Why low cardinality indexes negatively impact performance
Ember Crooks
September 26, 2013

Low cardinality indexes can be bad for performance. But, why? There are many best practices like this that DBAs hear and follow but don't always understand the reason behind. This article empowers DBAs to understand the logic behind why low cardinality indexes can be bad for performance or cause erratic performance. The topics this article covers include B-tree indexes, understanding index cardinality, hypothetical examples of the effects of low cardinality indexes, a real-world example of the effects of a low cardinality index, and tips on how to identify low cardinality indexes and reduce their impact on performance.

Low cardinality indexes
A lot of advice is available about identifying appropriate indexes to create or identifying bad indexes to drop. It may seem logical that an index on a field like account enabled, which has a very small set of unique values (yes, no), could substantially reduce a result set. In light of the geometry of B-tree indexes, an index with a low number of possible values can actually harm performance rather than help it.

What is index cardinality?
A table's cardinality is the number of rows or records in the table. For an index, cardinality is considered the number of unique values in the index. A unique index would have cardinality equal to the number of rows in the table. But a non-unique index can have a cardinality of anywhere from 1 to the number of rows in the table, depending on how many times each index key appears in the table.

Low cardinality indexes are those indexes with relatively few unique values. Columns like account enabled or published are likely to have only two unique values (yes and no). An index on such a column would have a cardinality of 2. Many tables can have a column that has a very small subset of unique values columns such as Status, Color, and Currency might be examples, depending on the data stored.

It is easy to find the cardinality of an existing index. The FULLKEYCARD column in syscat.indexes includes the cardinality of the full index: the unique combinations of the values of all index keys. A simple query to determine the FULLKEYCARD, assuming current RUNSTATS for an index called WSCOMUSR.I0000626 is in Listing 1:
Listing 1. Query to determine FULLKEYCARD for an Index

```sql
select FULLKEYCARD
from syscat.indexes
where indname='I0000626' and indschema='WSCOMUSR'
```

Often, listing all of the indexes on a table, along with their cardinalities, would be more useful. The query to list all indexes and their FULLKEYCARD for a table named WSCOMUSR.OFFER, with sample output, is in Listing 2:

Listing 2. Query to list all Indexes with their FULLKEYCARD by Table

```sql
select substr(INDNAME,1,25) as INDNAME, FULLKEYCARD
from syscat.indexes
where tabname='OFFER' and tabschema='WSCOMUSR'
```

<table>
<thead>
<tr>
<th>INDNAME</th>
<th>FULLKEYCARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL120207194412100</td>
<td>29383</td>
</tr>
<tr>
<td>I00000165</td>
<td>29383</td>
</tr>
<tr>
<td>I00000167</td>
<td>29378</td>
</tr>
<tr>
<td>I00000626</td>
<td>6</td>
</tr>
</tbody>
</table>

4 record(s) selected.

B-tree indexes

DB2 uses B-tree indexes that consist of multiple levels and are typically conceptually represented with three levels, shown in Figure 1:

**Figure 1. Three-tier B-tree index**

The root node is the top level and lists the end of a range of key values represented on each of the intermediate nodes in the middle level. There can be more than one level of intermediate nodes, with higher levels of intermediate nodes pointing to other intermediate nodes instead of leaf nodes. The intermediate nodes, in turn, point to the leaf nodes. The terms *node* and *page* are interchangeable for DB2. The leaf pages hold the key values along with Record IDs (RIDs) that tell DB2 where to find the data on the data pages of the table.

For optimal index access, DB2 would have a specific index key, and would only have to read the root page, one intermediate page, and one leaf page. DB2 could then use the RID on that leaf...
page to pull additional data from the table data page. RIDs themselves consist of a page number that is relative to either a table (SMS tablespaces) or relative to a tablespace (DMS tablespaces) and a slot number on that page. This access plan allows DB2 to access any row in a table, given an index key in a three-tier index, with just four page reads, regardless of the table size.

