CHAPTER 4

Index-Organized Tables

The simplest explanation of an index-organized table is that it is accessed like any other Oracle table (typically a heap-organized table) but is physically stored like an Oracle B-tree index. Index-organized tables are typically created on "thin" tables (tables without too many columns). Typically, multiple columns of the table make up the primary key of the index-organized table. The non-key columns can also be stored as part of the B-tree index. The proper configuration and use of index-organized tables is fairly specific and does not meet all application needs.

Understanding the Structure

From a user or developer perspective, an index-organized table (IOT) appears like a normal table. IOTs are stored in a B-tree structure. There must be a primary key on an index-organized table, as the data is stored in primary key order. Since there is no data segment, there are no physical `ROWID` values for index-organized tables. Figure 4-1 presents an example of an IOT.

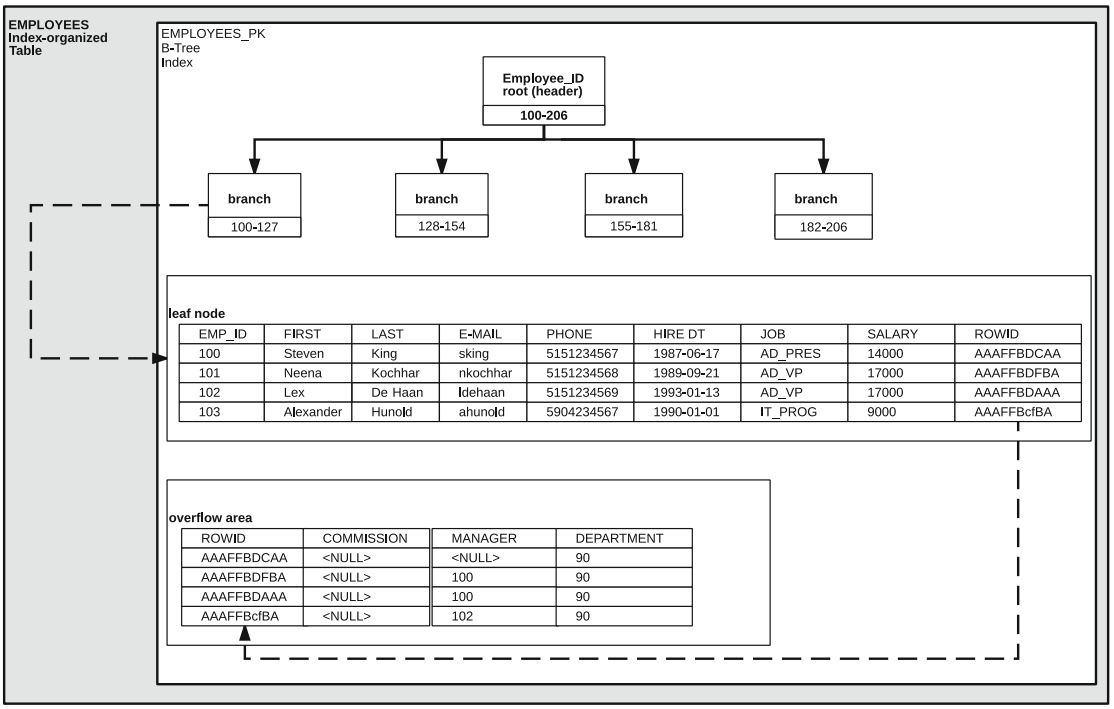

Figure 4-1. *Structure of an index-organized table*

D. Kuhn et al., *Expert Oracle Indexing and Access Paths*, DOI 10.1007/978-1-4842-1984-3_4

IOTs support many of the same features found in heap-organized tables, such as

- Constraints
- Partitioning
- Triggers
- LOB columns
- Parallelism
- Indexes (e.g., secondary indexes on IOTs)
- Global hash-partitioned indexes
- Online reorganization

Because all of the data within an index-organized table is stored within the index itself, there are physical differences in the way an index-organized table is stored, as compared to a normal B-tree index that supports a normal heap-organized table. Some of the unique aspects of IOTs are as follows:

- Secondary indexes use logical `ROWID`s rather than physical `ROWID`s.
- They *require* a primary key.
- Primary key compression can be used to save storage and reduce the size of an IOT.
- An overflow segment can be used for non-key column data.
- Secondary bitmap indexes require a defined mapping table.
- Non-key column data is stored in the leaf blocks of an IOT.

There are limitations on index-organized tables, although many of the limitations do not affect their use in the majority of applications. The following are some of these limitations:

- Rows exceeding 50% of a block must use an overflow segment.
- IOTs can't use virtual columns.
- Tables with more than 255 columns must have an overflow segment.
- Tables can't have more than 1,000 total columns.
- The primary key can't be more than 32 columns.

