Should You Drop Indexes on Exadata?

Arup Nanda, Long Time Oracle DBA and DMA

Abstract

You may have heard from many "reliable" sources that Exadata does not need to have database indexes and therefore you should drop all the indexes to save space and improve DML processing. Should you heed this advice? Not so fast. That is not an accurate statement. As with many cases, you should understand when and how databases indexes are useful even in the case of a database on Exadata.

Target Audience

Intermediate level and up Oracle DBAs and entry level Oracle Database Machine Administrators (DMAs).

Executive Summary

It's often recommended, particularly by Oracle, that all database indexes on Exadata be dropped, ostensibly since Exadata has a unique feature called Storage Indexes which are created and maintained automatically by Exadata Storage Server software running on the Cells. However, before you drop all the indexes, you should consider all the ramifications of that action. In this paper you will learn what Storage Indexes are, when they are used and, perhaps most important, when they are *not* used. It's during the latter cases the database indexes will prove useful and dropping them will lead to poor performance.

Background

For the purpose of the paper, "index" means database index. Storage indexes, which are truly not indexes, are mentioned as such. Dropping indexes in Exadata is considered to be a "best practice". However, for something to be a best practice, you need to know these three things about it and know it well.

1. Why it is better than the rest?
2. What happens if it is not followed?
3. When are they not applicable?

Unless you answer all these three completely and accurately, you should not consider something a best practice. Dropping indexes on Exadata is no exception. So, let's start with understanding what a storage index is and how it is useful in replacing database indexes.

Storage Index

To understand the value of Storage Indexes, you need to understand the limitations of a typical Oracle database. When a user session issues a SQL like this:

```
select name
from customers
where status ='ANGRY';
```

Oracle does not just go to the database files and get the NAME column from it. The least addressable unit in a database is not a column, or a row; it's a database block, which is typically 8 KB but could be of different sizes as per the blocksize of the tablespace. The server process of the session grabs the block containing the rows matching the query predicate and moves them to the SGA (or the PGA, as appropriate). From there it extracts the actual values and returns to the user.

Now assume there is no index on the STATUS column. In that case the server process will not immediately know which blocks to look for. Instead, it will move *all* the blocks of the table CUSTOMERS to the SGA or PGA and perform the scanning and filtering there. If the table is huge (and most tables in big databases are), this will cause a lot of disk reads, not to mention consuming tons of memory. Imagine this process occurring repeatedly and you can easily see why it will kill performance.

If you analyze the situation carefully, you will immediately realize that the problem lies in the storage. A typical storage system can perform activities like reading or writing blocks; but it has no idea what is *inside* those blocks. Applications—in this case, Oracle—use these blocks are containers to store data and the storage system is oblivious to the contents. This is why the storage system has to send all the blocks to the application's cache to perform the scanning. It's quite possible that there is just one customer who is angry; but Oracle server would have to ask for all 1 billion buffers of the table CUSTOMERS and the storage system has to retrieve them from the disk and send it. The problem of large I/O would have been easily solved had the storage system been able to filter the rows right there. If that were to happen, the storage system would have had to send only one block; not 1 billion—saving 99.999% of the I/O! Of course this is one simple example; but you can see the value in making the storage intelligent.

But to do the filtering at the storage level, you would need to build some type of a database there. That, while making it performant for reads, will make it highly impractical for writes. It also makes little sense to build a database on storage when a database—Oracle—is already running on the server. Exadata accomplishes that in a different way. The storage in Exadata is not a SAN; it's internal to Exadata. The disks are attached to a blade server known as a Storage Cell, or just a "Cell". There are 14 such Cells in a full rack of Exadata. The Cell runs a software called Exadata Storage Server (ESS). The ESS can understand different commands coming from the database server; not just I/O. For instance, the database can send a command to fetch a column of a table; not just a block. It can also send the predicate (e.g. STATUS='ANGRY'). This communication occurs over a protocol known as iDB.

