Tuning Access Paths: Red Flags to Look for.

Abstract: As DBAs and application developers we have all had to look at an access path at one time or another. Sometimes a performance issue can be easily seen in the access path, but other times it can be a little more difficult to spot. This is especially true when dealing with explain output that generates multiple query blocks. In this session we will look at explain output and highlight key areas that affect the performance of various SQL statements such as joins, sorts and subqueries. The intent is to focus our efforts on the pieces of the access path that have the most impact on the performance of our SQL.

Agenda

Topics to be covered

1) EXPLAIN review
2) Nested Loop / Merge Scan Join
3) Correlated / Non-Correlated Subquery
4) Access Path Red Flag Review

I.General review of key plan table columns
- A.Accesstype
- B.Join Method
- C.Matchcols
- D.Prefetch

II.Nested loop join .vs. Merge Scan join
- A.Processing differences
- B.Key performance indicators for each join type
- C.Review examples

III.Correlated .vs. Non-Correlated subqueries
- A.Processing differences
- B.Key performance indicators for each join type
- C.Review examples

V.Review of Red flags in Access path

Important PLAN_TABLE columns

QBLOCKNO: a logical block of a complex SQL statement (Union, subselect, etc..)

PLANNO: the order that the steps of the access path are performed within each logical block of SQL (i.e. within each QBLOCKNO).

JOIN METHOD: indicates the method used to join tables together (or an indication of a sort).
0- first table accessed
1- Nested Loop join
2- Merge Scan join
3 - sort ORDER BY, GROUP BY or DISTINCT
4 - Hybrid join

ACCESSTYPE: the way the table is accessed.
R - tablespace scan.
I - index access (Index name is put in ACCESSNAME).
N - index when an IN predicate is present in the SQL.

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefch	QBLK Type	Parent QBLK
1	1	MEMBER	0	R	0		N	S	SEL	0
1	2	ADRESS	1	I	1	X001A	Y	-	SEL	0

3

The list of PLAN_TABLE columns and descriptions can be found in the DB2 Application Programming and SQL guide (Chapter 7-4).

View online at: http://www-306.ibm.com/software/data/db2/zos/v8books.html

Important PLAN_TABLE columns

IDUG® 2007 North America

MATCHCOLS: number of index columns matched (ACCESSTYPE I or N).

ACCESSNAME: the name of the index if applicable (ACCESSTYPE I or N).

IX ONLY: can the index be used by itself without touching the table.

PREFETCH: Indicates that data pages will be read in advance.
S Sequential Prefetch (tablespace scan or high cluster index)
L List prefetch (low cluster index)

QBLOCK_TYPE: the type of SQL for each logical block of SQL (i.e. QBLOCKNO).
SELECT - select statement NCOSUB - non-correlated subquery
CORSUB - correlated subquery UNION - union or union all

PARENT QBLK: The Parent process that this child QBLOCK feeds into.

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK
1	1	MEMBER	0	R	0		N	S	SEL	0
1	2	ADRESS	1	I	1	X001A	Y	-	SEL	0

DB2 only uses Prefetch when it believes that many pages will need to be retrieved. This can be an **eye-catcher**.

4

GoFurther

The list of PLAN_TABLE columns and descriptions can be found in the DB2 Application Programming and SQL guide (Chapter 7-4).

View online at: http://www-306.ibm.com/software/data/db2/zos/v8books.html

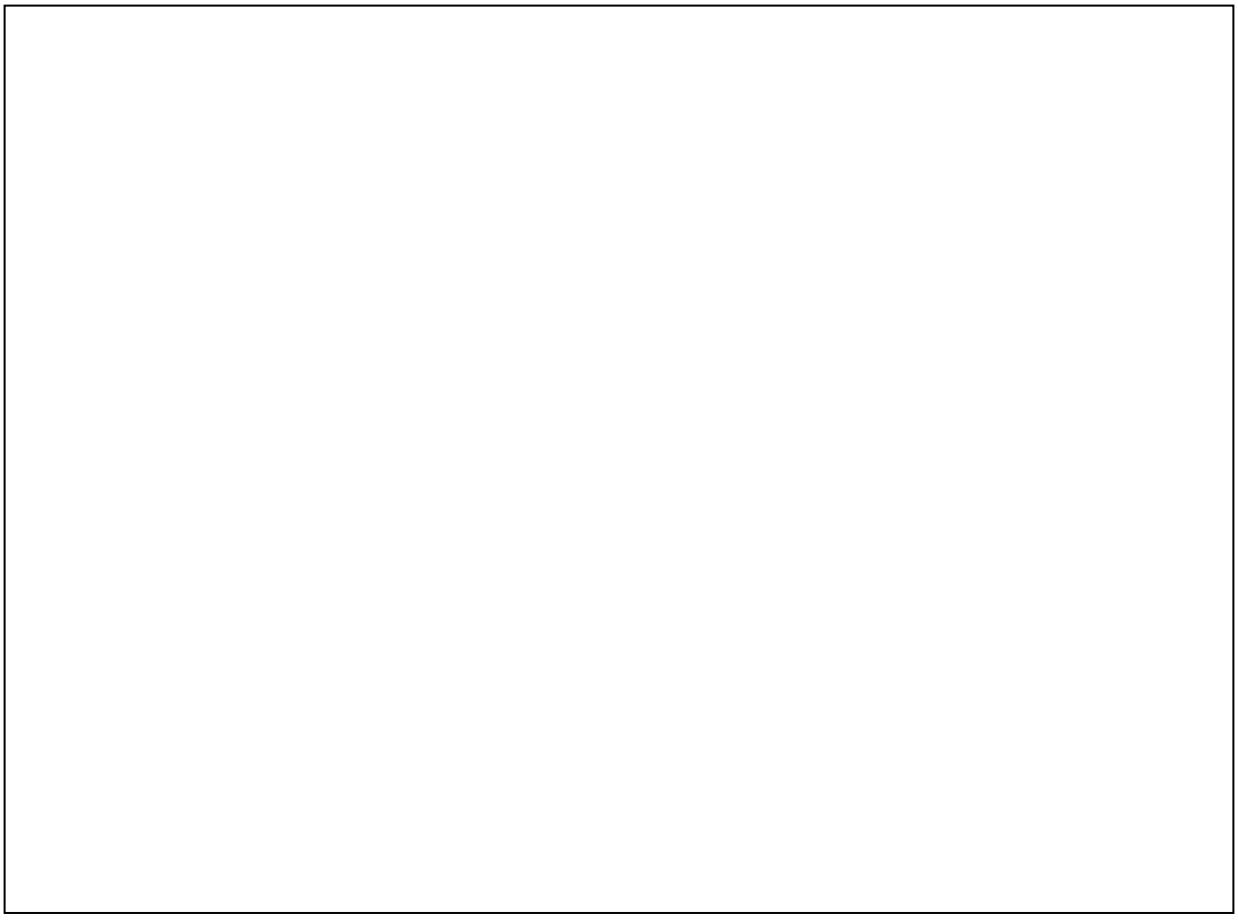
MATCHCOLS is an important indicator of performance. If a table is being scanned, or if an index is being scanned, then the MATCHCOLS will be 0. Scanning tables or indexes involves reading the object from top to bottom. If the object has millions of rows, then this can obviously take some time.

PREFETCH is an important eye-catcher for potential performance issues. But prefetch is not always bad. There are cases where prefetch is a very good mechanism for retrieving the data. However, prefetch is only used in cases where DB2 believes it will retrieve a lot of data (even if an index is being used). If DB2 is retrieving a lot of data, then that MAY be where the performance issue is. So an entry in the prefetch column is a good place to start looking for suspected performance issues.

Topics to be covered

1) EXPLAIN review
2) Nested Loop / Merge Scan Join
3) Correlated / Non-Correlated Subquery
4) Access Path Red Flag Review

GoFurther



Types of Joins

JOIN METHOD: indicates the method used to join tables together (or an indication of a sort).

0- first table accessed
1- Nested Loop join
2- Merge Scan join
3 - sort ORDER BY, GROUP BY or DISTINCT
4 - Hybrid join

The 2 main join methods work significantly different. Each can be very efficient in the right situation or each can perform terribly.

6

The JOIN METHOD is one of the most important columns when looking at multi-row explain output. Depending on the join method, the performance indicators to look for will be very different. The important performance indicators for 1- Nested Loop and 2- Merge scan will be covered in this session. The Hybrid join is not frequently seen, and will not be covered in this session.

The JOIN METHOD refers to the way that DB2 joins the tables together behind the scenes. This is not the same as the JOIN TYPE column which indicates if a LEFT OUTER, RIGHT OUTER or FULL OUTER JOIN will be done.

The JOIN TYPE column has implications on what the final result set will be (I.e. the results of a LEFT OUTER JOIN are different from the results of a RIGHT OUTER JOIN).

In contrast, the JOIN METHOD does not affect the final result. No matter which JOIN METHOD DB2 chooses (nested loop, merge scan, or hybrid), the final result will be the same. The JOIN METHOD really indicates how DB2 is putting the 2 tables together internally. And the following slides will show that the nested loop and merge scan algorithms are very different. And the performance characteristics of each are also very different.

Nested Loop Join (iterates through loop)


```
SELECT ...
  FROM MEMBER M
      ,ADRESS A
WHERE M.FNAME  > 'A'
  AND M.SSN    = A.SSN;
```

NESTED LOOP logic (JOIN METHOD = 1)
FOR **each** qualifying row of the **outer table** (MEMBER)
  search for matching row(s) on **inner table** (ADRESSS)
END-FOR

MEMBER table (500,000 rows)

SSN	Fname	Lname
111..	BOB	ZACHR
333..	ANDY	WEIS
222..	PAUL	WEIS
...	...	...
998..	LINDA	AVERY
999..	CAROL	AVERY

Outer table

ADRESS table (1,000,000 rows)

SSN	Line	Address
111..	1	Suite 650
222..	2	Apartment 2
333..	1	Main Street
222..	1	Freemont Ave
...	...	...
998	1	Building AZ
998	2	Industry Road
999	1	Floor 8
999	2	Maple Drive

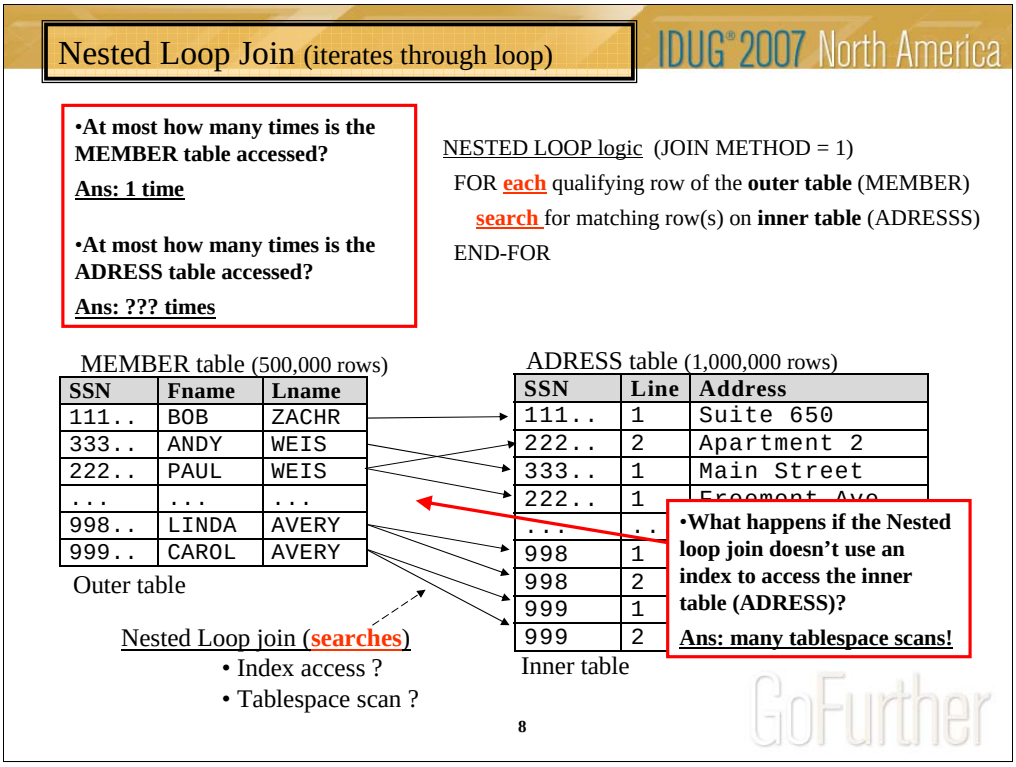
Inner table

Nested Loop join (**searches**)
- Index access ?
- Tablespace scan ?

7

The Nested Loop join is by far the most common type of join seen. The logic is very straightforward; for every qualifying row in the outer table (in this case MEMBER), the inner table (ADRESS) is searched to find matching rows. Because the inner table is searched once for each qualifying row of the outer table, it is very important that the inner table have an index available for the search. If no index is available, then the entire inner table will be scanned many times (once for each qualifying row of the outer table).

The inner and outer table are determined by the DB2 optimizer based on the size of the table and the number of rows that the optimizer believes will qualify.



Because the inner table is searched once for each qualifying row of the outer table, it is very important that the inner table have an index available for the search. If no index is available, then the entire inner table will be scanned many times (once for each qualifying row of the outer table). This could easily be a performance issue.

DB2 will still attempt to perform the join. Depending on the number of qualifying rows in the outer table, and the number of rows on the inner table that need to be scanned, the join may or may not finish. This could run for days.

Nested Loop Join (Sample access path)


```
SELECT . . .
FROM MEMBER M (1,500,000 rows)
    ,ADRESS A (1,000,000 rows)
WHERE M.FNAME > 'A'
  AND M.SSN = A.SSN
```

Indexes:

```
MEMBER                          Cluster
  X001M -> SSN                   (100)%
  X002M -> LNAME+FNAME            (55)%

ADRESS
  X002A -> LNAME                  (35)%
```

Join column(s)

There are 2 tablespace scans.
Which one should scare us the most?