In Figure 1, the index is on a column that is a single character. In looking for the index key value of \( E \), DB2 would start at the root node, and be pointed to the intermediate node that contains \( D \), \( F \), and \( J \). That intermediate node would then direct DB2 to the leaf node that contains \( E \) and \( F \), which includes the RID. Using the RID, DB2 then goes straight to the data page in the table to obtain data for that record.

This representation is really of a unique index. Low cardinality indexes by definition are not unique. The representation of the index keys and RIDs on an index leaf page for a unique index is in Figure 2:

**Figure 2. How Index entries are stored on the leaf pages for unique Indexes**

<table>
<thead>
<tr>
<th>Index Key</th>
<th>RID</th>
<th>RID</th>
<th>RID</th>
<th>RID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Key</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
</tr>
<tr>
<td>Index Key</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
</tr>
</tbody>
</table>

In non-unique indexes, any given index key appears only once on any given page. Leaf pages for non-unique indexes look more like Figure 3, with all RIDs for an index key being stored in RID order:

**Figure 3. How index entries are stored on the leaf pages for non-unique indexes**

<table>
<thead>
<tr>
<th>Index Key</th>
<th>RID</th>
<th>RID</th>
<th>RID</th>
<th>RID</th>
<th>RID</th>
<th>RID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Key</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
</tr>
<tr>
<td>Index Key</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
<td>RID</td>
</tr>
</tbody>
</table>

With this information and the fact that non-leaf pages include the top RID value for each leaf page when multiple leaf pages are full of the same key, the B-tree index in Figure 1 is altered to more accurately look like Figure 4:

**Figure 4. DB2 non-unique B-tree index**
Measuring differences

When talking about a page read or page scan in this article, it is not necessarily a physical I/O. It is not known whether the page access would simply be processing a page that is in a bufferpool, or an agent going out to disk to perform a physical I/O. It is hoped that it would be the former, and that prefetching would occur to make it so. For the purposes of this article, a page read, a page scan, and the processing of a page all mean the same thing because this article does not go down to the level of predicting which it will be. Obviously, a synchronous read of a page from disk by the agent satisfying a query would be much worse for performance, and the uncertainty there is part of what makes performance results relating to low cardinality indexes potentially erratic.

Hypothetical examples of the impact of low index cardinality

Working a hypothetical example can illustrate one reason that low cardinality indexes can be a problem.

For the examples in this section, the same query and the same result set are used, but as the cardinality of the index increases, the size of the table increases as well so that the result set size stays the same.

Starting with the simple table in Table 1:

Table 1. Description of a sample table for first hypothetical example

<table>
<thead>
<tr>
<th>ID</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A unique identifier with no index</td>
<td>Two possible values</td>
<td>Some text</td>
</tr>
</tbody>
</table>

Along with Table 1, there is also a standard B-tree index on the table created using the syntax in Listing 3:

Listing 3. Syntax for creating an Index for the hypothetical examples

```
CREATE INDEX indname ON table1 (type);
```

An equal or even distribution of data across all values in this low cardinality index is assumed.

Some round numbers to describe the table:

- 10,000 rows in the table
- 20 index RIDs fit on a leaf page
- 10 rows fit on a data page
- Index cardinality (FULLKEYCARD) is 2

Accessing the table using a full table scan would take: 10,000 rows / 10 data rows per page = 1,000 page scans

Finally, a simple query with the column of the low cardinality index in the where clause is in Listing 4:

```
```
Listing 4. Sample query for hypothetical table

```sql
SELECT id, description FROM table WHERE type = 'B'
```

In Figure 5 and all of the following figures, index access is represented in red, while direct table access is represented in green:

**Figure 5. Diagram of Index access vs. Table access for first hypothetical example**

![Diagram of Index access vs. Table access for first hypothetical example](image)

Figure 5 shows accessing 1 root node, 1 intermediate node, 250 leaf pages, and then anywhere between 500 and 1,000 table data pages. This means that satisfying the example query would take anywhere from 752-1,252 page reads using the index. On the other hand, a full table scan would be guaranteed to take 1,000 page reads each time. The index access, if chosen by the DB2 optimizer, could take anywhere from 75% to 125% of the page reads used by a table scan.