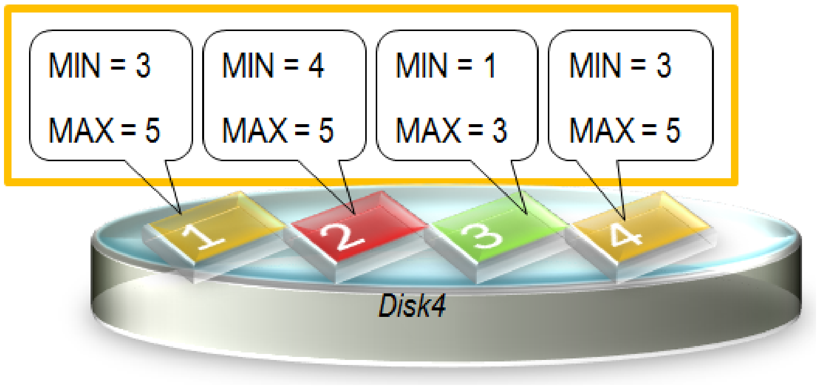

Figure 1 Storage Index on Disk

Once the ESS server gets the command, it looks into the disks to find out the blocks that can potentially contain the data. Fig 1 shows a disk connected to the Cell. The disks are divided into chunks of 1 MB each. The chunk may contain various columns. For a specific column, the ESS software stores the minimum and the maximum values. In Fig 1, you can see that in the first chunk, the minimum and maximum values of the column are 3 and 5 respectively. Similarly all the other chunks have their minimum and maximum recorded. When a query comes like this:

```
select … from thisTable where col1 = 2;
```

the ESS software checks this min/max value and can immediately determine that the first chuck will definitely *not* contain this value. Therefore it eliminates even reading data from it. Similarly it excludes chunks 2 and 4. In chunk 3, however, the minimum and maximum values are 1 and 3 respectively; so the value 2 *may* be present; so the ESS software reads that chunk. As you can see, instead of reading all the chunks, it read only one and as a result, eliminated 75% of the I/O. This is a great saving. This matrix of min/max values for the chunks is stored in the memory of the Cell and is known as a **Storage Index** for that column. In addition to min/max, Storage Indexes also contain a flag to indicate whether any value of the column in that chunk has a NULL value. This helps in queries with predicates such as "where col1 is null", which will not use a database index even if one is present.

Here are some facts about Storage Indexes:

- They are not built by the user or the DMA. Oracle automatically builds and maintains them.
- They do not point to the database blocks. Instead they point to "chunks" of space on the disks.
- They merely store for a Storage "Unit"
    - Max/Min Values
    - Whether nulls are present

  So, in many ways, this is akin to a column histogram created while gathering statistics; not really an index.
- They are for only some columns; not all columns in a table. Oracle automatically decides which columns to build it on
- They are on the memory of Cells; not disk. Therefore they disappear when the cell is shut down.

If you want to find out how much savings are as a result of the presence of Storage Indexes, you can use the following query:

```
select name, value/1024/1024 as stat_value
from v$mystat s, v$statname n
where s.statistic# = n.statistic#
and n.name in (
  'cell physical IO bytes saved by storage index',
  'cell physical IO interconnect bytes returned by smart scan')

NAME                                                            STAT_VALUE
--------------------------------------------------------------- ----------
cell physical IO bytes saved by storage index                      5120.45
cell physical IO interconnect bytes returned by smart scan         1034.00
```

From the output I know that the Storage Indexes saved about 5 GB of I/O in my own session. Had I not have the Storage Indexes (as would the case in a regular non-Exadata Oracle database), the session would have performed an additional 5 GB of physical reads. The number quickly adds up for many sessions. As I/O reduces, performance improves significantly.

Cell Processing

To understand how Storage Cells and other enhancements work in Exadata to reduce the dependence on database indexes, we need to understand two other concepts:

Offloading – it's the act of the database to offload some of its activities to be performed on storage cells

Smart Scan – it's the activity on database server that results in reduction in I/O

These two are related; but independent of one another. In other words, one could happen without the other.

Let's examine these in detail. **Smart Scan** means less I/O, which means:

- Faster disk access time
- Less data from storage to DB, which means less latency and better throughput
- Less number of buffers, which means the buffers will stay longer
- Less CPU consumption to manage these buffers
- Less data between database nodes in case of RAC

However, Smart Scan is not automatic. Here are the pre-requisite for Smart Scan:

- Direct Path operations as opposed to buffered operations
- Full Table or Full Index Scan operations are direct path. Other operations are not Smart Scan enabled.
- There has to be at least one predicate (WHERE condition). If there is no predicate, Smart Scan doesn't occur.
- Simple Comparison Operators, e.g. =, >, etc.

