Rdb Journal
technical article

Guide to Database Performance and Tuning: Bitmapped Scan

A feature of Oracle Rdb

By Mark Bradley
Oracle Rdb Relational Technology Group
Oracle Corporation

Table of Contents

BACKGROUND **3**

THE DYNAMIC OPTIMIZER **3**

Example 1: 3
INDEX ESTIMATION **4**
Example 2: 5
DYNAMIC COMPONENTS **5**
BACKGROUND ONLY TACTIC 5
FINAL PHASE (FIN) 6
FAST FIRST TACTIC 6
SORTED TACTIC 6
INDEX ONLY TACTIC 7

DBKEY LISTS **7**

LIMITATIONS OF DBKEY LISTS **8**

BBC SEGMENTS **9**

BITMAP SCANS **10**

BITMAPPED OR **11**
Example 3: 11
Example 4: 12
Example 5: 13
BITMAPPED AND **14**
Example 6: 14
Example 7: 16

Example 8:....16
BITMAPPED AND OR....18
Example 9:....19
Example 10:....21
Example 11:....22
Example 12:....23

FAST FIRST BITMAP SCANS....23

FAST FIRST EXAMPLE....24
Example 13:....25
BENEFITS IN FAST FIRST....26

EMPLOYING BITMAPPED SCANS....27

INTERACTIVE SQL....27
Example 14:....28
LOGICAL NAME....29
Example 15:....29
Example 16:....30
Example 17:....31
DYNAMIC SQL....31
Example 18:....32
Example 19:....33

BITMAPPPED SCAN COMPARISON....34

GLOSSARY....35

Background

For each table in a query, the Rdb optimizer must choose which indexes on that table will be used to retrieve the data. In some cases the choice of indexes is obvious but in other cases there may be several indexes that could be useful for retrieving the data.

When multiple indexes are read for the same table, Rdb needs an efficient mechanism to combine the results of each separate index scan.

This paper describes the various mechanisms used to combine index scans, and introduces a new mechanism termed bitmapped scan. Although Bitmapped Scab was introduced in Rdb V7.1, the functionality described by this paper is only available in Rdb V7.1.2 and later releases.

This paper assumes the reader is familiar with the structure of indexes in Rdb. For further information about indexes please refer to the Rdb documentation, and to the ***Rdb Internals and Data Structures manual*** from the Rdb Internals course.

The Dynamic Optimizer

For complicated queries there may not be any single index that provides the best query performance. When the optimizer detects that multiple indexes could be useful for retrieving data from a given table, it can record that list of indexes so that each time the query executes, Rdb can ensure the most efficient indexes are used.

Consider the SQL cursor in example 1.

Example 1:

```
SQL> DECLARE cur1 CURSOR FOR
cont>  SELECT * FROM employees
cont>  WHERE first name=:var1 and last name=:var2;
```

If there were one index on **first_name** and a second index on **last_name**, then Rdb would have to decide which of the two indexes would be most efficient for retrieving the data.

The actual values of var1 and var2 are likely to be different each time the cursor is opened. Therefore, the best index to use may be different each time the cursor is opened. Rdb attempts to scan the most efficient index first.

Suppose the index on **last_name** is scanned and a list constructed of all rows matching the condition "**last_name** = :var2".

Rdb must now decide if it is productive to scan the **first_name** index for rows that matched the condition "**first_name** = :var1". It may be more efficient to fetch those rows in the list and check the value of **first_name** in the row, rather than scan another index. This could be the case if there were only a handful of rows with the specified **last_name**, but many thousands of rows with the required **first_name**.

If Rdb decided to scan the first name index it must combine the results of that scan with the results of the previous scan on the **last_name** index. Logically, only those rows appearing in both lists would be useful for the query.

Rdb uses the ***Dynamic Optimizer*** to adjust the indexes used each time a database query is executed, and to combine the results of various index scans.

If the choice of index is obvious, the dynamic optimizer is not used, and the retrieval strategy is fixed to use the appropriate index. In this case the indexes used do not change between executions of the query, and the retrieval strategy is termed a ***static*** retrieval strategy.

The remainder of the paper describes queries that use the dynamic optimizer. Such queries are said to have a ***dynamic*** retrieval strategy.

Index Estimation

Each time a query with a dynamic retrieval strategy is executed[1], Rdb must decide which index to scan first. An estimate is made the number of rows each index is likely to return (termed ***Index Estimation***), and then the indexes are sorted by the expected number of rows returned.

Example 2 shows a sequence of commands where index estimation is performed.

[1] For example, each time a cursor is opened.

Example 2:

```
SQL> set flags 'strategy,execution'
SQL> declare cur1 cursor for
cont> select * from employees
cont> where first name=:var1 and last name=:var2;
SQL> begin
cont> set :var1='Alvin';
cont> set :var2='Toliver';
cont> end;
SQL> open cur1;
~S#0007
Leaf#01 FFirst EMPLOYEES Card=100
  BgrNdx1 EMP_LAST_NAME [1:1] Fan=12
  BgrNdx2 EMP FIRST NAME [1:1] Fan=14
~E#0007.01(1) Estim   Index/Estimate 1/1 2/1
```

Index estimation examines the index to see how many rows are likely to have the value requested, for example how many rows in the **last_name** index have the value 'Toliver'.

The last line in example 2 shows the estimation summary. Both indexes have been examined and both have been estimated to return 1 row. Since the number of rows expected from both indexes is the same, the order of the indexes has not changed.

For a more complete description of index estimation, please refer to the Rdb Journal Article describing ***Guide to Database Performance and Tuning: Predicate Estimation***.

Dynamic Components

When a dynamic retrieval strategy is chosen for a query, several different tactics can be employed.

Background Only Tactic

A background only tactic is intended to deliver all the selected rows as quickly as possible.

The first index is scanned, and a list of dbkeys constructed for rows that match the selection criteria for that index. In example 1 we may scan the index on last name to build a list of dbkeys for rows with a **last_name** value of 'Toliver'. Rdb will then decide if it is productive to scan further indexes.

If a second index is scanned, only dbkeys that appear in both indexes are retained. For example, the **first_name** index could be scanned to obtain a list of dbkeys for rows that have the **first_name** value of 'Alvin'.

A dynamic strategy may use more than two indexes, in which case Rdb would then have to decide if scanning a third index is considered productive, and so on. When Rdb is finished scanning indexes, the resulting dbkey list is passed to the final (Fin) phase.

Final Phase (Fin)

The Fin phase sorts the dbkey list from background and reads each data row. These rows will be delivered if all conditions in the query apply.

Fast First Tactic

A fast first tactic is intended to deliver the first few rows as quickly as possible.

As each dbkey is retrieved from the first background index, the row is fetched, and if it matches all the conditions of the query it is delivered.

When 1024 rows have been delivered, this tactic reverts to a background only tactic for the remaining rows.