Understanding the Advantages

There are specific advantages of IOTs, including the following:

- Storage space can be saved because the data is the index, so there is only one segment or set of segments in the database for an index-organized table, rather than the normal two segments that come with a heap-organized table and associated index(es).
- Query performance benefits can occur because there are fewer I/O requirements. Since the data is stored as part of the index, there is a potentially significant I/O reduction.
- DML performance benefits can occur because there is only the need to update the index segment(s), as there is no data segment(s) as part of the structure. There is no need to update the table and then any associated index as with heap-organized tables. Only the index needs to be updated.

Index-organized tables are most beneficial in OLTP environments for the following reasons:

- IOTs allow fast primary key access.
- IOTs allow online reorganization, which is essential in an OLTP environment.
- IOTs allow fast data retrieval in applications like Internet search engines.

The biggest challenge with index-organized tables is deciding when to use them. If you have tables that have several columns that make up the primary key, and the table itself is not dense as far as the number of columns, it *may* be a candidate as an IOT. However, this by itself is not reason enough to make a table into an index-organized table. There should be a tangible benefit gained from having a table structure be index-organized, and this may require some testing of your application. Generally, index-organized tables provide fast lookup of the primary key. They can be slower for inserts. Likewise, secondary index access isn't as fast as a normal B-tree index because index-organized table rows don't have the physical `ROWID` that would be found in a heap-organized table. Instead, IOTs use a logical `ROWID`, which isn't as exact as a physical `ROWID` and can become outdated over time. Because of this, index-organized tables may have to be rebuilt periodically for performance reasons. All in all, the use of index-organized tables should be limited and specific to a particular need. They are best used when fast primary key access is required. As mentioned, performing DML on index-organized tables is slower and over time can cause a degradation of query performance. Therefore, IOTs are best suited for data that is fairly static, with minimal DML activity.

Creating an Index-Organized Table

The data definition language (DDL) for an index-organized table is very similar to the DDL for a heap-organized table. The key difference is the use of the `ORGANIZATION INDEX` clause, which tells Oracle that you are creating an index-organized table; for example:

```
SQL> CREATE TABLE locations_iot
  2   (LOCATION_ID         NUMBER(4)         NOT NULL
  3   ,STREET_ADDRESS      VARCHAR2(40)
  4   ,POSTAL_CODE         VARCHAR2(12)
  5   ,CITY                VARCHAR2(30)      NOT NULL
  6   ,STATE_PROVINCE      VARCHAR2(25)
  7   ,COUNTRY_ID          CHAR(2)
  8   ,CONSTRAINT locations_iot_pk PRIMARY KEY (location_id)
  9  )
 10  ORGANIZATION INDEX;

Table created.
```

As previously stated, you must have a primary key defined on an IOT. Since the IOT is stored in a B-tree index structure, there is no physical `ROWID` stored with each row. That's why you must have a primary key on an IOT—so that each row can be uniquely identified.

The B-tree structure of an index-organized table is based on the primary key values. If you don't specify a primary key, you will get the following error:

```
SQL> CREATE TABLE locations_iot
  2   (LOCATION_ID         NUMBER(4)         NOT NULL
  3   ,STREET_ADDRESS      VARCHAR2(40)
  4   ,POSTAL_CODE         VARCHAR2(12)
  5   ,CITY                VARCHAR2(30)      NOT NULL
```

```
  6   ,STATE_PROVINCE      VARCHAR2(25)
  7   ,COUNTRY_ID          CHAR(2)
  8  )
  9  ORGANIZATION INDEX;

organization index
             *
ERROR at line 10:
ORA-25175: no PRIMARY KEY constraint found
```

For the most part, index-organized tables can be partitioned just like a heap-organized table. You can partition index-organized tables using the following partitioning methods: `range`, `list`, or `hash` partitioning. Using the `LOCATIONS_IOT` from the previous example, you can `list` partition the table by `STATE_PROVINCE` based on whether it is a domestic or an international state province, as shown in the following DDL:

```
SQL> CREATE TABLE locations_iot
  2   (LOCATION_ID         NUMBER(4)         NOT NULL
  3   ,STREET_ADDRESS      VARCHAR2(40)
  4   ,POSTAL_CODE         VARCHAR2(12)
  5   ,CITY                VARCHAR2(30)      NOT NULL
  6   ,STATE_PROVINCE      VARCHAR2(25)      NOT NULL
  7   ,COUNTRY_ID          CHAR(2)
  8   ,constraint locations_iot_pk primary key (location_id, state_province)
  9  )
 10  ORGANIZATION INDEX
 11  partition by list(STATE_PROVINCE)
 12  (partition p_intl values
 13  ('Maharashtra','Bavaria','New South Wales', 'BE','Geneve',
 14   'Tokyo Prefecture', 'Sao Paulo','Manchester','Utrecht',
 15   'Ontario','Yukon','Oxford'),
 16  partition p_domestic values ('Texas','New Jersey','Washington','California'));

Table created.
```