Changing some of the numbers in this example produces interesting results. The next figure examines what happens with an increased cardinality of the index and increased cardinality of the table so the end result still returns the same number of rows. The distribution of values is still expected to be equal across the values in the index. The revised table and conditions are in Table 2:

**Table 2. Description of a sample table for second hypothetical example**

<table>
<thead>
<tr>
<th>ID</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A unique identifier with no index</td>
<td>Four possible values</td>
<td>Some text</td>
</tr>
</tbody>
</table>

Round numbers to describe this table:

- 20,000 rows in the table
- 20 index RIDs fit on a leaf page
- 10 rows fit on a data page
- Index cardinality (FULLKEYCARD) is 4
Accessing the table using a full table scan would take: 20,000 rows / 10 data rows per page = 2,000 page scans

The modified diagram is in Figure 6:

**Figure 6. Diagram of Index access vs Table access for second hypothetical example**

In this case, using the index requires from 752 to 2,252 page reads, while a full table scan requires 2,000 page reads. It is still possible for a full table scan to be more efficient than index access, depending on the clustering of the table over this index.

Moving one step farther and changing conditions again while keeping the result set size identical produces the table described in Table 3:

**Table 3. Description of a sample table for third hypothetical example**

<table>
<thead>
<tr>
<th>ID</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A unique identifier with no index</td>
<td>Sixteen possible values</td>
<td>Some text</td>
</tr>
</tbody>
</table>

Round numbers to describe this table:

- 80,000 rows in the table
- 20 index RIDs fit on a leaf page
- 10 rows fit on a data page
- Index cardinality (FULLKEYCARD) is 16

Accessing the table using a full table scan would take: 80,000 rows / 10 data rows per page = 8,000 page scans

The distribution of rows across the values in the low cardinality index is equal across all 16 values. The modified diagram is in Figure 7:
Figure 7. Diagram of Index access vs Table access for third hypothetical example

This is an example where the index finally is guaranteed to do fewer page reads than the table scan. In all of the previous examples, it might have done fewer, but it also might have done more.

Each previous example gives a range of possible data page reads for index access to go back to the data pages. Where in this range the actual page reads are depends on how well clustered the table is over a particular index. If the table is clustered or nearly clustered on an index, then the data pages read will be lower because data pages will have multiple rows on them with the same value for the index key. If records with the same value for a low cardinality index are spread across more pages in the table, then the data pages read will be higher. The maximum possible data pages read is either the number of rows being returned or the total number of pages in the table.

These examples were simplified to eliminate other factors and focus on the issue at hand. The next section covers a real-world table and index.

**Real-world example of the impact of low index cardinality**

This example is from a base table in an IBM WebSphere® Commerce database. The table is MBRATTRVAL, and it stores attributes related to the users of a website. It has a small row size, because the VARCHAR is rarely populated for very long. The structure of the table is described in Table 4:

**Table 4. Description of the MBRATTRVAL table**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBRATTRVAL_ID</td>
<td>BIGINT NOT NULL</td>
</tr>
<tr>
<td>STORRENT_ID</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MEMBER_ID</td>
<td>BIGINT NOT NULL</td>
</tr>
<tr>
<td>ATTRTYPE_ID</td>
<td>CHAR(16) NOT NULL</td>
</tr>
<tr>
<td>MBRATTR_ID</td>
<td>BIGINT NOT NULL</td>
</tr>
</tbody>
</table>
There is an index on the STOREENT_ID column. This column splits entries into specific stores, which can be separate e-commerce websites. In this case, there are only seven possible values with an erratic distribution, shown in Listing 5:

### Listing 5. Distribution of values in low cardinality index on MBRATTRVAL table

```
select storeent_id, count(*) as count from MBRATTRVAL group by storeent_id with ur
STOREENT_ID  COUNT
----------  ------
10152       552784
10851       4888360
10852       362532
11352        188
12002        4477
12551        338
          22
```

There are a number of details needed to fill in a diagram similar to the hypothetical examples:

- Average number of data rows per table page
- Levels in the index
- Number of index RIDs per leaf page

Trying to predict the access path the DB2 optimizer would choose would require additional data, such as whether distribution statistics are collected. Prediction is not the point here; comparing an index-based access plan with a table scan is.