Sorted Tactic

A sorted tactic can only be used where there is an index that provides the required sorted order for the query. A particular sorted order may be needed because of an ORDER BY or a GROUP BY clause in the query, or because the table is being joined with another table using a matching style join.

The index that provides the required sorted order is always scanned to completion to ensure the correct sorted order. The other indexes are scanned from least cost to most cost to try to construct a dbkey list.

As each dbkey is fetched from the sorted order index, if a dbkey list is available, the row is only fetched if it is in that list. In this way fewer rows need to be read.

Index Only Tactic

An index only tactic may be used where an index exists that contains all the columns needed for the query. This tactic is a competition between the index only index, and the scan of other background indexes. The index only index is scanned, and at the same time the background indexes are scanned from least cost to most cost.

If the background dbkey list reaches 1024 dbkeys, then background is abandoned, and the index only index is used to deliver data. If the background index scan completes, then the index only index is abandoned, and normal background only is used.

Dbkey Lists

During execution of dynamic querys, dbkey lists are constructed. These dbkey lists take several forms.

When a query begins executing, an initial buffer called the ***tiny*** buffer is used. This tiny buffer is only large enough to hold 20 dbkeys.

If the tiny buffer fills up, a regular sized ***exe*** buffer is used. An exe buffer is only allocated when needed, and is large enough to hold 1024 dbkeys.

If the exe buffer fills up, then the dynamic optimizer will use ***dbkey bitmaps*** and ***temp tables***.

A temp table is really a temporary disk file that is used to store the actual dbkeys. When a temp table is used, the actual dbkeys are not stored in memory. Instead a hash bitmap is used to record the dbkeys in memory.

The hash bitmap is sized based on the table cardinality (the number of rows in the table), with a maximum of 1,000,000,000 bits or 125,000,000 bytes.

As each dbkey is read from the index it is written to the disk file (actually in batches of 1024 dbkeys), and also hashed into the bitmap and the corresponding bit is set.

Limitations of dbkey Lists

Dbkey lists work well for index scans that return a limited number of rows. They are CPU efficient and simple to create and maintain. However, once a regular exe buffer becomes full, and overflows to a temp table, execution starts incurring addition I/O to the temp table to write, then to sort the temp table, then again to read the temp table.

If you have several indexes to scan for the table, and each of them overflows the exe buffer, you would cause the following sequence of events:

- The first index is scanned, and all the dbkeys are written to a temp table on disk.
- All dbkeys are also hashed into an in memory bitmap.
- The second index is scanned, retaining only those dbkeys that hash to set bits in the first bitmap.
- A second temp table is created, and all retained dbkeys from the second index scan are written to disk.
- If the second index scan completes, the first temp table and bitmap are discarded.
- If the second index scan fails to complete, for example where the index is returning too many dbkeys, the second temp table and bitmap are discarded. Rdb calculates this *threshold limit* based on the number of dbkeys found in previous index scans.
- This process continues for any remaining indexes.
- Once all indexes have been scanned the resulting bitmap is discarded.
- The resulting temp table is sorted using VMS sort.
- The sorted dbkey list is read, each row is fetched, and if it matches all conditions in the query it is delivered.

Apart from the additional I/O necessary to manage the temp tables, there are additional limitations that become more apparent for very large tables.

The size of the bitmap created is based on the table cardinality. So for a table with a cardinality of around 1,000,000,000 or higher the maximum size bitmap will be allocated, even if the index scan only returns a few thousand dbkeys.

The hash algorithm is not perfect. You can have two different dbkeys that hash to the same bit which becomes more likely as the bitmap fills up.

This is the reason that the dbkeys must also be written to a temp table. There is no way to reconstruct the dbkeys from the in memory bitmap.

If an index scan returns a large percentage of the dbkeys in a table, a large proportion of the bits in the bitmap will be set. This means that on scanning a second index, testing for set bits in the first bitmap is likely to produce a lot of false positive results.

This may mean that the second dbkey list is much larger than it should be because of retained false positives.

BBC Segments

Within an index created using the clause **type is sorted ranked**[2], a key value with duplicates records the dbkeys for a given key value in ***Byte Aligned, Bitmap Compressed*** (BBC) segments.

Dbkeys represent the logical address in the database of the corresponding row. A dbkey is an eight byte structure consisting of:

- The logical area number, a 16 bit integer.
- The database page number, a 32 bit integer.
- The line number on the page, a 16 bit integer.

BBC segments use a patented compression algorithm to store a sequence of dbkeys without loss.

In a sorted list of dbkeys, the first dbkey in the list can be treated as a pure integer. That integer can then be rounded down to a multiple of eight and called the base dbkey. Each dbkey in the list can then be treated as a pure integer and represented as an offset upwards from the base dbkey. A

[2] Simply referred to as a *ranked index*.

virtual bitmap can then be constructed where the ordinal position in the bitmap represents the offset of each dbkey from the base dbkey.

The virtual bitmap can be further compressed by encoding the bitmap as a series of gaps and maps. A gap representing a sequence of all clear or all set bits, much like how a compression byte in a record represents repeating bytes. A map is a simple bitmap for a portion of the virtual bitmap.

Where there is a large gap in the dbkey sequence such segments can be chained together with each segment having a different base dbkey.

Bitmap Scans

In Rdb version 7.1.2, the dynamic optimizer has been enhanced to allow BBC technology to be used to store dbkey lists.

When used in a ranked index, multiple BBC segments are chained together to create virtual bitmaps larger than an index node. Chaining of BBC segments within an index node takes place only where large gaps exist in the dbkey list causing a jump in the base dbkey.

When used in memory by the dynamic optimizer, BBC segments are stored in a binary tree. This is done to allow arbitrary expansion and contraction of the virtual bitmap by adding, removing, and updating individual BBC segments without affecting the entire virtual bitmap.

As an alternative to other dbkey lists, temp tables and hash based dbkey bitmaps, BBC bitmaps have several advantages:

- They are much more compact than simple dbkey lists.
- They expand and contract based on the actual space used rather than being arbitrarily allocated.
- They are precise, and do not suffer from hash collisions or false positives when tested.
- There is no need for a secondary temporary file.
- They are inherently sorted, and do not require VMS sort.

The use of bitmap scan is enabled by using one of the following methods:

- Defining the RDMS$ENABLE_BITMAPPED_SCAN logical name,

- Using the interactive or dynamic SQL "SET FLAGS" statement with an option of BITMAPPED_SCAN,

The RDMS$SET_FLAGS logical name with the value BITMAPPED_SCAN cannot be used to enable bitmap scan as the presence or absence of the RDMS$ENABLE_BITMAP_SCAN logical name will override that setting.

One of the greatest advantages to bitmap scanning is when the input dbkey lists read from the indexes are already BBC bitmaps. In this case they can be copied directly rather than constructing them one dbkey at a time.

Bitmapped Or

In addition to the ability to use bitmaps to AND conditions together in the dynamic optimizer, bitmapped scan allows a list of OR conditions on different indexes to be performed as a dynamic optimizer component.

Example 3 shows a query with a more complex predicate.