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	R	0		N	S	SEL	0
1	2	ADRESS	1	R	0		N	S	SEL	0

Outer table: execute once

Inner table: execute many times

Nested Loop join

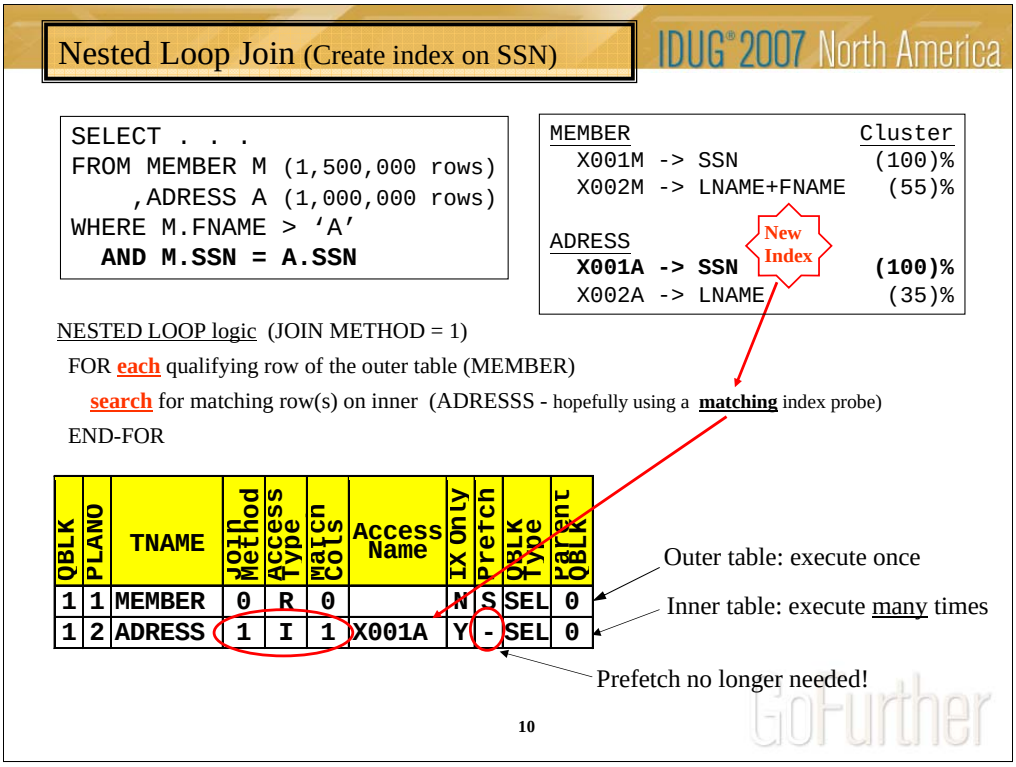
This SQL has a couple of performance issues. There are 2 tablespace scans being performed (which is not necessarily an issue). However, 1 of the scans is on the ADRESS table which is the inner table of a nested loop join. This means that the ADRESS table will be scanned 1 time for each qualifying row of the outer table (MEMBER). That could result in hundreds or thousands or millions of scans of the ADRESS table (1 million rows scanned each and every time). That is a serious performance issue.

The MEMBER table is also scanned. And this could be a performance concern, but it is not nearly as big an issue as the scan on the ADRESS table. The reason it's not as big a concern is because the MEMBER table will only be scanned 1 time (it's the outer table). So there will be 1 scan of the 1,500,000 row MEMBER table. That's generally not a huge issue.

The PREFETCH column shows that sequential prefetch is being used in both cases (which is standard for a tablespace scan).

The reason the ADRESS table is scanned is because there is no index for the join to use. The SQL joins the tables together by SSN number. However, the ADRESS table does not have an index specified on SSN. Therefore the only other option DB2 has is to scan the ADRESS table from top to bottom.

Note: This example is for illustration purposes. In reality DB2 would almost certainly NOT use this access path because of the multiple scans of the ADRESS table. However, if the statistics on the ADRESS table showed that there were very few rows (when there were really many rows), then this access path is possible.



To correct the multiple scans of the ADRESS table, an SSN index is added to the table. This allows DB2 to probe the SSN index rather than scan the ADRESS table. The index will be probed 1 time for each qualifying row from the outer table (MEMBER). Probing an index multiple times will be significantly faster than scanning the table multiple times.

The EXPLAIN shows that index X001A is now being probed matching on 1 column and there is no prefetch used on the ADRESS table. The prefetch is not needed because DB2 believes the index will provide sufficient filtering so that very few rows will be read with each probe (each probe by SSN will return few rows for that SSN). This is probably a fairly good access path. The only questionable access is the tablescan on the 1,500,000 row MEMBER table (which does use Prefetch). But that scan will only occur 1 time and is not that big of a performance issue.

This access path would be significantly faster than the access path seen on the previous slide.

Nested Loop join – Key to Performance

Nested loop - to perform well requires index access on **join columns** to the **inner table** (unless inner table is very small)

```
SELECT . . .
FROM MEMBER M (500,000 rows)
    ,ADRESS A (1,000,000 rows)
WHERE M.FNAME > 'VERA'
  AND M.SSN   = A.SSN
  AND M.LNAME = A.LNAME
ORDER BY A.CITY
```

Join columns to the inner table.

What columns do we suspect are in index X003A?

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	S	SEL	0
1	2	ADRESS	1	I	2	X003A	N	-	SEL	0
1	3		3							0

Should be looking at this line for index access and preferably no prefetch

When looking at a Nested Loop join, the most important factor is the join to the inner table. There should almost always be an index used to do the join (unless the table is very small). Also, if there is prefetch during the join, then that may indicate that even though an index is being used to do the join, the index does not provide much filtering. In other words the join will match many rows on the inner table even though an index is used. This could be a performance issue as well (although probably not quite as bad as a full tablespace scan on the join).

So the key to good performance for the nested loop join is to have the join done by an index that provides sufficient filtering that prefetch is not needed (as in the example shown above)

Q1: Are there red flags here?

IDUG® 2007 North America

```
SELECT . . .
FROM MEMBER  M (500,000 rows)
    ,PHONE  A (350,000 rows)
WHERE M.FNAME  ='JOCELYN'
  AND M.LNAME >='SMITH'
  AND M.LNAME = P.LNAME
  AND M.FNAME = P.FNAME
ORDER BY M.SSN
```

Indexes:

```
MEMBER                    Cluster
  X001M -> LNAME           (100%)
PHONE
  X001P -> SSN             (100%)
```

Is the index access OK? Why is Sequential prefetch used?

QBLK	PLANO	TNAME	Join Method	Access Type	Matcn Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X001M	N	S	SEL	0
1	2	PHONE	1	R	0		N	S	SEL	0
1	3		3							0

1.M.LNAME >='SMITH'

2.M.SSN = A.SSN

3.ORDER BY A.SSN

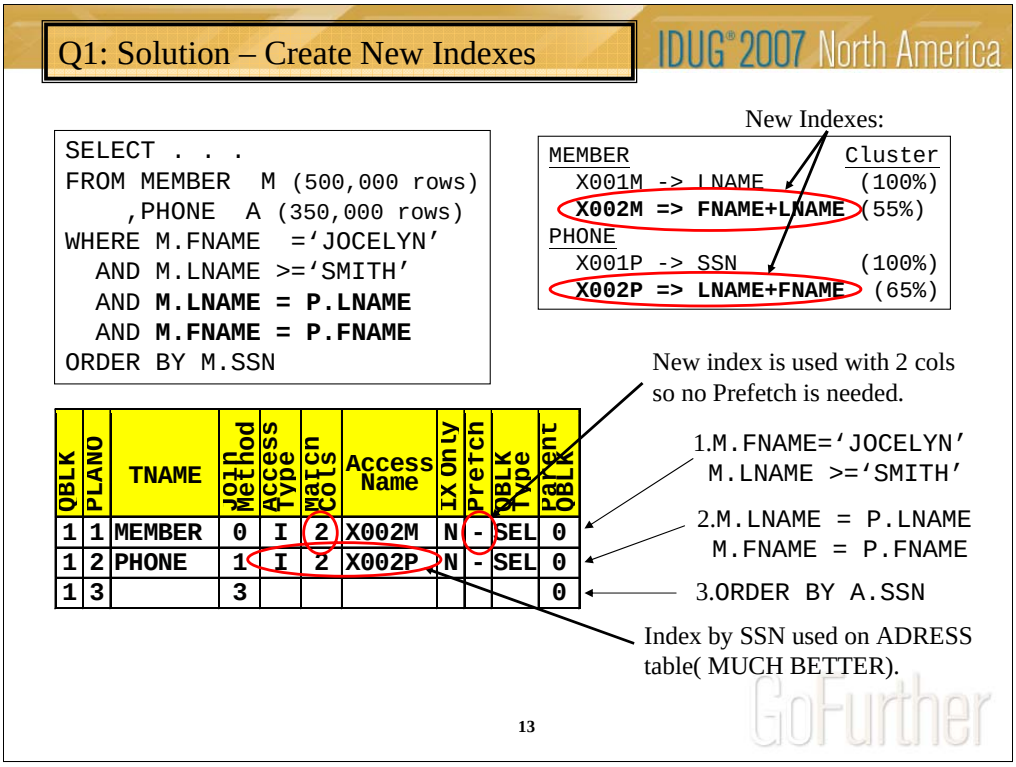
No index by LNAME/FNAME on the PHONE table!

GoFurther

This SQL has a clear performance issue. The Nested loop join to the PHONE table is not using an Index (the join is by LNAME / FNAME and there is no LNAME/FNAME index on the PHONE table). DB2 will still try to join the 2 tables together, but this will run for a very long time.

Also, the access to the MEMBER table is using and index probe, but is using Sequential prefetch. This means that DB2 believes that even though an index is used, the filtering is not very good (i.e it will match a lot of rows). And that is probably true in this case since the predicate says M.LNAME >= 'SMITH'. There are likely thousands of rows that have a LNAME greater than or equal to "SMITH". Because of this DB2 decides to use Prefetch to retrieve the many matching rows. In this case we know that we have further criteria that could provide more filtering (FNAME='JOCELYN'). However that criteria cannot be applied to the index because the FNAME column is not part of the index

So there are 2 potential performance issues with this access path. The Prefetch on the MEMBER table is a concern because it indicates that many rows will be retrieved. And the lack of an index when doing the Nested Loop join to the PHONE table is a severe issue because it means that the PHONE table will be scanned once for each row that is selected from the MEMBER table. That means there will be multiple scans of the PHONE table.

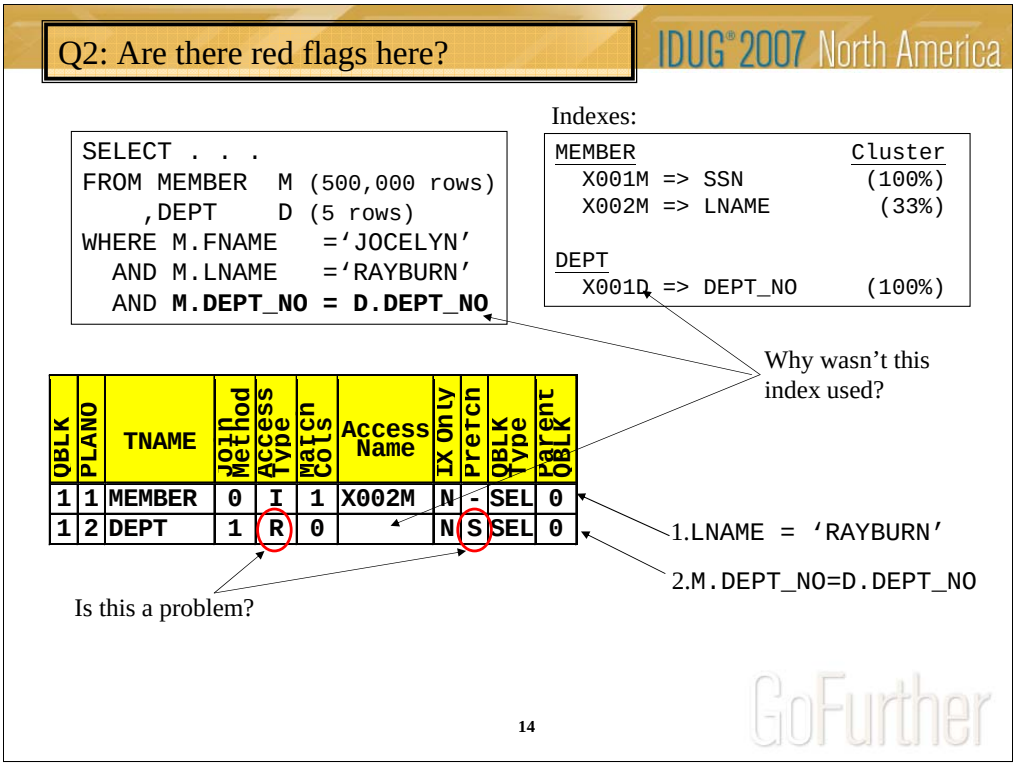


In this case, the creation of new indexes can correct the access path issues.

First an index is created on the MEMBER table that has FNAME and LNAME in it. Now the SQL can match on 2 predicates. Both the LNAME >= 'SMITH' and the FNAME = 'JOCELYN' can be used with the new index. There may be thousands of people with a last name greater than or equal to 'SMITH', but there will not be too many that also have a first name equal to JOCELYN. (Perhaps only a dozen or so rows may meet that criteria). Because so few rows are likely to meet both criteria, DB2 no longer needs to use Prefetch. (it is gone from the access path).

Also, an index on LNAME + FNAME is added to the PHONE table (these are the join columns to the inner table). Now the Nested Loop join can occur using an index instead of scanning the table. This means that DB2 can probe the inner table very efficiently to find exactly the matching LNAME+FNAME without having to scan the whole table (again prefetch is gone from the access path).