One query joining three system views supplies the values needed, shown in Listing 6:

### Listing 6. Query to return values critical for analysis of a low cardinality index

```
select CARD
    , (PAGESIZE-100)/(AVGROWSIZE+10) as avg_rows_per_table_page
    , NLEVELS
    , FULLKEYCARD
    , CASE
        WHEN LEVEL2PCTFREE = -1 and AVGNLEAFKEYSIZE > 0
        THEN (PAGESIZE-100)/(AVGNLEAFKEYSIZE+7)
        ELSE -1
        END as AVG_IND_RIDS_PER_NLEAF
    , CARD/NLEAF as AVG_IND_RIDS_PER_LEAF
from syscat.indexes i, syscat.tablespaces ts, syscat.tables t
where i.TBSPACEID=ts.TBSPACEID
    and i.tabschema=t.tabschema
    and i.tabname=t.tabname
    and indname='I0000327'
    and indschema='WSCOMUSR'
```
Average overhead per data page here is estimated at 100 bytes and overhead per data row is estimated at 10 bytes. Row overhead per index RID is estimated at 7 bytes. While these numbers do come from the 9.7 IBM DB2 Information Center, they are estimates and not exact values, so it is likely that values calculated using them will be in the right ballpark, but not exactly right on.

If the objects had a different PCTFREE than the default, this query would have to be altered to take that into account.

Other overhead could also increase the total space taken up by a table or an index.

In this case, 81 rows can fit on a data page in this table, and 1156 RIDs on an index leaf page on average.

First, look at how the diagram would look if the data were normally distributed across the seven values. Figure 8 shows what it looks like to plug the data into a similar diagram as used in the hypothetical examples. Because of this equal distribution, this example is still on the hypothetical side. Future examples will be more real-world.

**Figure 8. Diagram for an even data value distribution of full Index keys in a real-world example**

![Diagram](image)

It is assumed that a static value is provided in the query for this example. The query would look like Listing 7:

**Listing 7. Query used in even-distribution example**

```
select MBRATTRVAL_ID, MEMBER_ID, ATTRTYPE_ID, MBRATTR_ID
from MBRATTRVAL
where STOREENT_ID=12345 with ur
```
Much like the purely hypothetical examples, the index may reduce the number of pages read, or it might increase them. With a normal distribution, the index would read between 10,781 and 71,300 data pages to get 817,051 records. The table scan to retrieve the same data without using the index would use 70,610 pages.

Now, look at the same diagram with a frequently occurring value and a non-frequently occurring value.

First, the less frequent one: 12002, which occurs 4,477 times. The query used is in Listing 8:

**Listing 8. Query used in low-frequency example**

```sql
select MBRATTRVAL_ID, MEMBER_ID, ATTRTYPE_ID, MBRATTR_ID
from MBRATTRVAL
where STOREENT_ID=12002 with ur
```

The diagram is shown in Figure 9:

**Figure 9. Diagram of low-frequency value in a real-world example**

If the value specified is lower in frequency, DB2 will read significantly fewer pages using the index (between 62 and 4,483) than it would in a table scan (70,610). In this case, the low cardinality index is helpful in reducing pages read.