Example 3:

```
SQL> select *
cont> from cars
cont> where cmake = 'honda'
cont> and (ctype = 'wagon' or ccolour = 'yellow');
 CMAKE        CTYPE        CCOLOUR      CPLATE           CYEAR
 honda        wagon        black        ABC-119           1998
 honda        sedan        yellow       ABC-114           1990
2 rows selected
```

In example 3 we have indexes on the three columns used in the query **cmake**, **ctype**, and **ccolour**.

Theoretically Rdb could perform the following actions[3]:

[3] Versions of Rdb prior to 7.1.2, the optimizer was unable to perform this sequence of operations.

- The **cmake** index can be used to build a list of all cars made by ‘honda’.
- The **ctype** index can be used to build a list of all cars of type ‘wagon’.
- The **ccolour** index can be used to build a list of colour ‘yellow’.
- The results from the type, and colour scans can be logically ORed together to produce a list of all cars that are ‘yellow’ or a ‘wagon’.
- The resulting list can then be ANDed with the make list to produce a final dbkey list.

In Rdb 7.1.2, using bitmapped scan, the optimizer does a better job of building OR index lists. In addition, the optimizer will combine an OR index list with a list of AND indexes to produce a form of super dynamic retrieval strategy that is a combination of AND and OR indexes.

Even in the absence of AND indexes, the bitmapped scan feature will better recognize and utilize a list of OR indexes, even where the dynamic optimizer is not used. In this case true BBC bitmaps are not used for the OR index list.

Only where there are additional AND index scans, and the dynamic optimizer is used, do we currently use BBC bitmaps for dbkey lists. In the future the use of BBC bitmaps will be extended to support OR index lists in static retrieval strategies such as that shown in examples 4 and 5.

Example 4 shows the strategy for a purely OR based query without bitmap scan enabled. Example 5 shows the same query with bitmap scan enabled.

Example 4:

```
SQL> set flags 'strategy,detail(1)'
SQL> select *
cont> from cars
cont> where ctype = 'wagon'
cont> or ccolour = 'yellow' or cmake='honda';
Tables:
  0 = CARS
Conjunct: (0.CTYPE = 'wagon') OR (0.CCOLOUR = 'yellow') OR
(0.CMAKE = 'honda')
Get     Retrieval by index of relation 0:CARS
  Index name  YEARI [0:0]
 CMAKE        CTYPE        CCOLOUR      CPLATE           CYEAR
 toyota       wagon        black        ABC-111           1990
```

```
 honda          sedan          yellow         ABC-114              1990
 toyota         sedan          yellow         ABC-115              1990
 honda          sedan          blue           ABC-122              1992
 toyota         wagon          yellow         ABC-116              1996
 honda          wagon          black          ABC-119              1998
6 rows selected
```

Notice in example 4 that a full index scan (denoted by [0:0]) is performed to fetch all rows from the table. A conjunct operation is then performed on each row to test the set of conditions in the query.

Example 5:

```
SQL> set flags 'bitmapped_scan'
SQL> select *
cont> from cars
cont> where ctype = 'wagon' or ccolour = 'yellow' or
cmake='honda';
Tables:
  0 = CARS
OR index retrieval
  Conjunct: (0.CTYPE = 'wagon') OR (0.CCOLOUR = 'yellow')
  OR index retrieval
    Get     Retrieval by index of relation 0:CARS
      Index name  TYPEI [1:1]
        Keys: 0.CTYPE = 'wagon'
    Conjunct: NOT (0.CTYPE = 'wagon')
    Get     Retrieval by index of relation 0:CARS
      Index name  COLOURI [1:1]
        Keys: 0.CCOLOUR = 'yellow'
  Conjunct: NOT ((0.CTYPE = 'wagon') OR (0.CCOLOUR = 'yellow'))
  Get     Retrieval by index of relation 0:CARS
    Index name  MAKEI [1:1]
      Keys: 0.CMAKE = 'honda'
 CMAKE          CTYPE          CCOLOUR        CPLATE               CYEAR
```

```
 toyota         wagon          black          ABC-111              1990
 toyota         wagon          yellow         ABC-116              1996
 honda          wagon          black          ABC-119              1998
 honda          sedan          yellow         ABC-114              1990
 toyota         sedan          yellow         ABC-115              1990
 honda          sedan          blue           ABC-122              1992
6 rows selected
```

In contrast, example 5 shows that the bitmapped scan feature enabled the optimizer to detect and use the indexes specific for each branch of the OR query. In neither case is an actual BBC bitmap used to store the dbkey list, but what is important is the improved use of indexes to resolve the query. It should be noted that the strategy shown in example 5 may be chosen even without enabling bitmapped scan.

Rdb version 7.1.3 will support the use of dynamic tactics and bitmap scan functionality for simple OR queries without the need for an AND index. Please refer to the 7.1.3 release notes for further information.

Bitmapped And

A query that specifies conditions ANDed together that can be resolved from different indexes has traditionally been the domain of the dynamic optimizer. Rdb 7.1.2 supports the use of BBC bitmaps for bitmap scanning in the dynamic optimizer where the dynamic tactic is fast first or background only.

At present a bitmap scan will not be selected where the dynamic tactic is index only or sorted order. The complexity of handling the new BBC segments together with the special handling for sorted and index only tactics prevents use of bitmapped scan at this time. It is possible that this restriction will be removed in the future.

A simple query that invokes the dynamic optimizer is shown in example 6.

Example 6:

```
SQL> set flags 'strategy,execution'
SQL> select last_name, sex, dbkey
```

```
cont> from employees
cont> where last name = 'Smith' and sex = 'M'
cont> order by dbkey;
~S#0005
Sort
Leaf#01 BgrOnly EMPLOYEES Card=100
  BgrNdx1 EMP LAST NAME [1:1] Fan=12
  BgrNdx2 RSEX [1:1] Fan=19
~E#0005.01(1) Estim   Index/Estimate 1/1 2/65
~E#0005.01(1) BgrNdx1 EofData  DBKeys=2  Fetches=0+0  RecsOut=0 #Bufs=2
~E#0005.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0005.01(1) Fin     Buf      DBKeys=2  Fetches=0+0  RecsOut=2
 LAST NAME        SEX                    DBKEY
 Smith            M                     78:2:1
 Smith            M                    79:51:1
2 rows selected
```

In this case the bitmap scan feature was not enabled, so the regular dbkey handling mechanisms were employed.

The estimation trace line shows that the first background index, on **last_name**, has been estimated at finding one dbkey, while the second background index, on **sex**, has been estimated at 65 dbkeys. So the execution index order will remain unchanged.

The execution trace for background index 1 shows that it reached the end of it's index scan (EofData) after reading 2 dbkeys from the index.

The execution trace for background index 2 shows it had reached threshold limit, the point where further effort to scan the second index was considered non-productive.

The 'Fin' or final phase uses the 'Buf' or in memory dbkey buffer of dbkeys to deliver the rows requested.