The new index has both LNAME and FNAME in it. This is the ideal situation, but it would probably also be acceptable if only a 1 column index were created on LNAME or possibly even just on FNAME. It would not be as efficient as having both columns in the index, but it would certainly perform better than the full tablespace scan that was being done before.



In this case, there initially appears to be a problem with the Nested Loop join to the DEPT table. There is no index used and as a result sequential prefetch is invoked. However, the DEPT table only has 5 rows on it. There is no harm in scanning 5 rows (even if the Nested Loop join does the scan many times).

There is an index on the DEPT table that could have been used, but because there are only 5 rows in the DEPT table, DB2 chose to ignore the index.

In this case there are no red flags in the access path.

Types of Joins

JOIN METHOD: indicates the method used to join tables together (or an indication of a sort).

0- first table accessed
1- Nested Loop join
2- Merge Scan join
3 - sort ORDER BY, GROUP BY or DISTINCT
4 - Hybrid join

The 2 main join methods work significantly different. Each can be very efficient in the right situation or each can perform terribly.

15

The JOIN METHOD is one of the most important columns when looking at multi-row explain output. Depending on the join method, the performance indicators to look for will be very different. The important performance indicators for 1- Nested Loop and 2- Merge scan will be covered in this session. The Hybrid join is not frequently seen, and will not be covered in this session.

The JOIN METHOD refers to the way that DB2 joins the tables together behind the scenes. This is not the same as the JOIN TYPE column which indicates if a LEFT OUTER, RIGHT OUTER or FULL OUTER JOIN will be done.

The JOIN TYPE column has implications on what the final result set will be (I.e. the results of a LEFT OUTER JOIN are different from the results of a RIGHT OUTER JOIN).

In contrast, the JOIN METHOD does not affect the final result. No matter which JOIN METHOD DB2 chooses (nested loop, merge scan, or hybrid), the final result will be the same. The JOIN METHOD really indicates how DB2 is putting the 2 tables together internally. And the following slides will show that the nested loop and merge scan algorithms are very different. And the performance characteristics of each are also very different.

Merge scan join: Completely different

MEMBER list (sorted list by SSN)

SSN	Fname	Lname
111..	BOB	ZACHR
222..	PAUL	WEIS
333..	ANDY	WEIS
...	...	...
998..	LINDA	AVERY
999..	CAROL	AVERY

NO INDEX NEEDED

ADRESS list (sorted list by SSN)

SSN	Line	Address
111..	1	Suite 650
111..	2	Main Street
222..	1	Apartment 2
222..	2	Freemont Ave
...	...	...
998	1	Building AZ
998	2	Industry Road

A simple join of 2 **sorted** lists – Sort **size may be reduced** by filtering predicates

Step 1: Read the first record from the MEMBER list
Step 2: Read the first record from the ADRESS list
Step 3: If the SSN matches then it's a join. Now read the next record in the ADRESS list. Does it match? If yes then it's another join. Keep doing that until there isn't a match.
Step 4: Read the next record from the MEMBER list. Is it a match (go back to Step3. etc.. etc..

The Merge Scan join is entirely different from the Nested Loop join. The Nested Loop join is very dependent on having an index to insure good performance. The Merge Scan join does not need an index at all.

The reason that Merge Scan join does not need an index is because it uses sorted lists. The lists MUST be sorted in the same order. So DB2 will retrieve qualifying rows from each table and then sort them into a list. Then it is a simple matter of reading through each list exactly 1 time (and only 1 time).

A record from the MEMBER list is compared to a record from the ADRESS list. If they match then it is a join. Then another records is read from ADRESS to see if it matches. That process is continued until an ADRESS record is found that does not match. When that occurs a new record is read from the MEMBER table. Then the records are compared to see if they match. If they do then it's a match, and so on and so on. In this way, both lists will eventually be read from top to bottom. When that is finished, then the join is complete.

This is far different from the Nested Loop join where it was reading the outer table, and then searching for a matching row on the inner table. The Merge Scan join does no searching at all. It simply reads 2 sorted lists from top to bottom.

Merge scan join: Benefits and Drawbacks

Benefits:

Scans through each table at most 1 time (filtering predicates applied prior to scan)

Does not use an index to join the tables together

Computers are very efficient at reading files and comparing A to B

Drawbacks:

Both lists are sorted (but sort size may be reduced by filtering predicates or sort may be avoided completely by an index)

Sorting is slow compared to other things in DB2

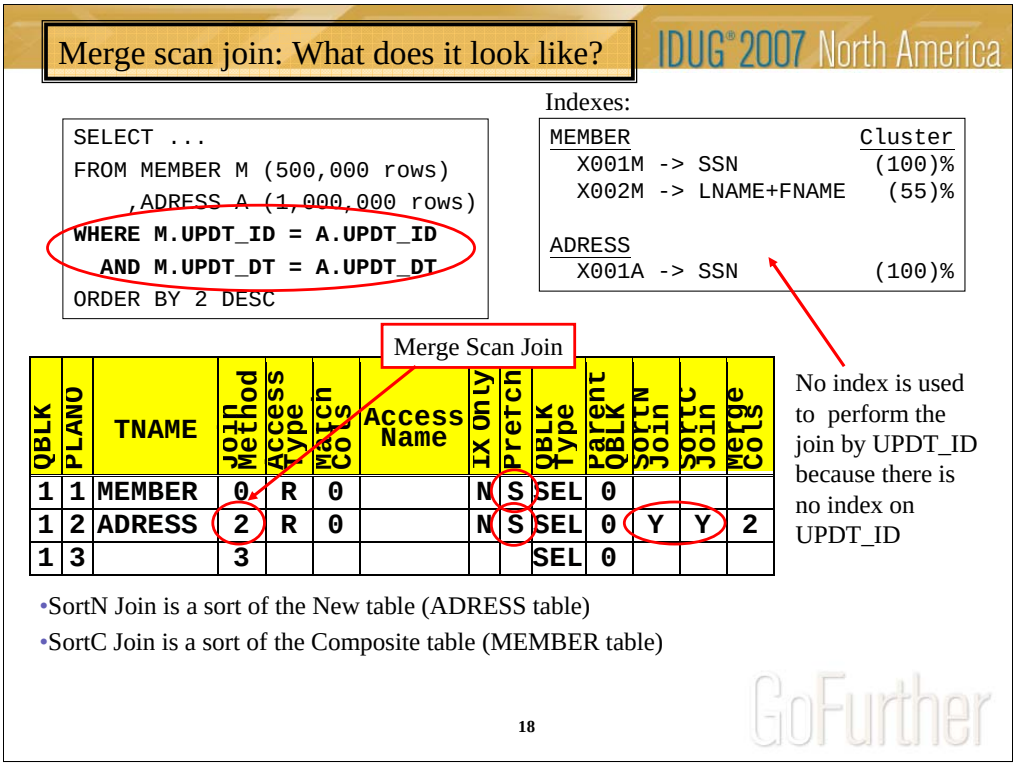
Often involves SCANing tables

The greatest benefit of the Merge Scan join is that it does not require an index on the join columns to perform well. This is quite useful in cases where no index exists to support the join. Also in cases where even if an index does exist it doesn't provide very good filtering. For example if the join criteria were based on a status code of "ACTIVE". There are probably thousands of rows on each table that have a status code of "ACTIVE". A Nested Loop join would not be practical since even if an index existed on the status code column it would still match thousands or rows for each join.

Merge scan on the other hand would simply sort the two lists and read through each of them one time. The Merge scan join could still perform quite well despite the poor index on the join column.

However, the major concern with the Merge Scan join is the time involved in performing the sort(s). Sorting can be time consuming. The more qualifying rows from each table, the longer the sort will take.

There are 2 things that can reduce the sorting time. First, if there are filtering predicates on the tables, they will be taken into account to reduce the size of the list that needs to be sorted. These predicates may or may not use indexes to reduce the size of the list (generally better performance if an index is used to quickly reduce the size of the sort). The second thing that can reduce the sort time is the existence of an index on the join column(s). Because indexes are always sorted in order, DB2 may use the index on the join column(s) to avoid the sort. It does not use the index to perform the actual joining of the tables, but rather it uses the index to avoid having to sort the list in order to join it.



In this example the tables are joined by an update id (UPD_ID) and update date (UPD_DT). The join columns probably do not provide very good filtering (it is possible that the UPD_ID is a batch job that updates thousands of rows for a given date (UPD_DT). Also, there is no filtering criteria provided for either table. So all of the rows from the MEMBER table are being joined to all of the rows from the ADRESS table based on UPD_ID & UPD_DT.

A Nested Loop join would not be a good choice since there is no index on the join columns. And even if there were an index it may not be a good access path since the join columns do not provide good filtering.

In this case, a much better access path is the Merge Scan join. Since so many rows are going to be joined anyway, the Merge Scan can simply sort both tables and then read the sorted lists from top to bottom 1 time. Also, there is no need for an index when a Merge Scan is used to join the tables together.

There are also 3 new columns to consider on the PLAN_TABLE.

- •SORTN_JOIN is the sort of the new table (i.e. the table listed on this line of the explain – in this example the ADRESS table)
- •SORTC_JOIN is the sort of the composite table (i.e. the table listed in the previous line of explain – in this example the MEMBER table)
- •MERGE_JOIN_COLS is the number of columns that the join is done on (in this case 2 columns – UPD_ID & UPD_DT)

Merge scan join: Why is it good?

IDUG® 2007 North America

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefch	QBLK Type	Parent QBLK	SortN Join	Sort C Join	Merge Cols
1	1	MEMBER	0	R	0		N	S	SEL	0			
1	2	ADRESS	2	R	0		N	S	SEL	0	Y	Y	2
1	3		3						SEL	0			

- **At most how many times is the MEMBER table accessed?**

  Ans: 1 time

- **At most how many times is the ADRESS table accessed?**

  Ans:1 times

- **What happens if the merge scan join doesn't use an index to access the inner table (ADRESS)?**

  Ans: nothing - merge scan doesn't need an index!

GoFurther

19

Because of the Merge Scan, each table is read through at most 1 time. And there is no concern that an Index is not used on the join columns. This may perform very well. Particularly if this is a batch application and this SQL is only executed a limited number of times.

Merge scan join: Why is it not so good?

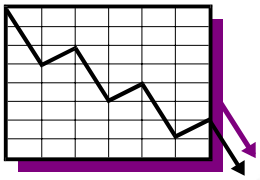
IDUG® 2007 North America

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefch	QBLK Type	Parent QBLK	SortN Join	Sort C Join	Merge Cols
1	1	MEMBER	0	R	0		N	S	SEL	0			
1	2	ADRESS	2	R	0		N	S	SEL	0	Y	Y	2
1	3		3						SEL	0			

2 tablespace scans

Lots of Sequential prefetch

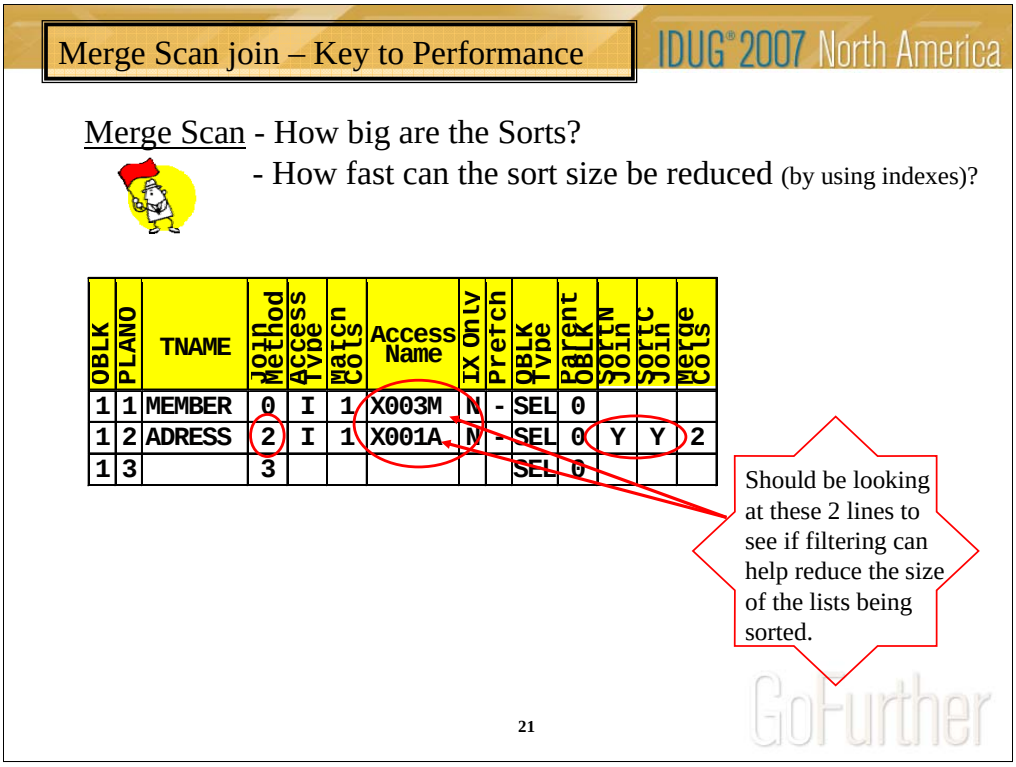
2 costly sorts



GoFurther

20

Although the Merge Scan is a good candidate in this case, it does have drawbacks. For instance, it will scan both the MEMBER and the ADRESS table (only 1 time each, but it does scan the tables from top to bottom). It does this to put the rows into a sorted list. Because of the scans, Sequential prefetch is used in both cases. Although this may be acceptable in a batch process, it would most likely not be a good idea for an online process. The scans, the prefetch and the sorts would take too long.



In a Merge Scan join, the key to good performance involves 2 things.

1) How Big are the Sorts? If there are many qualifying rows from each of the tables, then DB2 has to sort those rows. And of course the bigger the sort, the longer it will take.

