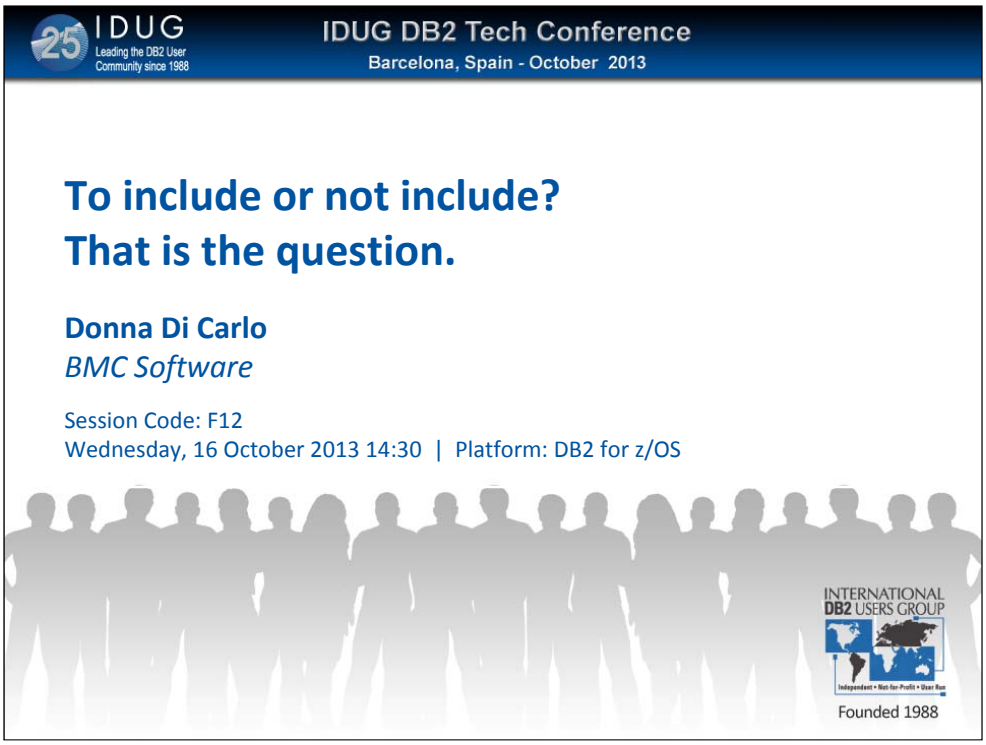
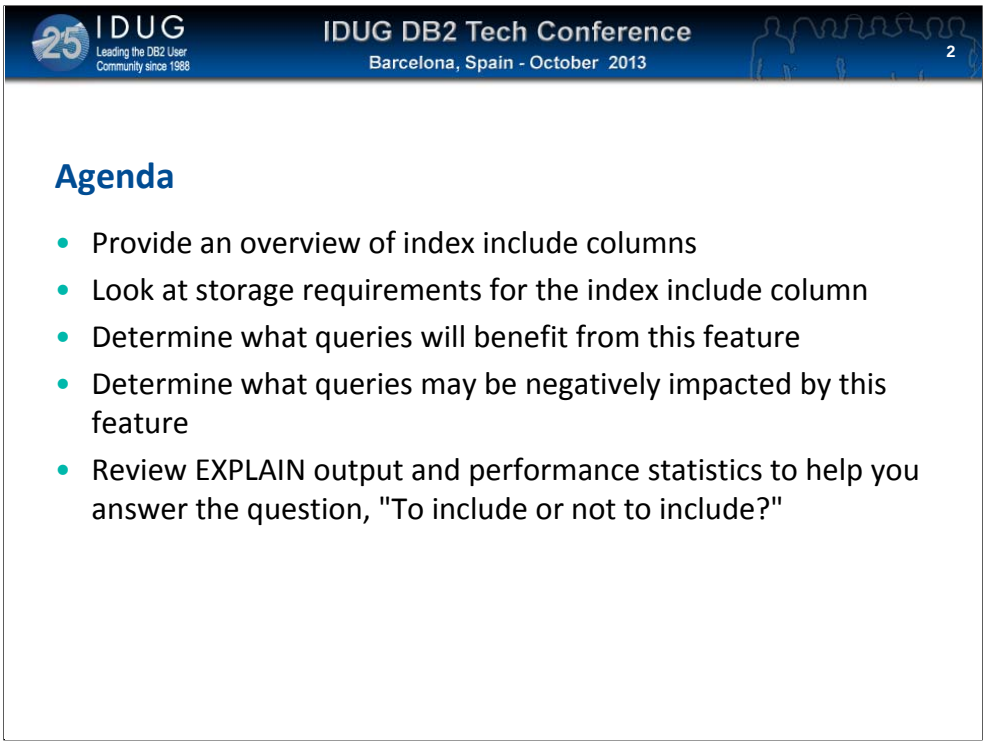


DB2 V10 introduced the ability to include non-key columns in a unique index, allowing index only access without affecting the unique key. That's great for the transaction benefiting from index only access, but what about the other transactions that access this index? Can they be negatively affecting?