You can't use composite partitioning in relation to index-organized tables. The following DDL snippet is attempting to create a composite `range-list` partitioned table:

```
SQL> CREATE TABLE locations_iot
  2  ...
 17  organization index
 18  partition by range(hire_date)
 19  subpartition by list( DEPARTMENT_ID)
 20  subpartition template
 21  (SUBPARTITION JOB10 VALUES ('10')
 22  ,SUBPARTITION JOB20 VALUES ('20')
 23  ,SUBPARTITION JOB30 VALUES ('30')
 24  ,SUBPARTITION JOB40 VALUES ('40')
 25  ,SUBPARTITION JOB50 VALUES ('50')
 26  ,SUBPARTITION JOB60 VALUES ('60')
 27  ,SUBPARTITION JOB70 VALUES ('70')
 28  ,SUBPARTITION JOB80 VALUES ('80')
```

```
 29  ,SUBPARTITION JOB90 VALUES ('90')
 30  ,SUBPARTITION JOB100 VALUES ('100')
 31  ,SUBPARTITION JOB110 VALUES ('110')
 32  (
 33  partition p1990 values less than ('1991-01-01'),
 ...
 45  );

subpartition template
*
ERROR at line 20:
ORA-25198: only range, list, and hash partitioning are supported for
index-organized table
```

This error clearly indicates that composite partitioning is not supported. For more information on the features of IOTs and their limitations, see the *Oracle Database Administrator's Guide* for your release of the database.

Adding an Overflow Segment

For index-organized tables, it is common, and even recommended, to create an overflow area for row data as part of the overall index-organized table structure. The typical index-organized table that includes an overflow area is structured as follows:

- B-tree index entry, which includes the following:
  - Primary key columns
  - Some non-key columns, depending on `PCTTHRESHOLD` and `INCLUDING` clause values specified
  - A physical `ROWID` pointer to the overflow segment
- Overflow segment, which contains the remaining non-key column values

In a normal B-tree index, the leaf node contains the index column key value, and then the `ROWID` for the row in the data segment. With index-organized tables, all the non-key column values are stored within the leaf blocks of the index by default. If the row data becomes very wide, the B-tree entries can become very large. This can slow data retrieval simply because the index must traverse more index blocks.

The overflow segment can aid in the efficiency of the overall B-tree index of an index-organized table by storing some of the non-key column values in an overflow data segment of the IOT, which is used solely to store these non-key column values. Associated with the overflow area is the `PCTTHRESHOLD` parameter, which specifies how column data goes to the overflow segment. If the length of a row is greater than the percentage of the index block specified by the `PCTTHRESHOLD` parameter (the default is 50), every column that exceeds the threshold will be stored in the overflow area. Also, you can specify the overflow segment to a specific tablespace, if desired.

■ **Tip** Use the `ANALYZE TABLE...LIST CHAINED ROWS` command to determine if you have set `PCTTHRESHOLD` appropriately.

In contrast to the PCTTHRESHOLD parameter, there is the INCLUDING clause, which specifies the last table column for the row data that will be stored in the B-tree index segment. All columns *after* the column specified by the INCLUDING clause will be stored in the overflow area. It is possible to specify both the PCTTHRESHOLD and INCLUDING clauses, as shown in the following example:

```
SQL> CREATE TABLE employees
  2  (
  3   EMPLOYEE_ID          NUMBER(6)         NOT NULL
  4   ,FIRST_NAME          VARCHAR2(20)
  5   ,LAST_NAME           VARCHAR2(25)      NOT NULL
  6   ,EMAIL               VARCHAR2(25)      NOT NULL
  7   ,PHONE_NUMBER        VARCHAR2(20)
  8   ,HIRE_DATE           DATE              NOT NULL
  9   ,JOB_ID              VARCHAR2(10)      NOT NULL
 10   ,SALARY              NUMBER(8,2)
 11   ,COMMISSION_PCT      NUMBER(2,2)
 12   ,MANAGER_ID          NUMBER(6)
 13   ,DEPARTMENT_ID       NUMBER(4)
 14   ,CONSTRAINT employees_pk PRIMARY KEY (employee_id)
 15  )
 16  ORGANIZATION INDEX
 17  TABLESPACE empindex_s
 18  PCTTHRESHOLD 40
 19  INCLUDING salary
 20  OVERFLOW TABLESPACE overflow_s

Table created.
```

Figure 4-1 shows an illustration of an index-organized EMPLOYEES table row as stored in the table, as well as the overflow segment. In the example, you can see that the primary key in the EMPLOYEES table is the EMPLOYEE_ID, and the root block, branch blocks, and leaf blocks are structured based on the primary key. Within the leaf blocks themselves is the primary key, as well as all of the non-key columns up through the SALARY column, which corresponds to the INCLUDING clause in the CREATE TABLE DDL statement. All column data after the SALARY column is therefore stored in the overflow segment.