2) How fast can the sort size be reduced? The size of the sort(s) can be reduced by applying filtering. (for example ZIP_C = '17888'). This can be done in 1 of 2 ways. Filtering could be done without an available index (which may be slow), or filtering can be done with the aid of an index (which would likely be faster. Either way the sort size is reduced, it's just a question of how fast is the sort size reduced.

In the example above there are indexes being used to provide filtering on both the MEMBER and the ADRESS table (and no Prefetch in either case). This is a good indication that the Merge Scan join will perform well. The filtering will quickly reduce the sort size (by using the indexes). And apparently the size will be sufficiently small that DB2 did not see the need for prefetch. That's a good indication that the sort size will quickly be reduced to a small number of rows. The small number of rows can be quickly sorted and the join performed.

Note: Even though there are 2 indexes being used in the Merge Scan, it is important to note that neither index is actually used in the join operation itself. The indexes are only there to reduce the sort size. There is no index needed to perform the actual join in a Merge Scan join.

Q3: Are there red flags here?


```
SELECT ...
FROM MEMBER M (500,000 rows)
    ,ADRESS A (1,000,000 rows)
WHERE M.FNAME = 'JOCELYN'
  AND M.UPDT_ID = A.UPDT_ID
  AND M.UPDT_DT = A.UPDT_DT
ORDER BY 2 DESC
```

Indexes:

```
MEMBER                      Cluster
  X001M -> SSN               (100)%
  X002M -> LNAME+FNAME        (55)%

ADRESS
  X001A -> SSN               (100)%
```

Added a good filtering predicate, but the entire MEMBER table is still being scanned

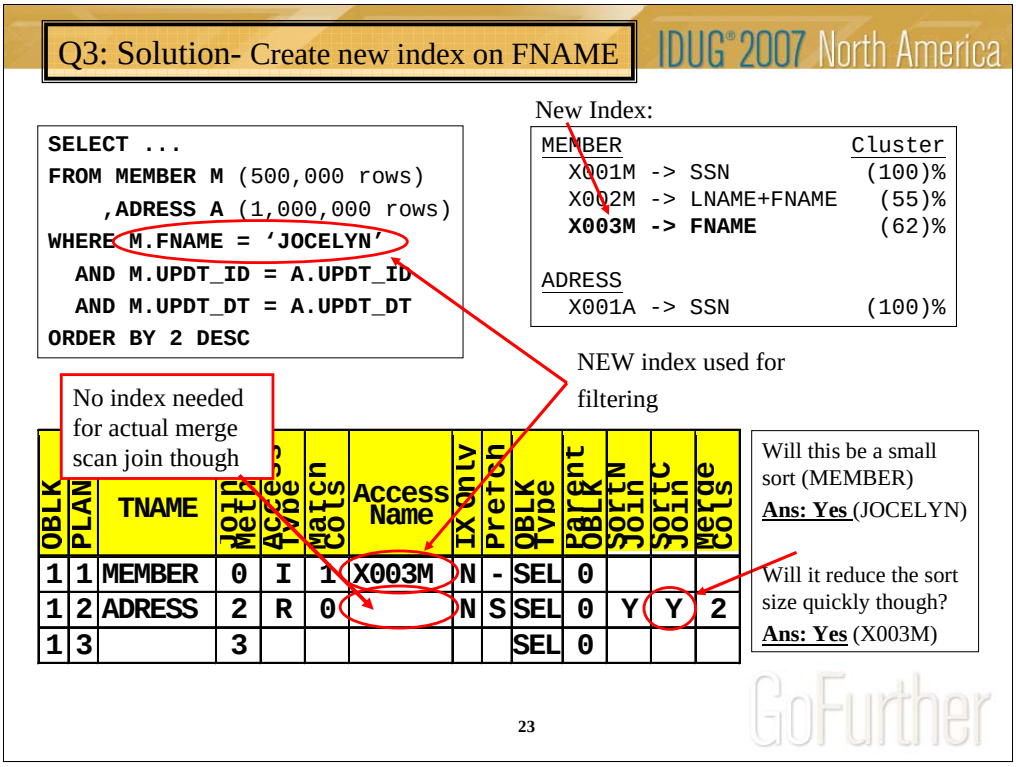
QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK	Sorte Join	Sorte Join	Merge Cols
1	1	MEMBER	0	R	0		N	S	SEL	0			
1	2	ADRESS	2	R	0		N	S	SEL	0	Y	Y	2
1	3		3						SEL	0			

Will this be a small sort (MEMBER)
Ans: Yes (JOCELYN)

Will it reduce the sort size quickly though?
Ans: No (no index)

In this case a filtering predicate is present (FNAME='JOCELYN'). This will reduce the number of rows that need to be sorted from the MEMBER table (perhaps a few dozen will have rows with a first name of 'JOCELYN'). However, the entire MEMBER table must still be scanned to find the rows that have a first name of 'JOCELYN'. (this is because there is no index on FNAME on the MEMBER table). So the scan will take some time, but once it is complete the sort will be quick (it only has to sort a few dozen rows).

This is better because it reduces the size of the sort. However further improvements could be made by adding an index on the FNAME column.



By adding an index on the MEMBER table using the FNAME column, the access path improves further. Now DB2 can use the FNAME index to quickly find the few dozen rows with an FNAME of 'JOCELYN'. The sort will also be quick because it only has a few dozen rows to sort. So the access to the MEMBER table is quite fast.

The access to the ADRESS table may still take some time since it has to scan the entire table and sort it. But because it's a Merge Scan join it will only need to scan the ADRESS table a single time.

This is probably not a bad access path and would work well in a batch process.

Q3: Continued – an even better solution

Add more predicates!

```
SELECT ...
FROM MEMBER M (500,000 rows)
    ,ADRESS A (1,000,000 rows)
WHERE M.FNAME = 'JOCELYN'
  AND A.SSN = '123456789'
  AND M.UPDT_ID = A.UPDT_ID
  AND M.UPDT_DT = A.UPDT_DT
ORDER BY 2 DESC
```

Indexes:

```
MEMBER                        Cluster
  X001M -> SSN                 (100)%
  X002M -> LNAME+FNAME          (55)%
  X003M -> FNAME                (62)%

ADRESS
  X001A -> SSN                 (100)%
```

Filtering predicates (not join predicates)

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefch	QBLK Type	Parent QBLK	SortN Join	Sortc Join	Merge Cols
1	1	MEMBER	0	I	1	X003M	N	-	SEL	0			
1	2	ADRESS	2	I	1	X001A	N	-	SEL	0	Y	Y	2
1	3		3						SEL	0			

Indexes are used for filtering… not for joining.

Both are small sorts now because of the filtering

An additional predicate is added to the SQL (SSN = '123456789'). Now there is a filtering predicate on both the MEMBER and the ADRESS table. And even better yet, the new predicate can also take advantage of the SSN index on the ADRESS table.

The access path shows that indexes are being used to access both the MEMBER and the ADRESS table. This is good because it means that DB2 can quickly find the few dozen rows from the MEMBER table and the few rows (possibly 1 row) from the ADRESS table. Then the few rows from both the MEMBER and the ADRESS table are sorted into separate lists. The Merge Scan join goes through the sorted lists (which are small) and then the answer is returned.

This is a very good access path, and would work well in either a batch or online process.

It is important to note that although indexes are used for both the MEMBER and the ADRESS table, the indexes are NOT used to perform the join between the 2 tables. The join is done using the UPD_ID & UPD_DT columns. Those columns are not part of any index on any table. This is fine because the Merge Scan join does not require an index to perform well.

Q4: Are there red flags here?

IDUG® 2007 North America

Multiple joins – one step at a time

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK	SortN Join	SortC Join	Merge Cols
1	1	DEPT	0	R	0		N	-	SEL	0			
1	2	ADRESS	1	I	0	X003A	N	S	SEL	0			
1	3	MEMBER	2	R	0		N	S	SEL	0	Y	Y	3
1	4		3						SEL	0			

This join first

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK	SortN Join	SortC Join	Merge Cols
		Result of join between DEPT & ADRESS											
1	3	MEMBER	2	R	0		N	S	SEL	0	Y	Y	3
1	4		3						SEL	0			

This join second

- DEPT table has 5 rows
- ADRESS table has 1 million rows
- MEMBER has 500,000 rows

25

GoFurther

When there is more than 1 join within a QBLKNO, then the Joins are performed in PLANNO order. So the join between DEPT & ADRESS is done first, then the join between that result and the MEMBER table will be done next.

This example starts with a nested loop join between the DEPT and ADRESS table. Then it moves to a Merge scan join between that result and the MEMBER table. For Merge Scan joins there are important sort columns:

- SortN Join is a sort of the New table (MEMBER table)
- SortC Join is a sort of the Composite table (Result of join between DEPT and ADRESS)

The real concern is the 2nd line of the Explain that shows the ADRESS table is being joined with a Nested Loop Join. It is using an Index (X003A) to do the join, but it is NOT matching on any columns (MATCH COLS = 0). This results in an index scan for each search of the inner table (ADDRESS). We can also see that Sequential Prefetch is being used here. The Nested loop join in this case will likely be a performance issue.

The scan on the MEMBER table is probably OK since this is part of a Merge Scan join and does not need an index on the join columns for good performance. It would be better if an index were used to provide filtering to reduce the sort size though.

Note: One good thing is that the Nested Loop join is coming FROM the DEPT table (which only has 5 rows). So at most we will scan the ADRESS table 5 times (still not good, but maybe not too terrible either).

Q5: Are there red flags here?

IDUG® 2007 North America

Multiple joins – one step at a time

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK	SortN Join	SortC Join
1	1	MEMBER	0	R	0		N	S	SEL	0	-	-
1	2	DEPT	2	R	0		N	S	SEL	0	Y	Y
1	3	PHONE	1	I	3	X003P	N	L	SEL	0	-	-
1	4	ADRESS	2	R	0		N	S	SEL	0	Y	Y
1	5	MEMBER	1	I	1	X001M	Y	-	SEL	0	-	-
1	6	COVGS	1	I	0	X002C	Y	S	SEL	0	-	-
1	4		3						SEL	0	-	-

- MEMBER has 500,000 rows
- DEPT table has 5 rows
- PHONE has 750,000 rows
- ADRESS table has 1 million rows
- COVGS has 2.5 million rows

26

There are many steps to this explain, lots of table scans and lots of prefetch. There are a couple of places that might be concerns.

1.On line 1-3 of the Explain is a Nested Loop join which uses an index matching on 3 columns. That is good, except that it also uses List prefetch. This means DB2 expects to retrieve a lot of rows. This will occur for each iteration of the Nested loop and could indicate a performance problem. A better index that provides better filtering may need to be created.

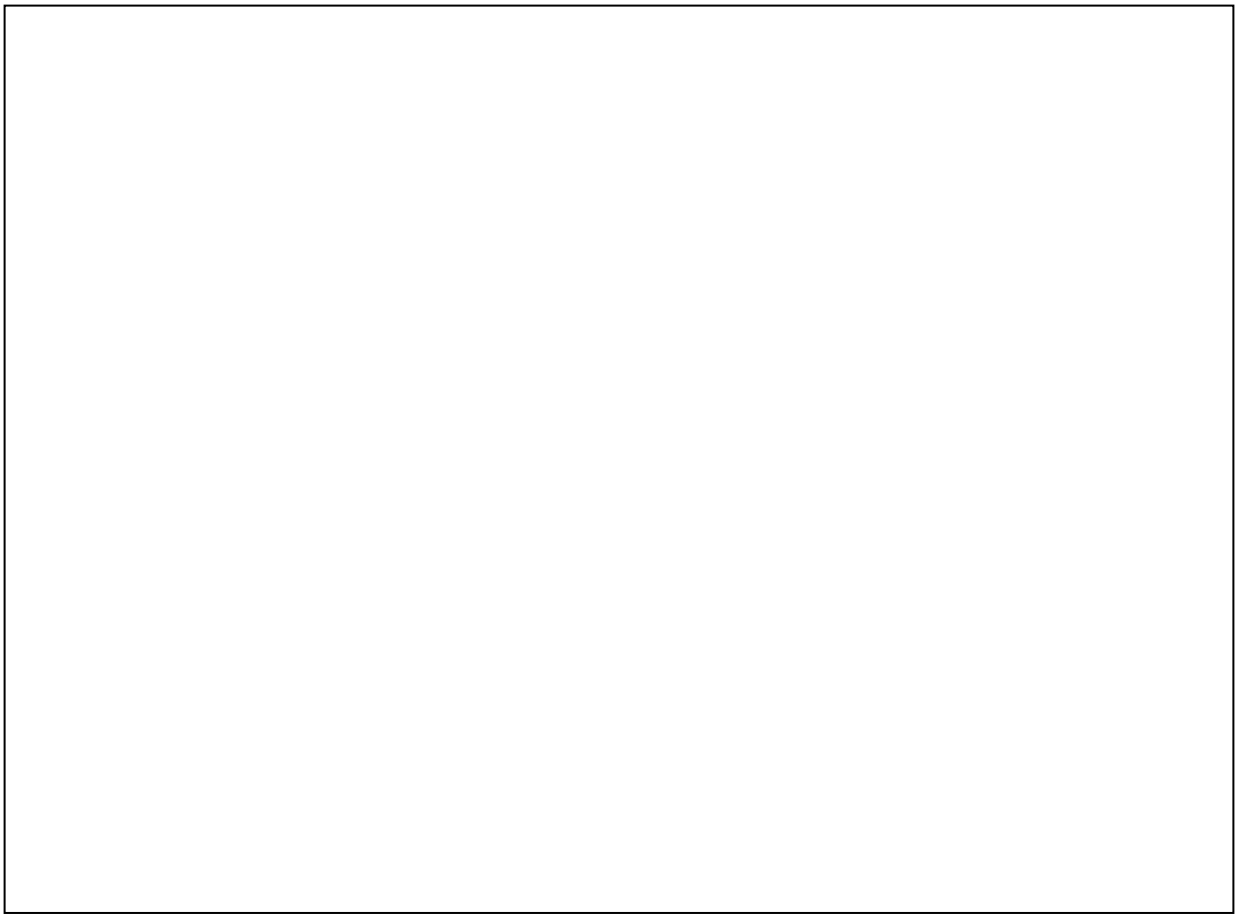
2.There are Merge Scan joins on lines 1-2 and 1-4 of the Explain. This may or may not be a big concern. The first Merge scan join is between the MEMBER and DEPT table. The DEPT table is only 5 rows so it's not a concern. The member table has 500,000 rows, which means that it will be scanned 1 time and a sort will be performed (there could be many rows that need to be sorted, we can't tell without the SQL). So it may or may not be an issue. The same is true for the merge scan join on line 1-4 of the Explain. So there may be some performance impact from these merge scan joins, but they are not the major concern in the access path.