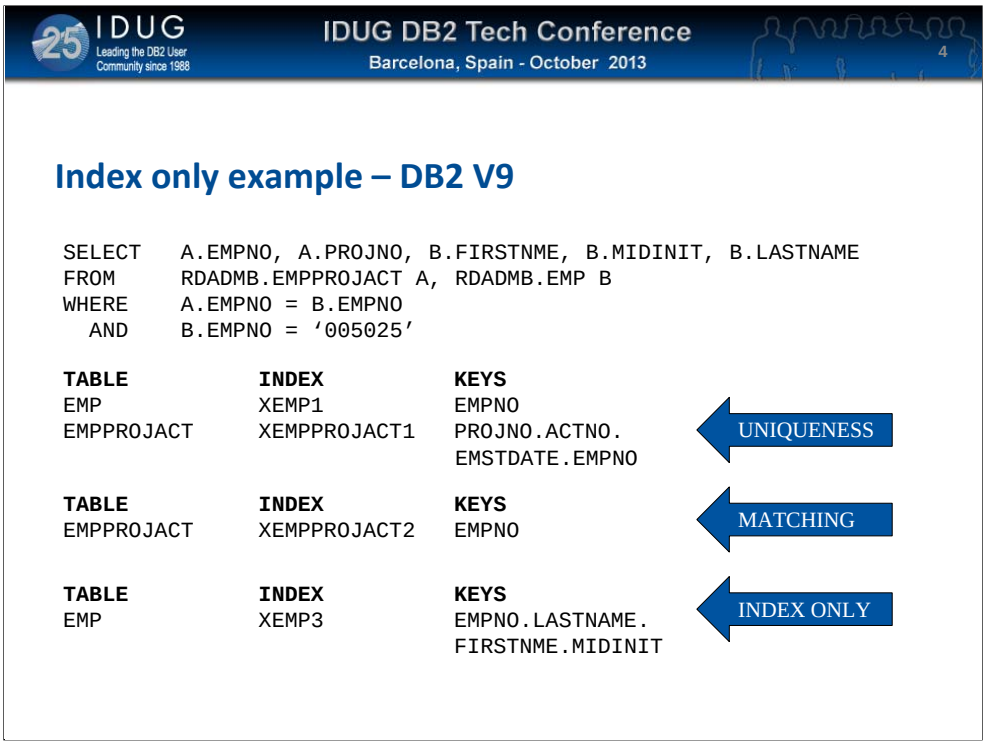


DB2 10 introduce the ability to add INCLUDE columns to a unique index. It can allow you to eliminate some redundant indexes without affecting query performance. You will save storage and save on INSERT and DELETE performance. But there are situations where adding an INCLUDE column can hurt query performance. In this presentation, we will look at why we might want to use this feature, how to implement it and how to measure your success.

Reasons for defining an index

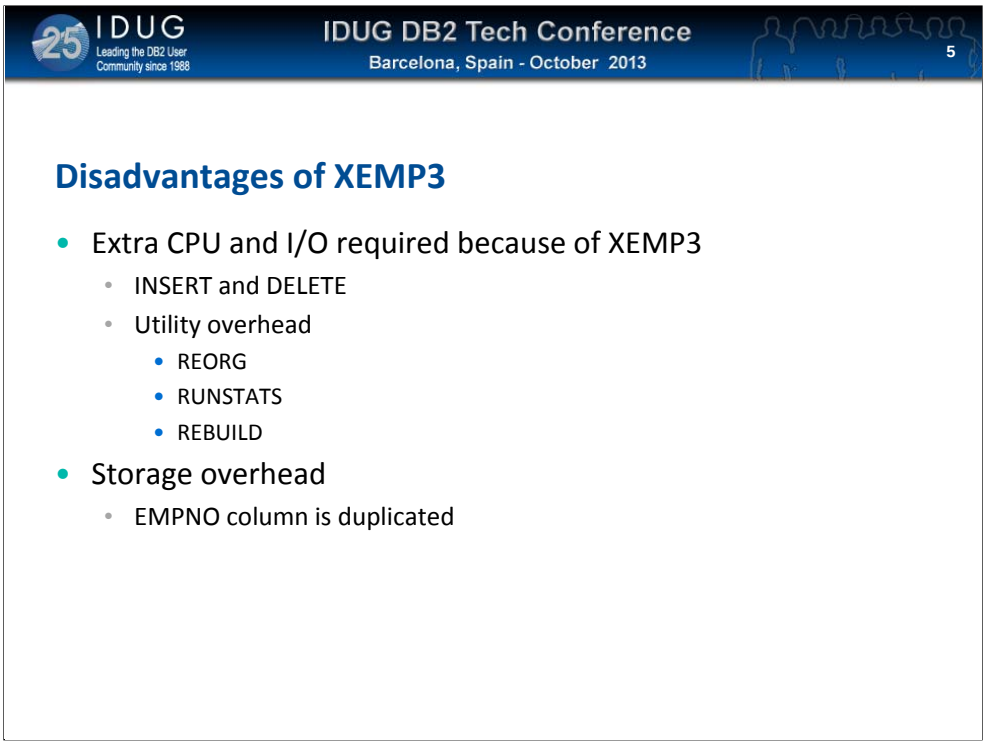
- Enforce uniqueness
- Ensure physical clustering of table data
- Provide quicker access to the data
  - Matching column access
  - Index screening
    - Index access without matching columns
  - Index only access
  - Other index methods

There are a lot of different reasons for defining an index. You can use an index to enforce uniqueness on your data or you can use it to cluster your table data and positively influence prefetch access. You can also define an index to give you quicker access to your data. A matching column access allows you drill directly down to your table data. Index screening uses the index to eliminate data that needs to be returned by your SQL. This usually occurs when you use a range type predicate that matches a column in your index. The less data you have to transmit between the application and DB2, the faster the response time. And then there's Index only access. This is where all the columns in your query can be found in the index. There is no need to reference the table space. This type of access can be quite desirable.



In this example, I have a query that joins the project activity table with the employee table to return all the project time reported by an individual. The employee table must have unique employee numbers, so I defined XEMP1 as a unique index. And the project activity table must have a unique index because an employee can't have duplicate project numbers within a week. These two indexes also help with table access. I get a matching index scan with XEMP1 and index screening with XEMPPROJACT1. But I think that I can improve things even more with a matching index scan on the project activity table. I want to go one step further and get index only access to the employee table by adding FIRSTNME, MIDINIT and LASTNAME to XEMP1. But if I do that, I will destroy the uniqueness rule. The only way to get index only access is to create another index, XEMP3. So now I have duplicated EMPNO in the two indexes.

```
METH ACC MTCH IX TBNAME     IXNAME
   0 I      1 N  EMP        XEMP1
   1 I      0 Y  EMPPROJACT XEMPPROJACT1
METH ACC MTCH IX TBNAME     IXNAME
   0 I      1 N  EMP        XEMP1
   1 I      1 N  EMPPROJACT XEMPPROJACT2
METH ACC MTCH IX TBNAME     IXNAME
   0 I      1 Y  EMP        XEMP3
   1 I      1 N  EMPPROJACT XEMPPROJACT2
```



I may have improved the performance of this query, but I could possibly caused other problems. If I have a lot of inserts or deletes executed against the employee table, I have cause added CPU and possibly I/O because I now have to process an extra index. I have also increased the elapsed time of my utilities. The number of indexes on a single table space increases the sort size. Sort performance is exponential. When sort is bad, it's really bad.