For performance reasons, the order of columns within an index-organized table is important, unlike normal heap-organized tables. This is simply because of the overflow segment. The most queried columns should *not* be placed in the overflow segment, simply because it is an extra I/O operation to retrieve the remaining column data for a given row. For this reason, the least queried columns should be placed on the trailing end of the table DDL, especially those after the column specified in the INCLUDING clause. In the table example, let's say that you determine through user interviews that the most queried columns on your EMPLOYEES table will be the JOB_ID, DEPARTMENT_ID, and MANAGER_ID columns. The initial DDL placed the DEPARTMENT_ID and MANAGER_ID columns in the overflow segment.

Based on the user interviews, it may be beneficial to move these two columns *above* the INCLUDING clause and possibly shift some other columns below the INCLUDING clause. The following example shows a modified DDL that rearranges the column order such that the DEPARTMENT_ID and MANAGER_ID are moved above the INCLUDING clause, and thus out of the overflow segment:

```
CREATE TABLE employees
(
 EMPLOYEE_ID          NUMBER(6)         NOT NULL
 ,FIRST_NAME          VARCHAR2(20)
 ,LAST_NAME           VARCHAR2(25)      NOT NULL
 ,EMAIL               VARCHAR2(25)      NOT NULL
 ,PHONE_NUMBER        VARCHAR2(20)
 ,HIRE_DATE           DATE              NOT NULL
 ,JOB_ID              VARCHAR2(10)      NOT NULL
 ,SALARY              NUMBER(8,2)
 ,MANAGER_ID          NUMBER(6)
 ,DEPARTMENT_ID       NUMBER(4)
 ,COMMISSION_PCT      NUMBER(2,2)
 ,MANAGER_ID          NUMBER(6)
 ,DEPARTMENT_ID       NUMBER(4)
 ,CONSTRAINT employees_pk PRIMARY KEY (employee_id)
)
ORGANIZATION INDEX
TABLESPACE empindex_s
PCTTHRESHOLD 40
INCLUDING DEPARTMENT_ID
OVERFLOW TABLESPACE overflow_s;
```

Based on your user interviews, it may also mean (based on the necessary queries against the EMPLOYEES table) that you decide not to create an overflow segment for the EMPLOYEES table. Creation of the overflow segment, and which columns to place there, should be done after careful analysis based on the proposed usage of the table columns.

If you choose to specify an INCLUDING clause within the DDL for an IOT, you must specify an OVERFLOW area, else you will receive the following error:

```
create table employees_iot
*
ERROR at line 1:
ORA-25186: INCLUDING clause specified for index-organized table without
OVERFLOW
```

Also, the Oracle data dictionary can become cluttered with entries for the overflow areas that have been dropped from an index-organized table. Recyclebin objects are normally seen in the DBA_SEGMENTS view, but for IOT overflow segments, you can see them in the USER_TABLES view (or appropriate ALL or DBA views), including those that have been dropped. The following query and results provide an example:

```
SQL> select table_name, iot_type from user_tables
  2  where iot_type like '%IOT%';

TABLE_NAME                     IOT_TYPE
------------------------------ ------------
SYS_IOT_OVER_77689             IOT_OVERFLOW
SYS_IOT_OVER_77692             IOT_OVERFLOW
SYS_IOT_OVER_77697             IOT_OVERFLOW
EMPLOYEES_IOT                  IOT
```

Therefore, purge the recyclebin to get rid of superfluous overflow entries.

```
SQL> purge recyclebin;

Recyclebin purged.
```

After you purge the recyclebin, the dropped overflow objects no longer show in the data dictionary.

```
SQL> select table_name, iot_type from user_tables
  2  where iot_type like '%IOT%'

TABLE_NAME                     IOT_TYPE
------------------------------ ------------
SYS_IOT_OVER_77697             IOT_OVERFLOW
EMPLOYEES_IOT                  IOT
```

■ **Tip** Always attempt to keep the most frequently accessed columns within the table itself—and outside of the overflow segment—for better access performance.

