Rdb Journal
technical article

Guide to Database Performance and Tuning: Predicate Estimation

A feature of Oracle Rdb

By Mark Bradley
Oracle Rdb Relational Technology Group
Oracle Corporation

Table of Contents

Background

The Rdb optimizer can use the index structures in a database to estimate how many rows may be selected from a table for a given set of conditions. This process is termed "Index Estimation" or more often simply "Estim".

Significant enhancements have been made to the Estim process. These enhancements have become necessary due to the discovery of limitations in the accuracy and reliability of the older code. In addition, this work forms the basis of future work that will broaden the application of estimation, return even more accuracy in estimation, and improve control of the cost of the estimation process itself.

This paper describes these enhancements and discusses the use and control of the Estim process. The first part of this paper describes features implemented for Rdb in both version 7.0.7 and version 7.1.2. The second part of the paper describes enhancements for version 7.1.2 that are not present in version 7.0.7.

This paper assumes the reader is familiar with the structure of indices in Rdb. For further information on the structure of Rdb indices please refer to the Rdb documentation, or to the Rdb Internals And Data Structures manual from the Rdb Internals course.

Predicate Estimation

Predicate estimation is used increasingly in the Rdb optimizer to determine the cost and productivity of various index scans.

When a particular query is executed, the conditions in the row selection expression, the “where” clause of a SQL statement, determine which rows will be selected. These conditions, or predicates, can be used to limit the parts of an index that are scanned to find data rows.

Consider the SQL query in example 1.

Example 1:

```
SQL> SELECT * FROM employees WHERE last_name='Toliver'
cont>  and first_name='Alvin';
```

If there were an index on **first_name** and a second index on **last_name**, then Rdb would have to decide which of the two indices would be most efficient for retrieving the data. To do this Rdb examines each index to find out roughly how many rows would be found through that index. For example, how many 'Toliver' rows would be found in the **last_name** index?

Uses Of Predicate Estimation

Dynamic Optimization

Historically Rdb uses estimation in the dynamic optimizer. The dynamic optimizer uses estimation to calculate costs of index scans on competing indices. This information is used to ensure the most efficient indices are scanned first.

Selectivity of Literal Values

During request compilation, the Rdb optimizer uses the selectivity of each expression to help cost various retrieval strategies to determine the most efficient method for retrieving the data.

Where an expression compares a literal value (e.g. WHERE field1=42), and an index exists on that field, the static optimizer can use **sampled selectivity** to obtain using the actual data an estimate for that predicate. In other words how many rows would actually have the value forty-two in the field called "field1"?

This feature can be enabled in Oracle Rdb V7.1.2 using the **optimize with sampled selectivity** clause on the query, or the SET OPTIMIZATION LEVEL 'sampled selectivity' statement.

Bitmap Scans

Where different indices can provide different information, such as the SQL query shown in example 1, Rdb can use the information from the different index scans to reduce the number of rows read.

For example, if the index on last name is scanned, the optimizer can build a list of the dbkeys for all rows that have **last_name** equal to 'Toliver'. It could then scan the index on **first_name** to build a second list of dbkeys for all rows that have **first_name** equal to 'Alvin'. These lists can be compared, and only when the dbkey of a row appears in both lists does the row need to be accessed.

Rdb can perform this kind of operation by building dbkey bitmaps. Dbkey bitmaps can be used to perform arbitrary comparisons including "bitmap AND" and "bitmap OR" operations. In the future, the bitmap scan feature will use estimation to determine if particular indices are useful for bitmap comparison.

Estimation Results

While the main purpose of estimation is to calculate the estimated number of dbkeys (rows) that scanning the index will find, it also records various other pieces of information:

- EST_TOOK_PLACE – Estimation took place for at least 1 index.
- EST_ZERO – Estimation is precisely zero.
- EST_RUN_NUM – Incremented for each successful estimation on this index.
- DBKEYS_AVG – Estimated number of DBKEYS, used to sort indices in dynamic.
- NODES_AVG – Estimated number of level 1 (leaf) nodes.
- EST_PRECISE – Set if estimation is precise.
- DBKEYS_MIN – Estimate based on expected error.
- NODES_MIN – Estimate based on expected error.
- SPLIT_LEVEL – End of descent in b-tree.

Once all possible indices have been estimated, the DBKEYS_AVG value for each index is used to sort the indices from least cost to most cost.

If the 'EXECUTION" debug flag is set, you will see the results of the estimation process. Example 2 shows the execution trace from a dynamic query that includes the "Estim" information.

Example 2:

```
SQL> set flags 'execution, strategy'
SQL> select count(*) from employees where first_name>'A'
cont> and employee_id>'0';
```

```
~S#0005
Aggregate
Leaf#01 BgrOnly 0:EMPLOYEES Card=100
  BgrNdx1 EMP_EMPLOYEE_ID [1:0] Fan=17
  BgrNdx2 EMP_FIRST_RANKED [1:0] Fan=14
~E#0005.01(1) Estim   Ndx:Lev/Seps/DBKeys 1:2/6\34 2:1/9/43
~E#0005.01(1) BgrNdx1 EofData  DBKeys=100  Fetches=0+0  RecsOut=0 #Bufs=4
~E#0005.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0005.01(1) Fin     Buf      DBKeys=100  Fetches=0+0  RecsOut=100

         100
1 row selected
```

In example 2, the indices are TYPE IS SORTED so the execution trace line for estimation, starting with "E#0005.01(1) Estim" would display:

- Ndx - The background index number for this strategy. In the example Ndx of 1 indicates the index EMP_EMPLOYEE_ID.
- Lev - The level in the index where the selected range spans more than one entry in an index node, termed the split level. The split level is 2 in this example.
- Seps - The number of entries (separators) in the index node that are included in the selected range on that index. This is actually one less than the number of next level nodes that would have to be read for that range. The number of entries is 6 in this example.
- DBKeys - The newly estimated number of database keys that will be selected using this index. This example shows 34 for Ndx number 1.

For indices that are **type is sorted ranked** the output meaning for two of these numbers is different:

- Lev - The estimated number of level 1 nodes that will have to be scanned for this index. For this example there will be 9 level 1 nodes scanned for the EMP_FIRST_RANKED index.
- Seps - The estimated minimum number of database keys that will have to be read from this index. This is the estimated number of database keys less the amount of error calculated for that estimate. The example estimates 43 dbkeys.

For the remaining fields in the output the meaning is the same.

The term “separator” is a little confusing in this case. Generally the term separator applies to entries in upper level index nodes that separate or distinguish the key values between adjacent nodes at the next level down the index. In the case of level 2 and above, the separator count is really one less than the number of next level down nodes that would have to be read for the selected range.

For level 1 nodes the separators are genuine key values. One entry for each unique key value stored in the index. In the case of a split level being level one, the separator count is really the count of unique keys that would be selected using this index.

Notice how a backslash character “\” may appear before the estimated number of dbkeys. This indicates the estimate is imprecise or approximate. The estimate may be precise, for example when the split level is level 1 and there are no duplicates. A forward slash character “/” is used in this position to indicate a precise estimate.

Estimation For Indices Of Type Is Sorted

For sorted indices (non-ranked) Rdb descends the index to find the point where the selected range spans more than one separator (more than 1 separator in the node). This is termed the split level.

Estimation on type is sorted (non-ranked) indices is not performed:

- If the index is partitioned.
- If the query is a range list.

Rdb uses the following information to determine the estimate:

- split_level – the level in the index where range was found to span more than one separator. Level 1 is the lowest level in the index, level two is the one above that and so on.
- seps – the number of separators in the range.
- fan – the index fan out factor. This is an estimate of the number of keys in an index node based on the uncompressed size of the key and the index node size.
- dup – the duplicity factor is the average number of rows per unique key value.

If the split level is level 1 then the number of separators is actually the number of unique keys in the selected range. If the split level is greater than 1 then the number of separators is always one less than the number of branches in the range.

Consider an index node with key values (10,20,30,40,50,60). These key values represent the minimum key values down each branch of the index. If the query were looking for the range

between 25 and 55 the optimizer would need to explore the branches with minimum keys of 20, 30, 40, and 50. But it guesses that only half of the branch for key 20 and half of the branch with key 50 will be selected. So the number of separators is set to 3. This is one less than the actual number of interesting branches.

The estimate is calculated using:

$$estimate = dup * seps * fan^{(split\_level - 1)}$$

So if, for example, if there are 2 keys for each key value (dups = 2), with the number of separators at the split level is 10, and the average number of dbkeys per index node (fan) of 9, and the split level at index level 3, the estimated number of dbkeys would be:

$$\begin{aligned} Estimate &= 2 * 10 * 9^{(3-1)} \\ &= 2 * 10 * 9^2 \\ &= 20 * 81 \\ &= 1620 \end{aligned}$$

If the index is of **type is sorted** and the estimated number of dbkeys for an index is less than or equal to ten, and precise, then no further indices will be estimated. Fetching the small number of rows would likely be more efficient than scanning further indices for estimation purposes. After every ten executions, this rule will be ignored and further indices will be estimated.