I have also increase my storage requirements because I have duplicated EMPNO. In this example, I have 500K rows and duplicated 12 bytes. A rough calculation shows that I have increased the storage requirements by 8 cylinders and that does not include other overhead that needs to be considered.

DB2 10 Index with INCLUDE columns

- Allows you to add non-key columns to a unique index
- Does not
  - Affect uniqueness
  - Enforce Referential Integrity constraint
- Does save resources by eliminating extraneous indexes

```
CREATE UNIQUE INDEX RDADMB.XEMP1
  ON    RDADMB.EMP(EMPNO ASC)
            INCLUDE(LASTNAME)
```

Looks identical to:

```
CREATE UNIQUE INDEX RDADMB.XEMP3
  ON    RDADMB.EMP(EMPNO ASC, LASTNAME ASC)
```

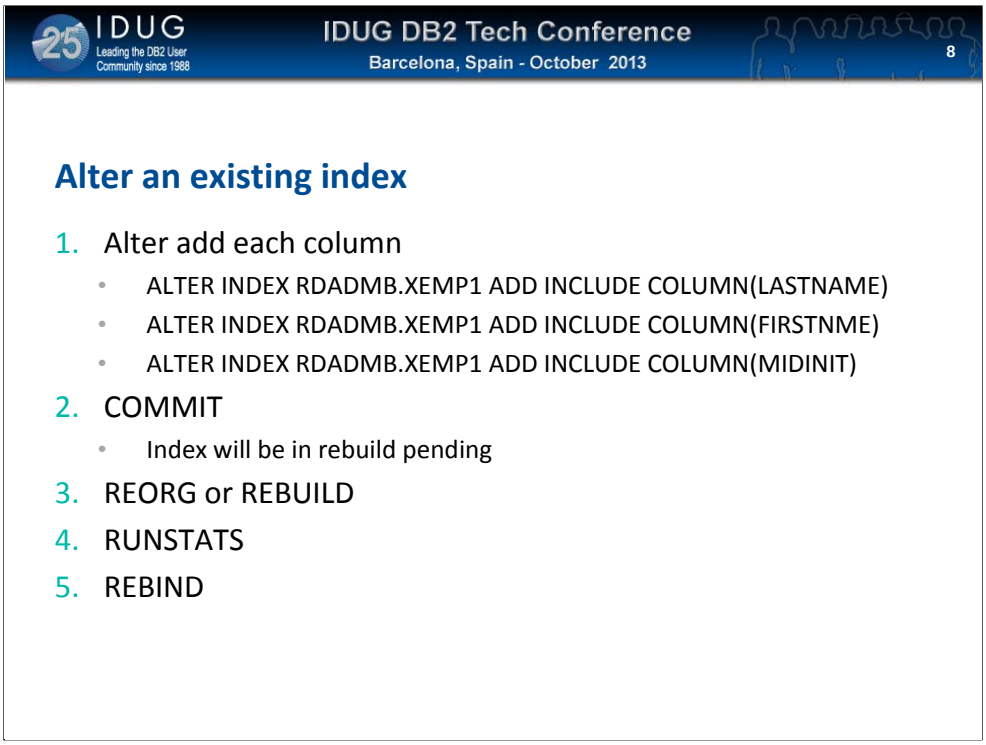
DB2 10 introduced the concept of the INCLUDE column. This allows you to add non-key type columns to a unique index. You no longer have to duplicate an existing index to get index only access. You can now drop any existing duplicate indexes. This save the overhead on inserts and deletes and with utilities. It also save on storage.

In this example, I have included LASTNAME to XEMP1. It will maintain uniqueness on EMPNO and it will store LASTNAME as a non-key column. If I wanted to add FIRSTNME and MIDINIT, I would have to alter the index 2 more times. You can only include one column per alter.

Restrictions

- Index must be UNIQUE
- Index can't be
  - System defined catalog index
  - Index controlled partitioning index
    - Partitioned index on table controlled partitioned space OK
  - Auxiliary index
  - Index on a foreign key
  - XML index
  - Index that has an expression
  - Hash overflow index
- INCLUDE column can't be LOB or DECFLOAT
- Can't ALTER ADD key if index has INCLUDE columns

In order to add INCLUDE columns, the index must be UNIQUE. UNIQUE WHERE NOT NULL is not allowed.



In many cases, you will probably alter an existing index. You must realize when you alter your index, you will take an outage for that index because it will be placed in rebuild pending status.

Here are the steps that you to take to add three include columns to XEMP1. First you need to perform an alter for each INCLUDE COLUMN. Once you execute the commit, the index will be set in rebuild pending status. Any new dynamic SQL will ignore this index. Any static SQL that was bound to use this index will fail. A REORG INDEX or REBUILD will remove the rebuild pending status.

Catalog changes

- Catalog
  - SYSKEYS.ORDERING
    - A – ascending
    - D – descending
    - R – random
    - Blank – INCLUDE column
  - SYSINDEXES.UNIQUE_COUNT
    - New column
    - 0 – no INCLUDE columns
    - > 0 – Number of columns used for uniqueness

DB2 10 added a new value for the ORDING column in SYSKEYS. This column designates if the columns in the index is ascending, descending or random order. If ORDERING is blank, it means the index column is an include column.

SYSINDEXES added a new column called UNIQUE_COUNT. It tells you if the index has INCLUDE columns. If it does, it tells you how many of the index columns are key columns.