Example 7 shows the same query when bitmap scanning is enabled.

Example 7:

```
SQL> set flags 'strategy,execution'
SQL> select last name, sex, dbkey
cont> from employees
cont> where last name = 'Smith' and sex = 'M'
cont> order by dbkey;
~S#0006
Sort
Leaf#01 BgrOnly EMPLOYEES Card=100       Bitmapped scan
  BgrNdx1 EMP_LAST_NAME [1:1] Fan=12
  BgrNdx2 RSEX [1:1] Fan=19
~E#0006.01(1) Estim   Index/Estimate 1/1 2/65
~E#0006.01(1) BgrNdx1 EofData  DBKeys=2  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0006.01(1) BgrNdx2 EofData  DBKeys=65  Fetches=0+0  RecsOut=0 #Bufs=2
~E#0006.01(1) Fin     Bitmap   DBKeys=2  Fetches=0+0  RecsOut=2
 LAST_NAME        SEX                      DBKEY
 Smith            M                       78:2:1
 Smith            M                      79:51:1
2 rows selected
```

Notice that Rdb has been more aggressive in pursuing the second background index. This time the second background index has been scanned to completion (EofData).

For a lot of queries the estimation phase, which samples into the index, will cause the index nodes to be cached (in local buffers, global buffers, or row caches), so collecting the dbkeys from a ranked index where we can expect few I/O's to be expended can be I/O efficient.

In this case, the number of dbkeys has not been reduced by the second index scan. Both of the dbkeys with **last_name** = 'Smith' happen to have **sex** = 'M'. Specifying the DETAIL(1) debug flag we can see more of what was done for this query. Example 8 is the same as example 7 with the DETAIL(1) debug flag.

Example 8:

```
SQL> set flags 'strat,exec,bitmap,detail(1)'
SQL> select last name, sex, dbkey
cont> from employees
```

```
cont> where last name = 'Smith' and sex = 'M'
cont> order by dbkey;
~S#0007
Tables:
  0 = EMPLOYEES
Sort: 0.DBKEY(a)
Leaf#01 BgrOnly 0:EMPLOYEES Card=100 Bitmapped scan
  Bool: (0.LAST NAME = 'Smith') AND (0.SEX = 'M')
  BgrNdx1 EMP LAST NAME [1:1] Fan=12
    Keys: 0.LAST NAME = 'Smith'
  BgrNdx2 RSEX [1:1] Fan=19
    Keys: 0.SEX = 'M'
~Estim  EMP LAST NAME Sorted: Split lev=1, Seps=1 Est=1
~Estim  RSEX Ranked: Nodes=1, Min=65, Est=65 Precise IO=0
~Estim  RLEAF Cardinality=  1.0000000E+02
~E#0007.01(1) Estim   Index/Estimate 1/1 2/65
~E#0007.01(1) BgrNdx1 FillMap2  DBKeys=1 Fetches=0+0
~E#0007.01(1) BgrNdx1 FillMap2  DBKeys=1 Fetches=0+0
~E#0007.01(1) BgrNdx1 Or  Map2  DBKeys=2 Fetches=0+0
~E#0007.01(1) BgrNdx1 EofData  DBKeys=2  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0007.01(1) BgrNdx2 FillMap2  DBKeys=65 Fetches=0+0
~E#0007.01(1) BgrNdx2 EofData  DBKeys=65  Fetches=0+0  RecsOut=0 #Bufs=2
~E#0007.01(1) BgrNdx2 And Map1  DBKeys=2 Fetches=0+0
~E#0007.01(1) Fin     Bitmap   DBKeys=2  Fetches=0+0  RecsOut=2
 LAST NAME        SEX                     DBKEY
 Smith            M                      78:2:1
 Smith            M                     79:51:1
2 rows selected
```

To understand the extended output provided by the DETAIL(1) flag you have to understand a little about how the dynamic optimizer reads an index.

In a sorted ranked index an entire BBC duplicates chain is processed in one step. The BBC duplicates chain is simply copied into an in memory BBC binary tree. However, when the index being scanned is unique, or a non-ranked index (hashed, or sorted), there is no BBC duplicates chain. So in this case dbkeys are fetched from the index one at a time.

In this example, background index 1 is an index on **last_name** and is type is sorted (non-ranked). So dbkeys are fetched one at a time. Each time a dbkey or duplicates chain is fetched, you will see a "FillMap" line. This indicates that a BBC binary tree has been created in memory for that dbkey or BBC duplicates chain.

If a prior read of the index has created an existing BBC tree in memory, the new tree will have to be logically ORed with the existing tree.

So in this example you see that:

- Background index 1 has filled a map with 1 dbkey. This is because the index is not ranked (i.e. just type is sorted) and dbkeys are only fetched one at a time.
- Background index 1 then fills a second bitmap with the next dbkey from the index.
- The two bitmaps are then logically ORed together to produce a bitmap of two dbkeys.
- Background index 1 then reaches the end of it's scan (EofData) and processing can move on to the next index.

In this case, background index 2 is a ranked index on **sex**. So all the dbkeys for **sex** = 'M' come from a single BBC duplicates chain. So the first FillMap for index 2 loads 65 dbkeys into an in memory bitmap and immediately reaches EofData.

Because background index 2 is the second AND background index. The bitmap from index 1 and the bitmap from index 2 have to be logically ANDed together. This produces a dbkey bitmap of 2 dbkeys.

The final (Fin) stage uses the resulting bitmap to deliver the rows for the query.

It may seem strange to create bitmaps with only one dbkey, but bitmap scanning is optimized for processing large BBC duplicates chains on indexes of type is sorted ranked. Any additional overhead on small queries may not be noticeable unless the query is executed many times over.

Bitmapped And Or

When a combination of AND and OR conditions are applied to various indexed fields, Rdb will attempt to perform a bitmap AND/OR operation.

For a regular dynamic strategy, Rdb can combine a list of up to 7 background AND index scans. The indexes selected for sorted and index only tactics are not count towards this limit.

If the query is such that a list of OR indexes are available, that list of OR indexes is treated as a single background index. For version 7.1.2, Rdb supports only one such OR index list in the dynamic tactic.

Example 9 shows a simple query that uses a combination of AND and OR operations.

