Chapter 4

Index Organized Tables

When compared to the standard B-tree/table combination, Index Organized Tables (IOTs) can dramatically reduce the number of logical and physical reads for certain application work loads. IOTs are structured such that both key and non-key columns can be collocated in the index. Take, for example, the standard employee table:

```
CREATE TABLE emp (empno NUMBER, name VARCHAR2(100), salary NUMBER, photo VARCHAR2(2000));
```

Queries such as

```
SELECT name FROM emp WHERE empno = n;
or
SELECT salary FROM emp WHERE empno = n;
```

benefit by storing the name and salary fields directly in the index. This is particularly true when the cardinality of the table is large and/or the size of the less frequently queried columns (PHOTO) is large. At first glance, it may seem that reducing the number of physical reads for these queries from, say, 2 to 1, might not be much of a performance gain. However, if these types of queries are a significant part of your application load, you may have reduced the required number of disk drives to handle the throughput from 50 to 25; a potentially significant cost savings.

There can be disadvantages to this type of organization: The plusses and minus of IOTs are the topic of this article.

Our EMP example

First let us layout a somewhat familiar couple of example tables – a normally organized table, “EMP”, and an index organized table, “EMPIOT”. In both cases, the EMPNO field will be a unique primary key and we will store 1000 rows in each.

```
SQL> CREATE TABLE emp (empno    varchar2(5)
  2                  , name     varchar2(1000)
  3                  , salary   number
  4                  , photo    varchar2(2000)
  5                  , CONSTRAINT    empno_pk PRIMARY KEY (empno)
  6  )
  7      STORAGE (INITIAL 2000K)
  8  ;
Table created.
SQL> CREATE TABLE empiot (empno    varchar2(5)
  2                     , name     varchar2(1000)
  3                     , salary   number
  4                     , photo    varchar2(2000)
  5                     , CONSTRAINT    empnoiot_pk PRIMARY KEY (empno)
  6  )
  7    ORGANIZATION INDEX
  8      INCLUDING   salary
  9      PCTTHRESHOLD 50
 10      STORAGE (INITIAL 2000K)
 11    OVERFLOW
 12      STORAGE (INITIAL 2000K)
 13  ;
Table created.
```

```
SQL> execute sys.dbms_random.initialize(10);
PL/SQL procedure successfully completed.
SQL> insert into emp
  2    select to_char(rownum - 1, 'FM00000')                       empno
  3         , rpad(o.object_name,
  4                20   + mod(abs(sys.dbms_random.random),  50)) name
  5         , rownum * 10                                          salary
  6         , rpad('x',
  7                1000 + mod(abs(sys.dbms_random.random), 500)) photo
  8  from all_objects o, all_objects o2
  9  where rownum <= 1000;
1000 rows created.
SQL> insert into empiot
  2    select to_char(rownum - 1, 'FM00000')                       empno
  3         , rpad(o.object_name,
  4                20   + mod(abs(sys.dbms_random.random),  50)) name
  5         , rownum * 10                                          salary
  6         , rpad('x',
  7                1000 + mod(abs(sys.dbms_random.random), 500)) photo
  8  from all_objects o, all_objects o2
  9  where rownum <= 1000
 10  ;
1000 rows created.
```