Physical page changes

- Index header page
  - HPGIFLGS2.HPGIINCLCOLIDX – yes if index has INCLUDE columns
  - HPGIOLTH – maximum ordered key length
  - HPGIORDCOLS – number of columns used for uniqueness
- Leaf page

XEMP1

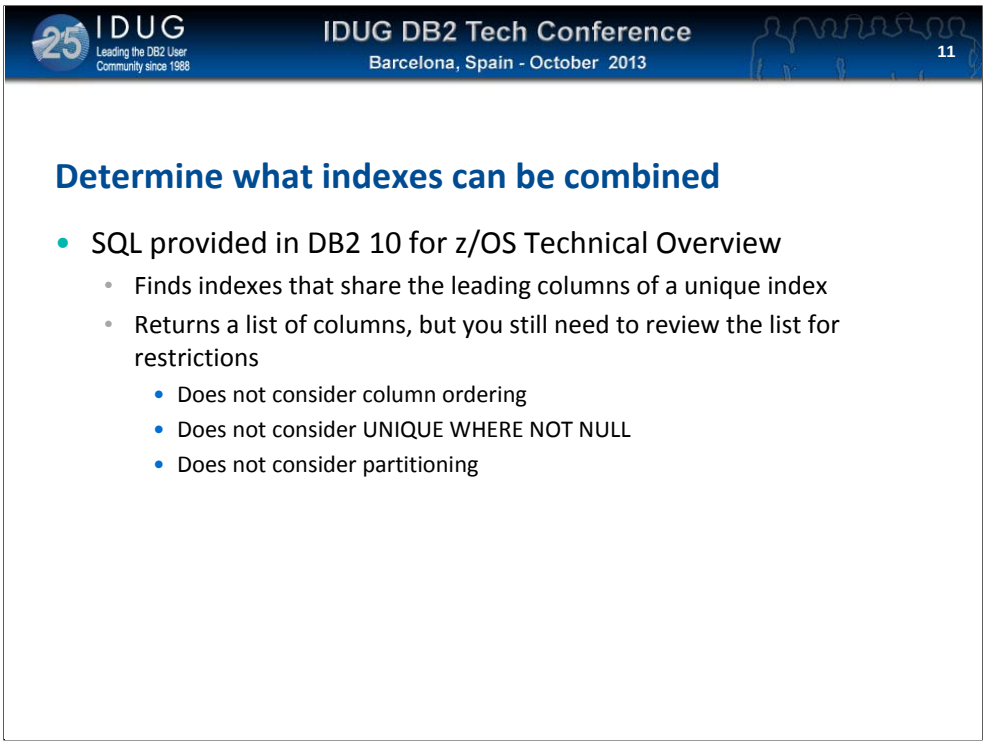
EMPNO	LASTNAME	FIRSTNME	MIDINIT	RID

XEMP3

EMPNO	LASTNAME	FIRSTNME	MIDINIT	RID

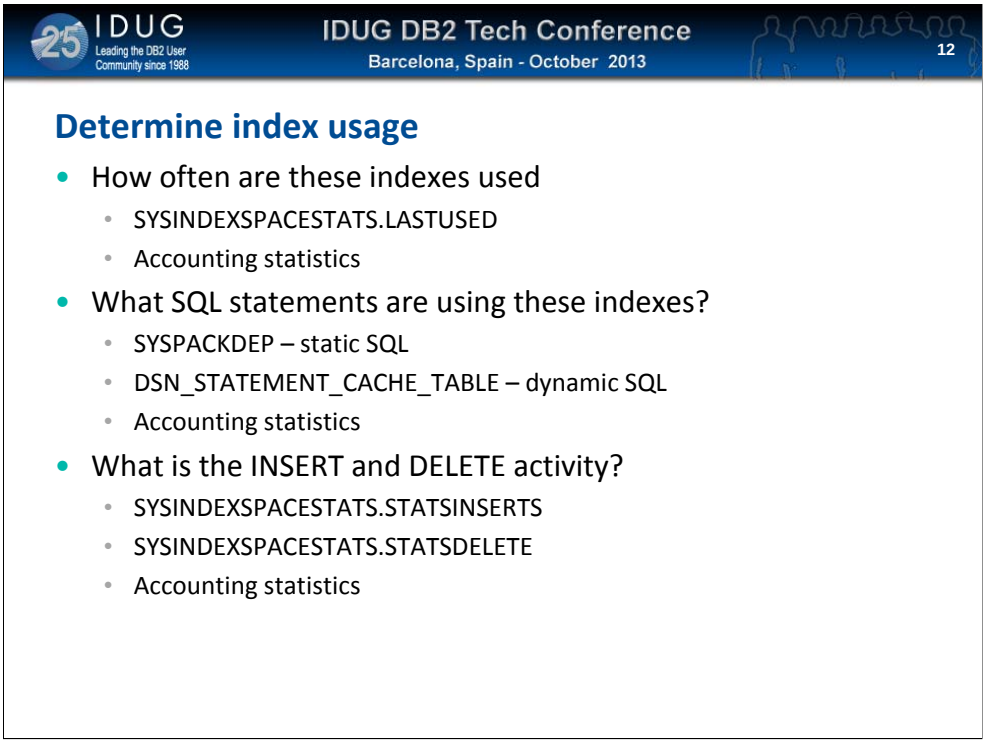
The index with include columns looks almost identical to the index that defines all the columns as keys. The header page contains a few new fields. There is a flag that indicates the index has include columns. There are also two new fields that give the maximum length of the key and the number of key columns.

If you were to look at a leaf page, you will find that the key columns are always the first columns. But, you would not be able to tell if all the columns were keys or some of the columns were include columns.



The first step in implementing INCLUDE columns is identifying the indexes that can be combined. The DB2 10 for z/OS Technical Overview provides a very nice SQL statement that will search your catalog and tell you what indexes might be eligible for INCLUDE columns. What is does is look for indexes with leading columns that match the key columns of a unique index. It's a great start, but you will need to verify that the indexes and columns follow the restrictions.