The estimate for an index is considered precise where the split level is level 1 in which case the separators are actual key values. If the index allows duplicates, then Rdb cannot tell how many duplicates there are without reading the duplicate nodes, so the estimate is not considered precise.

If the estimate is precise and zero, then Rdb will consider performing a “zero shortcut”. A zero shortcut means that since this index has no interesting rows, the query will not return any rows. This could happen in example 1 above, if the index on last_name did not contain the key value ‘Toliver’. In that case looking for rows with first_name of ‘Alvin’ is pointless.

A zero shortcut cannot be used if the isolation level is read committed, or if the cursor is a “WITH HOLD” cursor, because some other process may insert rows with that key value during execution of the request.

Estimation For Indices Of Type Is Hashed

Oracle Rdb version 7.0 does not support estimation on indices of type is hashed.

Version 7.1.2 will contain support for estimation of hashed indices. Refer to the section below describing 7.1.2 enhancements for details.

Estimation For Indices Of Type Is Sorted Ranked

The next few sections of this document describe the estimation of indices of "TYPE IS SORTED RANKED".

Except where noted, the functionality described here is available in Oracle Rdb versions 7.0.7 and 7.1.2.

Theoretical Basis

The theoretical basis for the redesign of estimation on ranked indices came from various design documents and patents by Gennady Antoshenkov. In particular, two patents motivated the design:

- Performance-Related Estimation Using Pseudo-Ranked Trees, Gennady Antoshenkov 10-May-1995.
- Simple Random Sampling On Pseudo-Ranked B+ Trees, Gennady Antoshenkov 25-Jul-1991.

The rest of this document will discuss the design and implementation of estimation for ranked indices in Rdb.

Design Overview

A simple overview is provided here, as an introduction to the implementation

If you consider a range of key values from a theoretical minimum value to a theoretical maximum value, you can divide that range into sub-ranges based on the separators in an index. Each separator in an index describes a sub-range of key values that is bounded by the separator and the next

sibling separator to the right. The left or low separator is included in the range, and the right or high separator is excluded.

The notation used here uses a square bracket "[" to suggest the key value is included in the range and the round bracket ")" to suggest the key value is excluded from the range.

The Rdb executive provides a callback routine for estimation that returns true or false sub-ranges of key values given an arbitrary key value. For example, if the index lookup was of the form "KEY >= 'M'", and the callback routine was called with a key of 'A', it would return the range of keys [MIN,'M') as a false interval. False meaning that key values in that range are not interesting to the query.

The [MIN, 'M') range above describes a range of key values from theoretical minimum key value to a real key value of 'M' where the low end of the range is included, and the high end excluded.

By repeatedly calling this routine, the optimizer can divide the sub ranges obtained from the index nodes into:

- Known true ranges – where the index branch described by the separators is known to be completely included in a true interval obtained from the callback routine.
- Known false ranges – where the index branch described by the separators is known to be completely included in a false interval obtained from the callback routine.
- Mixed intervals – where the index branch described by the separators spans across a true/false interval boundary.

It is obvious that a false interval is not productive to descend since key values in that range will not be selected by the query.

With a ranked index, the ranking information can be used to give an estimate of the number of rows and dbkeys selected by the query. This is calculated by including the ranking values of all true intervals, excluding all false intervals, and assuming half of any mixed intervals.

The major advantage of this design over the simplistic descent to split level method, used by non-ranked indices, is that it can be adapted to work with partitioned indices and range list queries.

A range list query is one where more than one range of key values is selected. Examples of such queries could be "WHERE key_value IN (1,2,3)" or "WHERE key_value < 10 or key_value > 100".

The current implementation does not estimate partitioned indices.

The actual implementation is in two phases:

1. The *descend phase* where the optimizer descends the index, refining the true, false, and mixed intervals, until the split level is found.
2. The *refine phase* where the optimizer descends the index into mixed intervals to more precisely define the part of the index that needs to be scanned, and by descending into true intervals to obtain more precise cardinality information closer to the leaf nodes.

The first phase of implementation executes phase 1 (descend to the split level) as the default behavior, and allows for phase 2 (refine) as a user selected option. It is intended that the second refine phase become a default once field test is complete.

Gennady Antoshenkov proposed that refining of the estimates beyond the split level be performed by randomly descending based on the cardinality of the interval. This means that a branch of the index that had a very high estimated cardinality would be most likely to be picked for refinement.

He also proposed that mixed intervals are good candidates for refinement.

One can easily see the benefit of refinement if by imagining an index root node pointing to two next level nodes. The root node would have a null separator for the first entry, and some real key for the second. Imagine a query that selects a very small range of values that spans the second separator value. The root node is now the split level, and the estimate will be half the right branch plus half the left branch giving half the table population, which may be highly inaccurate.

Estimation Errors

Errors in the estimated cardinality result from two factors:

1. Ranking information within the index is not precise, and is allowed to drift up to 33% between levels before an update is forced. If the ranking information in a level 2 index node suggests that a particular level 1 node has 100 rows in it, Rdb may in fact find as many as 133 rows or as few as 67 rows. This problem is compounded at higher levels in the tree.
2. For mixed intervals the optimizer guesses that half the key values in that branch of the index will be selected. So for a mixed interval with a cardinality of 100 the guess would be 50 rows, but in fact as many as 100 or as few as zero rows may be selected in that branch.

By calculating these errors the optimizer can assess where refinement is likely to be most productive.

Of course, refinement is performed at a cost in that it costs CPU time and potentially real I/O to descend an index, so limits must be placed on refinement so that the cost is not unreasonable. For example, if the optimizer has already estimated one index and it shows a fairly confident and low estimate, is it even productive to estimate another index?

Currently estimation for ranked indices is always attempted where possible, unless a precise estimate of zero is returned (zero shortcut). In the non-ranked case, estimates of 10 or fewer will shortcut further estimation.

Gennady Antoshenkov proposed that sampling into true and mixed intervals could dramatically reduce errors in estimation. By descending randomly to the leaf level and computing the difference at each level between the actual and stored cardinality information, the optimizer could measure the actual versus theoretical cardinality drift. While there is a sound mathematical basis for such sampling, in practice, one would wonder how much additional I/O this may cause. It is believed this would only be practical where the number of necessary I/O's for query execution is likely to be large, and consequently outweigh the cost of sampling.

Implementation

The Rdb executive is responsible for allocating a structure called an ESTIM block. The ESTIM block is fundamentally a set of arrays, where each element in the array represents a key value derived at runtime from data provided for the query.

The entries are maintained as a linked list where adjacent entries in the list describe a range of key values. Such a range of key values is termed an "interval". Various flags are used to describe the type of interval:

- TRUE – The interval commences in a known true interval.
- MIXED – This is an index node that spans into or out of a true or false interval. It should also be marked as true or false based on the interval it starts in. Only index node entries are marked as mixed.
- FALSE – The interval commences in a known false interval.

- PICKED – This separator has been expanded by reading the child node. This flag is only set on index nodes that have been expanded (read), but the entry may be retained if it also represents a range boundary of a known true or false interval.
- EXEC – This separator has been provided by the Rdb executive, the relational layer of Rdb. This applies to keys that represent a range boundary of a true or false interval.
- NODE – This separator represents an entry from an index node. This flag and EXEC may both be set if a range boundary falls on a separator value.
- MIN – The separator represents the theoretical minimum key. No key value is actually associated with this entry.
- MAX – The separator represents the theoretical maximum key. No key value is actually associated with this entry.

The actual key values (separators) are not stored in the ESTIM block as they are considered to be too dynamic. Instead a separate block of memory is allocated and attach to the ESTIM block. This memory is not easily re-used as separators are created and destroyed during the estimation process. Instead, Rdb waits until it appears full, at which point a decision is made if it really is full or simply needs de-fragmenting.

In any event, a second separator buffer is allocated with the same size (or larger if necessary) and the separators are shuffled into the new buffer. The new buffer becomes the primary, and the second is retained (also off the ESTIM block) until needed. It is possible that the new buffer will need to expand, in which case the old (smaller) buffer is freed and a new (larger) one created.

There are two user visible outputs from the new estimation code. First, the ESTIM block can now be found in bugcheck dumps, and second, the user can use the set flags “EXECUTION, DETAIL (2)” to display the ESTIM block after each descend or refine phase.

The ESTIM block can contain 1024 entries. Each entry representing a boundary of a known true or false interval, or a key value read from an index node. While this size is somewhat arbitrary, in practice, it allows us to descend an index until the range of key values spans over a thousand key values in the index nodes. This is more than sufficient to obtain an estimate that is reasonably accurate.

Example 3 shows an ESTIM block representing an infinite unknown interval, when initialized.