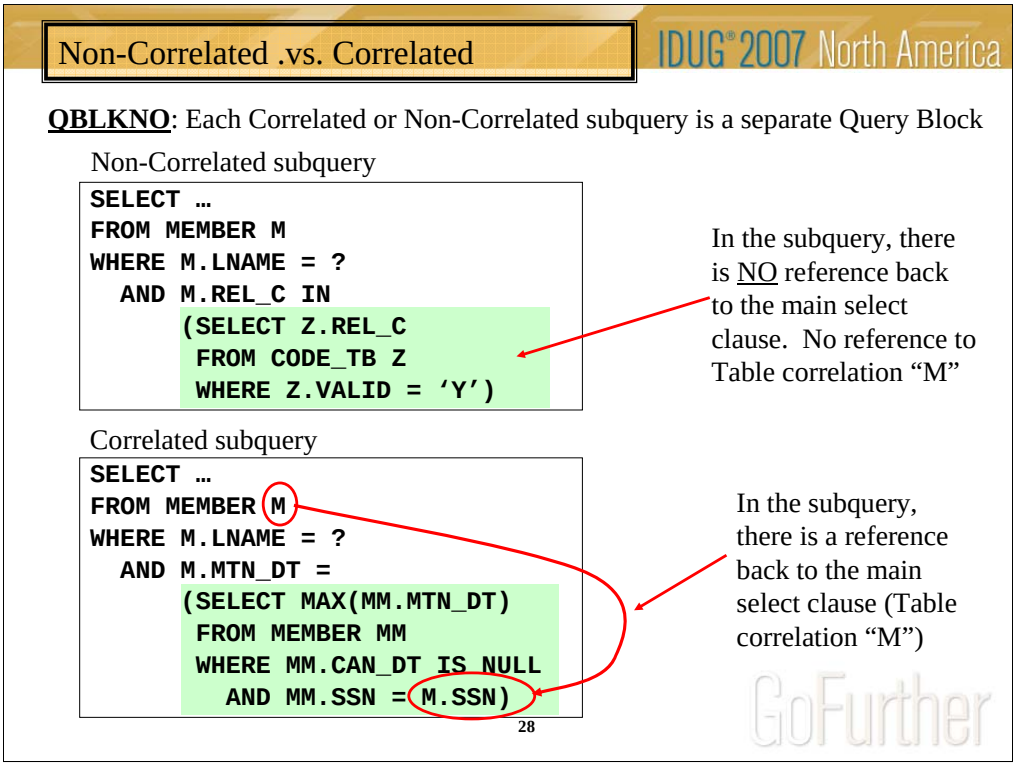
By far, the biggest concern is on line 1-6 of the Explain. This is a Nested loop join to the COVGS table. The join is using an index, but it is matching on 0 columns (MATCHCOLS = 0). This means that for each search of the COVGS table there will be an index scan (from top to bottom). And because it's a nested loop join there may be multiple searches (possibly hundreds, thousands or even millions). This is a clear performance issue and needs to be addressed.

Topics to be covered

1) EXPLAIN review
2) Nested Loop / Merge Scan Join
3) Correlated / Non-Correlated Subquery
4) Access Path Red Flag Review

GoFurther

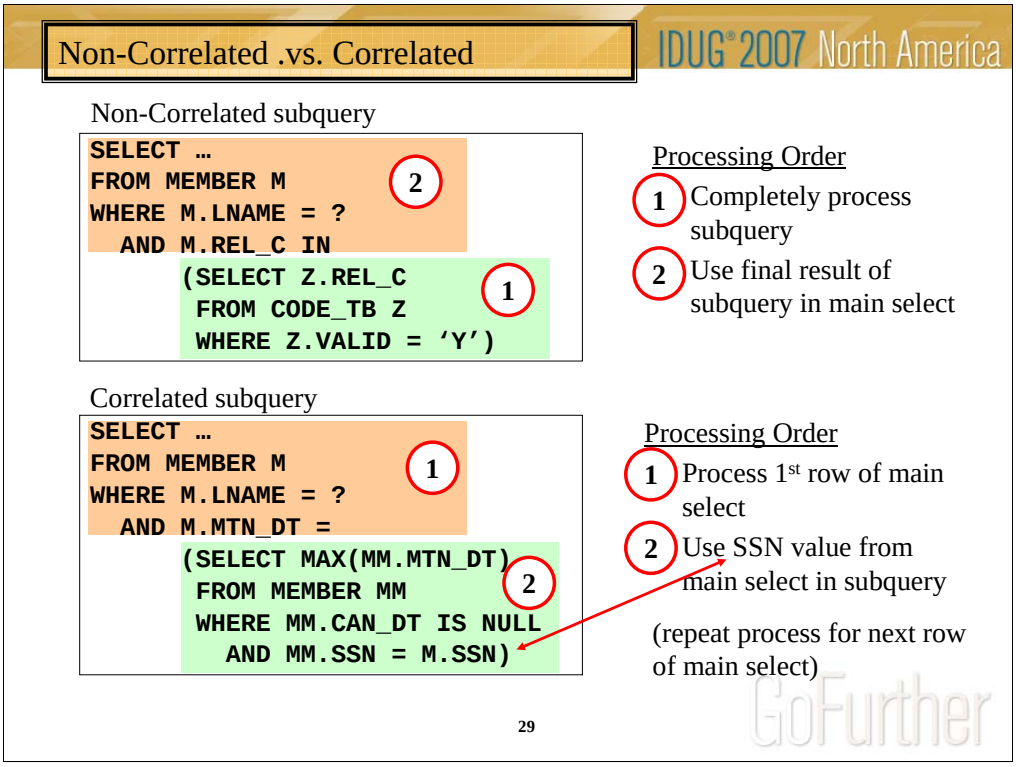




Non-Correlated and Correlated subqueries look similar, but they are really very different.

The Non-correlated subquery can be thought of as a stand alone query. It does not make any reference to the main select clause (there is no correlation to the main select). All of the predicates are local to the subquery itself. In the example above, all of the predicates in the non-correlated subquery refer to the CODE_TB table. None of them reference the main select clause table (MEMBER table).

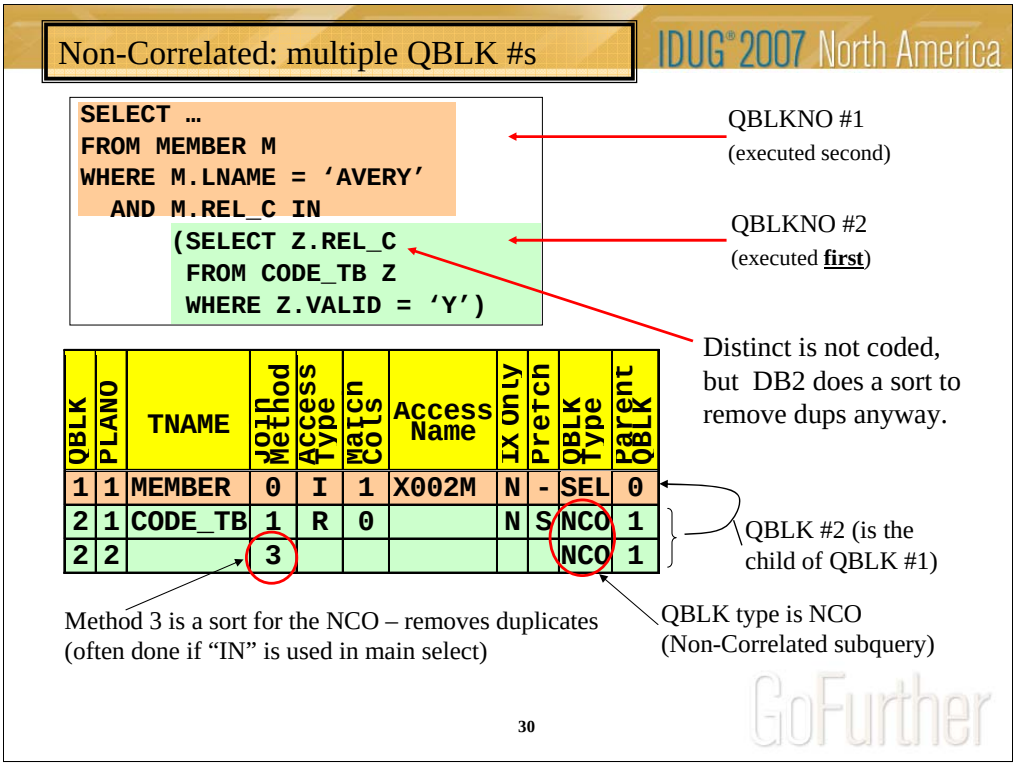
The Correlated subquery however does reference the main select clause. It has a predicate that refers to the MEMBER table (AND MM.SSN = M.SSN). The M.SSN refers to table M which is the MEMBER table from the main select clause. This means there is a correlation between the subquery and the main select clause.



One of the biggest differences between a non-correlated and correlated subquery is the order that the pieces are processed in.

Non-Correlated: In the non-correlated subquery, the subquery is completely processed first. The result of the subquery is materialized prior to processing the main select. It is important to note that the subquery is ONLY done 1 time.

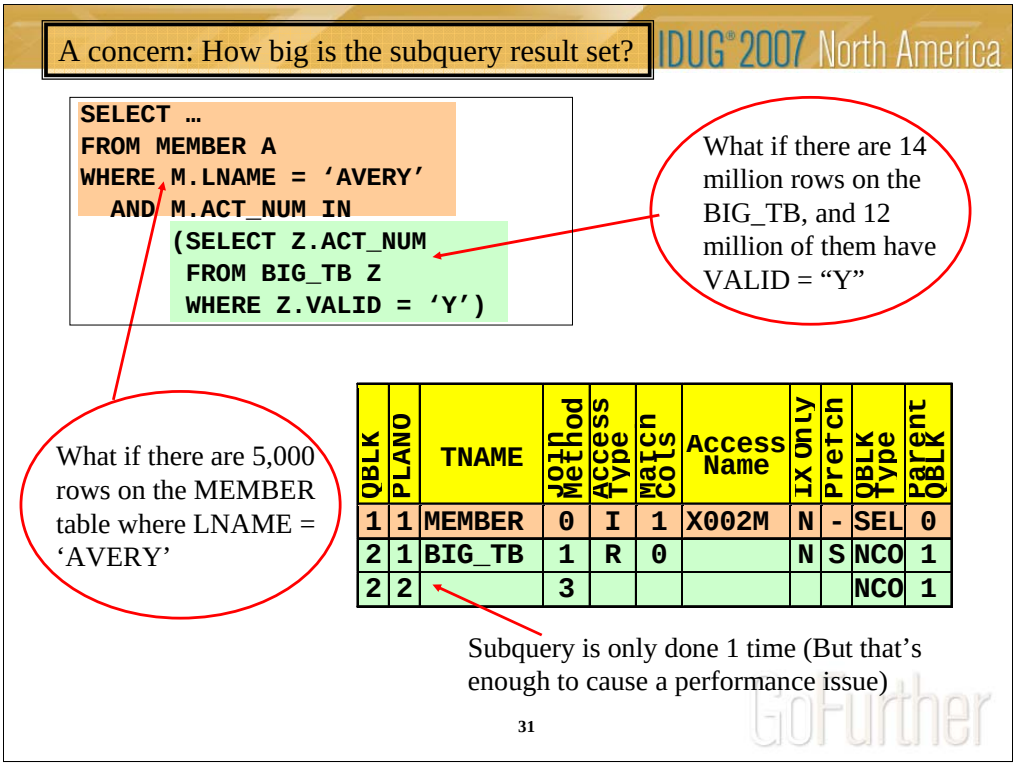
Correlated: In the correlated subquery, processing starts with the main select. It will process local predicates first and then start accessing the correlated subquery. It will execute the correlated subquery for each qualifying row of the main table. So it may execute the correlated subquery hundreds or thousands of times depending on how many rows qualify from the main table. So in the example above, if there are 10,000 rows that qualify from the main table(M.LNAME=?), then the query will execute the correlated subquery 10,000 times. So it is important that the correlated subquery execute efficiently.



In QBLK#2 (the non-correlated subquery) there is a Method=3 which is a sort. However there is no distinct, union or orderby in the query.

The sort is done internally by DB2 to remove any duplicates that might exist in the subquery. So, the subquery retrieves all the valid REL_C values and then sorts them to remove any duplicates in the list. Then the list is used by the main select.