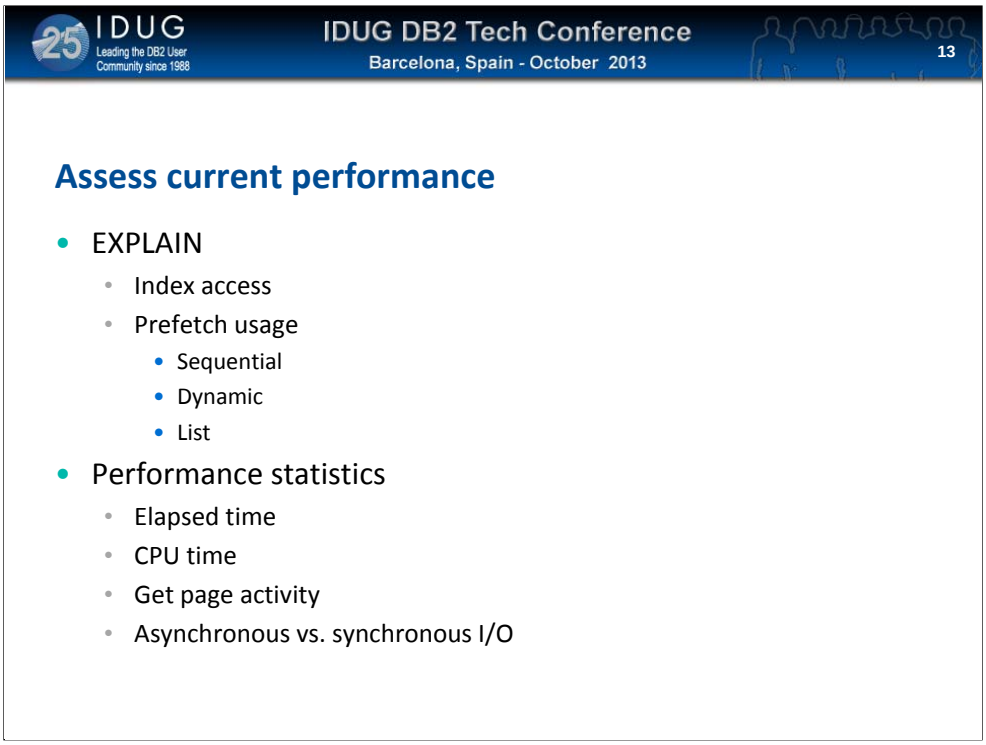
The SQL doesn't consider column ordering. If one column is ascending and the matching column is descending, you really don't have a match. It doesn't consider UNIQUE WHERE NOT NULL. Internally, these types of indexes look different than a UNIQUE index and can't have INCLUDE columns. It also doesn't distinguish between index or table controlled partitioning.



Now that you have a list of indexes to combine, you will need to take a closer look at how these indexes are used. You can use the Real Time Statistics column LASTUSED to see if SQL is using this index. Or you can use accounting statistics to determine if the index is being used.

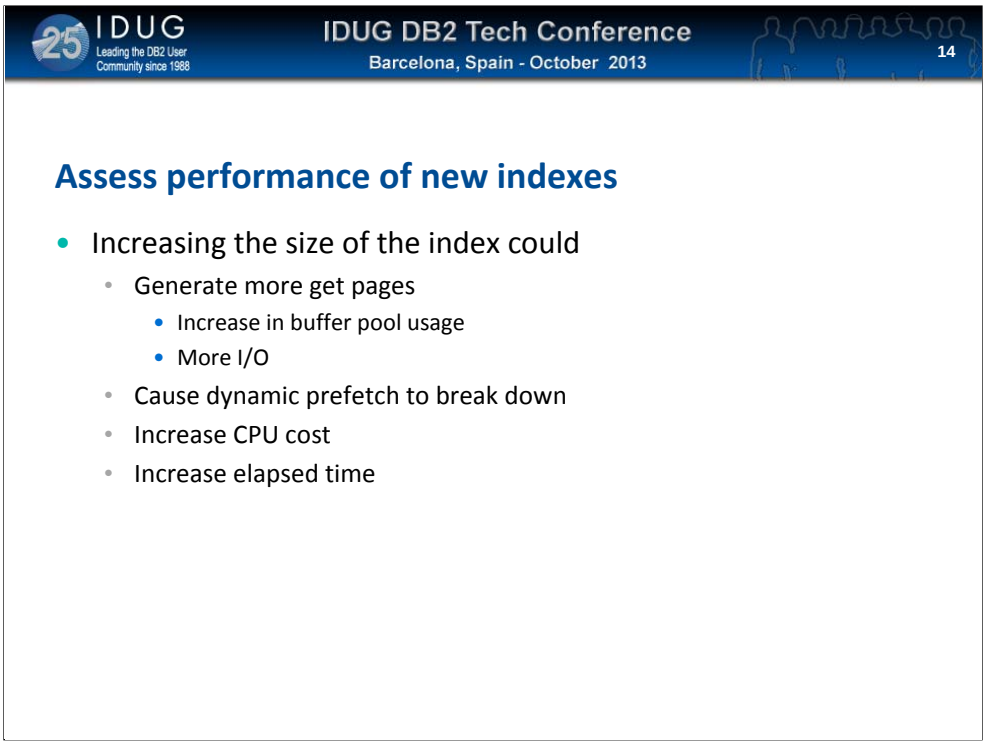
Now you will want to look at the SQL that is using the index. You can search SYSPACKDEP for your static SQL. Or you can use the statement cache table for your dynamic SQL. You want to get an idea of how often these statements are being executed.

You will also want to look at insert and delete activity to see how much you might save by dropping the extra index.



Now that you have narrowed down your list of indexes, you need to assess the performance of the SQL that utilize these indexes. You will want to look at current access paths using EXPLAIN. Notice the index access method. The optimizer may have chosen a matching index access, an index scan or some other kind of access. You will also want to note if there is any planned prefetch activity. You will see if the optimizer will try to use sequential prefetch, dynamic prefetch or list prefetch. Dynamic prefetch will read groups of sequential pages asynchronously. It will dynamically switch to synchronous I/O when it detects if data is out of order. List prefetch will use an index to find the RIDS that qualify for your predicate, order these RIDS and access the table data with prefetch, defined by the order of the RID list.