There are some other reasons for Smart Scan not occurring:

- Cell server is not offload capable. It happens when the diskgroup attribute cell.smart_scan_capable is set to FALSE;
- Does not occur on clustered tables, Index Organized Tabless, etc.

Process Offloading feature moves some of the activities which occur on a database node in a typical Oracle database not on Exadata to the Cell servers. Since Cells have the storage, this process offloading makes a huge difference to performance. Here are the various offloaded processes:

1. Column Projection

   Suppose a user issues this query

   ```
   select cust_id, sale_amt from sales
   ```

   In a typical Oracle database, the server process gets the block and then extracts the columns cust_id, sales_amot from the buffer. In process offloading, these columns can be extracted from the storage directly. This saves a lot of traffic between disk and the database host.

2. Predicate Filtering

   Suppose the user issues the following SQL:

   ```
   select … from customers where status = 'ANGRY'
   ```

   Instead of returning all the blocks, it can return only the relevant blocks where the predicate will be true.

3. Function Offloading

   When users issue this SQL:

   ```
   select min(sale_amt) from sales;
   ```

   Traditional database moves the blocks to the host memory and then performs a sort operation (typically in sort area, spilling over to the temporary tablespace in case sort area is full). Process Offloading can perform the sort operation right on the Cell.

4. Virtual Columns

   These are columns not stored in the table; but rather their values are computed at the runtime. This computation is performed at the database instance. However, in process offloading, because the storage is smart and identify values, this can occur at the Cells.

From the two features, you can see why Exadata is highly performant. The Smart Scan and Proces Offloading features make the use of Storage Index possible. Additionally some features are possible only because of the Storage Index.

Where Indexes Help

With this foundation place, let's now take a deep dive into Storage Indexes and where they fail to deliver. In those situations a database index would have been helpful.

DATA DISTRIBUTION

Since they work by storing min and max values of columns, they are beneficial only when the min and max values are somehow restrictive. Consider two example SIs shown in Fig 2. The lower one has chunks with varying degrees of min and max values. However in the upper one, the min and max are 1 and 5 respectively in all chunks. If you issue a query like this:

```
select … from table where col1 = 2;
```

The lower storage index will help eliminate some I/O. The upper one will match all chunks and hence will not perform any I/O reduction. The data distribution is crucial to the effectiveness of the SIs; not only their presence.

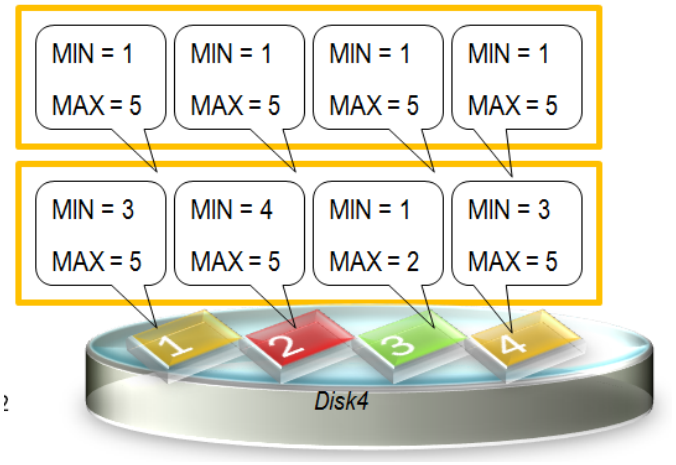

Figure 2 Restrictive Storage Indexes

8 COLUMN LIMIT

Remember one very important aspect of SIs. When you define a database index, it's on a column and all the rows in all the blocks of that segment are represented in that index. However, in SI's, each chunk has its own SI. So it's possible that column c1 has an SI on chunk 1 but that column does not have an SI in chunk 2. Instead column c2 may have an SI on chunk 2, and so on. The SIs are built based on usage; so if chunk 1 didn't get enough requests for c2, an SI will not be built in that chunk. Oracle creates SIs on a chunk on up to 8 columns. And those columns could be different on different chunks. Fig 3 shows an example. Here the first chunk has an SI on column C1, which is not on SIs on chunk 2 and 3. Similarly C9 and C10 are on SIs of chunks 2 and 3 but not on chunk 1. This means when a query comes with C1 as the predicate the I/O savings will happen only on chunk 1 and not on chunks 2 and 3.