Example 3:

```
Estimator block @1790210
Tail of separators: 0
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0     E I  Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: NULL
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
  Key: NULL
# of nodes: 2;
Minimum Level: 0  Detail Level: 5
Compression disabled: Compression bytes @17741f4 run length 0.
Entry=0.000000+/-0.000000 Leaf=0.000000+/-0.000000
IO Limit: -1  IO Count: 0
Free Nodes  Stored: 1022  Actual: 1022
Primary separator block @171a020 of 2048 bytes
Second separator block  @0 of 0 bytes
```

For each entry (separator) in the block the output will show:

- Sep 0 – This will be the entry number in the separator array used to store this entry.
- TMFPENIA flags – these are shorthand for the flags shown above. If a flag is set for that entry, the corresponding character is output. If the flag is clear, the position in the flags field is blank.
- Card – This is the cardinality of the branch or entry. This field is only valid for entries out of indices. Range boundaries that do not also represent index entries do not have a cardinality. In simple terms this is the number of dbkeys down that branch of an index.
- Leaf – This is the number of level 1 nodes down a branch of an index. It is not used for range boundaries unless it is also an index entry.
- Next – This is a pointer to the next logical entry in the separator array. Entries are continually used and re-used, so entries may not be used in order.
- Lev – This is used for index entries (separators) to indicate the level in the index where the entry was obtained.
- Dbkey – Following the level is the dbkey associated with that separator. This will be a pointer to the next level down the branch in the index.
- Key – This is the actual key value for the separator display in hex and ASCII. In both hex and ASCII the key is read from left to right. This is contrary to tradition for hex display, but consistent with the comparison rules for index key entries, which are always compared as ASCII text.

The minimum level tells us the lowest level read in the index. This would be the split level after the descend phase, and the minimum level read after a refine phase.

The detail level is the value of the DETAIL_LEVEL flag if the EXECUTION flag is set. The ESTIM block will be dumped for detail level 2 and above.

If run length compression is enabled for the index, the Rdb executive will store the pointer to the candidate bytes, and the minimum run length in the ESTIM block so that KODA can decompress the key values. Compressed key values will be decompressed as they are reconstructed from the index node.

When an actual estimation has been performed the “Entry” and “Leaf” fields show the estimated number of entries (or dbkeys) and the estimated number of leaf (or level 1) index nodes in the selected range. These values are shown plus or minus the error computed for that estimate. For example, if the output showed “Entry=10.000000+/-5.000000” this would indicate an estimate of 10 dbkeys (entries) plus or minus an estimated error of 5 dbkeys.

The “IO Limit” and “IO Count” fields are used for control over the estimation process and are described later.

The free node counts are the number of available slots in the estim block. If the stored and actual values disagree, the estim block is corrupt.

Each separator in the index node is entered into an entry in the ESTIM block, and the appropriate location in the linked list found for inserting (or updating) the entry.

If the insertion point is a known false interval then the separator can be discarded as long as the range represented by that separator does not span into or out of the false interval into a true or unknown interval.

As each new entry is added (or updated), the entry is used to refine the known true/false/mixed intervals and to eliminate any further known false index nodes.

Once the entire index node has been processed, the optimizer scans the resulting ESTIM block to determine if a further descends is needed to find the split level. If more than one entry in the ESTIM block represents an index node that has not been read, and is a true or mixed interval, Rdb has reached the split level.

Example 4 shows ESTIM block after processing the root node. In this case the root node has two separators. The first (or left most) is of course a null representing the minimum possible key value. The second is '.05441' where the '.' Represents the null byte.

The query in this case used WHERE KEY = '00005', so the only interesting separator in the node is the minimum (first) one.

Each key separator is passed to the EXEC callback routine. When the first (minimum) key is passed, the callback routine passes back [MIN,'.00005') as a known false interval. When '.05441' is passed, ['.00006',MAX) is returned as a known false interval. This information is also recorded in the ESTIM block.

Example 4:

```
Estimator block @1800a50
Tail of separators: 18
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0  M  ENI  Card=5440.000000, Leaf=272.000000, Next=2, Lev=4: 72:1444:0
  Key: NULL
Sep 2     E    Card=0.000000, Leaf=0.000000, Next=5, Lev=0: Null DBK
  Key: ' 03030303035' '.00005'
Sep 5   F E    Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: ' 03030303036' '.00006'
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
  Key: NULL
# of nodes: 4;
Minimum Level: 4  Detail Level: 5
Compression disabled: Compression bytes @17741f4 run length 0.
Entry=0.000000+/-0.000000 Leaf=0.000000+/-0.000000
IO Limit: -1  IO Count: 0
Free Nodes  Stored: 1020  Actual: 1020
Primary separator block @1799c60 of 2048 bytes
Second separator block  @0 of 0 bytes
------ End of ESTIM block ------
```

Notice how the MIN entry has both the NODE and EXEC flags set. This represents an index entry that is a mixed interval.

The other key value in the root node was '.05441' which is wholly contained in a known false interval and was discarded.

Normal Estimation

This is termed normal estimation only because it is the current default behavior. In the future, refinement rules may be enabled by default. Refinement is described in later sections of this document.

Estimation by default is performed to the split level. The root node of the index is read, and each key is inserted into the ESTIM block, or discarded, as appropriate.

After expanding each index node, the ESTIM block is checked to see how many entries in the block correspond to unread index nodes in mixed or true intervals. If there is only one such node, that node is processed in the same way.

Shortcut termination is only performed on ranked indices if a precise estimate of zero is returned.

Note: If a table has a mix of ranked and non-ranked indices, the non-ranked indices may cause shortcut termination for ten or fewer rows, but the ranked indices will not.

Estimation of ranked indices is currently not performed if the index is partitioned.

After each index node is expanded, the estim block is scanned to calculate the estimated cardinality. The estimate is calculated as the sum of the cardinality of all true intervals plus half the cardinality of any mixed intervals.

Range List Queries

Where a query is a range list, there may be multiple true ranges to examine.

Example 5:

```
SQL> select * from employees where (last_name='Ames' or last_name>'Z')
cont> and employee_id>'0';
```

```
~S#0012
Tables:
  0 = EMPLOYEES
Leaf#01 FFirst 0:EMPLOYEES Card=98
  Bool: ((0.LAST_NAME = 'Ames') OR (0.LAST_NAME > 'Z'))
          AND (0.EMPLOYEE_ID > '0')
  BgrNdx1 EMP_EMPLOYEE_ID [1:0] Fan=17
    Keys: 0.EMPLOYEE_ID > '0'
  BgrNdx2 EMP_LAST_NAME [1:0,1:1] Fan=12
    Keys: r0: 0.LAST_NAME > 'Z'
          r1: 0.LAST_NAME = 'Ames'
```

In example 5, ESTIM on the LAST_NAME index must search for two ranges

1. LAST_NAME=’Ames’
2. LAST_NAME > ‘Z’

The callback routine will return an appropriate true or false range around each tested key value. Each returned range is stored in the ESTIM block. The optimizer can therefore be sure to account for all needed branches of the index.

Example 6 shows the ESTIM block after estimation on the LAST_NAME index for the query above.

Example 6:

```
Estimator block @19160e0
Tail of separators: 170
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0  MF ENI  Card=11.000000, Leaf=1.000000, Next=2, Lev=2: 66:697:3
  Key: NULL
Sep 2     E    Card=0.000000, Leaf=0.000000, Next=5, Lev=0: Null DBK
  Key: ' 0416d65732020202020202020202020' '.Ames          '
Sep 5   F E    Card=0.000000, Leaf=0.000000, Next=6, Lev=0: Null DBK
  Key: ' 0416d65732020202020202020202021' '.Ames         !'
Sep 6  MF  N   Card=9.000000, Leaf=1.000000, Next=4, Lev=2: 66:740:4
  Key: ' 054' '.T'
Sep 4     E    Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: ' 05a2020202020202020202020202021' '.Z            !'
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
```

```
  Key: NULL
# of nodes: 6;
Minimum Level: 2  Detail Level: 2
Compression disabled: Compression bytes @183dc84 run length 0.
Entry=10.000000+/-13.000000 Leaf=2.000000+/-0.000000
IO Limit: 34  IO Count: 1
Free Nodes  Stored: 1018  Actual: 1018
Primary separator block @182d160 of 2048 bytes
Second separator block  @0 of 0 bytes
------ End of ESTIM block ------
```

This index has only two levels, so level 2 is the root node. In this case it is also the split level. Each separator is describe below.

- Sep 0: This is the minimum separator. It is both an EXEC (range limit) and an index node. It represents a pointer to the level one node with key values >= ‘’ (a null key) and also the beginning or low bound of a range. The callback routine returned the range [MIN, ’Ames’) as a false interval. This key range is marked as mixed because it spans out of the known False range, and therefore may include interesting key values.
- Sep 2: This is the upper bound of the known false interval returned from the EXEC callback routine. It does not represent an index node, so keys in the preceding index node (Sep 0) span into this range. So even though the range [‘Ames ’, ‘Ames !’) would be a true interval, there are no real key values in that range. The preceding mixed index node (Sep 0) would contain that key value if it exists in the index. So the preceding node spans into this range making it a mixed range.
- Sep 5: Ranges are always represented as low included, high excluded. So to represent a range such as LAST_NAME=’Ames’, A high bound key is manufactured by incrementing the low order byte of the key. This turns the last space character into an exclamation mark. As well as representing the upper bound of the interval [‘Ames ‘, ‘Ames !’), this also represents the lower bound of the false interval [‘Ames !’, ‘Z !’).
- Sep 6: This index node was the highest key value found in the false interval [‘Ames !’, ‘Z !’), and therefore may contain key values that span into the next range (i.e. > ‘Z’). So this node is marked as a mixed node.
- Sep 4: This is the upper bound of the known false interval [‘Ames !’, ‘Z !’), so represents the start of a mixed (unknown) interval.
- Sep 1: represents the theoretical maximum key value.