The EXPLAIN is only half of the picture. There is no guarantee that the optimizer will choose this access path during execution. To see what actually happens, you need to look at the DB2 performance statistics. You will want to note the average response time and CPU time. You will also want to look at get page activity. Be aware that a get page doesn't necessarily mean a physical I/O. Buffer pool activity will determine if a physical I/O is require. Higher get page activity could translate into more physical I/Os.



Increasing the size of your index will, in most cases, increase the amount of get pages. If the amount of get pages is significant, you could harm your buffer pool hit ratio, causing more physical I/O.

Dynamic prefetch uses skip sequential access. Having a larger index may cause this process to break down and generate more sequential I/O.

You will see a rise in your CPU cost and that all contributes to a longer response time.

EXPLAIN

- If a query is using the extraneous index, the optimizer will not show any access path difference when using the index with INCLUDE columns
- If the query is using the unique index, adding INCLUDE columns can change or increase the cost of the access path.

How to proceed

- Identify indexes that can be consolidated
- Find transactions that use the affected indexes
- Analyze EXPLAIN output
- Look at before and after statistics
  - Space allocation
  - Affects of INSERT and DELETE activity
  - Get page counts for identified queries

Data base layout - before

TABLE	INDEX	KEYS
ACT	XACT1	ACTNO
	XACT2	ACTKWD
DEPT	XDEPT1	DEPTNO
	XDEPT2	ADMRDEPT
	XDEPT3	DEPTNO.CITY.STATE
EMP	XEMP1	EMPNO
	XEMP2	WORKDEPT
	XEMP3	EMPNO.LASTNAME. FIRSTNAME.MIDINIT
EMPPROJACT	XEMPPROJACT1	PROJNO.ACTNO.EMSTDATE.EMPNO
	XEMPPROJACT2	EMPNO

```
TBNAME            CARD    NPAGES COMP  NACTIVE PARENT CHILD BPOOL DATABASE LOCK PARTS PGSIZE SEGSIZE PCTPAGES  FARINDREF N
ACT                 18         1    0       6      0     0 BP0   DMDDBA   R        0      4       4       25          0
   IXNAME          1STKEY     FULLKEY NLEAF NLEVELS CLUSTERRATIO CLUSTERING CLUSTERED   FAROFF  NEAROFF LEAFDIST UNIQUE TY
   XACT1             18          18     1       2 1.0000000000 N          Y               0        0        0 P      2
      KEY COLUMN         COLNO ORDERING
      ACTNO                 1 A
   XACT2             18          18     1       2 1.0000000000 N          Y               0        0        0 U      2
      KEY COLUMN         COLNO ORDERING
      ACTKWD                2 A
DEPT                100         3    0       6      0     2 BP0   DMDDBA   P        0      4       4       75          0
   IXNAME          1STKEY     FULLKEY NLEAF NLEVELS CLUSTERRATIO CLUSTERING CLUSTERED   FAROFF  NEAROFF LEAFDIST UNIQUE TY
   XDEPT1             100         100     1       2 1.0000000000 N          Y               0        0        0 P      2
      KEY COLUMN         COLNO ORDERING
      DEPTNO                1 A
   XDEPT2              10          10     1       2 1.0000000000 N          Y               0        0        0 D      2
      KEY COLUMN         COLNO ORDERING
      ADMRDEPT              3 A
   XDEPT3             100         100     2       2 1.0000000000 N          Y               0        0       50 U      2
      KEY COLUMN         COLNO ORDERING
      DEPTNO                1 A
      CITY                  4 A
      STATE                 5 A
EMP              500000     14706    0    14716      1     1 BP0   DMDDBA   P        0      4       4       99          0
   IXNAME          1STKEY     FULLKEY NLEAF NLEVELS CLUSTERRATIO CLUSTERING CLUSTERED   FAROFF  NEAROFF LEAFDIST UNIQUE TY
   XEMP1           500000      500000  1793       3 1.0000000000 Y          Y               0        8        0 P      2
      KEY COLUMN         COLNO ORDERING
      EMPNO                 1 A
   XEMP2              89          89   691       3 0.9998440000 N          Y              78   299231        0 D      2
      KEY COLUMN         COLNO ORDERING
      WORKDEPT              5 A
   XEMP3           500000      500000  6250       3 1.0000000000 N          Y               0        8        0 U      2
      KEY COLUMN         COLNO ORDERING
      EMPNO                 1 A
      LASTNAME              4 A
      FIRSTNME              2 A
      MIDINIT               3 A
EMPPROJACT    25916182    242208    0   242351      1     0 BP0   DMDDBA   R        0      4       4      100          0
   IXNAME          1STKEY     FULLKEY NLEAF NLEVELS CLUSTERRATIO CLUSTERING CLUSTERED   FAROFF  NEAROFF LEAFDIST UNIQUE TY
   XEMPPROJACT1       133    25916182 193.5K      4 1.0000000000 N          Y               0      141        0 U      2
      KEY COLUMN         COLNO ORDERING
      PROJNO                2 A
      ACTNO                 3 A
      EMSTDATE              5 A
      EMPNO                 1 A
   XEMPPROJACT2    500000      500000 37226       3 0.0120228743 N          N        25915988        0        0 D      2
      KEY COLUMN         COLNO ORDERING
      EMPNO                 1 A
```

Identify indexes

UNIQUE INDEX	INCLUDE COLS	COLTYPE	COLNAME
RDADMB.XDEPT1	RDADMB.XDEPT3	UNIQUE	DEPTNO
		INCLUDE	CITY
		INCLUDE	STATE
RDADMB.XEMP1	RDADMB.XEMP3	UNIQUE	EMPNO
		INCLUDE	LASTNAME
		INCLUDE	FIRSTNME
		INCLUDE	MIDINIT

I ran the SQL to tell me what indexes could be combined. The SQL will list the unique index and the index it could be combined with. It identifies the unique columns and then tells you what columns can be included. It highlighted two indexes in my data base.