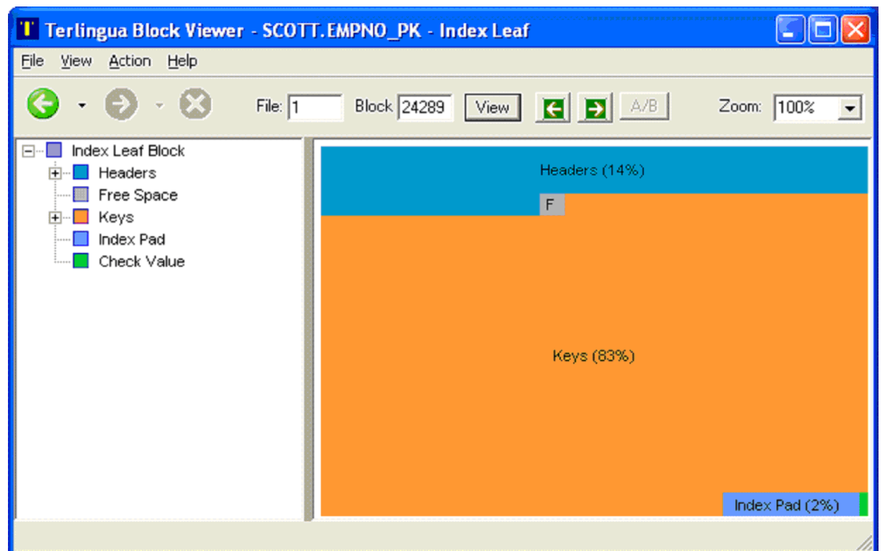
The following helper table and view illustrate the relationship between the three objects Oracle creates for the IOT.

```
SQL> create table helper as
  2    select o.name                      name
  3         , o.obj#                      obj#
  4         , i.file#                     file#
  5         , i.block#                    block#
  6         , i.bo#                       bobj#
  7         , 'index'                     typ
  8     from ind$ i, obj$ o
  9    where o.obj# = i.obj#
 10  union all
 11    select o.name                      name
 12         , o.obj#                      obj#
 13         , t.file#                     file#
 14         , t.block#                    block#
 15         , decode(t.file#, 0, t.bobj#) bobj#
 16         , 'table'                     typ
 17     from tab$ t, obj$ o
 18    where o.obj# = t.obj#
 19  ;
Table created.
SQL> select obj#
  2       , bobj#
  3       , typ
  4       , name
  5       , file#
  6       , block#
  7  from helper
  8  start with        obj# = 6673
  9  connect by prior bobj# = obj#
 10  ;
      OBJ#      BOBJ# TYP   NAME                               FILE#     BLOCK#
---------- ---------- ----- ------------------------------ ---------- ----------
      6673       6671 index EMPNOIOT_PK                             1      23787
      6671       6672 table EMPIOT                                  0          0
      6672            table SYS_IOT_OVER_6671                       1      23287
```

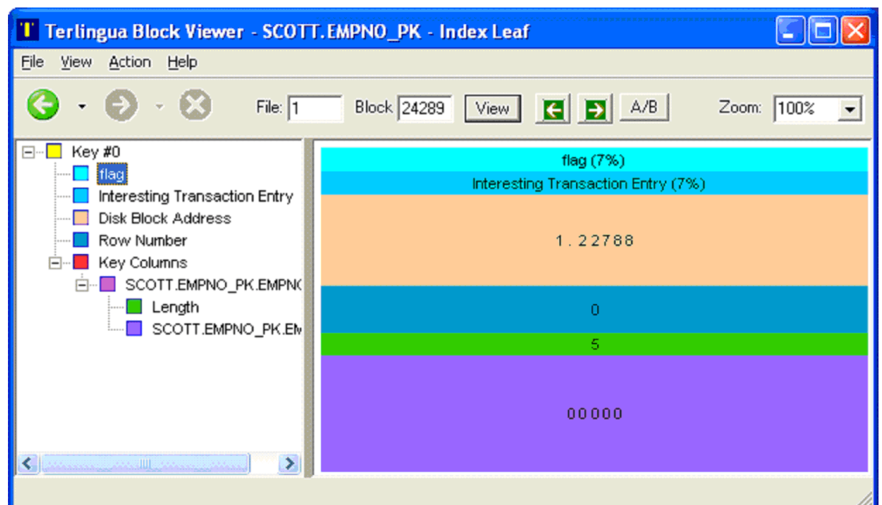
The first object is the index itself (EMPNOIOT_PK) - containing the index key (EMPNO) and one or more non-key columns (NAME and SALARY). The second object is a placeholder in the Oracle data dictionary. It does not take up any space on disk (as evidenced by FILE#, BLOCK# both being 0). EMPIOT is inserted into the dictionary by Oracle to be a reference for all the columns. That is, all the columns of the EMPIOT table stored in COL$ have a foreign key reference to this table (object number 6671). The actual location of any overflow columns

(e.g. PHOTO), are stored in the table aptly named "SYS_IOT_OVER_6671." In summary, we have the EMPNOIOT_PK index containing columns EMPNO, NAME and SALARY, the SYS_IOT_OVER_6671 table containing the overflow columns (PHOTO) and the EMPIOT table as a placeholder in the data dictionary to serve as a reference for all the columns.