Example 9:

```
SQL set flags 'strategy,detail(1),execution,bitmapped_scan'
SQL> select last_name, first_name, middle_initial, state
cont> from employees
cont> where last_name > 'S' and (state='MA' or middle_initial='R')
cont> order by dbkey;
~S#0010
Tables:
  0 = EMPLOYEES
Sort: 0.DBKEY(a)
Conjunct: (0.LAST_NAME > 'S') AND ((0.STATE = 'MA') OR (0.MIDDLE_INITIAL = 'R'))
OR index retrieval
  Leaf#01 BgrOnly 0:EMPLOYEES Card=100     Bitmapped scan
    BgrNdx1 RSTATE [1:1] Fan=19
      Keys: 0.STATE = 'MA'
        OrNdx1 RMIDDLE [1:1] Fan=19
          Keys: 0.MIDDLE_INITIAL = 'R'
    BgrNdx2 RNAME [1:0] Fan=12
      Keys: 0.LAST NAME > 'S'
~Estim  RSTATE Ranked: Nodes=1, Min=9, Est=9 Precise IO=0
~Estim  RMIDDLE Ranked: Nodes=1, Min=3, Est=3 Precise IO=0
~Estim  RLEAF Cardinality=  1.0000000E+02
~Estim  RNAME Ranked: Nodes=2, Min=0, Est=10 IO=0
~Estim  RLEAF Cardinality=  1.0000000E+02
~E#0010.01(1) Estim   Index/Estimate 1/9 2/10
~E#0010.01(1) BgrNdx1 FillMap2  DBKeys=9 Fetches=0+0
~E#0010.01(1) BgrNdx1 EofData  DBKeys=9  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0010.01(1) Or Ndx1 FillMap2  DBKeys=3 Fetches=0+0
~E#0010.01(1) Or Ndx1 EofData  DBKeys=3  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0010.01(1) Or Ndx1 Or__Map2  DBKeys=12 Fetches=0+0
```

```
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=3 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=5 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=6 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=8 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=9 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=11 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=12 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=13 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=14 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=15 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=16 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=17 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=19 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=20 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=21 Fetches=0+0
~E#0010.01(1) BgrNdx2 EofData  DBKeys=21  Fetches=0+0  RecsOut=0 #Bufs=11
~E#0010.01(1) BgrNdx2 And Map1  DBKeys=3 Fetches=0+0
~E#0010.01(1) Fin     Bitmap   DBKeys=3  Fetches=0+0  RecsOut=3
 LAST NAME        FIRST NAME   MIDDLE INITIAL   STATE
 Siciliano        George       NULL             MA
 Smith            Roger        R                NH
 Sullivan         Len          R                NH
```

```
3 rows selected
```

The first background index is on the column **state**. It is selecting from the state index all rows with **state** = 'MA'. This index also has an OR index - Or index 1 is on **middle_initial** that is selecting all rows with **middle_initial** = 'R'. The two index scans must both complete to make a complete index scan.

The second background index is on **last_name**, and is used to locate all rows with **last_name** > 'S'.

Let us consider the execution trace for the first background index, which is really a list of two OR indexes. Example 10 shows this part of the trace from example 9.

Example 10:

```
~E#0010.01(1) BgrNdx1 FillMap2  DBKeys=9 Fetches=0+0
~E#0010.01(1) BgrNdx1 EofData  DBKeys=9  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0010.01(1) Or Ndx1 FillMap2  DBKeys=3 Fetches=0+0
~E#0010.01(1) Or Ndx1 EofData  DBKeys=3  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0010.01(1) Or Ndx1 Or  Map2  DBKeys=12 Fetches=0+0
```

Lets look at the sequence of events during this portion of execution.

- BgrNdx1 FillMap2 DBKeys=9 – This indicates that the first BBC bitmap has been read from background index 1, the index on state. Nine dbkeys were found with state='MA'. Because the index is ranked, the BBC bitmap was simply copied to an in memory BBC tree.
- BgrNdx1 EofData DBKeys=9 – This shows that the index scan had completed after accumulating 9 dbkeys.
- Or Ndx1 FillMap2 DBKeys=3 – Here OR index number 1 is being scanned. This is the index on **middle_initial**. The BBC duplicates chain was copied into an in memory BBC tree and contains 3 dbkeys.
- Or Ndx1 EofData DBKeys=3 – The scan of OR index 1 has completed, and accumulated 3 dbkeys.
- Or Ndx1 Or__Map2 DBKeys=12 – The BBC tree from background index 1 has been ORed with the BBC tree from OR index 1 to create a BBC tree containing 12 dbkeys. This represents the dbkeys for all rows with **state** = 'MA' or **middle_initial** = 'R'.

This represents a completed index scan to the dynamic optimizer. An index scan must be complete to be useful for retrieving data. A partial index scan will be discarded.

During query execution, the dynamic optimizer monitors I/O consumption, and places limits on how many I/O's will be expended scanning an index. In this way it is possible that not all available indexes will actually be used during query execution.

If the background index is an OR index list, as in this case, all OR index scans must complete to have a valid dbkey list.

Example 11 shows the next portion of the execution trace, which is the start of the scan of background index 2.

Example 11:

```
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or__Map2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=1 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=3 Fetches=0+0
~E#0010.01(1) BgrNdx2 FillMap2  DBKeys=2 Fetches=0+0
~E#0010.01(1) BgrNdx2 Or  Map2  DBKeys=5 Fetches=0+0
```

Even though the index here is a ranked index, the index is on the **last_name** field and is fairly unique. So only small bitmaps are being manipulated.

- BgrNdx2 FillMap2 DBKeys=1 – The index scan has returned one dbkey. So an in memory BBC is created for that dbkey.
- BgrNdx2 FillMap2 DBKeys=1 – The second fetch from the index again returned 1 dbkey. This dbkey is added to a second BBC tree.
- BgrNdx2 Or__Map2 DBKeys=2 – The two BBC trees are ORed together to produce a BBC tree with 2 dbkeys.
- BgrNdx2 FillMap2 DBKeys=1 – The next fetch from the index again returns 1 dbkey. An in memory BBC is created for that dbkey.
- BgrNdx2 Or__Map2 DBKeys=3 – The new BBC of 1 dbkey is ORed with the old BBC of 2 dbkeys to produce a new BBC of 3 dbkeys.

- BgrNdx2 FillMap2 DBKeys=2 – The next fetch from the index returned two dbkeys, so the BBC segment is copied into an in memory BBC tree.
- BgrNdx2 Or__Map2 DBKeys=5 – The new BBC of 2 dbkeys is ORed with the old BBC of 3 dbkeys, producing a BBC of 5 dbkeys.

This process continues. For each new unique key, the BBC is ORed with the existing BBC. In example 9 this can be seen to continue until the scan of background index 2 completes. Example 12 shows the execution trace from that point.

Example 12:

```
~E#0010.01(1) BgrNdx2 EofData  DBKeys=21  Fetches=0+0  RecsOut=0 #Bufs=11
~E#0010.01(1) BgrNdx2 And Map1  DBKeys=3 Fetches=0+0
~E#0010.01(1) Fin     Bitmap   DBKeys=3  Fetches=0+0  RecsOut=3
```

Remember that we have already scanned background index 1, which was really a list of OR indexes. We have a BBC tree for that scan.

At this point we have just completed the scan of background index 2. The execution trace shows:

- BgrNdx2 EofData DBKeys=21 – Background index 2 has reached the end of it's scan and has accumulated 21 dbkeys.
- BgrNdx2 And_Map1 DBKeys=3 – The BBC tree from background index 1, which contained 12 dbkeys, is ANDed with the BBC tree from background index 2, containing 21 dbkeys, producing a BBC tree containing 3 dbkeys.
- Fin Bitmap DBKeys=3 – The final phase uses the in memory BBC to fetch and deliver the 3 rows.