Data base - after

TABLE	INDEX	KEYS
ACT	XACT1	ACTNO
	XACT2	ACTKWD
DEPT	XDEPT1	DEPTNO **INCLUDE** (CITY.STATE)
	XDEPT2	ADMRDEPT
EMP	XEMP1	EMPNO **INCLUDE** (LASTNAME.FIRSTNAME.MIDINIT)
	XEMP2	WORKDEPT
EMPPROJACT	XEMPPROJACT1	PROJNO.ACTNO.EMSTDATE.EMPNO
	XEMPPROJACT2	EMPNO

```
TBNAME            CARD    NPAGES COMP  NACTIVE PARENT CHILD BPOOL DATABASE LOCK PARTS PGSIZE SEGSIZE PCTPAGES  FARINDREF N
ACT                 18         1    0       6      0     0 BP0   DMDDBA   R        0      4       4       25          0
   IXNAME          1STKEY     FULLKEY NLEAF NLEVELS CLUSTERRATIO CLUSTERING CLUSTERED   FAROFF  NEAROFF LEAFDIST UNIQUE TY
   XACT1              18          18     1       2 1.0000000000 N          Y               0        0        0 P      2
      KEY COLUMN         COLNO ORDERING
      ACTNO                1 A
   XACT2              18          18     1       2 1.0000000000 N          Y               0        0        0 U      2
      KEY COLUMN         COLNO ORDERING
      ACTKWD               2 A
DEPT                100          3    0       6      0     2 BP0   DMDDBA   P        0      4       4       75          0
   XDEPT1             100         100     2       2 1.0000000000 N          Y               0        0       50 P      2
      KEY COLUMN         COLNO ORDERING
      DEPTNO               1 A
      CITY                 4
      STATE                5
   XDEPT2              10          10     1       2 1.0000000000 N          Y               0        0        0 D      2
      KEY COLUMN         COLNO ORDERING
      ADMRDEPT             3 A
EMP              500000     14706    0    14716      1     1 BP0   DMDDBA   P        0      4       4       99          0
   XEMP1            500000     500000  6250       3 1.0000000000 Y          Y               0        8        0 P      2
      KEY COLUMN         COLNO ORDERING
      EMPNO                1 A
      LASTNAME             4
      FIRSTNME             2
      MIDINIT              3
   XEMP2               89          89   691       3 0.9998440000 N          Y              78   299231        0 D      2
      KEY COLUMN         COLNO ORDERING
      WORKDEPT             5 A
EMPPROJACT     25916182    242208    0   242351      1     0 BP0   DMDDBA   R        0      4       4      100          0
   IXNAME          1STKEY     FULLKEY NLEAF NLEVELS CLUSTERRATIO CLUSTERING CLUSTERED   FAROFF  NEAROFF LEAFDIST UNIQUE TY
   XEMPPROJACT1       133    25916182 193.5K       4 1.0000000000 N          Y               0      141        0 U      2
      KEY COLUMN         COLNO ORDERING
      PROJNO               2 A
      ACTNO                3 A
      EMSTDATE             5 A
      EMPNO                1 A
   XEMPPROJACT1    500000      500000 37226       3 0.0120228743 N          N        25915988        0        0 D      2
      KEY COLUMN         COLNO ORDERING
      EMPNO                1 A
```

Storage Savings

- DEPT table
  - With redundant index
    - XDEPT1 + XDEPT3 = 2 cylinders
  - With INCLUDE columns
    - XDEPT1 = 1 cylinder
  - Savings of 1 cylinder
- EMP table
  - With redundant index
    - XEMP1 + XEMP3 = 51 cylinders
  - With INCLUDE columns
    - XDEPT1 = 36 cylinder
  - Savings of 15 cylinder

The first thing I did was look at the storage savings. I did a LISTCAT on the four indexes to look at the allocations. If my main concern is DASD usage, I won't see any relief by combining the indexes on the department table.

Insert/delete get page savings

- Get page count for XEMP1 and XEMP3

	XEMP1 + XEMP3	XEMP1 w/INCLUDE	% DECREASE
INSERT	31224	15605	50
DELETE	45247	14805	67

The next thing I looked at was the insert and delete activity against these indexes. I generated 7800 insert and deletes and looked at the get page activity. In this chart, I am just comparing the get pages for the two indexes. It looks like a saving of 50% for inserts and 67% for deletes. That seems like a pretty good saving.

Insert/delete get page savings (cont.)

- Total get page count

	TOTAL SQL	TOTAL w/INCLUDE	% DECREASE
INSERT	374573	344683	8
DELETE	583847	554509	5

But that's not the whole picture. When you do an insert or delete, your SQL performs get pages for the catalog, the table space and the other indexes. If I look at the big picture, my total saving is 8% and 5%. It dawned on me that this table had a primary key and a foreign key. Maybe that is why there was a lot of extra get page activity on this table space. So, I dropped the RI and tried my inserts again.

Insert/delete get page savings (cont.)

- Total get page count after removing RI

	TOTAL SQL	TOTAL w/INCLUDE	% DECREASE
INSERT	265355	235477	11
DELETE	373228	290476	22

Removing the RI did decrease the get page count and now I can see my savings might have a small impact.

Example 1 - EXPLAIN

```
SELECT   DEPTNO, CITY, STATE,
         EMPNO, FIRSTNME, MIDINIT, LASTNAME
FROM     RDADMB.DEPT, RDADMB.EMP
WHERE    DEPTNO = WORKDEPT
ORDER    BY DEPTNO, CITY, STATE
```