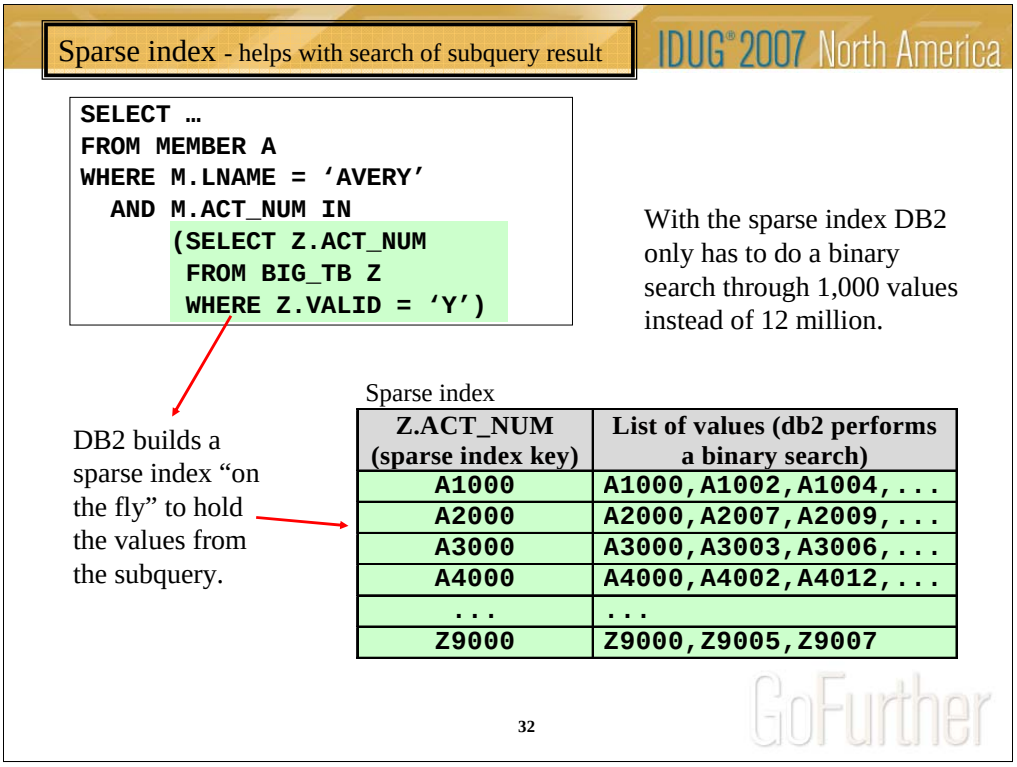
The sort is almost always done when an IN is used to access the subquery. If an "exists" or "equal" predicate were used instead, then the sort would not be seen.



If the result of the subquery is a 12 million rows, then those rows need to be sorted to remove duplicates. That will likely be a long running sort.

And once the sort is done and duplicates are removed there may still be a large result set (for example 6 million rows). The result of the subquery is then interrogated by the main select for each qualifying row of the main select. So for example if there are 5,000 rows with a last name of 'AVERY', then that means DB2 will interrogate the result set of the subquery 5,000 times. (That means DB2 will look through 6 million values 5,000 times).

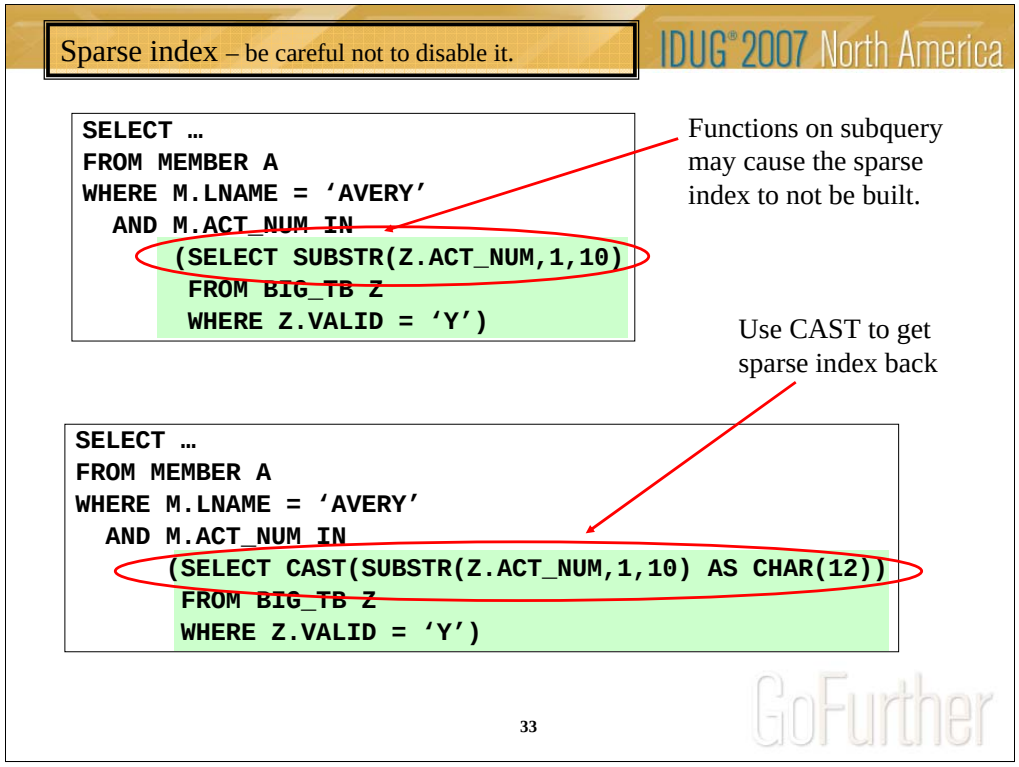
Note: There is some help in looking through the 6 million values though. The following slides cover the concept of a sparse index that is created by DB2 on the fly to help with the search.



DB2 has the ability to build a Sparse index on the values returned from the subquery. This can greatly enhance the performance when there are a large number of values in the subquery result.

The sparse index has keys based on ranges of values. So in the example above, the sparse index will have a key on every 1,000th value (this is just an example for illustration purposes, how DB2 creates the sparse index in reality is probably much different). Each keyed entry has a corresponding list of values (so each keyed entry would have at most a list that contains 1,000 values).

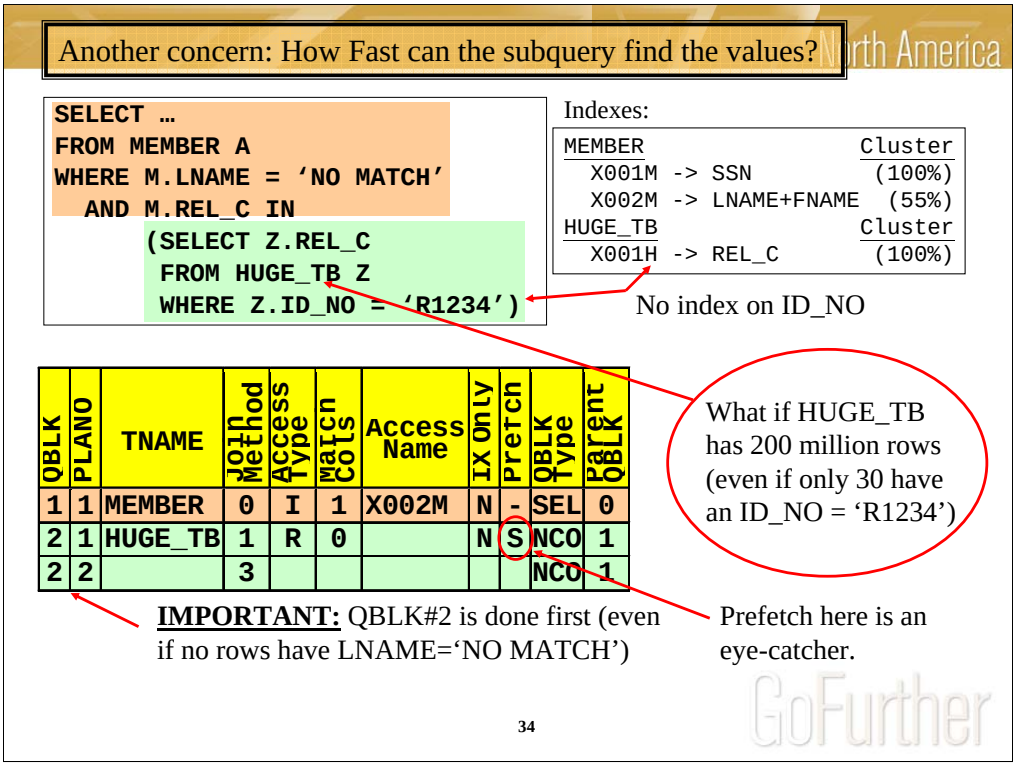
When DB2 starts to process the outer query it will use the sparse index to see if the ACT_NUM is in the subquery. So for example if a MEMBER row had an ACT_NUM of "A3006", then DB2 would quickly navigate the sparse index to the A3000 entry, and then do a binary search through the list associated with the A3000 entry (this would be a binary search on at most 1,000 values). That is much faster than having to look through all 12 million values that the subquery may return.



One thing to be aware of is that using functions in the subquery can cause DB2 to NOT build the sparse index. This can have a significant negative impact on performance. Without the sparse index, DB2 will have to look through all 12 million values.

However, the CAST function can re-enable the use of the sparse index. The CAST function should be used to CAST the subquery value to be the exact length of the column in the outer table. In this case the CAST function is used to CAST the ACT_NUM to be a CHAR(12) which is the exact length of the ACT_NUM column in the MEMBER table.

Note: Credit should go to Suresh Sane from DST systems for coming up with the idea of using the CAST function to get DB2 to build the sparse index.



General performance rules apply when looking at a non-correlated subquery. The important eye catcher is the Prefetch column. In this case the HUGE_TB table is being scanned and this incurs Sequential prefetch (Prefetch = 'S'). This may be a problem if the table is large (which it is in this case).

In a non-correlated subquery the subquery is done first, and it is **only done 1 time**. So if the HUGE_TB is only 15 rows then a single scan of the table is not a concern. However if the HUGE_TB is 200 million rows, then even a single scan could be a problem. Especially if this were some type of online transaction.

IMPORTANT: non-correlated subqueries are only executed 1 time which is good. However, they are also always executed first. This means that even if there are other predicates that will result in no rows ever coming back from the main select, the subquery is still executed. So in the example above even if there are no rows with LNAME = 'NO MATCH', the subquery still gets executed. So it scans 200 million rows to create a subquery result, and then finds out that there are no rows on the MEMBER table with LNAME = 'NO MATCH'. That's a lot of rows read just to return a +100 no rows found.

Non-Correlated: Key to Performance

IDUG 2007 North America

Non-Correlated -How big is the subquery (how many values does it return)?

-How fast will the subquery build the list of values?

```
SELECT …
FROM MEMBER A
WHERE M.LNAME = 'AVERY'
  AND M.REL_C IN
      (SELECT Z.REL_C
       FROM CODE_TB Z
       WHERE Z.VALID = 'Y')
```

We want to keep the number of values returned by the subquery as low as possible (and find them quickly)

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	CODE_TB	1	I	1	X003C	N	-	NCO	1
2	2		3						NCO	1

35

GoFurther

1. How big is the subquery result? If the subquery returns many values then those values are put in a temporary area and sorted to remove duplicates. A large sort can take time. Also, after the rows are sorted DB2 still has to look through the list for each row that qualifies from the main table (in this case the MEMBER table).

For example: If the subquery returns 500,000 rows, then the 500,000 rows are sorted to remove duplicates. If that leaves 400,000 rows, then that is still a large result set. Once the 400,000 row result set is materialized, then DB2 goes through the main table (MEMBER table). For each row on the MEMBER table that has an LNAME of "AVERY", DB2 will look through the 400,000 row result set (the result set that was created by the subquery). If there are 5,000 rows on the MEMBER table with an LNAME of "AVERY", then DB2 has to look through the 400,000 row result set each time (5,000 searches of a 400,000 row result set – note: DB2 may build a sparse index to help with this search).

2. The other main factor in performance is how "fast" DB2 can find the subquery result set. If the table has 75 million rows on it and the subquery scans those rows then that will take some time. Even if only 5 rows actually meet the subquery requirements, if it has to scan 75 million rows to find the 5 that qualify, that will take some time.

Even if no rows have an LNAME='AVERY', the subquery is still executed first. So DB2 may scan 75 million rows to build the subquery result. After that it will check to see if any rows on the MEMBER table have an LNAME='AVERY'. And there may be none.

Q6: Are there red flags here?


```
SELECT ...
FROM MEMBER M
WHERE M.LNAME >= 'SMITH'
  AND M.FNAME IN
      (SELECT CC.FNAME
       FROM COVGS CC
       WHERE CC.LNAME = 'YOUNG')
```

Indexes:

```
MEMBER                      Cluster
  X002M -> LNAME             (100%)
COVGS
  X001C -> SSN               (100%)
```

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	S	SEL	0
2	1	COVGS	0	R	0		N	S	NCO	1
2	2		3					-	NCO	1

Why is sequential prefetch used?

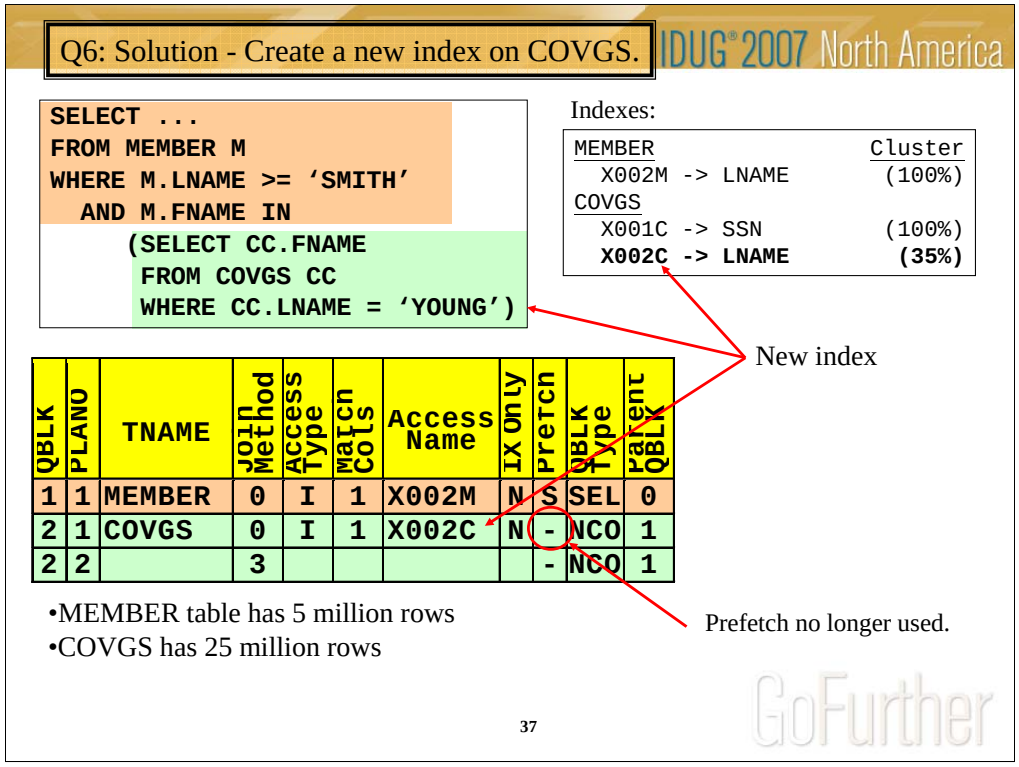
- MEMBER table has 5 million rows
- COVGS has 25 million rows

How many times will the COVGS table be scanned?