Fast First Bitmap Scans

While the aim of a bitmap scan is to efficiently deliver the entire selected row set by using as many productive indexes as possible, Rdb still supports the use of bitmap scans when the first few rows need to be delivered quickly.

The dynamic optimizer by default tries to deliver the first 1,024 rows as quickly as possible. Where it can, it uses the Fast First dynamic tactic to do this.

There are cases however where this is not possible. For example, if a physical sort must be done, or if some other part of the retrieval strategy depends on the data from the table in question. In these cases Rdb will automatically select a different dynamic tactic, such as the background only tactic.

The user can also bias the optimizer, using the "OPTIMIZE FOR FAST FIRST" or "OPTIMZE FOR TOTAL TIME" clause in an SQL statement, or using the /OPTIMIZATION_LEVEL = FAST_FIRST or /OPTIMIZATION_LEVEL = TOTAL_TIME compiler qualifiers.

When a fast first tactic is chosen, the dynamic optimizer will try to deliver the first 1,024 rows as quickly as possible. To achieve this, it scans the background indexes from least cost to most cost, and as each new dbkey is retrieved from the index, the row is immediately fetched, and if it passes all the conditions of the query it will be immediately delivered.

Once 1,024 rows have been delivered, the dynamic optimizer assumes that the entire selection is going to be retrieved, and reverts to the same behavior as a background only tactic.

If the bitmapped scan feature is enabled, the dbkeys will be stored in an in memory bitmap. There is no significant difference, during the fast first phase, between using a regular dbkey list and an in memory BBC tree. If, however, 1,024 dbkeys have been read, and the strategy reverts to a background only tactic, bitmapped scanning as described above will be employed.

Fast First Example

When a fast first tactic is used, dbkeys are read from the index scan one at a time, regardless of the index type or the presence or absence of duplicates.

As each dbkey is retrieved it is recorded in an in memory BBC tree.

Because multiple indexes may be scanned, and since different indexes may return the same dbkey, as each dbkey is retrieved, the existing in memory BBC tree is checked for the presence of that dbkey. If the dbkey is present, then it is known that the associated row has already been delivered, so the next dbkey is fetched from the index scan.

If the dbkey is not present, then the row will be fetched, and if all conditions specified in the query are satisfied, the row will be delivered and the dbkey added to the in memory BBC tree.

Once 1,024 dbkeys are fetched, execution reverts to the background only tactic, and bitmap at a time operations will be used. By bitmap at a time it is meant that an entire BBC duplicates chain is used to construct an in memory BBC tree and the resulting tree is ANDed or ORed as approriate. This is much more efficient than single dbkey at a time operations.

Example 13 shows a query similar to that shown in example 9, however in this case the fast first dynamic tactic is employed.

Example 13:

```
SQL> set flags 'strategy,detail(1),execution,bitmapped_scan'
SQL> select last name, first name, middle initial, state
cont> from employees
cont> where last_name > 'S' and (state='MA' or middle_initial='R');
~S#0004
Tables:
  0 = EMPLOYEES
Conjunct: (0.LAST NAME > 'S') AND ((0.STATE = 'MA') OR (0.MIDDLE INITIAL = 'R'))
OR index retrieval
  Leaf#01 FFirst 0:EMPLOYEES Card=100      Bitmapped scan
    BgrNdx1 RSTATE [1:1] Fan=19
      Keys: 0.STATE = 'MA'
        OrNdx1 RMIDDLE [1:1] Fan=19
          Keys: 0.MIDDLE INITIAL = 'R'
    BgrNdx2 RNAME [1:0] Fan=12
      Keys: 0.LAST_NAME > 'S'
~Estim  RSTATE Ranked: Nodes=1, Min=9, Est=9 Precise IO=1
~Estim  RMIDDLE Ranked: Nodes=1, Min=3, Est=3 Precise IO=2
~Estim  RLEAF Cardinality=  1.0000000E+02
~Estim  RNAME Ranked: Nodes=2, Min=0, Est=10 IO=3
~Estim  RLEAF Cardinality=  1.0000000E+02
~E#0004.01(1) Estim   Index/Estimate 1/9 2/10
~E#0004.01(1) BgrNdx1 EofData  DBKeys=9  Fetches=0+0  RecsOut=9 #Bufs=0
 LAST_NAME        FIRST_NAME   MIDDLE_INITIAL   STATE
 Siciliano        George       NULL             MA
 Sullivan         Len          R                NH
~E#0004.01(1) Or Ndx1 EofData  DBKeys=3  Fetches=0+0  RecsOut=12 #Bufs=0
~E#0004.01(1) FgrNdx  FFirst   DBKeys=12  Fetches=0+11  RecsOut=12`ABA
~E#0004.01(1) Fin     Bitmap   DBKeys=12  Fetches=0+0  RecsOut=12
 Smith            Roger        R                NH
```

```
3 rows selected
```

The absence of bitmap execution trace lines is indicative of the absence of bitmap operations themselves.

The execution trace lines in example 13 tell us the sequence of operations:

- BgrNdx1 EofData DBKeys=9 – Background index 1 is scanned, and 9 dbkeys are returned. Each dbkey will have been fetched, and tested, and if all conditions are met, they will have been delivered and inserted into an in memory bitmap.
- Or Ndx1 EofData DBKeys=3 – Or index 1 has been scanned and 3 new dbkeys are added to the in memory bitmap.
- FgrNdx FFirst … ABA – Because a complete background index has been scanned, and fast first has fetched and filtered these rows, fast first is abanodoned.

Benefits In Fast First

While at face value there seems little benefit from employing a BBC bitmap, rather than a regular dbkey list, during a fast first tactic, the use of BBC bitmaps has two major advantages.

First, if more than 1024 dbkeys are delivered, and the strategy reverts to a background only tactic, bitmap at a time operations can be employed.

Second, bitmapped scans understand how to employ OR index retrieval. So the use of bitmaps allows the enhanced use of index AND and index OR combinations.

Employing Bitmapped Scans

The performance benefits, if any, of bitmapped scans over traditional methods will depend a great deal on the nature of the query, the structure of the database and indexes, and the number and size of BBC duplicates chains in the selected areas of the indexes being scanned.

For this reason it is recommended that this feature be employed selectively. It is also recommended that performance testing be undertaken to ensure that the intended performance characteristics are materialized.

Bitmapped scans are not enabled by default. The application developer or database administrator must enable bitmapped scans in the situations where it is thought that they could be beneficial.

Enabling the bitmapped scan feature does not guarantee that bitmapped scanning will be employed. If the optimizer believes that a non-dynamic retrieval strategy is most efficient, or if a sorted order or index only dynamic tactic is considered most efficient, then bitmapped scanning cannot be used. No warning or error is generated for such queries, the bitmapped scan feature is simply not used.