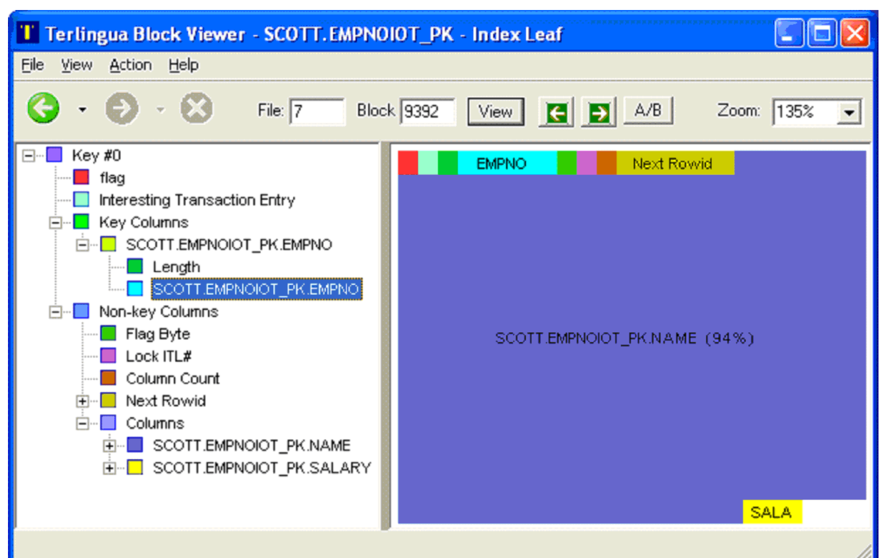
Let us take a look at some of the structures on disk, using our block viewer. First we will start with the basic B-tree index, EMPNO_PK. Right is a screen shot of the leaf block containing the first key ('00000').



Right is a screen shot showing the details of what is contained in the first key. The detail of the first index key confirms the rowid of the employee '00000' is 22788.0.1 when expressed in block.row.file format. Of course, since this is a normal B-tree index block, only the key value and rowid can be found in the index block - no non-key values can be found here.



Now, in contrast, let us look at a detailed screenshot for the index on EMPIOT. The non-key values of NAME and SALARY are stored directly in the index key along with the key, value. Note also the forward reference to the rowid containing the overflow column(s) as identified by "Next Rowid".



Performance Issues

Now that we have a solid understanding of how IOTs look on disk, let us explore some of the performance gains we can expect. In the following sections, the performance statistics are generated by SQL*Plus, with tracing turned on via the command "set autotrace on". As we are primarily interested in the plan output and the consistent gets, the logs of the examples have been edited for brevity.

Single Row Fetch

First let us compare a simple single column fetch via the primary key between the B-tree index and the IOT.

```
SQL> select empno, rpad(name,20) from emp    where empno = '00000';
EMPNO RPAD(NAME,20)
----- --------------------
00000 AGGXMLIMP
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     INDEX (UNIQUE SCAN) OF 'EMPNO_PK' (UNIQUE)
Statistics
----------------------------------------------------------
          3  consistent gets
SQL> select empno, rpad(name,20) from empiot where empno = '00000';
EMPNO RPAD(NAME,20)
----- --------------------
00000 DUAL

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=506)
   1    0   INDEX (UNIQUE SCAN) OF 'EMPNOIOT_PK' (UNIQUE) (Cost=1 Card
          =1 Bytes=506)
Statistics
----------------------------------------------------------
          2  consistent gets
```