Since the highest key value examined in the index was ‘T’ the optimizer hasn’t tested any key values in the last interval [‘Z !’, MAX). Therefore, the optimizer doesn’t actually know precise

true or false information in that range. The preceding index node is marked as mixed because it spans into that unknown range.

Key Only Boolean

It is possible that a range may be scanned in the index and conditions tested against the key value in the index prior to reading the actual data row. Testing of conditions against key values in the index is termed "key only boolean".

When a complete key value is read from an index, for example from a level 1 node, then ESTIM will evaluate any boolean conditions against the key value in the index. When processing upper level index nodes (level 2 and above), the key value is rarely a complete key, so key only boolean is not applied.

The callback routine normally provides a true or false interval based on which key values in the index would be scanned for the query. For example, if the query was of the form "WHERE key >= 'L'" and the optimizer was testing the key value 'S' (using the callback routine), the callback routine would normally return ['L', MAX) as a true interval.

In the case of false intervals, or where the key value being tested is less than a full-length key, this is the value that is actually returned. However, when the key value is complete, the key only Boolean is applied. In this case the result, true or false, of the boolean evaluation is returned. The range returned has a low bound of the key being tested, and a high bound is constructed by incrementing that key value. Thus the true or false returned represents a range of exactly one key value, the one being tested.

When testing a level 1 key entry, the optimizer ignores the range boundaries returned, and simply consider whether the key value is in a true or false interval. If a level 1 entry is false it is discarded. Therefore, testing of level 1 entries does not accumulate redundant range boundaries in the ESTIM block. The range boundaries are ignored, and the optimizer simply marks the entry as true or discards it if it is false.

Consider the following data in Example 7.

Example 7:

```
SQL> select last_name,first_name from employees
cont> where last_name < 'Bn';
 LAST_NAME        FIRST_NAME
 Ames             Louie
 Andriola         Leslie
 Babbin           Joseph
 Bartlett         Dean
 Bartlett         Wes
 Belliveau        Paul
 Blount           Peter
7 rows selected
```

It turns out that in the index on (last_name, first_name) all these key values are in the same level 1 index node. So in the following query (example 8) specifies the condition that last_name is less than 'Bn', the split level will be level 1. The condition on first_name is applied as a key only boolean.

The following query shows the result of applying key only boolean to the entries above.

Example 8:

```
SQL> select * from employees where last_name < 'Bn'
cont>   and first_name < 'J'
cont>   and employee_id > '0';
~S#0030
Tables:
  0 = EMPLOYEES
Leaf#01 FFirst 0:EMPLOYEES Card=98
  Bool: (0.LAST_NAME < 'Bn') AND (0.FIRST_NAME < 'J') AND (0.EMPLOYEE_ID > '0')
  BgrNdx1 EMP_EMPLOYEE_ID [1:0] Fan=17
    Keys: 0.EMPLOYEE_ID > '0'
  BgrNdx2 EMP_LN_FN [0:1] Fan=9
    Keys: 0.LAST_NAME < 'Bn'
    Bool: 0.FIRST_NAME < 'J'
Estimator block @19160e0
Tail of separators: 41
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0 TM PENI  Card=7.000000, Leaf=1.000000, Next=3, Lev=2: 66:1028:2
```

```
  Key: NULL
Sep 3 T    N   Card=1.000000, Leaf=0.000000, Next=2, Lev=1: 79:22:4
  Key: ' 0426172746c657474202020202020 04465616e202020202020' '.Bartlett
.Dean      '
Sep 2   F E    Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: ' 0426e2020202020202020202020' '.Bn            '
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
  Key: NULL
# of nodes: 4;
Minimum Level: 1  Detail Level: 2
Compression disabled: Compression bytes @183e434 run length 0.
Entry=1.000000+/-0.000000 Leaf=1.000000+/-0.000000
IO Limit: 34  IO Count: 2
Free Nodes  Stored: 1020  Actual: 1020
Primary separator block @182d160 of 2048 bytes
Second separator block  @0 of 0 bytes
------ End of ESTIM block ------
```

Notice that the first entry in the ESTIM block has the “P” Picked flag set. This is the entry from the level two node that was read to find the appropriate level 1 node. This remains behind in the ESTIM block because level 1 key tests are not a range test, but rather just a test on the specific key value. The picked entry remains behind because it is also a range boundary, as indicated by the “E” Exec flag.

Of all the entries in this level 1 node, only entry 3 remains. This is the only entry that returned “true” when the key only Boolean was applied.

Refinement Of Estimates On Ranked Indices

While the default behavior for estimation on ranked indices is to perform the descend phase to read the index to the split level, this behavior can be controlled using the REFINE_ESTIMATES flag.

For testing purposes, this flag will enable various rules for the refinement process. In the future some of these rules are expected to become default behavior. At that time, the REFINE_ESTIMATES flag can be used to disable these refinement rules, or to restore the old behavior.

The REFINE_ESTIMATES flag is interpreted as a binary bitmap of flags. Each bit enables or disables the corresponding rule. The statement SET FLAGS 'REFINE_ESTIMATES(1)' would enable rule 1, SET FLAGS 'REFINE_ESTIMATES(3)' would enable rule 1 and rule 2, and SET FLAGS 'REFINE_ESTIMATES(7) would enable rule 1, rule 2, and rule 3, and so on. To enable refinement to be controlled, Rdb tracks the actual number of I/O operations performed by the ESTIM process, and places restrictions on the estimation process based on the number of I/O operations, as well as other factors.

If the REFINE_ESTIMATES flag is not set, or if it is set to zero, the refinement rules are ignored, and all ranked indices will be estimated to the split level.

If any REFINE_ESTIMATES flags are set, then Rdb will attempt the refine phase. This is intended to read further down the index beyond the split level until a reasonable estimate is obtained.

Values for the REFINE_ESTIMATES flag	
Values	**Meaning**
1	Use I/O Limit on Descend
2	Use I/O Limit on Refine
4	Limited refinement on True and Mixed Cardinality
8	Limit the error to 10%
16	Try to be precise
32	Enable Estimation for Hashed Indices
64	Use I/O to Limit the Hashed Estimation

Rule 1 – Use I/O To Limit Descend

If REFINE_ESTIMATES(1) is set, then the I/O count will be used to limit the descend phase of estimation.

When a query with a dynamic strategy is executed, Rdb will estimate each of the background indices. The minimum estimated number of dbkeys is used as an overall I/O limit for the estimation process. The intent is to limit the amount of work performed during estimation in proportion to the number of rows that must be read to satisfy the query.

When the first index is estimated, the estimated number of dbkeys for that index becomes the total I/O limit for the estimation process. Only I/O's performed on ranked indices count towards the I/O limit.

As each succeeding index is estimated, the minimum estimate from any index, ranked or non-ranked, is set as the I/O limit.

Rule 1 forces the descend phase to honor the I/O limit.

The first estimated index does not have an I/O limit. If the first index estimation consumed ten I/O's and returned an estimate of 20 rows, then the I/O limit would be set to 20, and the current I/O count will be set to 10.

Since it is known that the first index estimate is 20 rows (or dbkeys), it is not unreasonable to assume that these rows should not take much more than 20 I/O's to fetch. Of course, if they are cached or buffered, it could take much fewer than 20 I/O's, but if they were badly fragmented it could take much more than 20 I/O's.

If another index is estimated, then each I/O causes the count to increment. If the count of I/O's exceeds the limit, then estimation for that index will terminate, and the estimate will be the best estimate possible once the current (just fetched) index node is processed.

Note that I/O's are only counted after each index node is fetched. If an index node happened to be fragmented, then multiple I/O operations may be required to read the complete index node. Therefore, the I/O count may jump, rather than incrementing one by one.

If an index were in global buffers or in a row cache, then no I/O would be performed, and the count would not be incremented.

It is possible that the number of I/O's performed estimating previous indices causes the I/O count to exceed the I/O limit. Therefore, a read may not even be attempted on a subsequent index even if that index were in a row cache or global buffer pool.

This rule is only applied to ranked indices. If the strategy includes non-ranked indices, then non-ranked indices may be estimated regardless of the I/O limit.

Rule 2 – Use I/O To Limit Refine

If REFINE_ESTIMATES(2) is set, the I/O count will be used to limit the refine phase of estimation.

The minimum estimated number of dbkeys is used as a limit on the total number of I/O's. The number of I/O's consumed during estimation refinement count towards the I/O count, and if the I/O count exceeds the I/O limit, then refinement of the estimate on this index is terminated.

The estimate returned will be the best estimate possible once the current (just fetched) index node is processed.

Rule 3 – Limit Refine On True and Mixed Cardinality

If REFINE_ESTIMATES(4) is set, the refine process will terminate when the sum of the cardinality for known true branches exceeds the sum of the cardinality for mixed branches.

The refine phase descends mixed branches in preference to known true branches. This is because the error in the estimate of a mixed branch is likely to far exceed the error due to cardinality drift in a known true branch.