The Rdb debug flag values “strategy” and “execution” can be used to determine if and when an actual bitmapped scan is employed.

The bitmapped scan feature must be enabled at the time a database query is compiled and optimized. It is at the time of query compilation that the optimizer chooses the execution strategy, and hence, whether to use a bitmapped scan.

The following sections demonstrate the various methods that can be employed to enable bitmapped scans.

Interactive SQL

The interactive SQL SET FLAGS command can be used to enable and disable bitmapped scans.

A SET FLAGS ‘BITMAPPED_SCAN’ command must be issued prior to a select statement, or prior to opening a cursor, where you wish to employ a bitmapped scan.

Once a cursor is opened, the bitmapped scan feature can be disabled without changing the strategy for the existing cursor, no matter how many times the cursor is closed and opened.

Any select statement is compiled at the time the statement is entered. So the bitmapped scan feature must be enabled at the time of issuing the select statement.

Example 14 shows the use of the set flags statement with a cursor in interactive SQL.

Example 14:

```
SQL> set flags 'strategy,execution'
SQL> declare c1 cursor for
cont> select count(*)
cont> from employees
cont> where state = 'NH' and last name > 'L';
SQL> set flags 'bitmap'
SQL> open c1;
~S#0003
Aggregate
Leaf#01 BgrOnly EMPLOYEES Card=100       Bitmapped scan
  BgrNdx1 RSTATE [1:1] Fan=19
  BgrNdx2 RNAME [1:0] Fan=12
~E#0003.01(1) Estim   Index/Estimate 2/46 1/90
~E#0003.01(1) BgrNdx2 EofData  DBKeys=49  Fetches=1+1  RecsOut=0 #Bufs=0
~E#0003.01(1) BgrNdx1 EofData  DBKeys=90  Fetches=0+0  RecsOut=0 #Bufs=24
~E#0003.01(1) Fin     Bitmap   DBKeys=42  Fetches=0+29  RecsOut=42
SQL> set flags 'nobitmap'
SQL> fetch c1;

          42
SQL> close c1;
SQL> open c1;
~E#0003.01(2) Estim   Index/Estimate 2/46 1/90
~E#0003.01(2) BgrNdx2 EofData  DBKeys=49  Fetches=1+1  RecsOut=0 #Bufs=24
~E#0003.01(2) BgrNdx1 EofData  DBKeys=90  Fetches=0+0  RecsOut=0 #Bufs=24
~E#0003.01(2) Fin     Bitmap   DBKeys=42  Fetches=0+29  RecsOut=42
SQL> fetch c1;
```

```
          42
SQL> close c1;
```

Notice how the bitmapped scan feature need only be enabled at the time the cursor is first opened. This directs the optimizer, while compiling the query, to attempt to use a bitmapped scan.

Disabling the bitmapped scan feature has no effect on subsequent executions of the query once it is compiled. If a bitmappped scan strategy is chosen by the optimizer then a bitmapped scan will be performed evey time the cursor is opened.

Logical Name

If you would like to enable bitmap scan for execution of a specific program, you can use the RDMS$ENABLE_BITMAPPED_SCAN logical name. By using the logical name you will enable bitmap scan for all queries in the program.

Example 15 is a simple embedded SQL in C program that performs one query.

Example 15:

```
$ ty bitmap.sc
#include <stdio.h>
#include <stdlib.h>
#include <sql rdb headers.h>
main( )
{
    int cnt;
    int     return status;

    EXEC SQL INCLUDE SQLCA;
    EXEC SQL DECLARE ALIAS FILENAME mf personnel;
    EXEC SQL WHENEVER SQLERROR GOTO HANDLE_ERROR;

    EXEC SQL SET TRANSACTION READ ONLY;
    EXEC SQL SELECT COUNT(*) INTO :cnt FROM EMPLOYEES
                WHERE STATE = 'NH'
```

```
                AND SEX = 'M';
    EXEC SQL COMMIT;

    printf("\nQuery returned %d rows.\n",cnt);
    exit(1);

HANDLE ERROR:
    sql signal();
    exit(0);
}
```

Example 16 shows the use of the RDMS$ENABLE_BITMAPPED_SCAN logical name to enable bitmap scan for this program. The RDMS$SET_FLAGS logical is also used here with the values of 'execution' and 'strategy' to demonstrate that a bitmapped scan is used.

Example 16:

```
$ define rdms$enable bitmapped scan 1
$ define rdms$set flags "strategy,execution"
$ run bitmap
~S#0001
Firstn  Aggregate
Leaf#01 BgrOnly EMPLOYEES Card=100       Bitmapped scan
  BgrNdx1 RSTATE [1:1] Fan=19
  BgrNdx2 RSEX [1:1] Fan=19
~E#0001.01(1) Estim   Index/Estimate 2/65 1/90
~E#0001.01(1) BgrNdx2 EofData  DBKeys=65  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0001.01(1) BgrNdx1 EofData  DBKeys=90  Fetches=0+0  RecsOut=0 #Bufs=33
~E#0001.01(1) Fin     Bitmap   DBKeys=61  Fetches=0+36  RecsOut=61

Query returned 61 rows.
```

Note that using the logical name in this way will cause all queries, including metadata queries, to use bitmapped scan where possible.

Normally the RDMS$SET_FLAGS logical name would be used to set the various flags values in the same way as the interactive SQL SET FLAGS command is used. However the presence, or absence, of the RDMS$ENABLE_BITMAPPED_SCAN logical name overrides any setting for this flag in the RDMS$SET_FLAGS logical, so this method cannot currently be used.

Example 17 demonstrates that, while there is no error reported, RDMS$SET_FLAGS cannot currently be used to enable bitmapped scan.

Example 17:

```
$ deassign rdms$enable_bitmapped_scan
$ define rdms$set_flags "strategy,execution,bitmapped_scan"
%DCL-I-SUPERSEDE, previous value of RDMS$SET_FLAGS has been superseded
$ sql$
SQL> show flags

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   STRATEGY,PREFIX,WARN_DDL,MAX_SOLUTION,EXECUTION(100)
   ,MAX_RECURSION(100),NOBITMAPPED_SCAN
```

Dynamic SQL

If you wish to write a program and employ bitmapped scan for a specific query, dynamic SQL statements can be used to execute a "SET FLAGS" statement the first time a query is executed.

Bitmapped scan must be enabled the first time a query is executed, or the first time a cursor is opened. This will cause the optimizer to attempt to use bitmapped scan in the retrieval strategy for that query.

Once the query is compiled, bitmapped scan can be enabled or disabled without affecting the retrieval strategy for that query, no matter how many times it is executed.

Example 18 is a C program with embedded SQL that uses the dynamic SQL interface to enable and disable bitmapped scan.