In the normal B-tree/table combination, Oracle requires three consistent gets, one for the root block, one for the leaf block, and one to fetch the row from the table itself. In the IOT, as the name field is stored in the index record itself, only two consistent gets are required, one for the root block, and one for the leaf block. If all of these objects could fit in the Oracle buffer cache, there would not be a dramatic performance gain. However, as is often the case, if the base table is too large to fit in the buffer cache, and yet the index portion of the IOT could fit in the cache, this would save a physical read (a potentially dramatic savings). Just to clarify this point, consider some specific sizes for our B-tree/table implementation versus our IOT implementation. The following query reveals the nature of the EMP rows:

```
SQL> select avg(totalsize)
  2       , min(totalsize)
  3       , max(totalsize)
  4       , count(*)
  5    from table_rows
  6   where owner = 'SCOTT'
  7     and name  = 'EMP'
  8  ;
AVG(TOTALSIZE) MIN(TOTALSIZE) MAX(TOTALSIZE)   COUNT(*)
-------------- -------------- -------------- ----------
      1305.569           1044           1579       1000
```

The rows vary in size from 1044 bytes to 1579 bytes with and average length of about 1300 bytes. Consider an example of a 10 million row EMP table, deployed on a server with one

thousand megabytes of memory, devoted to the Oracle buffer cache. The following table approximates the size of the objects involved in an IOT implementation versus those of a standard B-tree/table implementation.

Object Name	***Size per each***	***Total Size (in Meg)***
EMP (empno, name, salary, photo)	*1300*	*13,000*
EMP_PK (empno)	*5*	*50*
EMPIOT_PK (empno, name, salary)	*50*	*500*
SYS_IOT_OVER_EMPIOT (photo)	*1250*	*12,500*

With a 1 gig buffer cache, a single row lookup of an employee's name in the standard B-tree/ table implementation is likely to cause one physical I/O to retrieve the given employee from EMP (none for the two index blocks, 1 for the table block). With the IOT implementation, the entire 10,000,000 long collection of empno.name.salary tuples (total size of 500 meg) is likely to be in the buffer cache. A name (or salary) retrieval via primary key is, therefore, not likely to cause a single physical read. If this is the dominant form of retrieving data in this system, IOTs could provide a substantial performance gain as well as reducing the demand for disk drives.

Index Range Scans

IOTs significantly outperform the standard B-tree/table model during index range scans. Consider a query to retrieve the average salary of a range of employees.

```
SQL> select avg(salary)
  2    from emp
  3   where empno >= '00000'
  4     and empno < '00100'
  5  ;
AVG(SALARY)
-----------
        505
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   3    2       INDEX (RANGE SCAN) OF 'EMPNO_PK' (UNIQUE)
Statistics
----------------------------------------------------------
         48  consistent gets
SQL> select avg(salary)
  2    from empiot
  3   where empno >= '00000'
  4     and empno < '00100'
  5  ;
AVG(SALARY)
-----------
        505
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=17)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (RANGE SCAN) OF 'EMPNOIOT_PK' (UNIQUE) (Cost=2 Car
          d=10 Bytes=170)
Statistics
----------------------------------------------------------
          3  consistent gets
```