If the total cardinality of known true branches exceeds the total cardinality of mixed branches, the estimate is considered accurate enough. This is because the effect on the estimate due to those mixed branches cannot be greater than half the mixed branch cardinality.

Rule 4 – Limit The Error To Ten Percent

If REFINE_ESTIMATES(8) is set, as each index node is processed, the estimate and corresponding error is calculated. Once the estimated error falls below 10% of the actual estimate the estimate is considered close enough.

In practice, this rule may be unlikely to occur, as errors in an estimate are likely to exceed 10% for any estimate that has not reached level 1 of the index.

In addition, if Rule 3 is disabled, and the optimizer does not limit the refinement phase based on the true/mixed cardinality, this rule is more likely to come into effect.

Rule 5 – Try To Be Precise

If REFINE_ESTIMATES(16) is set, the refine phase will be attempted.

If no other rules are enabled, Rdb will attempt to refine the estimate all the way to level one, or until the ESTIM block becomes full.

Use of any of the other rules will place the appropriate limits on how far the estimation process will continue.

This rule need only be used if none of the other refinement rules are to be enforced, and you wish the refinement process to be pursued as far as possible.

If any of the refinement rules are enabled, then refinement will be attempted.

Estimation On Large Selections

When performing index estimation where the number of rows selected from each index is large, compared to the depth of the indices being estimated, the I/O limits impose little restriction on the estimation process.

This is done because the cost of performing index estimation is very likely to be small in comparison to the cost of retrieving all the selected rows.

Consider that the job of index estimation in the dynamic optimizer is to realistically re-order the available indices based on the number of rows that would be found for each execution of a request.

If the cost of index estimation is not significant compared to the cost of retrieving the data, then obtaining precise estimates is more important than preserving I/O's.

Estimation On Small Selections

When estimation is being performed and the number of selected rows is small, the I/O consumed during estimation is much more significant to the overall query performance.

If estimation on one index suggests that scanning that index would return two rows, there appears little point descending a second index, which may consume many I/O's, when the data rows could be fetched and processed using the first index in not more than two I/O's.

Of course this argument ignores the effect of fragmentation, which may increase the number of I/O's required to fetch the data. It also ignores the fact that buffering or row cache may reduce the I/O.

By using the smallest estimate obtained so far to limit the total number of I/O's consumed during estimation, a realistic limit is placed on the resources consumed by the estimation process itself. This limit attempts to minimize the impact on overall query performance due to the estimation process.

A Worked Example of Refinement

In example 9, the table T3 has three integer columns and each column has the same unique value. There are 3,000,000 rows in the table with the values 1 to 3,000,000.

There is a separate sorted ranked index on each of the three fields.

The query selects a very small range, exactly two values, from each index:

Example 9:

```
SQL> set flags 'strategy,detail,execution'
SQL> select count(*) from t3
cont>         where (f1 between 1270394 and 1270395)
cont>         and   (f2 between 2125762 and 2125763)
cont>         and   (f3 between 2994 and 2995);
~S#0001
Tables:
  0 = T3
Aggregate: 0:COUNT (*)
Leaf#01 BgrOnly 0:T3 Card=3000000
  Bool: (0.F1 >= 1270394) AND (0.F1 <= 1270395) AND (0.F2 >= 2125762) AND (0.F2
        <= 2125763) AND (0.F3 >= 2994) AND (0.F3 <= 2995)
  BgrNdx1 I31 [1:1] Fan=831
    Keys: (0.F1 >= 1270394) AND (0.F1 <= 1270395)
  BgrNdx2 I33 [1:1] Fan=1664
    Keys: (0.F3 >= 2994) AND (0.F3 <= 2995)
  BgrNdx3 I32 [1:1] Fan=3
    Keys: (0.F2 >= 2125762) AND (0.F2 <= 2125763)
~E#0001.01(1) Estim   Ndx:Lev/Seps/DBKeys 1:1/2/2 2:1/2/2 3:1/2/2
~E#0001.01(1) BgrNdx1 EofData  DBKeys=2  Fetches=19+6  RecsOut=0 #Bufs=1
~E#0001.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0001.01(1) Fin     Buf      DBKeys=2  Fetches=0+1  RecsOut=0

           0
1 row selected
```

This query was performed using the default behavior, so each index is descended to the split level. For this very small range, the split level was found to be level 1 in the index, and a precise estimate of two rows was found for each index.

Note that each index is descended separately to find the key value range for that index. So in this example three separate estimations are performed.

In the following examples, the output has been edited to remove all but the ESTIM output. This is simply for brevity as the strategy in each case is identical.

If the values are changed slightly so that the range spans across separators very high up in the b-tree structure for each index, the estimates for the same size ranges can be markedly different as shown in example 10.

Example 10:

```
SQL> select count(*) from t3
cont>         where (f1 between 1270395 and 1270396)
cont>         and   (f2 between 2125763 and 2125764)
cont>         and   (f3 between 2995 and 2996);
~E#0018.01(1) Estim Ndx:Lev/Seps/DBKeys 1:2/0\1358 2:2/0\2994 3:147622/0\590490
```

When this query is run again with the refinement rules enabled a markedly different result is observed:

Example 11:

```
SQL> select count(*) from t3
cont>         where (f1 between 1270395 and 1270396)
cont>         and   (f2 between 2125763 and 2125764)
cont>         and   (f3 between 2995 and 2996);
~E#0000.00(1) Estim   Ndx:Lev/Seps/DBKeys 1:2/2/2 2:_367500 3:_367500
```

Two important things have happened here:

1. The first index to be estimated has now obtained a far more accurate estimate, even though the split level was very high up in the index.
2. The second and third indices are no longer being estimated.

In example 9, with the default refinement rules and a very low split level, a large amount of I/O is needed to descend the second and third index, even though only two rows would be selected out of the first index. With three million rows, these indices have up to fifteen levels, so the estimation is very expensive in comparison to reading just two rows.

However, example 10 with default rules shows that a very bad estimate due to a split level high up in the index structure does not return a precise enough estimate to be useful.

Example 11 demonstrates the use of two refinement rules:-

1. Refinement is attempted until a reasonable estimate is returned.
2. Estimation will not consider an additional index once the number of I/O's consumed exceeds the smallest estimated number of rows.

The following examples show the output when the 'EXECUTION,DETAIL(2)' flags is set, and refinement rules enabled. Note that the output is broken up into pieces to describe the various parts of the output.

Example 12:

```
SQL> set flags 'execution,detail(2),refine_estimates(15)'
SQL> select count(*) from t3
cont>         where (f1 between 1270395 and 1270396)
cont>         and   (f2 between 2125763 and 2125764
cont>         and   (f3 between 2995 and 2996);
Estimator block @189ba60
Tail of separators: 1129
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0   F ENI  Card=1496.000000, Leaf=1.000000, Next=3, Lev=2: 57:881:0
  Key: NULL
Sep 3  MF  N   Card=1358.000000, Leaf=1.000000, Next=2, Lev=2: 57:5555:0
  Key: ' 080135d2e' '...].'
Sep 2 T   E    Card=0.000000, Leaf=0.000000, Next=4, Lev=0: Null DBK
  Key: ' 08013627b' '...b{'
Sep 4 TM   N   Card=1358.000000, Leaf=1.000000, Next=5, Lev=2: 57:5560:0
  Key: ' 08013627c' '...b|'
Sep 5   F E    Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: ' 08013627d' '...b}'
```

```
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
  Key: NULL
# of nodes: 6;
Minimum Level: 2  Detail Level: 2
Compression disabled: Compression bytes @1821cd4 run length 0.
Entry=1358.000000+/-1765.400024 Leaf=2.000000+/-0.000000
IO Limit: -1  IO Count: 14
Free Nodes  Stored: 1018  Actual: 1018
Primary separator block @18aa4a0 of 2048 bytes
Second separator block  @18a8ad0 of 2048 bytes
------ End of ESTIM block ------
```

Example 12 shows the estimator block after the descend phase on the first index.

Notice how separator 3 is a mixed separator because it spans out of the false interval into the true interval beginning with separator 2. For this reason it is marked as mixed. Separator 4 is also marked as mixed because it spans out of the true interval.

The estimate has found the split level at level two, but it is not very accurate, 1358 dbkeys plus or minus 1765.

In this case the I/O limit is specified as –1. Since optimizer has not yet estimated any index, we don't have an I/O limit. Fourteen I/O's have been used to reach this level, as shown by the I/O Count.

Example 13:

```
Estimator block @189ba60
Tail of separators: 1129
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0   F ENI  Card=1496.000000, Leaf=1.000000, Next=2, Lev=2: 57:881:0
  Key: NULL
Sep 2 T   EN   Card=1.000000, Leaf=0.000000, Next=4, Lev=1: 62:12590:16
  Key: ' 08013627b' '...b{'
Sep 4 T    N   Card=1.000000, Leaf=0.000000, Next=5, Lev=1: 62:12590:17
  Key: ' 08013627c' '...b|'
Sep 5   F EN   Card=1.000000, Leaf=0.000000, Next=1, Lev=1: 62:12590:18
  Key: ' 08013627d' '...b}'
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
```

```
  Key: NULL
# of nodes: 5;
Minimum Level: 1  Detail Level: 2
Compression disabled: Compression bytes @1821cd4 run length 0.
Entry=2.000000+/-0.000000 Leaf=2.000000+/-0.000000
IO Limit: -1  IO Count: 25
Free Nodes  Stored: 1019  Actual: 1019
Primary separator block @18aa4a0 of 2048 bytes
Second separator block  @18a8ad0 of 2048 bytes
------ End of ESTIM block ------
```

Example 13 is the estimator block after the refinement phase on the first index. Notice that estimation has descended to level one and found only two useful key values. These are separators two and four.