Compressing an Index-Organized Table

You can use a concept called *key compression* on index-organized tables to save storage space and compress data. It's called "key compression" because it can eliminate repeated values of the key columns. You can use key compression with either a `CREATE TABLE` statement or an `ALTER TABLE` statement. The following is a sample DDL of a `CREATE TABLE` statement with key compression enabled:

```
SQL> CREATE TABLE employees_iot
  2  (
  3   EMPLOYEE_ID          NUMBER(7)         NOT NULL
  4   ,FIRST_NAME          VARCHAR2(20)
  5   ,LAST_NAME           VARCHAR2(25)      NOT NULL
  6   ,EMAIL               VARCHAR2(25)      NOT NULL
  7   ,PHONE_NUMBER        VARCHAR2(20)
  8   ,HIRE_DATE           DATE              NOT NULL
  9   ,JOB_ID              VARCHAR2(10)      NOT NULL
 10   ,SALARY              NUMBER(8,2)
 11   ,COMMISSION_PCT      NUMBER(2,2)
 12   ,MANAGER_ID          NUMBER(6)
 13   ,DEPARTMENT_ID       NUMBER(4)
15   ,CONSTRAINT employees_iot_pk PRIMARY KEY (employee_id, job_id)
17  )
18  ORGANIZATION INDEX COMPRESS 1
19  TABLESPACE empindex_s
20  PCTTHRESHOLD 40
21  INCLUDING salary
22  OVERFLOW TABLESPACE overflow_s;

Table created.
```

If you have a pre-existing table on which you want to enable key compression, you can simply use the ALTER TABLE...MOVE statement to enable the compression.

```
SQL> ALTER TABLE employees_iot MOVE TABLESPACE empindex_s COMPRESS 1;

Table altered.
```

You can only use key compression when there are multiple columns as part of the primary key; otherwise, you receive the following message when creating the table:

```
CREATE TABLE employees_iot
*
ERROR at line 1:
ORA-25193: cannot use COMPRESS option for a single column key
```

For obvious reasons, you can't use the same number of key columns as is within the primary key for the key compression factor specified in the COMPRESS clause because that represents a unique value, and therefore no key compression is possible. You receive the following error if you attempt to create the table with the same compression factor as the number of primary key columns:

```
CREATE TABLE employees_iot
*
ERROR at line 1:
ORA-25194: invalid COMPRESS prefix length value
```

The compression occurs when there are duplicates within the columns of the primary key. For instance, if the employee with EMPLOYEE_ID 100 worked several jobs over the years, they would have several entries for the EMPLOYEE_ID/JOB_ID combination. For rows with duplicates of the EMPLOYEE_ID itself, all repeated values would be compressed. Table 4-1 provides a brief example of the results from key compression.

Table 4-1. *Example of Key Compression for Employee_ID 100*

Employee_ID	Job_ID	Employee_ID Value Compressed?
100	AD_ASST	NO (first entry)
100	IT_PROG	YES
100	AD_VP	YES
100	…	YES for all subsequent entries

Building Secondary Indexes

The index-organized table can be viewed the same as a heap-organized table in that if other indexes are needed to speed query performance, secondary indexes can be added to index-organized tables because they can be added on heap-organized tables. The following example shows how to create a secondary index on DEPARTMENT_ID in the EMPLOYEES_IOT table:

```
SQL> CREATE INDEX employees_iot_1i
  2  ON employees_iot (department_id);
```

You can also create secondary indexes on partitioned IOTs.

```
SQL> CREATE INDEX employees_iot_1i
  2  on employees_iot (department_id)
  3  LOCAL;
```

The key difference between secondary indexes on heap-organized tables and secondary indexes on index-organized tables is that there is no physical ROWID for each row in an index-organized table because the table data is stored as part of the B-tree index. Therefore, all access to data within an index-organized table is based on the primary key.

Instead of the normal physical ROWIDs to locate table rows, index-organized tables use a logical ROWID, which is used by any secondary indexes on the IOT in order to retrieve data. The logical ROWID is the equivalent to a physical guess of the row location based on the ROWID when the index entry was first created. Based on the physical guess, Oracle scans through leaf blocks searching for a match. The physical guess doesn't change over time, even if a row's physical location changes. For instance, leaf block splits can occur over time, which can fragment the index and change a row's physical location. Because the physical guess is not updated even if a row location changes, the physical guesses can become outdated or stale over time.

You can get information from the data dictionary to determine if the physical guesses for an IOT are stale by querying the PCT_DIRECT_ACCESS column of USER_INDEXES, as follows:

```
SQL> select index_name, index_type, pct_direct_access
  2  from user_indexes;

INDEX_NAME                     INDEX_TYPE                  PCT_DIRECT_ACCESS
------------------------------ --------------------------- -----------------
EMPLOYEES_IOT_PK               IOT - TOP                                   0
EMPLOYEES_PART_1I              NORMAL                                    100
```

If the PCT_DIRECT_ACCESS value falls below 100, it means that the secondary index entries are becoming migrated, and the physical guess can start to be inaccurate enough that extra I/O operations start occurring and performance starts to degrade. Once the PCT_DIRECT_ACCESS falls below 80, performance degradation starts becoming more noticeable, and the index may be a good candidate for a rebuild operation.