Compare the 48 consistent gets needed by the standard B-tree/table versus the 3 consistent gets needed by the IOT. The range scan of the EMPNOIOT_PK index only took 3 consistent gets (one for the root block, and one for each of the first two blocks of the index) because employees in the range '00000'-'00100' fit on the lowest two index leaf blocks.

```
SQL> select branchlmchex leftmostchild
  2    from index_blocks
  3  where owner       = 'SCOTT'
  4    and name        = 'EMPNOIOT_PK'
  5    and commonlevel = 1
  6  ;
LEFTMOSTCHILD
-------------
0001.00005ced
SQL> select rpad(char01,8) empno
  2       , nextdbahex     leafdba
  3   from iot_keys
  4  where owner      = 'SCOTT'
  5    and name       = 'EMPNOIOT_PK'
  6    and indexlevel = 1
  7  order by 1
  8  ;
EMPNO    LEAFDBA
-------- -------------
00059    0001.00005cee
00117    0001.00005cef
00174    0001.00005cf0
00231    0001.00005cf1
00287    0001.00005cf2
00343    0001.00005cf3
004      0001.00005cf4
00456    0001.00005cf5
00511    0001.00005cf6
00568    0001.00005cf7
00624    0001.00005cf8
0068     0001.00005cf9
00735    0001.00005cfa
0079     0001.00005cfb
00846    0001.00005cfc
009      0001.00005cfd
00956    0001.00005cfe
17 rows selected.
SQL> select rpad(max(char01),8) sample
  2       , mydbadothex         leafdba
  3       , count(*)            n
. 4   from iot_keys
  5  where owner      = 'SCOTT'
  6    and name       = 'EMPNOIOT_PK'
  7    and indexlevel = 0
  8    and char01     < '00100'
  9  group by mydbadothex
 10  order by 1
 11  ;
SAMPLE   LEAFDBA                N
-------- ------------- ----------
00058    0001.00005ced         59
00099    0001.00005cee         41
```

The first query fetched the disk block address (dba) of the leaf block containing the lowest key values (a.k.a. the "leftmost child") from the root index block. The second query fetched the terminal key for each of the entries in the root block. From this query, we can see that keys less than '00059' lie in the leftmost child, and keys greater than or equal to '00059', but less than '00117', lie in block '0001.00005cee'. The final query confirms this by fetching all the keys in the leaf blocks containing key values less than '00100'. We can see 59 such keys in the leftmost child, '0001.00005ced', and 41 in '0001.00005cee'. Armed with this information, we can see why the query required only 3 consistent gets.

The Neutrals

The following operations with IOTs seem to have no significant performance gain or detriment when compared to a standard B-tree/table implementation. Keep in mind, having no detriment is a positive outcome, since this allows IOTs to be used in a wider range of applications. That is, for a wider range of applications, administrators can reap the benefits of IOTs without paying a significant cost.

Inserts

I would speculate a normal insertion of a row into our standard B-tree/table implementation would normally take 5 block gets:

1. fetch index root block
2. update index leaf block (to confirm uniqueness and pre-insert new key, without known rowid)
3. fetch table segment header to find block on free list
4. update table block to insert new row (noting new rowid)
5. update index leaf block (to update key entry with new rowid).

I would speculate an insertion of a row into an IOT should take the same 5 block gets:

1. fetch index root block
2. update index leaf block (to confirm uniqueness, pre-insert new key, place as much non-key column values as space permits, rowid still unknown)
3. fetch overflow table segment header to find block on free list
4. update overflow block to insert columns/column fragments
5. update the index leaf block (to update key entry with new rowid).

```
SQL> insert into emp
  2    values('01006'
  3         , rpad('John Doe', 50)
  4         , 30000
  5         , rpad('Photo', 1200)
  6  );
1 row created.
Statistics
----------------------------------------------------------
          4  db block gets
          1  consistent gets
SQL> commit;
Commit complete.
SQL> insert into empiot
  2    values('01006'
  3         , rpad('John Doe', 50)
  4         , 30000
  5         , rpad('Photo', 1200)
  6  );
1 row created.
Statistics
----------------------------------------------------------
          4  db block gets
          1  consistent gets
SQL> commit;
Commit complete.
```

These statistics seem to confirm what we expected – a single row insertion takes the same amount of block gets. This is good news; basic, single row inserts, should not be any slower.

Size of index

An obvious downside of IOTs is that of a larger index. With non-key values stored along with the key values in the index, the IOT index will be larger than a standard B-tree index. In our above example, with 10,000,000 EMP rows, the size of the standard B-tree index was 50

megabytes, while the size of the IOT index was ten times larger (500 megabytes). Now in a system with 1,000 megabytes of Oracle buffer cache, this might not make such a difference, but in a system with, say 100 megabytes of buffer cache, accessing name via primary key is likely to cause a physical read for either the IOT implementation, or the standard B-tree implementation. Avoiding physical I/Os, the gain for using IOTs, has been largely unrealized. However, unless the buffer cache is unfortunately sized, or the non-keys values stored in the IOT are unduly large, I do not believe the larger index would have a dramatic negative performance impact.