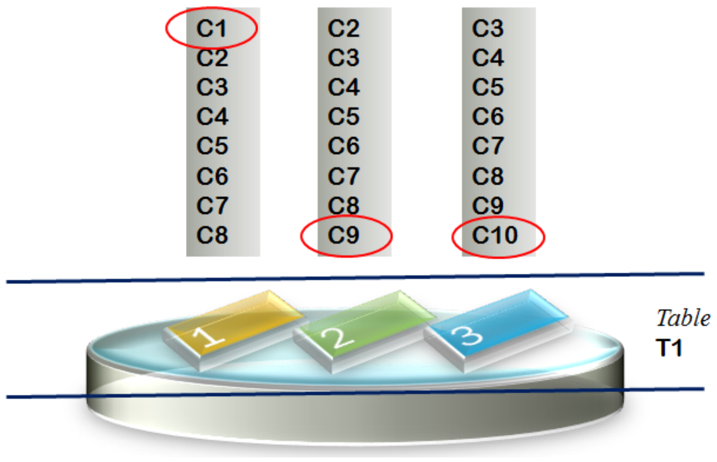

Figure 3 Multiple Columns SI

No Predicate

Consider the following query:
select sum(sale_amt) from sales;

There is no predicate in this query and no predicates means a full table scan, generally. However, if there is an index on SALE_AMT, that index data will be used by Oracle to compute the sum. If you didn't have that index, and there is a storage index on the column on all the chunks on the disks, will the storage index be used? Unfortunately, no. Since there is no predicate, storage index can't be used. Besides, remember, storage indexes are just histograms; not actual values. So operations like aggregation will not take advantage of them.

Consider another operation: select ... from sales order by sale_amt. This SQL is a sort operation; not aggregation. Again, the database index on SALE_AMT could have been used. A storage index will do nothing for this query.

Function Based Indexes

Consider a query like this

```
select … from sales where to_char(sale_dt,'YY') = '13'
```

A traditional index on sale_dt can't be used. You can create a function based index on to_char(sale_dt,'YY') which will immensely help the query. But the presence of storage indexes will not help at all.

SI Exclusions

Two types of tables—Index Organized Tables and Clustered Tables—are not taken up for creation of Storage Indexes, at least not as of writing of this paper. These structures help immensely in improving performance of the queries. You can also define secondary indexes on the IOTs and clustered tables. Dropping these indexes will hurt performance.

Query Structures

Certain types of query can't take advantage of SIs even if they are present. Consider the following:

```
select sale_amt from sales where status != 'SHIPPED'
```

This could have used an index on the STATUS columns, if present. A storage index is not used for non-equality operations; so this query will not use it. A database index would have been helpful.
Similarly, the storage indexes do not get used for wild card match operations.

```
select sale_amt from sales where city like 'NEW YORK%'
```

If we had a database index, this operation may have used it and got benefit in performance. A storage index will not be used.
As you can see from the previous discussion, dropping indexes would bring in bad performance. Storage indexes do not help, even if present. Hopefully you got enough food for thought on the idea of dropping database indexes on Exadata. In summary, full table scans are not as bad in Exadata as in a non-Exadata database—in many cases—but in some cases they are and you should not drop indexes without considering all the ramifications.