The estimate is now a precise estimate of two.

At this point 25 I/O's have been consumed. This is a result of very large and consequently highly fragmented index nodes.

Example 14:

```
Estimator block @189ba60
Tail of separators: 0
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0     E I  Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: NULL
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
  Key: NULL
# of nodes: 2;
Minimum Level: 0  Detail Level: 2
Compression disabled: Compression bytes @181fcb4 run length 0.
Entry=0.000000+/-0.000000 Leaf=0.000000+/-0.000000
IO Limit: 2  IO Count: 25
Free Nodes  Stored: 1022  Actual: 1022
Primary separator block @18aa4a0 of 2048 bytes
Second separator block  @18a8ad0 of 2048 bytes
------ End of ESTIM block ------
```

Example 14 is the estimator block after the attempt to descend the second index. The estimator block is empty because the I/O count is 25, but the I/O limit is 2. So the descent was not even attempted.

Example 15:

```
Estimator block @189ba60
Tail of separators: 0
Flags are True, Mixed, False, Picked, Exec, Node, mIn, mAx.
Sep 0     E I  Card=0.000000, Leaf=0.000000, Next=1, Lev=0: Null DBK
  Key: NULL
Sep 1     E  A Card=0.000000, Leaf=0.000000, Next=-1, Lev=0: Null DBK
  Key: NULL
# of nodes: 2;
Minimum Level: 0  Detail Level: 2
Compression disabled: Compression bytes @18209f4 run length 0.
Entry=0.000000+/-0.000000 Leaf=0.000000+/-0.000000
IO Limit: 2  IO Count: 25
Free Nodes  Stored: 1022  Actual: 1022
Primary separator block @18aa4a0 of 2048 bytes
Second separator block  @18a8ad0 of 2048 bytes
------ End of ESTIM block ------
```

Example 15 is the estimator block after attempting the descent on the third index. Once again no descent was attempted due to the I/O limit.

Example 16:

```
~E#0000.00(1) Estim   Ndx:Lev/Seps/DBKeys 1:2/2/2 2:_367500 3:_367500
```

Example 16 is the execution trace line for the estimation process. The first index returned a precise estimate of two rows.

The underscore character "_" indicates that estimation on indices two and three was not attempted. The estimate for those indices remains the estimate that was made by the static optimizer, or by estimation during a previous execution of the request.

Estimation As A Learning Engine

For queries that execute just once and are then discarded, such as those generated from 4GL applications, or query tools, a balance must be maintained between the cost of the estimation process and the benefit to query execution performance.

As has been shown, this cost is most significant where the number of rows selected is relatively small.

Rdb's behavior in this case must be relatively conservative.

However, where a query executes many times, even for small selections, there is more opportunity to adapt the estimation process over time.

Consider the case where the first index estimated returns an estimate of two rows. It would not seem reasonable to spend ten I/O's to estimate a second index when two I/O's would have fetched all the necessary rows.

But what if this request was executed many thousands of times, and it turns out that a second index, would have returned zero rows?

Future work on the dynamic optimizer is intended to detect this type of situation, and even where the number of rows is small, ensure that estimation is performed from time to time.

Of course, Rdb would like to ensure that estimation is productive. So if estimation is performed, and this does not affect the order of indices, then it was not productive, and can be considered wasteful.

On the other hand, if estimation is performed, and a new index order results, then estimation is productive.

Currently in design is the ability for Rdb to adapt the estimation process over multiple executions of the request.

Queries that do not benefit from the estimation phase will learn to stop performing ESTIM, and queries that do benefit will learn to perform estimation more often, even at a slightly higher cost in terms of I/O.

New Features In Oracle Rdb 7.1.2

While the features described above are common to both Oracle Rdb 7.0.7 and 7.1.2, further work has been undertaken to enhance the index estimation feature that will only be available in version 7.1.2 and later.

The features are based on customer feedback, internal testing, bug reports, and wish list items.

Estimation On Indices Of Type Is Hashed

Oracle Rdb 7.1.2 supports estimation on indices of type is hashed. To enable this feature you must use the refine estimates flag.

The estimation on hashed indices can be controlled using I/O limiting in much the same way as indices of type is sorted ranked.

In the future it is anticipated that estimation on hashed indices will be enabled by default. In any event, the REFINE_ESTIMATES flag can be used to disable this feature.

Rule 6 – Enable Estimation for Hashed Indices

If REFINE_ESTIMATES(32) is set, estimation on hashed indices is enabled.

Hash indices can only be used for retrieval where a full and complete key value is available from the query or strategy.

A query that specifies a list of key values, for example “WHERE KEY IN (1,2,3,4)”, could also use a hashed index. This is termed a range list in the same way as described previously. Rdb supports estimation on hashed indices for range list queries.

An index may be spread over multiple storage areas, termed horizontal partitioning. While Rdb does not currently support estimation on sorted indices that are partitioned, estimation on hashed indices can be performed regardless of the number of partitions.

Estimation on a hashed index will retrieve the correct hash bucket for the requested key value, locate the key value in the hash bucket, and return the duplicate count stored in the hash bucket[1].

This means that for a hashed index that has no overflow or fragmentation, hashed index estimation can be performed in a maximum of 1 I/O. For a range list query this would mean one I/O per key value.

Rule 7 – Use I/O to Limit Hashed Estimation

If REFINE_ESTIMATES(64) is defined, the count of I/O's is used to limit the estimation process for hashed indices.

While estimation on a hashed index would normally be expected to take exactly one I/O, this is not always the case. If an area containing a hashed index is incorrectly allocated, or if for some other reason the hash index structures have fragmented, then retrieving the index structures may take several I/O operations.

Rdb will first read and reconstruct the system record. In extreme cases of fragmentation, the system record itself may be fragmented.

Next Rdb must read the hash bucket for the index. If the target page has filled up, the hashed bucket may be fragmented and it may take several I/O operations to reconstruct the hash bucket.

I/O operations are checked once after reading the system record, and again after attempting to read the hash bucket. Therefore, the I/O count may be seen to jump rather than incrementing smoothly one at a time.

If the I/O limit is reached prior to reconstructing the hash bucket, the estimation on the current index is abandoned.

[1] Notice that Rdb does not have to scan all the duplicate buckets to count the number of records with the given key value.

Zero Shortcut Reordering

In previous versions, if an index was scanned for estimation, and Rdb could be certain that no rows exist for the specified key value(s), then execution of the request would not attempt the actual read of the data, but would perform a "Zero Shortcut".

When a zero shortcut was performed, Rdb did not sort the indices from least to most expected cost since execution of the request was not going to actually fetch the data.

However, consider the case where two indices were available, and the query executed again and again. If the query executed the first time, and found some non-zero estimate for the first index, the second index would be estimated. It is possible that the second index arrived at a precise estimate of zero, and thus a zero shortcut would be performed.

If the query executed a second time, the first index would again be estimated, and may return some non-zero estimate before estimation is attempted on the second index.

In Rdb 7.1.2, if index estimation causes a zero shortcut, that index is swapped to the head of the list to ensure it is always the first index scanned on the next execution. This can significantly reduce the I/O consumed by the estimation process where a query frequently produces a zero shortcut.

Example 17 shows the old behavior where an index is repeatedly estimated even though another index provides a zero shortcut.

Example 17:

```
~E#0005.04(1) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
~E#0005.04(2) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
~E#0005.04(3) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
~E#0005.04(4) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
~E#0005.04(5) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
~E#0005.04(6) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
~E#0005.04(7) Estim  Ndx:Lev/Seps/DBKeys 1:1/1/1 2:1/0/0 ZeroShortcut
```

Example 18 shows the new output where the first zero shortcut causes the indices to be swapped so that the zero shortcut index is estimated first.

Example 18:

```
~E#0006.04(1) Estim   Index/Estimate 2/0 1/1 ZeroShortcut
~E#0006.04(2) Estim   Index/Estimate 2/0 1_1 ZeroShortcut
~E#0006.04(3) Estim   Index/Estimate 2/0 1_1 ZeroShortcut
~E#0006.04(4) Estim   Index/Estimate 2/0 1_1 ZeroShortcut
~E#0006.04(5) Estim   Index/Estimate 2/0 1_1 ZeroShortcut
~E#0006.04(6) Estim   Index/Estimate 2/0 1_1 ZeroShortcut
~E#0006.04(7) Estim   Index/Estimate 2/0 1_1 ZeroShortcut
```

Notice how on the second and subsequent execution, the background index 1 shows an underscore "_" character. This indicates that estimation was not performed for that index due to the zero shortcut from the estimate on background index 2.