Non-key column retrieval

Clearly the retrieval of a column (via primary key) not stored in the index portion of the IOT, but in the overflow portion, will not benefit from being in an IOT. The following statistics illustrate this.

```
SQL> select empno, length(photo) from emp    where empno = '00000';
EMPNO LENGTH(PHOTO)
----- -------------
00000          1107
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     INDEX (UNIQUE SCAN) OF 'EMPNO_PK' (UNIQUE)
Statistics
----------------------------------------------------------
          3  consistent gets
SQL> select empno, length(photo) from empiot where empno = '00000';
EMPNO LENGTH(PHOTO)
----- -------------
00000          1392
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=1006)
   1    0   INDEX (UNIQUE SCAN) OF 'EMPNOIOT_PK' (UNIQUE) (Cost=1 Card
          =1 Bytes=1006)
Statistics
----------------------------------------------------------
          3  consistent gets
```

Both the IOT and the standard B-tree/table implementation required 3 consistent gets to retrieve the length(photo) - index root block, index leaf block, table block. At first glance this does not appear to be a disadvantage of IOTs only a lack of an advantage. Remember, however, the IOT index blocks store far fewer keys per block as illustrated by the following query (uniquely provided by our product).

```
SQL> select mydbadot             mydba
  2       , round(avg(keysize))  keysiz
  3       , count(*)             nkeys
  4    from index_keys
  5   where owner       = 'SCOTT'
  6     and name        = 'EMPNO_PK'
  7     and indexlevel  = 0
  8  group by mydbadot
  9  ;
MYDBA            KEYSIZ      NKEYS
------------ ---------- ----------
1.24289               5        243
1.25104               5         28
2.2162                5        243
2.2163                5        243
2.2164                5        243
```

```
SQL> select mydbadot              mydba
  2       , round(avg(keysize))    keysiz
  3       , round(avg(nonkeysize)) nonkeysiz
  4       , count(*)               nkeys
  5    from iot_keys
  6   where owner       = 'SCOTT'
  7     and name        = 'EMPNOIOT_PK'
  8     and indexlevel  = 0
  9   group by mydbadot
 10   ;
MYDBA            KEYSIZ  NONKEYSIZ      NKEYS
------------ ---------- ---------- ----------
1.23789               5         56         59
1.23790               5         57         58
1.23791               5         57         57
1.23792               5         57         57
1.23793               5         59         56
1.23794               5         59         56
1.23795               5         58         57
1.23796               5         58         56
1.23797               5         61         55
1.23798               5         57         57
1.23799               5         59         56
1.23800               5         59         56
1.23801               5         61         55
1.23802               5         61         55
1.23803               5         60         56
1.23804               5         61         54
1.23805               5         60         56
1.23806               5         57         44
18 rows selected.
```

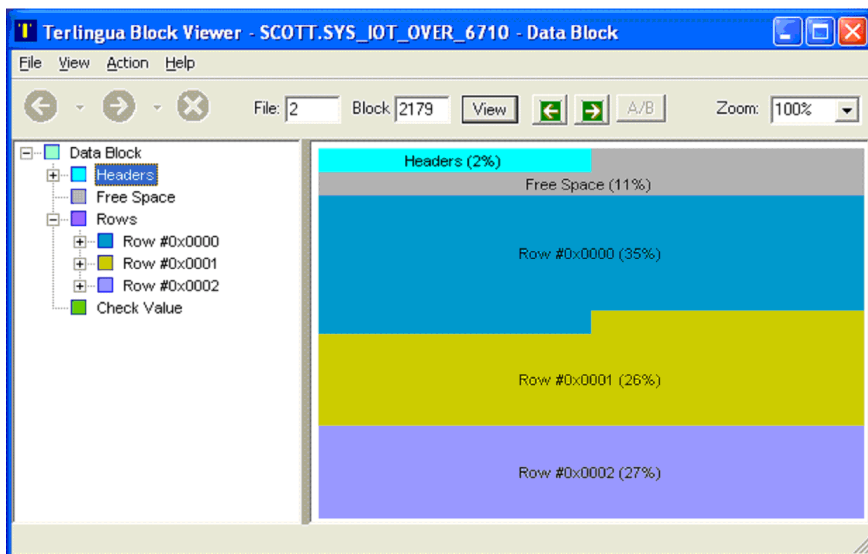
Oracle packs 4 to 5 times as many keys per block in the standard B-tree/table implementation versus that of the IOT implementation. This will cause the IOT implementation to create a new index level 4-5 times sooner than that of the standard implementation. In fact, if the buffer cache was just the wrong size, the IOT implementation may need to perform two physical reads for this operation (one for the index leaf block and a second for the overflow table block). All this having been said, however, I would consider the increased size of an IOT index a "neutral" and would not be a significant downside for using IOTs.