Index Tips on Exadata

Now that you know that database indexes help immensely in Exadata in many cases, let's see some tips and tricks for using them properly. The first is using flash cache. **Flash Cache** is a storage area of flash drives located on the storage cells. They are similar in concept to SAN caches. When the database node asks for blocks of I/O, the blocks are moved from disk to the flash cache and then to the database node. When the same blocks are requested again, the flash cache fulfils the request without going to the disk—saving significant disk I/O. However, this is where its similarity ends with SAN Cache. Unlike SAN Cache, the flash cache in Exadata is visible to the Oracle database. Therefore, the contents of flash cache are not just blocks of disk but blocks of specific segments. The DBA can even choose to pin a specific segment in the flash cache. Suppose a table called PERSON is requested quite often in the database. But the table is big and the sessions usually do a full table scan. Therefore either blocks never make it to the buffer cache, or get aged out quickly when they do make it to there. This is where the DBA can help by pinning this *table*—not blocks or areas of the disk—to the flash cache.

```
alter table person storage (cell_flash_cache keep)
```

This is something not possible in SAN caches which do not know what a table PERSON is and where on the disk it is located. You can use this technique to pin often used objects including indexes in the flash cache. Here is an example of pinning an index IN_T2 in the flash cache:

```
alter index in_t2 storage (cell_flash_cache keep);
```

The actual table T2 may not be there on flash, or perhaps only some popular blocks are there. The fact that the index is on the cache helps the physical I/O dramatically. To find out what objects are available on the flash cache, you should run this command in the CellCLI (the command line interpreter tool for managing Cells):

```
CellCLI> list flashcachecontent attributes -
>        cachedKeepSize, cachedSize, hitCount, -
>         hoursToExpiration, missCount -
>         where objectnumber = 382380;
```

You can expire objects from flash cache if you think they do not add any value and are not popular. You can also pin partitions, not the entire table, in the flash cache. This is useful when you have a time-defined range partitioned table where more recent partitions are accessed more frequently than the rest. When the Cell server senses increased access to a specific set of blocks, it may decide to move them to flash cache anyway; but you can expedite that, or pin it permanently if you know those specific segments.

DROP THE INDEXES?

Finally, there will always be cases where a full table scan will get all the help from Smart Scan and Process Offloading, making the access better or at least as good as indexed scans. In that case, the indexes are just deadwood eating valuable space. They also impact DML statements because they need maintenance. So, should you just drop them?

The big question is—what will be the impact? Fortunately Oracle 11g has a trick that sort of allows you to try before you buy for dropping indexes. Instead of dropping the indexes, you can make them invisible.

```
SQL> alter index i1 invisible;
```

This statement makes the database maintain the index during DMLs; but optimizer ignores it in its access paths. That allows you gauge the impact had you actually dropped the index. Examine the performance impact of the index being gone. You can gauge the presence of the index in a specific session only by issuing the following:

```
SQL> alter session set optimizer_use_invisible_indexes = true;
```

Now gauge the impact in this session. Only this session can use the index; other can't. Perhaps you can tune the query to be better, or faster. If, despite all the effort, you see that the indexes actually were better, just make them visible. Now all the sessions will see the index. Since you didn't drop the index, you don't need to create it.

Going along the same thought, you may be curious to see if the Smart Scan and Offloading are adding any value at all. You can test it by selectively disabling these features using database parameters.

- To disable offloading

```
alter system set cell_offload_processing = false;
```

- To disable storage indexes alone

```
alter system set "_kcfis_storageidx_disabled" = true;
```

You can set these parameters in a session too to limit the impact to a smaller scope. You can then examine in even more fine grained detail if your indexes are really that redundant in an Exadata system. In those cases where index actually hurts performance, you can just force full table scan through a FULL hint in the SQL or the optimizer_mode hint in the session. There is no reason to drop the index.

CONCLUSION

- Full table scans in Exadata
    - may be faster compared to non-Exadata
    - may not be faster than index scans in Exadata
    - may benefit from Storage Indexes

  The operative word here is "may". They may or may not be faster, effective, etc. You must understand how they work and exploit their features.
- Storage Indexes are not same as DB Indexes. Instead of storing a pointer to the block where the data is stored, the storage indexes store the min/max/null values for chunks of disk—akin to histograms. They simply tell which chunks not to look for; they do not tell which chunks to look for. So, they are sort of an *anti*-index.
- Dropping DB indexes helps in some cases, but not all. There are many situations where a DB index would have been helpful.
- You can test by making DB indexes invisible and checking for performance. If the indexes were helpful, simply make them visible once again.
- You can force full table scans in those cases where index hurts, instead of dropping them.