In order to refresh the logical ROWIDs over time, there are two primary ways to address the issue:

- Rebuild the secondary index.
- Update the block references for the index.

The first way to refresh the logical ROWIDs within secondary indexes is simply by rebuilding the index(es). Rebuilding secondary indexes built on index-organized tables is no different from rebuilding indexes on heap-organized tables.

```
SQL> ALTER INDEX employees_1i REBUILD;
```

Of course, depending on the size of the table, rebuilding one or more secondary indexes can take time, and with shrinking maintenance windows and ever-increasing availability windows on databases, it can be problematic to rebuild indexes on large tables on a regular basis.

An alternative to rebuilding your secondary indexes and a quick way to fix stale physical guesses within your secondary indexes is to use the ALTER INDEX…UPDATE BLOCK REFERENCES command, which quickly realigns stale physical guesses without having to rebuild an entire index.

```
SQL> ALTER INDEX employees_part_1i UPDATE BLOCK REFERENCES;
```

You can also place bitmap indexes on IOTs as secondary indexes. Refer to Chapter 3 for examples of creating bitmap indexes on an IOT. Within the bitmap index, since there is an entry for each row in a given table, there is normally a ROWID, along with the bitmap and data value corresponding to the indexed column. Since there are no physical ROWID values with an index-organized table, a bitmap index that is built on an index-organized table must be managed differently. When creating the bitmap index on the IOT, you must include a mapping table within the bitmap index. Again, see Chapter 3 for an example of how to build a bitmap index on an index-organized table.

A mapping table is simply a heap-organized table that is used to store the logical ROWID values. The mapping table is essentially an object that replaces the physical ROWID representation with a logical ROWID representation for the rows in the table. So, within the bitmap index itself, the physical ROWID is from the mapping table, rather than from the base table. Then the mapping table is accessed to retrieve the logical ROWID in order to access the data from the index-organized table. Figure 4-2 shows an example of a bitmap index with a mapping table.