The Downsides

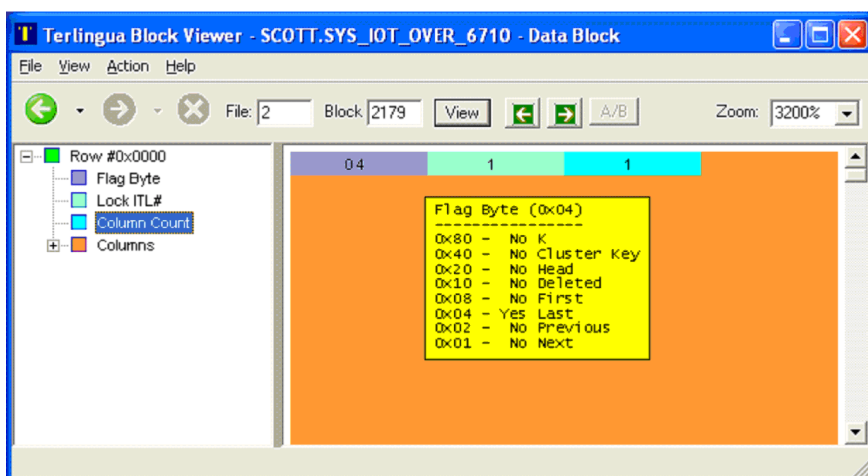
We have seen some of the performance gains IOTs can bring to a system, but are there any downsides? The good news is that I have only been able to find one significant performance detriment to using IOTs. Unfortunately, it is such a detriment that it may make IOTs a non-option for your environment.

Full Table Scans

The only significant weakness of Oracle's IOT implementation (and I believe this to be such a weakness as to be considered a bug) is that Oracle does not store any column number information in the overflow table. By this I mean that, taken alone, the row fragments stored in the overflow table (e.g. the PHOTO column) do not contain a column offset.



Left is a complete block from the overflow table containing three row fragments which illustrates this lack of column offset.



Left is a detailed look at one of the fragments, confirming that no column offset is stored in the header of the row (only flag, ITL entry, and column count).

The implications of this are dramatic when looking at full table scan type queries. Take for example queries of the form:

```
select avg(length(photo)) from emp;
vs.
select avg(length(photo)) from empiot;
```