In this case, there are a couple of concerns. The first is the prefetch on the MEMBER table. This is done even though a 1 column index match is being used. This is because the predicate LNAME >= 'SMITH' will qualify a large number of rows. There is nothing we can do to correct this situation (unless additional predicates are available).

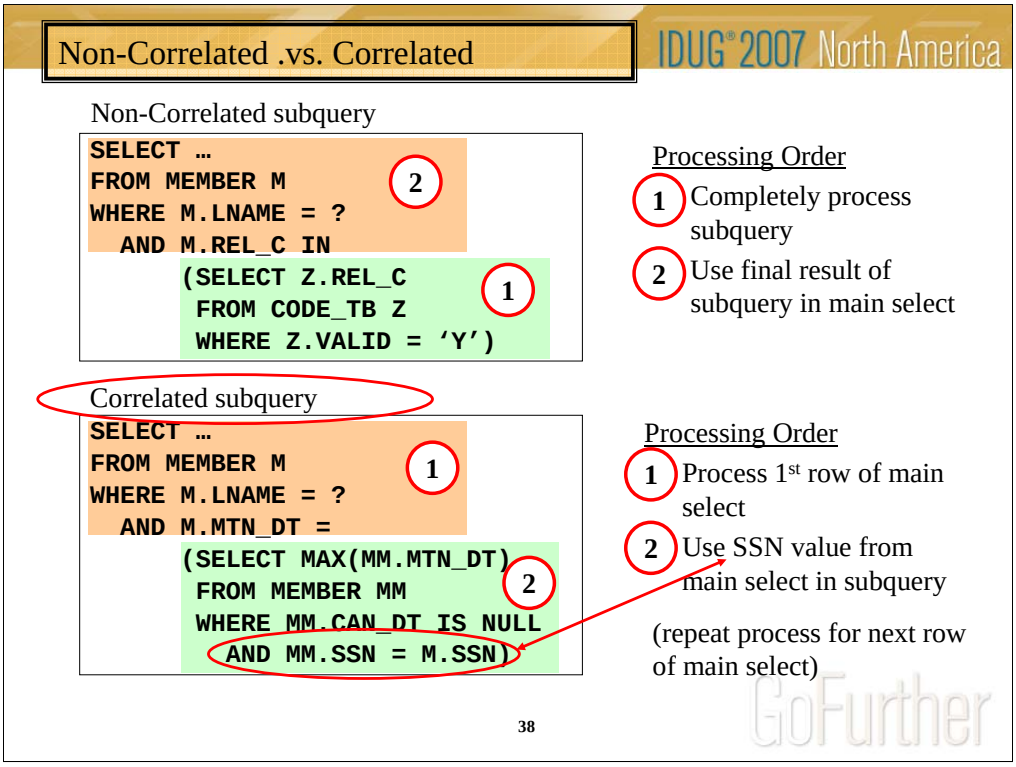
The bigger concern however is the access to the COVGS table in the non-correlated subquery. The COVGS table is being scanned and it has 25 million rows in it. Since this is a non-correlated subquery, the scan will only happen 1 time. However scanning 25 million rows is still a fairly big concern.

The result set of the query will be sorted, but it may not be too big since only those rows with a LNAME = 'YOUNG' will be sorted.



A solution to the problem is to add a NAME index on the COVGS table. Now the explain shows that the non-correlated subquery will access the COVGS table using an index access. And because so few rows have a NAME = 'YOUNG' there is no prefetch needed.

There is still prefetch on the MEMBER table but that cannot be avoided in this case.



One of the biggest differences between a non-correlated and correlated subquery is the order that the pieces are processed in.

Non-Correlated: In the non-correlated subquery, the subquery is completely processed first. The result of the subquery is materialized prior to processing the main select. It is important to note that the subquery is ONLY done 1 time.

Correlated: In the correlated subquery, processing starts with the main select. It will process local predicates first and then start accessing the correlated subquery. It will execute the correlated subquery for each qualifying row of the main table. So it may execute the correlated subquery hundreds or thousands of times depending on how many rows qualify from the main table. So in the example above, if there are 10,000 rows that qualify from the main table(M.LNAME=?), then the query will execute the correlated subquery 10,000 times. So it is important that the correlated subquery execute efficiently.

Correlated: multiple QBLK #s

IDUG® 2007 North America

QBLKNO: Each Correlated or Non-Correlated subquery is a separate Query Block

Correlated subquery

```
SELECT …
FROM MEMBER M
WHERE M.LNAME = ?
  AND M.MTN_DT =
      (SELECT MAX(MM.MTN_DT)
       FROM MEMBER MM
       WHERE MM.CAN_DT IS NULL
         AND MM.SSN = M.SSN)
```

QBLKNO #1
(executed first)

QBLKNO #2
(executed second)

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	MEMBER	0	I	1	X001M	N	-	COR	1

QBLK type is COR
(Correlated subquery)

39

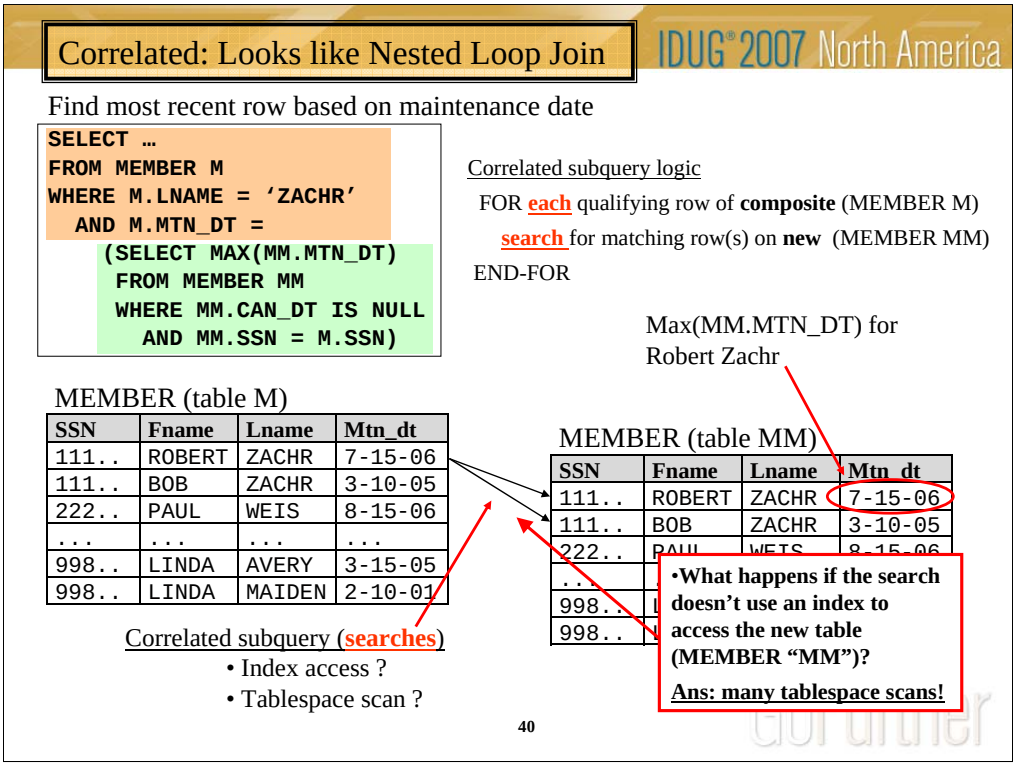
GoFurther

There are a number of things that can cause DB2 to divide a single SQL statement into multiple Queryblocks (QBLKS). A correlated or non-correlated subquery will cause multiple query blocks.

In the example above there will be a QBLK for the main select, and also a QBLK for the subquery (in this case a correlated subquery). This is a correlated subquery because the subquery refers back to the main select (MM.SSN = M.SSN)

Within each QBLK there may be multiple steps (multiple PLANNOs – for joins or sorts).

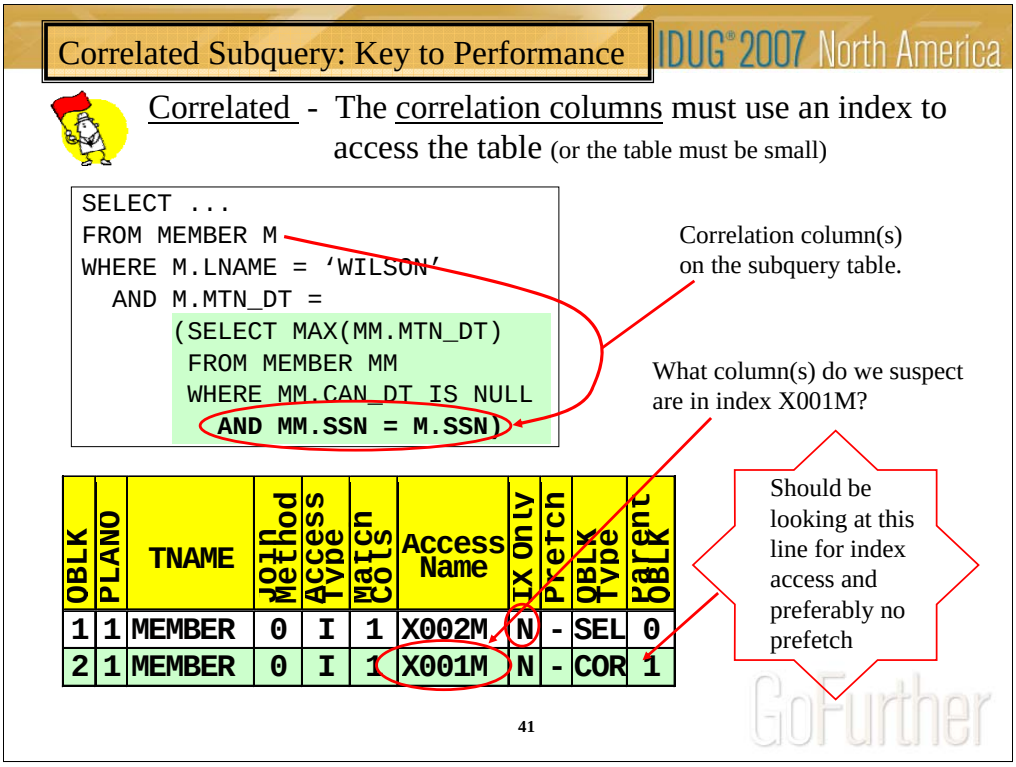
IMPORTANT: The QBLKNOs may not be execute in numerical order. However, within each QBLKNO the PLANNOs will be done in numerical order. In the example above however, the QBLKS are executed in numerical order. Also, if there are multiple steps within either of the QBLKs, the multiple steps would be performed in PLANNO order (within each individual QBLK).



Correlated subqueries execute in a fashion that is very similar to a Nested loop join. Qualifying rows from the outer table, called the **composite** table (in this case the MEMBER "M" table) are found. Then for each of those rows the inner table, called the **new** table, is searched (in this case the same table – MEMBER "MM"). The key factor is that the inner table is searched. If the search is done efficiently using an index then performance will be good. However, if the inner table is searched by a tablespace or non-matching index scan then performance will be bad (unless the inner table is very small).

What this means is that if there are 500 rows on the MEMBER table with a LNAME= 'ZACHR', then it will search the inner table 500 times. If the search is a tablespace scan, then that is 500 tablespace scans.

This is very similar to the same performance concerns we look for in Nested loop joins (which were discussed earlier in the presentation).



When looking at a correlated subquery, the most important factor is the access to the subquery table. The reason for this is that the SQL will search the subquery 1 time for each qualifying row of the outer table. There should almost always be an index used to access the subquery table (unless the table is very small). Also, if there is prefetch during the access to the subquery table, then that may indicate that even though an index is being used, the index does not provide much filtering. In other words the access to the subquery table will match many rows even though an index is used. This could be a performance issue as well (although probably not quite as bad as a full tablespace scan).

In the case above if 500 rows match LNAME='WILSON', then the SQL will execute the subquery 500 times. If the subquery access is not efficient then it will severely degrade the performance. Index access by the correlation columns is a key to good performance. In this case the correlated reference is on the SSN column. So an index on SSN should be present on the MEMBER table to allow for efficient access.

If no SSN index was on the MEMBER table, then the subquery would scan the MEMBER table 500 times. If the MEMBER table is 5 million rows, then that is 500 scans of a 5 million row table.

So the key to good performance for the correlated subquery is to have the access done by an index that provides sufficient filtering that prefetch is not needed (as in the example shown above)

Q7: Are there red flags here?


```
SELECT …
FROM MEMBER M
WHERE M.LNAME = 'WILSON'
  AND M.MTN_DT =
      (SELECT MAX(MM.MTN_DT)
       FROM MEMBER MM
       WHERE MM.CAN_DT IS NULL
         AND MM.SSN = M.SSN)
```

Indexes:

```
MEMBER                          Cluster
  X002M -> LNAME+FNAME           (55)%
```

There is no index by SSN on the MEMBER table

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	MEMBER	0	R	0		N	S	COR	1

How big a concern is this scan? (ans: major issue)

•MEMBER table has 5 million rows

The reason the correlated subquery is not using an index is because there is no index on the MEMBER table by SSN. And the correlation reference is on the SSN column (MM.SSN = M.SSN).

