilconkinski	4	2

The two new expression fields were a helpful step. In fact, they show the correct EmpLevel information for Sally Smith and for the people at Level 2 (i.e., Adams, Bender, Brown, Kennson, Lonning and Osako). However, the 2 is just a hard-coded placeholder to help visualize your next step. Lisa Kendall and several other employees need to be at Level 3.

d) Add a self-referencing INNER JOIN statement

We can use the CTE for this. A recursive CTE requires an INNER JOIN to connect the recursive query to the CTE itself. Go ahead and write an INNER JOIN statement binding the recursive query 'Emp' to the CTE 'EmployeeList AS EL' ON Emp.ManagerID = EL.EmpID.



```
WITH EmployeeList AS
   (SELECT Boss.EmpID, Boss.FirstName, Boss.LastName, Boss.ManagerID,
    1 AS EmpLevel
    FROM Employee AS Boss
    WHERE Boss.ManagerID IS NULL
    UNION ALL
    SELECT Emp.EmpID, Emp.FirstName, Emp.LastName, Emp.ManagerID,
    EL.EmpLevel + 1
    FROM Employee AS Emp
    INNER JOIN EmployeeList AS EL
    ON Emp.ManagerID = EL.EmpID
    WHERE Emp.ManagerID IS NOT NULL)
SELECT * FROM EmployeeList
```

	EmpID	FirstName	LastName	ManagerID	EmpLevel
1	11	Sally	Smith	NULL	1
2	1	Alex	Adams	11	2
3	2	Barry	Brown	11	2
4	3	Lee	Osako	11	2
5	4	David	Kennson	11	2
6	5	Eric	Bender	11	2
7	7	David	Lonning	11	2
8	6	Lisa	Kendall	4	3
9	8	John	Marshbank	4	3
10	12	Barbara	O'Neil	4	3
11	13	Phil	Wilconkinski	4	3
12	9	James	Newton	3	3
13	10	Terry	O'Haire	3	3