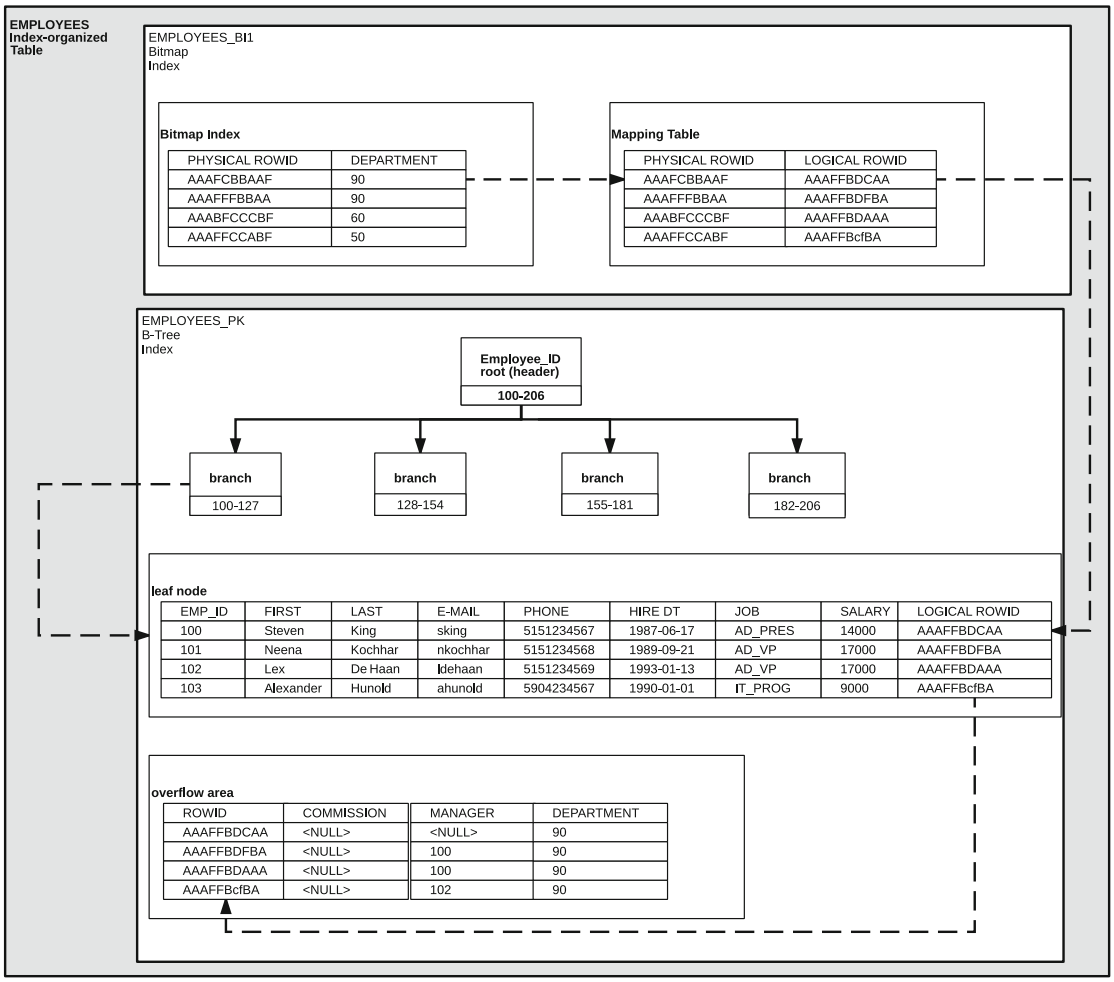

Figure 4-2. *A bitmap index within an index-organized table*

Rebuilding an Index-Organized Table

An index-organized table is a B-tree index. It can become fragmented over time and can incur the same issues as a normal B-tree index: an expanded index depth over time, an unbalanced tree, and sparse blocks, to name a few. Therefore, you can rebuild an index-organized table as you would a normal B-tree index. The obvious difference is that because it is regarded as a table, you rebuild an IOT with the `ALTER TABLE` command, as shown in the following example:

```
SQL> ALTER TABLE employees_iot MOVE;

Table altered.
```

If you want to move the IOT to a different tablespace, simply specify the tablespace within the `ALTER TABLE` clause, as shown in the following example:

```
SQL> ALTER TABLE employees_iot MOVE TABLESPACE emp_s;

Table altered.
```

When an IOT is rebuilt, the overflow segment is not rebuilt by default. Since similar fragmentation issues can occur with the overflow segment, it's a good idea to always rebuild the overflow segment whenever you rebuild the IOT itself, as shown in the following examples:

```
SQL> ALTER TABLE employees_iot MOVE overflow;

SQL> ALTER TABLE employees_iot MOVE tablespace emp_s
  2  overflow tablespace overflow_s;

Table altered.
```

You can also rebuild an IOT with the `ONLINE` clause, which means that the existing structure can be accessed during the rebuild operation.

```
SQL>  alter table employees_iot move tablespace users online;

Table altered.
```

An index-organized table can be partitioned just as any other heap-organized table can be partitioned. If you are rebuilding a partitioned IOT, you can't rebuild it in one step—that is, the entire table; if you do, you receive the following error:

```
SQL> ALTER TABLE employees_iot MOVE;
ALTER TABLE employees_iot MOVE
            *
ERROR at line 1:
ORA-28660: Partitioned Index-Organized table may not be MOVEd as a whole
```

If you wish to rebuild an entire partitioned IOT, you must do it one partition at a time. You need to get the partition names from the index itself using the USER_IND_PARTITIONS view (or, of course, optionally the equivalent ALL or DBA views), and then issue the ALTER TABLE…MOVE PARTITION command in order to move each partition of an IOT, as shown in the following example:

```
SQL> select partition_name
  2  from user_ind_partitions
  3* where index_name = 'EMPLOYEES_IOT_PK';

PARTITION_NAME
------------------------------
P1990
...
P1999
P2000
PMAX

SQL> ALTER TABLE employees_iot MOVE PARTITION p1990;

Table altered.
```

You must rebuild the IOT with an ALTER TABLE command. If you attempt to rebuild an IOT via the primary key index, you receive the following error:

```
SQL> alter index employees_iot_pk rebuild;
alter index employees_iot_pk rebuild
*
ERROR at line 1:
ORA-28650: Primary index on an IOT cannot be rebuilt
```

Converting to or from an Index-Organized Table

You may have an existing table that you want to convert either to an index-organized table or from an index-organized table due to application needs. In either case, the conversion process is straightforward. The simplest method is to use the "create table as select" syntax. The following examples show how to perform these conversions using this method.