Finally, the most frequent value: 10851, which occurs 4,888,360 times. The query used is in Listing 9:

**Listing 9. Query used in low-frequency example**

```sql
select MBRATTRVAL_ID, MEMBER_ID, ATTRTYPE_ID, MBRATTR_ID
from MBRATTRVAL
where STOREENT_ID=10851 with ur
```

The diagram is shown in Figure 10:
In the case of the final diagram, at best, the index is going to save 10,259 page reads (70,610 table scan pages minus 64,587 best-case total index access pages), while at worst, it will cost an additional 4,236 page reads (70,610 table scan pages minus 74,846 worst-case total index access pages).

These diagrams show three important things:

- Significantly more page reads might be added by index access, especially using a frequently occurring value.
- Knowing the how well data is clustered over a particular index is critical to know where actual index page read numbers would be in the range.
- Current runstats that include distribution statistics are critical so DB2 isn't expecting a normal distribution and then getting something else.

Low cardinality indexes provide erratic performance depending on frequency of the value and index clustering within the table. DBAs don't always know whether queries will be using the frequent or the less frequent values when creating the indexes.

**How low cardinality indexes affect inserts, updates, and deletes**

For non-unique indexes, each index key appears only once on each leaf page. This can save considerable storage space, and it shows up in the previous calculations in the difference between the number of entries stored on each leaf page (1156 in the real-world example) and the number of entries stored on non-leaf pages (735 in the real-world example). This also helps to reduce the space taken up by the index.

While compression is beyond the scope of this article, if indexes are compressed, this space and therefore page reads can be even further reduced because DB2 does not have to store the entire RID, but only the difference between RIDs.

In previous versions, with type 1 indexes, DB2 had to scan through the entire RID list if an update to the index key, insert, or delete was done for just one row. With type 2 indexes, the RIDs are
stored in the index in order for each index key. DB2 also stores the maximum RID associated with each lower level page on the non-leaf pages of the index. In this context, think of the RID like just another column in the index key. Once the first part of the key is matched, DB2 can navigate the index for the RID. This means that for each row updated, DB2 will need to access the index root page, an index intermediate page, and an index leaf page. DB2 then performs a binary search on the leaf page to find the particular RID it is looking for. This would add three page reads for every insert, update, and delete for each three-level index, making the treatment of non-unique indexes much like the treatment of unique indexes in this area. This is why any index that may help select performance will also likely hurt insert, delete, and sometimes update performance.

Updates to the index key value would first have to locate the RID of the row being updated, delete it, and then insert that RID into the proper place in the index. If PCTFREE and subsequent activity have allowed free space on the located page, then it will simply be added. If not, then a new leaf page would have to be added into the index structure, and the non-leaf pages updated accordingly. Default PAGE SPLIT behavior would likely include moving approximately half of the index rows from one page to the new leaf page. This update processing would be more of a concern for frequently updated values. Columns like “published” or “store_id” may not change often, but this might be more of a concern for things designed to change (for example, “order status”).

Inserts can require searching through a list of RIDs to find the right order within the RID list, but again, that search is within the list on a leaf page after DB2 has traversed the B-tree index structure. They could also lead to an additional leaf page on occasion, in the same manner as updates.

**Avoiding problems with low cardinality indexes**

This article has covered several things that define whether low cardinality indexes can help or hurt performance of a particular statement.

**Large row size when compared to the row size of the table**

When the number of index keys that fits on an index page is close to the number of data rows that fit on a data page, index access is likely to introduce additional page reads beyond what a full table scan would do. To compare the table row size and the index row size, use a query like the one in Listing 10 to look at all indexes for a particular table:

**Listing 10. Query to compare the Index key size to the Table row size**

```sql
select
    substr(t.TABNAME,1,12) as tabname,
    substr(i.INDNAME,1,20) as indnameA,
    AVGROWSIZE as RSIZE,
    AVGLEAFKEYSIZE as KSIZE,
    100*(DECIMAL((FLOAT(AVGLEAFKEYSIZE)/FLOAT(AVGROWSIZE)),7,5)) as PCT_OF_ROWSIZE
from syscat.indexes i, syscat.tables t
where i.tabschema=t.tabschema and i.tabname=t.tabname
    and type='T'
    and AVGROWSIZE>0
    and t.tabname='SRCHTERM'
```
This data can be useful when thinking about index row size as compared to the table row size. Many low cardinality indexes also have index keys that are very small, so combining this with computing the index cardinality might be an easy way to filter problem low cardinality indexes down to a manageable number in an existing database.