As the statement creating EMPIOT specifically stated, Oracle was not to store any PHOTO data in the index; Oracle should be able to simply perform a full table scan on the overflow table to compute the avg(length(photo)). However, since it is also possible for the salary column to spill into the overflow table, and since Oracle does not store column offsets with the overflow data, the length of each photo has to be computed through the index. That is, the only way for Oracle to know that the data in the overflow table is indeed PHOTO data, and not SALARY data, is to go to the row fragment via the key. This creates an inordinate number of single block physical reads when compared to the standard B-tree/table implementation as evidenced by the following statistics.

```
STATISTICS                                   TEST LOAD
-------------------------------------------------------------------------------------------------
SQL> create table bstat as
  2  select kcfiofno
  3       , kcfiopyr
  4       , kcfiopbr
  5  from x$kcfio
  6  where kcfiopyr != 0
  7  ;
Table created.
                                             SQL> select avg(length(photo)) from emp;
                                             AVG(LENGTH(PHOTO))
                                             ------------------
                                                       1244.511

                                             Execution Plan
                                             ----------------------------------------------------

                                                0      SELECT STATEMENT Optimizer=CHOOSE
                                                1    0   SORT (AGGREGATE)
                                                2    1     TABLE ACCESS (FULL) OF 'EMP'
                                             Statistics
                                             ----------------------------------------------------
                                                     477  consistent gets
                                                     475  physical reads

SQL> select n.kcfiofno                           fileno
  2       , n.kcfiopyr - o.kcfiopyr              pysrd
  3       , n.kcfiopbr - o.kcfiopbr              bksrd
  4       , round((n.kcfiopbr - o.kcfiopbr) /
  5               (n.kcfiopyr - o.kcfiopyr), 2) blksperread
  6  from bstat o, x$kcfio n
  7  where n.kcfiofno != 0
  8    and n.kcfiofno  = o.kcfiofno
  9    and n.kcfiopyr - o.kcfiopyr != 0
 10  ;
    FILENO      PYSRD      BKSRD BLKSPERREAD
---------- ---------- ---------- -----------
         1         30        475       15.83
SQL> create table bstat as
  2  select kcfiofno
  3       , kcfiopyr
  4       , kcfiopbr
  5  from x$kcfio
  6  where kcfiopyr != 0
  7  ;
Table created.
                                             SQL> select avg(length(photo)) from empiot;
                                             AVG(LENGTH(PHOTO))
                                             ------------------
                                                       1250.481
                                             Execution Plan
                                             ----------------------------------------------------

                                                0      SELECT STATEMENT Optimizer=CHOOSE
                                                1    0   SORT (AGGREGATE)
                                                2    1     INDEX (FAST FULL SCAN)
                                                             OF 'EMPNOIOT_PK' (UNIQUE)
                                             Statistics
                                             ----------------------------------------------------
                                                     475  consistent gets
                                                     473  physical reads

SQL> select n.kcfiofno                           fileno
  2       , n.kcfiopyr - o.kcfiopyr              pysrd
  3       , n.kcfiopbr - o.kcfiopbr              bksrd
  4       , round((n.kcfiopbr - o.kcfiopbr) /
  5               (n.kcfiopyr - o.kcfiopyr), 2) blksperread
  6  from bstat o, x$kcfio n
  7  where n.kcfiofno != 0
  8    and n.kcfiofno  = o.kcfiofno
  9    and n.kcfiopyr - o.kcfiopyr != 0
 10  ;    FILENO      PYSRD      BKSRD BLKSPERREAD
---------- ---------- ---------- -----------
         1        456        473        1.04
```

The statistic of note is that the normal B-tree/table implementation only performed 30 physical reads, averaging 15.83 blocks per read, while the IOT implementation erformed 456 physical reads, averaging 1.04 blocks per read. In a real world environment, the IOT implementation would have taken 15 times longer (456/30) than the standard B-tree/table

implementation. I believe this severe performance hit would be removed if Oracle simply chose to place column offsets in with each of the row fragments stored in the overflow table. Unfortunately, until this issue is resolved, applications which occasionally require a full table scan of their non-indexed columns, may find IOTs prohibitively expensive to use.

Conclusion

I hope this article has given the reader a clearer understanding of the fundamentals of index organized tables. Understanding the fundamentals is the key to understanding when index organized tables may be right for your application and, perhaps more importantly, when they may be dreadfully wrong. If you have a further interest in index organized tables, go to *www.tlingua.com* and you will find an advanced article on index only tables covering secondary indices on IOTs, benefits of compressed indices, and a more detailed look at the performance characteristics of loading IOTs.

As always, I hope this article has provided some important insight into the functioning of Oracle as well as given the reader some appreciation of the value our tools can provide to both the novice and veteran database administrator. If you have found the information herein to be valuable, please mention it in your e-mails or postings to Oracle web sites.