Table Cardinality Estimation

In previous versions of Rdb, estimation on a **sorted** (non-ranked) index would read the root index node, and attempt to use the depth of the b-tree and the number of entries in the root node to estimate how many rows actually existed in the table.

This information was used to ensure that limits were set appropriately, for example, on when to switch from indexed to sequential retrieval for the query.

This did not happen for other index types, and proved to be very inaccurate in some cases where the index had a lot of duplicates, or was very sparsely or very densely populated. This occurred because the estimate was based on an arbitrary uncompressed key size, and full index nodes.

Ranked indices, within the index structure, contain a reasonable estimate of the number of rows down each branch of the index. That is normally far more accurate than the estimate possible with non-ranked indices.

Rdb 7.1.2 will use the ranking information in a **sorted ranked** index to estimate the table cardinality each time a ranked index is estimated.

Ranked estimates of table cardinality are always used in preference to those from non-ranked indices for the same query.

Hashed indices cannot be used to re-estimate table cardinality for the query.

In example 19, you can see the difference in the cardinality estimate between a sorted and a ranked index. The estimate from a ranked index is much more accurate.

Example 19:

```
~Estim  Ndx1 Sorted: Split lev=1, Seps=1 Est=3
~Estim  RLEAF Cardinality=  2.0000000E+01
~Estim  Ndx2 Ranked: Nodes=1, Min=20, Est=20 Precise IO=0
~Estim  RLEAF Cardinality=  1.2100000E+03
```

The sorted index gives a table cardinality estimate of 20 rows, while the ranked index gives an estimate of 1210 rows. The actual table cardinality for this query is 1540 rows. The cardinality information in the **sorted ranked** index structure means that, in general, the estimate from a **sorted** index is less useful than that from a **sorted ranked** index.

Estimation For One Background Index

In the past, index estimation was only performed if there was more than one background index. Unfortunately, this meant that for volatile tables, the table cardinality for the query was never updated. This could lead to premature switches to sequential retrieval where the limit was based on an obsolete table cardinality.

In Rdb 7.1.2, index estimation is performed even where there is only one background index.

This ensures that the table cardinality estimate for the query is always based on the best information available.

Together with the better accuracy available from ranked indices this should help ensure that the threshold for the switch to sequential retrieval should be based on the current table cardinality at all times.

In example 20, a user attaches to a database and queries a table containing no rows. On first reference to the table, the user will record the current cardinality.

A second user then inserts approximately 60,000 rows into the table. The first user has no way of identifying the presence of the new rows, so when the table is queried again, the dynamic optimizer

prematurely switches to sequential retrieval. This causes a large amount of I/O to be performed during the FIN phase.

Example 20:

```
~E#0005.01(1) BgrNdx1 FtchLim  DBKeys=1016  Fetches=4+42  RecsOut=0
~E#0005.01(1) Fin     Seq      DBKeys=60451  Fetches=0+2329  RecsOut=26
```

Example 21 shows the effect of the 7.1.2 functionality. The background index is now estimated, and estimation causes the cardinality for this query to be updated. Because the cardinality is more accurate, the threshold for the switch to sequential is now set appropriately, and execution uses the index scan to completion.

Example 21:

```
~Estim  Ndx1 Ranked: Nodes=155, Min=0, Est=3110 IO=1
~Estim  RLEAF Cardinality=  4.3821000E+04
~E#0004.01(1) Estim   Index/Estimate 1/3110
~E#0004.01(1) BgrNdx1 EofBuf   DBKeys=1024  Fetches=3+42  RecsOut=0
~E#0004.01(1) BgrNdx1 EofBuf   DBKeys=2048* Fetches=0+41  RecsOut=0
~E#0004.01(1) BgrNdx1 EofBuf   DBKeys=3072* Fetches=0+40  RecsOut=0
~E#0004.01(1) BgrNdx1 EofData  DBKeys=4058* Fetches=0+38  RecsOut=0 #Bufs=1017
~E#0004.01(1) Fin     TTbl     DBKeys=4058  Fetches=0+1017  RecsOut=26
```

With SET FLAGS ‘EXECUTION,DETAIL(1)’ it can be seen that the index was estimated, and also that estimation gave us an updated cardinality estimate of 4.3821000E+04 or 43,821. This is reasonably close to the actual table cardinality of 60,000 rows.

Enhanced Debugging

Because of the extra functionality in 7.1.2, and because of some confusion in the old debugging information for index estimation, the debug output has been significantly enhanced.

The old output displayed one line with a summary of the estimation process. Apart from the new estimate for each index, this summary line tried to show output meaningful for the index type (sorted or ranked). With the introduction of hashed estimation, this extra information became increasingly cryptic.

So the summary line has been amended to display only the new index order, and the new estimate for each index. Example 22 shows the new summary line format.

Example 22:

```
SQL> set flags 'strategy,execution'
SQL> select count(*) from t2 where f1=50 and f2=1 and f3=1;
~S#0007
Aggregate
Leaf#01 BgrOnly T2 Card=10
  BgrNdx1 I21 [1:1] Fan=1
  BgrNdx2 I22 [1:1] Fan=17
  BgrNdx3 I23 [1:1] Fan=17
~E#0007.01(1) Estim   Index/Estimate 1_3 2/3 3/100
~E#0007.01(1) BgrNdx1 EofData  DBKeys=50  Fetches=0+0  RecsOut=0 #Bufs=1
~E#0007.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0007.01(1) Fin     Buf      DBKeys=50  Fetches=0+0  RecsOut=1~S#0006
```

Notice how the estimation summary line simply shows the index number and the new estimation (or number of dbkeys) for the index.

The underscore "_" notation has been retained to indicate that estimation was not performed, or failed, for that index.

To obtain more complete information, the DETAIL(1) flag can be set. Example 23 shows the same query with DETAIL(1).

Example 23:

```
SQL> set flags 'strategy,detail(1),execution'
SQL> select count(*) from t2 where f1=50 and f2=1 and f3=1;
~S#0008
Tables:
  0 = T2
Aggregate: 0:COUNT (*)
Leaf#01 BgrOnly 0:T2 Card=10
  Bool: (0.F1 = 50) AND (0.F2 = 1) AND (0.F3 = 1)
  BgrNdx1 I21 [1:1] Fan=1
```

```
    Keys: 0.F1 = 50
  BgrNdx2 I22 [1:1] Fan=17
    Keys: 0.F2 = 1
  BgrNdx3 I23 [1:1] Fan=17
    Keys: 0.F3 = 1
~Estim  I21 Hashed: Nodes=0, Est=3 Disabled IO=0
~Estim  I22 Sorted: Split lev=1, Seps=1 Est=3
~Estim  I23 Ranked: Nodes=1, Min=100, Est=100 Precise IO=0
~Estim  RLEAF Cardinality=  2.6830000E+03
~E#0008.01(1) Estim   Index/Estimate 1_3 2/3 3/100
~E#0008.01(1) BgrNdx1 EofData  DBKeys=50  Fetches=0+0  RecsOut=0 #Bufs=1
~E#0008.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0008.01(1) Fin     Buf      DBKeys=50  Fetches=0+0  RecsOut=1
```

The DETAIL(1) flag does cause additional information to be displayed in the strategy when the STRATEGY flag is also set, but it also enhances the execution trace for index estimation.

Ndx1 is a **hashed** index, therefore the output will show the estimated number of duplicate hash buckets (Nodes), the new estimate for this index (Est), the status of this estimation, and the count of I/O's after the estimation on this index completed. In this case, the status is "Disabled" indicating that hashed index estimation is disabled.

For a hashed index the status can be "Disabled", or "IO limit". In either case, no estimate was obtained. If an estimate is actually obtained, "Precise" will be displayed indicating the estimate is precise. Values for hashed indices are always precise.

Ndx2 is a **sorted** (non-ranked) index, therefore the information displayed is consistent with the old format. That is, the split level, the number of separators, and the new estimate. If the estimate is precise, the word "Precise" will also be displayed.

Ndx3 is a ranked index, therefore the output shows the estimated number of level 1 nodes that need to be read (Nodes), the minimum number of rows (Min) that are estimated based on the error in the estimate, and the new estimate (Est).

If the estimate was terminated for some reason, the reason for termination will be displayed. Termination can occur for several reasons:

- Descend IO Limit – The I/O limit rules were enabled, and the descent to the split level reached the I/O limit.

- Refine IO Limit – The refine phase of estimation was enabled, and the refine phase reached the I/O limit.
- True > Mixed – During estimation, the refine phase estimated the cardinality of known true intervals to exceed the cardinality of all mixed branches (see above).
- Error < 10% - When the error in the estimate was calculated during the refine phase, it was found to be less than 10% of the estimate. If this rule is enabled, the estimate will not be refined further.
- Estim block full – During the estimation process, the Estim block became full. If this occurs while processing the index root node, then no estimate is possible. However at lower levels a reasonable estimate will be available from the estimate at the next higher index node level.

If a ranked estimate is considered precise, then "Precise" will also be displayed.

An Effect On Performance Of Index Estimation

One reported problem showed a complex query with several dynamic tactics in a nineteen table join.