Without INCLUDE

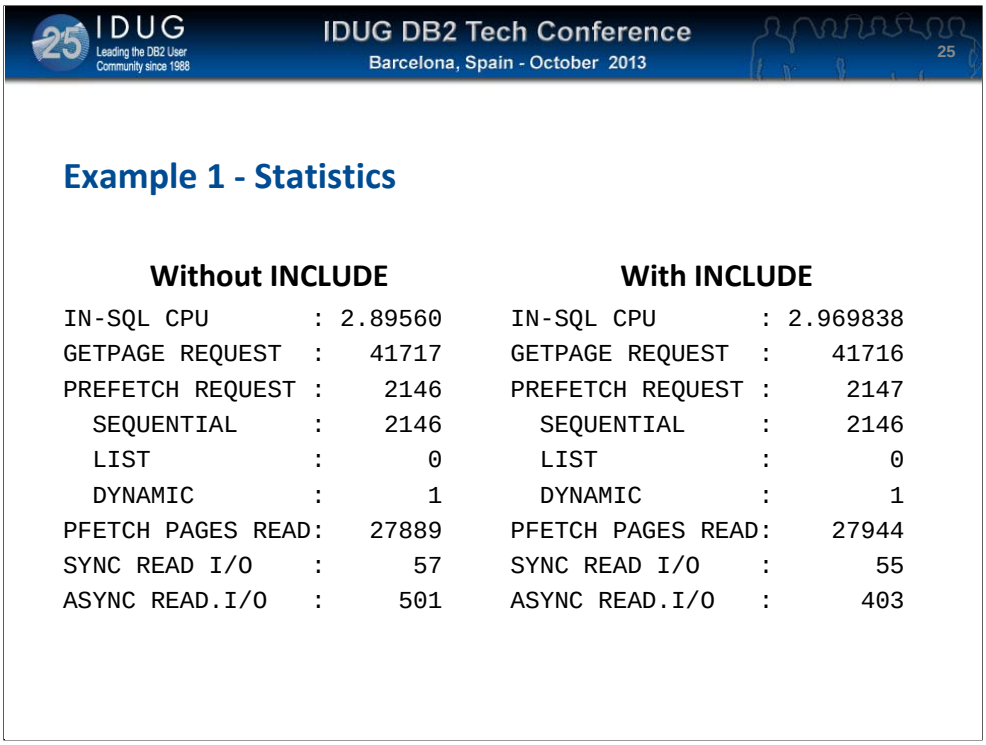
COST*RATE	METH	ACC	MTCH	IX	TBNAME	IXNAME	NU	J	O	G	CU	J	O	G	PRE
0.331200	0	I	0	Y	DEPT	IDEPT3	N	N	N	N	N	N	N	N	
41547.582	2	R	0	N	EMP		N	Y	N	N	N	N	N	N	S

With INCLUDE

COST*RATE	METH	ACC	MTCH	IX	TBNAME	IXNAME	NU	J	O	G	CU	J	O	G	PRE
0.331200	0	I	0	Y	DEPT	IDEPT1	N	N	N	N	N	N	N	N	
41547.582	2	R	0	N	EMP		N	Y	N	N	N	N	N	N	S

Next, I want to compare access paths for some of the heavier hit SQL. You can do this in a couple different ways. You can migrate your table spaces to a test subsystem and run an EXPLAIN against the new configuration . You will want to make sure you have performed RUNSTATS after the migration. Or you can use a vendor tool that allows you model what happens when you modify your indexes

In this example, you can see that this SQL uses the redundant index for index only access. When you drop this index and add CITY and STATE as include columns, the optimizer cost and access path is exactly the same. That's because these two indexes look the same to the optimizer.



Now I wanted to see if the statistic would be the same, too. As you can see, the statistics are pretty close.

Example 2 - EXPLAIN

```
SELECT   DEPTNO, DEPTNAME, EMPNO, FIRSTNME, MIDINIT,
         LASTNAME
FROM     RDADMB.DEPT, RDADMB.EMP
WHERE    DEPTNO = WORKDEPT
ORDER    BY EMPNO, LASTNAME, FIRSTNME, MIDINIT
```

Without INCLUDE

COST*RATE	METH	ACC	MTCH	IX	TBNAME	IXNAME	NU	J	O	G	CU	J	O	G	PRE
8726.0195	0	I	0	N	EMP	XEMP1	N	N	N	N	N	N	N	N	S
11502.035	1	I	1	N	DEPT	XDEPT1	N	N	N	N	N	N	N	N	

With INCLUDE

COST*RATE	METH	ACC	MTCH	IX	TBNAME	IXNAME	NU	J	O	G	CU	J	O	G	PRE
10551.601	0	I	0	N	EMP	IEMP1	N	N	N	N	N	N	N	N	S
11502.035	1	I	1	N	DEPT	IDEPT1	N	N	N	N	N	N	N	N	

The next SQL uses the unique index on the employee table. When we add the INCLUDE columns to this index, it will be much larger. The optimizer recognizes that and increases the cost for that access.

Example 2 - Statistics

Without INCLUDE			With INCLUDE		
IN-SQL CPU	:	4.977964	IN-SQL CPU	:	5.851382
GETPAGE REQUEST	:	332924	GETPAGE REQUEST	:	704728
PREFETCH REQUEST	:	524	PREFETCH REQUEST	:	664
SEQUENTIAL	:	0	SEQUENTIAL	:	0
LIST	:	0	LIST	:	0
DYNAMIC	:	524	DYNAMIC	:	664
PFETCH PAGES READ	:	16509	PFETCH PAGES READ	:	20990
SYNC READ I/O	:	20	SYNC READ I/O	:	30
ASYNC READ.I/O	:	4	ASYNC READ.I/O	:	8

As you can see, the CPU time increased by about 18% and the get page count more than doubled. You may want to give more thought as to whether you want to add INCLUDE columns. You improved the INSERT and DELETE get page access by 10 – 20%, but you degraded the performance of this SQL by 18%. At this time, I would want to compare CPU savings for my INSERT and DELETE SQL.

Summary

- Proceed cautiously
- INCLUDE columns can save resources
- Can possibly degrade performance
- Be sure to measure all the affects

Recent APARs

- PM82595
  - Partitioned index with INCLUDE columns
  - Query with parallelism
  - May cause storage overlay
  - Causes a problem with RI that is fixed by PM88442
    - SQLCODE = -531 when updating a parent key
- PM69341
  - Primary key of parent table contains INCLUDE columns
  - LOAD child table
  - REASON=X'00E20005' DSNSVBK

Donna Di Carlo
BMC Software
donna_di_carlo@sbcglobal.net

F12
To include or not include? That is the question.