This will be a very long running query since the subquery does a scan of a 5 million row table multiple times (once for each qualifying row of the outer table (LNAME = 'WILSON')

Q7: Solution - Create a new index on SSN


```
SELECT …
FROM MEMBER M
WHERE M.LNAME = 'WILSON'
  AND M.MTN_DT =
      (SELECT MAX(MM.MTN_DT)
       FROM MEMBER MM
       WHERE MM.CAN_DT IS NULL
         AND MM.SSN = M.SSN)
```

Indexes:

```
MEMBER                          Cluster
  X001M -> SSN                   (100)%
  X002M -> LNAME+FNAME            (55)%
```

Add a new index

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	MEMBER	0	I	1	X001M	N	-	COR	1

Much better access path

•MEMBER table has 5 million rows

Prefetch no longer used.

The solution in this case is to add an index on the MEMBER table for the SSN column. After the index is added the access path shows that the subquery is using the index to efficiently search the MEMBER table.

This will be a much better access path. So if 5,000 rows qualify from the outer table, then it it will do 5,000 index probes into the subquery table (which will be fairly quick).

Q8: Are there red flags here?


```
SELECT ...
FROM COVGS C
WHERE C.SSN LIKE '123456%'
  AND EXISTS
    (SELECT 1
     FROM DEPT DD
     WHERE DD.DEPT_N = C.DEPT_N
       AND DD.DEPT_NAME = ?)
```

Indexes:

```
COVGS                          Cluster
  X001C -> SSN                 (100%)
DEPT
  X001D -> DEPT_CODE           (100%)
```

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK
1	1	COVGS	0	I	1	X001C	N	-	SEL	0
2	1	DEPT	0	R	0		N	S	COR	1

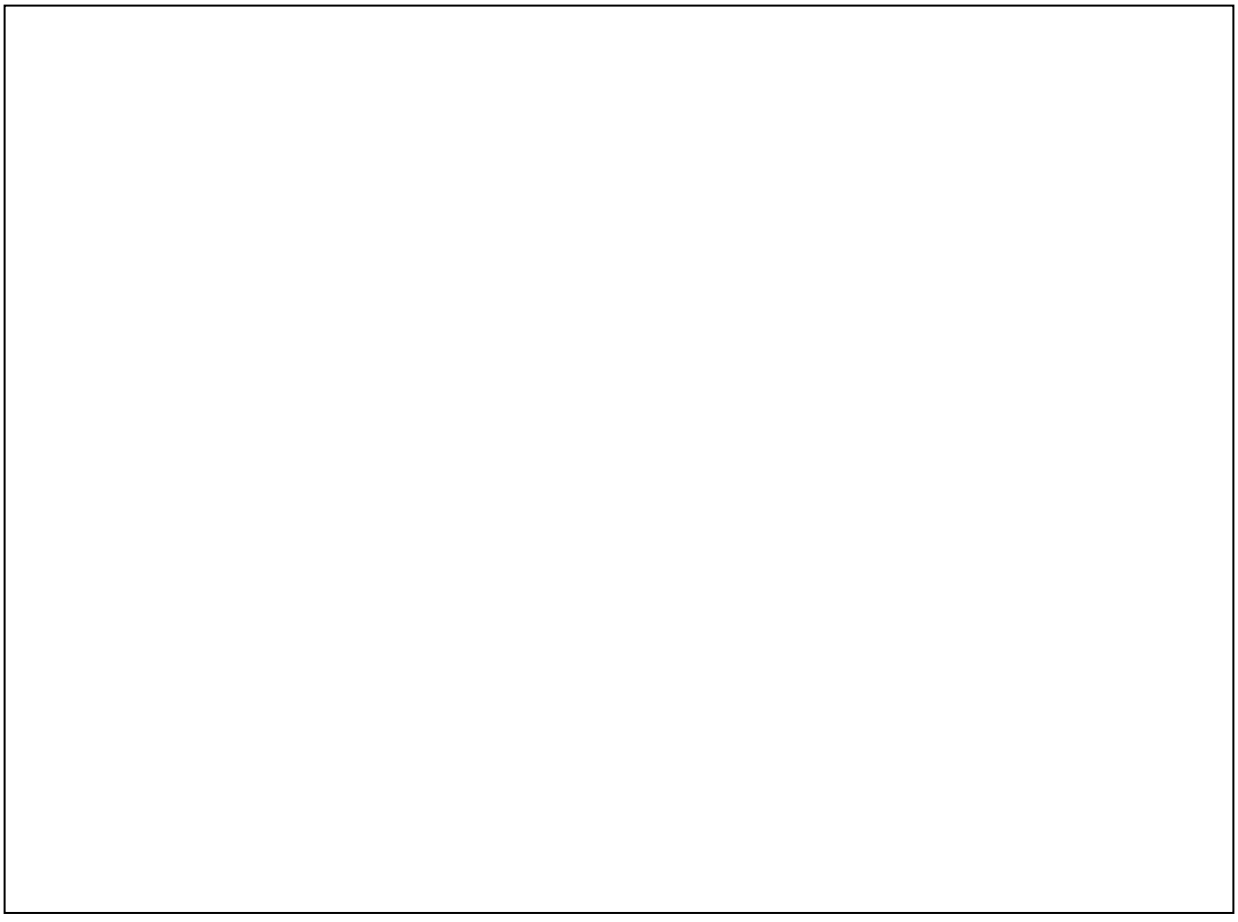
- COVGS has 25 million rows
- DEPT has 5 rows

Initially it may seem like there is a concern because the correlated subquery is searching the DEPT table using a tablespace scan. If the DEPT table were large this would definitely be a problem. However, because the DEPT table only has 5 rows this is not a concern.

This access path is fine.

Topics to be covered

1) EXPLAIN review
2) Nested Loop / Merge Scan Join
3) Correlated / Non-Correlated Subquery
4) Access Path Red Flag Review



Access Path Red Flag Review

Nested loop - to perform well requires index access on join columns to the inner table (unless inner table is very small)

Merge Scan - How big are the Sorts?
- How fast can the sort size be reduced (by using indexes)?

Non-Correlated -How big is the subquery (how many values does it return)?
-How fast will the subquery build the list of values?

Correlated - The correlation columns must use an index to access the table (or the table must be small)

The access path red flags that were reviewed during the presentation. Each particular type of SQL has a different performance characteristic to focus on.

Review Exercise #1

IDUG® 2007 North America

```
SELECT ...
FROM MEMBER M
     PHONE  P
WHERE M.SSN = '123456789'
  AND M.EMPID = P.EMPID)
```

Indexes:

```
MEMBER                          Cluster
  X001M -> SSN (UNIQUE)         (100%)

PHONE
  X001P -> PHONE_NO             (100%)
```

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X001M	N	-	SEL	0
1	2	PHONE	1	R	0		N	S	SEL	0

Unique index insures that at most 1 row will qualify from the MEMBER table.

How many times will the PHONE table be scanned?

- MEMBER has 5 million rows
- PHONE has 750,000 rows

47

GoFurther

This is a fairly simple looking nested loop join. Initially it appears that there is a serious performance issue since the PHONE table is being joined WITHOUT the use of an index. This normally means multiple scans of the PHONE table would be needed. The PHONE table would be scanned 1 time for each qualifying row of the outer table (MEMBER).

However, in this case because the MEMBER table is accessed using an UNIQUE index, the most rows that will qualify from the outer table is 1 (possibly none if the SSN doesn't exist on the MEMBER table). That means at most there will be a single scan of the PHONE table during the Nested loop join.

This access path may be OK for batch processes (a single scan of a 750,000 row table).

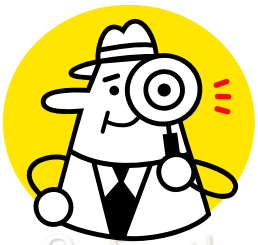
Review Exercise #2

Are there any warning flags in this Access Path?

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	ADRESS	0	I	0	X001M	N	S	COR	1

- MEMBER has 5 million rows
- ADRESS has 10 million rows

Non-Matching index scan on a 10 million row table (performed multiple times)



48

This is a correlated subquery. So the subquery will be executed multiple times (possibly thousands of times). So the access to the subquery table (ADRESS) must be efficient. However it looks like there is a non-matching index scan when accessing the ADRESS table. And the SEQUENTIAL prefetch indicator is on. Since the ADRESS table has 10 million rows on it, the scan will take time. And if it has to scan the index multiple times then it will take a very long time.

Possibly a new index on the ADRESS table should be created, or additional WHERE predicates should be added to the SQL.

Review Exercise #3

Are there any warning flags in this Access Path?

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Pretch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	ADRESS	0	I	0	X001M	N	S	NCO	1
2	2		3					-	NCO	1

- MEMBER has 5 million rows
- ADRESS has 10 million rows

Note: the subquery on the ADRESS table only returns **7 rows** out of the 10 million.

Non-Matching index scan is a concern, but it is only done once

GoFurther

49

This is a Non-correlated subquery. So the subquery will only be executed 1 time. So the scan on the 10 million row ADRESS table is a bit of a concern. But it's not nearly as big a concern as the previous example when a correlated subquery was being used.

Also, apparently there are only 7 rows that the subquery returns so the sort will be quick.

However, for optimal performance, perhaps a new index should be considered on the ADRESS table, or maybe predicates can be added to the SQL to take advantage of existing indexes.

Review Exercise #4

Are there any warning flags in this Access Path?

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
2	1	ADRESS	0	I	0	X001M	N	S	NCO	1
2	2		3					-	NCO	1

- MEMBER has 5 million rows
- ADRESS has 10 million rows

Note: the subquery on the ADRESS table returns **8 million** of the 10 million rows

Non-Matching index scan is a concern, but it is only done once. But the 8 million row sort is a big concern.

50

This is a Non-correlated subquery. So the subquery will only be executed 1 time. So the scan on the 10 million row ADRESS table is a bit of a concern. But it's not nearly as big a concern as the previous example when a correlated subquery was being used.

However, there are apparently 8 million rows that will be returned by the subquery. This will require some time to sort (JOIN METHOD = 3). Also, once it is sorted to remove duplicates DB2 will have to look through the large result set to see if there are matches.

This is likely a performance issue. An attempt should be made to add additional predicates to the subquery to reduce the number of rows returned by the subquery.

And for optimal performance a new index should be considered on the ADRESS table to allow the suquery to use index access.

Review Exercise #5

IDUG® 2007 North America

Are there any warning flags in this Access Path?

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	IX Only	Prefch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
1	2		3					-	SEL	0
2	1	COVGS	0	R	0		N	S	NCO	1
2	2		3					-	NCO	1
3	1	ADRESS	0	R	0		N	S	COR	1

- MEMBER has 5 million rows
- COVGS has 25 million rows
- ADRESS has 10 million rows

The tablescan on the correlated subquery is probably the biggest concern. (although the scan and sort on COVGS is also troublesome)

51

GoFurther

This access path has both a non-correlated and a correlated subquery. Both of the subqueries appear to have concerns.

The non-correlated subquery (QBLK #2) is an access to the 25 million row COVGS table. This is a scan of the table and a sort is also done to remove duplicates. However because it is a non-correlated subquery it is only done 1 time.

The correlated subquery is an access to the 10 million row ADRESS table. This is also a scan of the table, but no sort is needed since duplicates are not removed.

Initially the larger concern may seem to be the non-correlated subquery access to the 25 million row table. However this is scanned at most 1 time. The bigger concern is actually the correlated subquery on the 10 million row ADRESS table. Because it is a correlated subquery it may be executed many times (possibly thousands of scans of a 10 million row table).

Indexes should be considered for both the non-correlated and correlated subqueries (especially the correlated subquery).

The SQL that created this access path is on the next slide.

Review Exercise #5 (continued)


```
FROM MEMBER M
WHERE M.LNAME = 'MILLER'
  AND M.SSN IN
      (SELECT CC.SSN
       FROM COVGS CC
       WHERE ACTIVE = 'Y')
  AND M.SSN =
      (SELECT MAX(AA.SSN)
       FROM ADRESS AA
       WHERE AA.LNAME = M.LNAME)
ORDER BY FNAME,SSN
```

QBLK	PLANO	TNAME	Join Method	Access Type	Match Cols	Access Name	Ix Only	Prefetch	QBLK Type	Parent QBLK
1	1	MEMBER	0	I	1	X002M	N	-	SEL	0
1	2		3					-	SEL	0
2	1	COVGS	0	R	0		N	S	NCO	1
2	2		3					-	NCO	1
3	1	ADRESS	0	R	0		N	S	COR	1

QBLK#2 is a scan of a 25 million row table. However it's a non-correlated subquery so it is only done 1 time.

QBLK#3 is a scan of a 10 million row table. In this case it's a correlated subquery so it may be done many times (possibly thousands of times)

This access path has both a non-correlated and a correlated subquery. Both of the subqueries appear to have concerns.

The non-correlated subquery is an access to the 25 million row COVGS table. This is a scan of the table and a sort is also done to remove duplicates. However because it is a non-correlated subquery it is only done 1 time.

The correlated subquery is an access to the 10 million row ADRESS table. This is also a scan of the table, but no sort is needed since duplicates are not removed.

Initially the larger concern may seem to be the non-correlated subquery access to the 25 million row table. However this is scanned at most 1 time. The bigger concern is actually the correlated subquery on the 10 million row ADRESS table. Because it is a correlated subquery it may be executed many times (possibly thousands of scans of a 10 million row table).

Indexes should be considered for both the non-correlated and correlated subqueries (especially the correlated subquery).

THANK YOU FOR ATTENDING!

Session: G08

Tuning Access Paths: Red Flags to Look for!

Joe Burns

Highmark Inc.

joseph.burns@highmark.com

GoFurther