Example 18:

```
$ ty bitmap.sc
#include <stdio.h>
#include <stdlib.h>
#include <sql rdb headers.h>

main( )
{
    int cnt,i;
    int return_status;
    int first time = TRUE;
    char bitmap on[30] = "SET FLAGS 'BITMAPPED SCAN'";
    char bitmap_off[30] = "SET FLAGS 'NOBITMAPPED_SCAN'";

    EXEC SQL INCLUDE SQLCA;
    EXEC SQL DECLARE ALIAS FILENAME mf_personnel;
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT COUNT(*) FROM EMPLOYEES
                WHERE STATE = 'NH'
                AND SEX = 'M';
    EXEC SQL WHENEVER SQLERROR GOTO HANDLE ERROR;

    EXEC SQL SET TRANSACTION READ ONLY;

    for (i=1; i<4; i++) {
        if (first time) {
            EXEC SQL EXECUTE IMMEDIATE :bitmap on;
        }
        EXEC SQL OPEN C1;
        if (first time) {
            EXEC SQL EXECUTE IMMEDIATE :bitmap_off;
            first time = FALSE;
        }
        EXEC SQL FETCH C1 INTO :cnt;
        EXEC SQL CLOSE C1;
        printf("Query returned %d rows.\n",cnt);
    }
```

```
    EXEC SQL COMMIT;
    exit(1);

HANDLE ERROR:
    sql signal();
    exit(0);
}
```

In example 19 we execute the program shown in example 18. The RDMS$SET_FLAGS logical is used here with the value of "STRATEGY,EXECUTION" to demonstrate that bitmapped scan was used.

Example 19:

```
$ define rdms$set flags "strategy,execution"
$ run bitmap
~S#0001
Aggregate
Leaf#01 BgrOnly EMPLOYEES Card=100       Bitmapped scan
  BgrNdx1 RSTATE [1:1] Fan=19
  BgrNdx2 RSEX [1:1] Fan=19
~E#0001.01(1) Estim   Index/Estimate 2/65 1/90
~E#0001.01(1) BgrNdx2 EofData  DBKeys=65  Fetches=0+0  RecsOut=0 #Bufs=0
~E#0001.01(1) BgrNdx1 EofData  DBKeys=90  Fetches=0+0  RecsOut=0 #Bufs=33
~E#0001.01(1) Fin     Bitmap   DBKeys=61  Fetches=0+36  RecsOut=61
Query returned 61 rows.
~E#0001.01(2) Estim   Index/Estimate 2/65 1/90
~E#0001.01(2) BgrNdx2 EofData  DBKeys=65  Fetches=0+0  RecsOut=0 #Bufs=33
~E#0001.01(2) BgrNdx1 EofData  DBKeys=90  Fetches=0+0  RecsOut=0 #Bufs=33
~E#0001.01(2) Fin     Bitmap   DBKeys=61  Fetches=0+36  RecsOut=61
Query returned 61 rows.
~E#0001.01(3) Estim   Index/Estimate 2/65 1/90
~E#0001.01(3) BgrNdx2 EofData  DBKeys=65  Fetches=0+0  RecsOut=0 #Bufs=33
~E#0001.01(3) BgrNdx1 EofData  DBKeys=90  Fetches=0+0  RecsOut=0 #Bufs=33
~E#0001.01(3) Fin     Bitmap   DBKeys=61  Fetches=0+36  RecsOut=61
Query returned 61 rows.
```

The important thing to notice from this example is that bitmapped scan was only enabled the first time the cursor was opened.

This method is useful where you wish to employ bitmapped scan for a specific query, without affecting other queries in the executable.

This method can also be used from the SQL module language.

Bitmappped Scan Comparison

The following table shows execution times and I/O counts for several queries with and without bitmapped scan enabled.

	NOBITMAPPED_SCAN		BITMAPPED_SCAN		I/O
Query #	DIO	CPU Sec.	DIO	CPU Sec.	Improvement
1	116	00.05	150	00.13	-29%
2	405	00.14	99	00.07	+76%
3	412	00.;24	253	00.25	+39%
4	70	00.06	74	00.37	-5%
5	46	00.01	46	00.02	0%
6	817	00.68	582	00.79	+29%

Notice how there can be a dramatic difference in performance using bitmap scans.

The best performance improvement occurs for complex queries that retrieve and combine large BBC duplicates chains from indexes of type is sorted ranked.

For indexes with few or no duplicates, there can even be a performance penalty for using bitmap scan.

Glossary

B-tree: A binary tree of nodes that constitute a sorted or ranked index.

Background Index: When the dynamic optimizer is used, all potentially useful competing indexes are formed into a background index list.

Background Index Number: A number assigned to each background index so that it can be uniquely identified.

Bitmapped Scan: The ability to use in memory compressed dbkey bitmaps to perform arbitrary logical AND and OR operations.

Boolean: A single condition or predicate on a query.

Bugcheck Dump: A dump of in memory data structures in response to an unexpected condition.

Cardinality: A count. For example the cardinality of a table is how many rows are stored in that table.

Dbkey: The database key is the logical address of a row in the database.

Debug Flag: A feature of Rdb that can be used to display diagnostic information.

Dynamic: The dynamic optimizer is that component that adapts the use of different indexes for each execution of a query.

Estimate: An approximation of the number of rows that would be selected from a given index.

Estimation: The process of using an index structure to find the approximate number of rows that match a set of conditions.

Exe buffer: Dynamic optimizer execution buffer. These buffers can hold 1024 dbkeys.

Exec: The Rdb Executive is that component that handles compiling and driving execution of a query. Journaling, buffering, locking and other lower level functions a performed by KODA.

Fanout Factor: The estimated number of separators that will fit in an index node.

Fin: Final stage of execution of a dynamic query, where rows are delivered.

Hashed Index: A hashed index uses a mathematical function to locate the appropriate index structure. These indexes are useful only for queries that provide a complete key value.

Index Estimation: See Estimation.

Key Only Boolean: The testing of some condition against the key value in an index to determine if the row should be included in the selection.

KODA: That component that performs low level functions such as locking, journaling, and buffering of data.

Optimizer: That component of Rdb that determines how to access the database for a given database query.

Partitioned: An index is partitioned if different key value ranges are stored in different storage areas.

Predicate: An assertion or condition constraining the rows accessed by a database query.

Query: A database query is a statement that manipulates or retrieves data. Typical queries will include select, delete and update statements. A stored procedure or multi-statement procedure may contain many queries.

Range List: A query that specifies more than one simple range for an index.

Ranked Index: An index defined as "type is sorted ranked". These index structures differ from sorted indexes in that upper level index entries contain statistics on how many rows are located down each branch of the index structure. This is the cardinality of each index branch.

Separator: A key value, in an index node or estimator block, that defines the start or end of a key range.

Shortcut: The early termination of a process or function.

Sorted Index: An index defined as "type is sorted". Also referred to as a non-ranked index.

Tiny Buffer: A dynamic optimizer tiny dbkey buffer. These buffers can hold 20 dbkeys.

Zero Shortcut: Early termination of query execution because index estimation found no rows for a selected range.

ORACLE

Oracle Rdb
Guide to Database Performance and Tuning: Bitmapped Scan
June 2004

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com