First, let's say you have a heap-organized table that meets the fundamental requirements of an index-organized table, and you wish to convert that table to an IOT, as follows:

```
CREATE TABLE locations_iot
 (LOCATION_ID
 ,STREET_ADDRESS
 ,POSTAL_CODE
 ,CITY
 ,STATE_PROVINCE
 ,COUNTRY_ID
 ,CONSTRAINT locations_iot_pk PRIMARY KEY (location_id)
 )
 ORGANIZATION INDEX
 as select * from locations;

Table created.
```

Conversely, if you have an index-organized table that is causing performance problems, and you want to convert to a heap-organized table, you could do as shown in the following example:

```
CREATE TABLE locations
 (LOCATION_ID
 ,STREET_ADDRESS
 ,POSTAL_CODE
 ,CITY
 ,STATE_PROVINCE
 ,COUNTRY_ID
 ,CONSTRAINT locations_pk PRIMARY KEY (location_id)
 )
 as select * from locations_iot;

Table created.
```

After you have completed the operation, you'd then have to drop the original table, of course, and if necessary, rename the newly created table to the desired name.

Based on how an application and its data can change over time, you may need to convert a table from a heap-organized table to an IOT or from an IOT to a heap-organized table. The aforementioned examples are one simple way to do this.

Reporting on Index-Organized Tables

Getting information from the Oracle data dictionary on index-organized tables is straightforward. Look at the following query, which gives the fundamental information regarding the IOTs within your database:

```
SQL> select i.table_name, i.index_name, i.index_type, i.pct_threshold,
  2         nvl(column_name,'NONE') include_column
  3  from user_indexes i left join user_tab_columns c
  4  on (i.table_name = c.table_name)
  5  and (i.include_column = c.column_id)
  6  where index_type = 'IOT - TOP';

TABLE_NAME      INDEX_NAME         INDEX_TYPE PCT_THRESHOLD INCLUDE_COLUMN
--------------- ------------------ ---------- ------------- ------------------
LOCATIONS_IOT   LOCATIONS_IOT_PK   IOT - TOP             50 NONE
EMPLOYEES_PART  EMPLOYEES_PART_PK  IOT - TOP             50 NONE
COUNTRIES       COUNTRY_C_ID_PK    IOT - TOP             50 NONE
EMPLOYEES_IOT   EMPLOYEES_IOT_PK   IOT - TOP             40 SALARY
```

From this query, you get the following information:

- The table name
- The index name(s), which includes the primary key and any secondary indexes on the table
- The index type, which will be designated as `'IOT - TOP'` for index-organized tables
- The `PCTTHRESHOLD` for the table
- The `INCLUDING` column, if specified

You need to do an outer join to the USER_TAB_COLUMNS view to get the column name for the column specified by the INCLUDING clause, which is optional when creating an index-organized table. The COLUMN_ID column on the USER_INDEXES view specifies the column number of the column for the INCLUDING clause. If there is no INCLUDING clause specified on the index-organized table, the COLUMN_ID column will be populated with a default value of '0' or with the value from the USER_TAB_COLUMNS COLUMN_ID column.

If you look at the USER_TABLES view, both the IOT itself and the overflow segment are shown.

```
SQL> select table_name, iot_type, segment_created from user_tables;

TABLE_NAME                     IOT_TYPE     SEG
------------------------------ ------------ ---
SYS_IOT_OVER_77704             IOT_OVERFLOW YES
EMPLOYEES_IOT                  IOT          YES
```

If querying DBA_SEGMENTS to get the actual physical characteristics of the IOT itself, as well as the overflow segment, remember to use the primary key segment_name; the table name itself will not be specified within the DBA_SEGMENTS view, since the IOT is essentially an index segment.

```
  1  select segment_name, segment_type
  2  from dba_segments
  3* where segment_name like '%IOT%'
SQL> /

SEGMENT_NAME                   SEGMENT_TYPE
------------------------------ ------------------
SYS_IOT_OVER_77704             TABLE
EMPLOYEES_IOT_PK               INDEX
```

Summary

Index-organized tables have a specific niche in applications and are not really suitable for extensive use. The following are some guidelines to determine if a table is a good candidate for an IOT:

- Is it a table with a small number of columns?
- Is it a table made up of a composite primary key (several columns of the table)?
- Does the table require fast primary key access?
- Is the data in the table fairly static—with little DML activity?

IOTs are generally better suited for OLTP applications than data warehouse applications, simply because OLTP applications often have a requirement for very fast lookup of primary key data. IOTs are generally avoided in the data warehouse simply because a data warehouse typically does bulk loading of data, and the performance of inserts on IOTs is slower. This is especially noticeable with a large volume of data. Also, if there are access requirements to place many secondary indexes on the IOT, it can generally be slower just because no physical ROWID exists within an IOT. This can slow the access of data, especially over time, as physical guesses become stale. Of course, these are guidelines. Deciding whether to use IOTs within your application depends on your specific data loading and data retrieval requirements. You should also consider your available maintenance windows, which can be used in part to rebuild IOTs when they become fragmented.

All this said, the index-organized table is a valuable tool. Knowing the features, advantages, and disadvantages can help you decide where and when to properly implement an index-organized table within your application.