**Uneven distribution of data and frequently occurring values**

To analyze indexes based on this data, it is important to know what values applications are using when writing queries that might use a low cardinality index. It is easy enough to write a query to return values and their frequency, but it will depend on the table structure. Listing 11 shows an example of this:

**Listing 11. An example of looking at data distribution across Index keys**

```sql
select storeent_id, count(*) as count from MBRATTRVAL group by storeent_id with ur
```

<table>
<thead>
<tr>
<th>STOREENT_ID</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10152</td>
<td>552784</td>
</tr>
<tr>
<td>10851</td>
<td>4888360</td>
</tr>
<tr>
<td>10852</td>
<td>362532</td>
</tr>
<tr>
<td>11352</td>
<td>188</td>
</tr>
<tr>
<td>12002</td>
<td>4477</td>
</tr>
<tr>
<td>12551</td>
<td>338</td>
</tr>
<tr>
<td>-</td>
<td>22</td>
</tr>
</tbody>
</table>

4 record(s) selected.

In this case, the possible values are of wildly skewed distribution; this means that the index is likely to provide erratic performance, and probably about as likely to slow down queries of the most frequently occurring values as it is to speed them up. If distribution statistics are gathered and used by queries, then it might help queries with less frequently occurring values.

**Data in the table is not well clustered over the low cardinality index**

In the examples worked through in this article, low cardinality index performance was more likely to be better than a table scan if the table was well clustered over the index. This may actually change your strategy for choosing clustering indexes: if there is a low cardinality column that is frequently queried with performance critical queries, it may be worth either turning it into a clustering index or reorganizing the table on that index. There are other considerations for clustering indexes and clustering reorgs that are outside the scope of this article.
If detailed statistics have been collected, then the column CLUSTERFACTOR in SYSCAT.INDEXES holds this information. If there are no detailed statistics, then it is CLUSTERRATIO.

Listing 12 shows a query to find that information for a specific table:

**Listing 12. Query to find clustering information**

```sql
select substr(INDNAME,1,18) as indname,
decimal((100*CLUSTERFACTOR),5,2) as CLUSTERFACTOR
from SYSCAT.INDEXES
where tabname='ORDERS' and tabschema='WSCOMUSR'
with ur
```

<table>
<thead>
<tr>
<th>INDNAME</th>
<th>CLUSTERFACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL120212212527630</td>
<td>80.75</td>
</tr>
<tr>
<td>I0000176</td>
<td>77.00</td>
</tr>
<tr>
<td>I173124</td>
<td>90.74</td>
</tr>
<tr>
<td>I0000652</td>
<td>88.68</td>
</tr>
<tr>
<td>I0000653</td>
<td>100.00</td>
</tr>
<tr>
<td>I0000654</td>
<td>94.98</td>
</tr>
<tr>
<td>I0000892</td>
<td>100.00</td>
</tr>
<tr>
<td>I0000933</td>
<td>100.00</td>
</tr>
<tr>
<td>I0001267</td>
<td>100.00</td>
</tr>
</tbody>
</table>

9 record(s) selected.

The closer this number is to 100, the better the clustering of the table over this index.

**Index with a drastically low cardinality**

The lower the cardinality of the index compared to the cardinality of the table, the less likely the index is to be useful. Due to all of the other factors here, there is no magic number that defines an index with a cardinality that is very low. I would take a very hard look at any index where the FULLKEYCARD is less than 5% of the table cardinality. To calculate this for the indexes on the table, use a query such as the one in Listing 13:

**Listing 13. Query to examine index FULLKEYCARD in relation to table CARD**

```sql
select substr(INDNAME,1,18) as indname,
CARD,
FULLKEYCARD,
decimal((100*(float(FULLKEYCARD)/float(CARD))),5,2) as card_pct
from SYSCAT.INDEXES i, SYSCAT.TABLES t
where t.tabname=i.tabname and t.tabschema=i.tabschema
and t.tabname='ORDERS'
and t.tabschema='WSCOMUSR'
with ur
```

<table>
<thead>
<tr>
<th>INDNAME</th>
<th>CARD</th>
<th>FULLKEYCARD</th>
<th>CARD_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL120212212527630</td>
<td>929728</td>
<td>929728</td>
<td>100.00</td>
</tr>
<tr>
<td>I0000176</td>
<td>929728</td>
<td>904244</td>
<td>97.25</td>
</tr>
<tr>
<td>I173124</td>
<td>929728</td>
<td>150612</td>
<td>16.19</td>
</tr>
<tr>
<td>I0000652</td>
<td>929728</td>
<td>160047</td>
<td>17.21</td>
</tr>
<tr>
<td>I0000653</td>
<td>929728</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>I0000654</td>
<td>929728</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>I0000892</td>
<td>929728</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>I0000933</td>
<td>929728</td>
<td>1</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Distribution statistics are not collected or parameter markers do not allow DB2 to make use of distribution statistics

Doesn't the DB2 optimizer take care of this and ensure it is not choosing an access plan that requires more data pages? Well, yes, it frequently does. However, the optimizer is only as good as the data and statistics available to it. The optimizer is also capable of mistakes.

To properly find an access plan that truly fits the data, collect distribution statistics and detailed statistics on all indexes. Also, the use of parameter markers (which show up in queries in the package cache as ?), disallows DB2 from making proper use of distribution statistics. DBAs can control what kind of statistics are collected, but they may not be able to control the use of parameter markers. There are advantages that may lead developers or vendors to choose parameter markers.

The real problem comes in when DB2 has to make an assumption that the distribution is standard, and then comes up with skewed results it can cause sort overflows and other effects that can both erratically and severely impact the performance of only certain instances of queries.

Adding columns to inflate index cardinality

While the best indexes are usually well thought-out composite indexes, sometimes people randomly add a column to a low cardinality index, just to make the cardinality of an index appear better. This can help with problems with inserts, updates, and deletes, but at the expense of the selects. If the column is added as the leading column in the index, the index would not then be able to serve queries that do not also specify the added column. The index takes significantly more space and therefore requires DB2 to read even more pages.

Adding a column that is also needed by the query to either encourage index-only access or help satisfy other conditions of the query is perfectly valid and might be a good choice.

Summary

Low cardinality indexes are bad for performance in two major ways. First, they may cause erratic performance if the DB2 optimizer chooses to use them in inappropriate situations. This is more likely to happen if distribution statistics are not used or are not available. Second, low cardinality indexes cause the same performance impacts to inserts, updates, and deletes that other indexes do. If the optimizer does not use them, they can be a waste of disk space and of system resources for upkeep. While the insert, update, and delete behavior for non-unique indexes is an improvement from the behavior of type 1 indexes, it still does not make sense to negatively impact the performance of inserts, updates, and deletes for an index that will rarely be used in a positive way.

The methods in this article help identify the worst potential problem indexes and eliminate them or avoid adding them in the first place. If low cardinality indexes do exist in a database, it is also
critical to ensure that distribution statistics are gathered and used to give the DB2 optimizer the information it needs to use these indexes most efficiently.
Related topics

- Check the [DB2 9.7 Information Center Page](http://ibm.com/developerWorks) with details on estimating index sizes.
- Learn more from the [DB2 9.7 Information Center for Linux, UNIX and Windows](http://ibm.com/developerWorks).

© Copyright IBM Corporation 2013
[Trademarks](http://ibm.com/developerworks/ibm/trademarks/)