The query was performing badly under Rdb 7.0. Rdb version 6.0 was over 400 direct I/O's and version 7.0 was around 1152 direct I/O's. Analysis showed that many of the inner cross blocks that used dynamic tactics frequently select zero or very few rows. With zero or very few rows for each execution of a dynamic tactic, it is obvious that the cost of estimation can be significant to the overall performance of the query.

In the reported case, because of nested loop joins, there were around 40,000 executions of dynamic leaf tactics for the single query.

All indices on the database were of TYPE IS SORTED, so for very few or zero rows, the zero shortcut, and shortcut for 10 or fewer rows was helping to minimize wasted I/O's during index estimation.

Example 24 is a brief extract of a couple of sections from the execution trace for Rdb V7.0.

Example 24:

```
~E#0004.04(428) Estim   Ndx:Lev/Seps/DBKeys 2:1/0/0 1:_1 ZeroShortcut
~E#0004.04(429) Estim   Ndx:Lev/Seps/DBKeys 2:1/0/0 1:_1 ZeroShortcut
~E#0004.04(430) Estim   Ndx:Lev/Seps/DBKeys 2:1/0/0 1:_1 ZeroShortcut
~E#0004.04(431) Estim   Ndx:Lev/Seps/DBKeys 2:1/0/0 1:_1 ZeroShortcut
~E#0004.04(432) Estim   Ndx:Lev/Seps/DBKeys 2:1/0/0 1:_1 ZeroShortcut
~E#0004.04(433) Estim   Ndx:Lev/Seps/DBKeys 1:1/1/1 2:4/1\1195
~E#0004.04(434) Estim   Ndx:Lev/Seps/DBKeys 1:1/1/1 2:_1195
~E#0004.04(435) Estim   Ndx:Lev/Seps/DBKeys 1:1/1/1 2:_1195
~E#0004.04(436) Estim   Ndx:Lev/Seps/DBKeys 1:1/1/1 2:_1195
.
.
.
~E#0004.08(9890) Estim   Ndx:Lev/Seps/DBKeys 2:1/2/2 1:_3
~E#0004.08(9891) Estim   Ndx:Lev/Seps/DBKeys 2:1/2/2 1:1/5/5
~E#0004.08(9892) Estim   Ndx:Lev/Seps/DBKeys 2:1/2/2 1:_5
~E#0004.08(9893) Estim   Ndx:Lev/Seps/DBKeys 2:1/2/2 1:_5
~E#0004.08(9894) Estim   Ndx:Lev/Seps/DBKeys 2:1/2/2 1:_5
~E#0004.08(9895) Estim   Ndx:Lev/Seps/DBKeys 2:1/2/2 1:_5
```

For sorted indices on Rdb version 7.0, the query executed in 1,152 direct I/O's.

As an experiment it was decided to test the use of ranked indices. At the time, ranked indices did not have any of the features described here. The I/O rose to 1973 and CPU time increased dramatically.

It was known from the execution trace with sorted indices, that both zero shortcut, and shortcut for 10 or fewer rows was playing a major part in the performance of the query in question. Most of the observed I/O was being consumed on indices. Buffering also played a significant role.

A review was undertaken to see how I/O could be minimized during execution of dynamic tactics where the number of selected rows was small.

From this reported query, the following issues came to light with respect to estimation on ranked indices.

- The cost of estimating a non-productive index could easily outweigh the cost of retrieval using a productive index.

- In the case of zero shortcut, it was highly probable that the next execution would zero shortcut or find very few rows on the same index.
- Inaccurate estimates based on high split levels could erroneously imply that a useful index was less effective.
- Index estimation on ranked indices was inherently limited, and could produce inaccurate results.

The last was a matter of first priority, and the ranked estimation code was completely re-written for both Rdb 7.0 and 7.1. This alone reduced the I/O to 700 I/O's.

Next the REFINE_ESTIMATES flag was used to enable estimation refinement on ranked indices, and the I/O dropped to 504 I/O's.

By converting to Rdb 7.1, the zero shortcut reordering eliminated a further 20 data I/O's, but addition metadata reads balanced this out.

The net effect of the changes was that Rdb 7.1 with ranked indices and refinement rules could perform the same query with fewer data reads than Rdb 6.0 with sorted indices. The difference in observed I/O can be attributed to increased metadata access during database attach and request compilation.

The most important element was to minimize the I/O for queries that select a small number of rows. For queries that select a large number of rows, the cost of estimating and scanning additional indices is far outweighed by the cost of retrieving the data rows themselves.

How to Use and Monitor the New Features

The features described in this paper are only used by the dynamic optimizer, or by the static optimizer when the **sampled selectivity**[2] feature is enabled.

The dynamic optimizer is enabled by default, and it is possible that many queries may use this feature even without your knowledge. When examining a retrieval strategy either using the SET FLAGS 'STRATEGY' statement or by defining the logical name RDMS$DEBUG_FLAGS "S", the presence of a "Leaf" in the strategy indicates that the dynamic optimizer is being used.

Also keep in mind that there are cases for which index estimation is not used:

[2] The sampled selectivity feature is described in the Oracle Rdb V7.1.2 Release Notes.

1. For an index of **type is sorted**, estimation is not performed if the query is a range list or the index is partitioned.
2. For an index of **type is sorted ranked**, estimation is not performed if the index is partitioned.

The default behavior is to estimate only sorted indices, and only descend to the split level.

Example 25 shows the default behavior on a relatively trivial query.

Example 25:

```
SQL> set flags 'strategy,detail(1),execution'
SQL> select count(*) from t2
cont>   where f1 between 1 and 3
cont>   and (f2=1 or f2=10000)
cont>   and f3 in (1,10000,20000,30000);
~S#0003
Tables:
  0 = T2
Aggregate: 0:COUNT (*)
Leaf#01 BgrOnly 0:T2 Card=100000
  Bool: (0.F1 >= 1) AND (0.F1 <= 3) AND ((0.F2 = 1) OR (0.F2 = 10000)) AND ((
        0.F3 = 1) OR (0.F3 = 10000) OR (0.F3 = 20000) OR (0.F3 = 30000))
  BgrNdx1 I22 [(1:1)2] Fan=17
    Keys: r0: 0.F2 = 10000
          r1: 0.F2 = 1
  BgrNdx2 I23 [(1:1)4] Fan=1
    Keys: r0: 0.F3 = 30000
          r1: 0.F3 = 20000
          r2: 0.F3 = 10000
          r3: 0.F3 = 1
  BgrNdx3 I21 [1:1] Fan=17
    Keys: (0.F1 >= 1) AND (0.F1 <= 3)
~Estim  I22 Ranked: Nodes=309, Min=0, Est=6205 IO=1
~Estim  RLEAF Cardinality=  1.0000000E+05
~Estim  I23 Hashed: Nodes=0, Est=3 Disabled IO=1
~Estim  I21 Sorted: Split lev=4, Seps=1 Est=5534
```

```
~E#0003.01(1) Estim   Index/Estimate 2_3 3/5534 1/6205
~E#0003.01(1) BgrNdx2 EofData  DBKeys=4  Fetches=2+6  RecsOut=0 #Bufs=4
~E#0003.01(1) BgrNdx3 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0003.01(1) Fin     Buf      DBKeys=4  Fetches=0+4  RecsOut=0

           0
1 row selected
```

Background index 1 is a ranked index I22 on the field F2. The query specifies that "F2=1 or F2 = 100000". For a ranked index the normal behavior is to descend the index until the selected range(s) span more than one branch in the index node.

It is easy to see that the two key values 1 and 100,000 are very likely to be down different branches from the root node. Each of these key values only occurs once, so the correct estimate should be 2 rows, but the "~Estim" trace line for Ndx1 shows an estimate of 6205 rows.

Background index 2 is a hashed index. As the "~Estim" trace line shows, estimation on hashed indices is disabled by default, so this index is not estimated. Instead, the static optimizers guess of 3 rows is retained as the estimate.

Background index 3 is a sorted index on the field F1. Sorted index estimation descends to the split level. In this case the "~Estim" trace line shows that the split level was level four resulting in an estimate of 5,534 rows. This happened because the range of two key values happened to span a separator in the index root node.

To enable the new features simply add up the REFINE_ESTIMATES value corresponding to the features that are required. To obtain the very best estimate, regardless of cost, use REFINE_ESTIMATES(16) to enable refinement of ranked estimates, and REFINE_ESTIMATES(32) to enable hashed index estimation. Without any other rules estimation attempts to be as precise as possible regardless of cost. Combining these value using logical OR yields 48, and so SET FLAGS 'REFINE_ESTIMATES(48)' should be used.

Example 26 shows the result of requesting precise estimates. Because the query and strategy are the same as example 23, they have been cut from the example for brevity.

Example 26:

```
SQL> set flags 'refine_estimates(48),strategy,detail,execution'
SQL> select ...
~Estim  I22 Ranked: Nodes=2, Min=2, Est=2 Precise IO=6
~Estim  RLEAF Cardinality=  1.0000000E+05
~Estim  I23 Hashed: Nodes=0, Est=4 Precise IO=14
~Estim  I21 Sorted: Split lev=4, Seps=1 Est=5534
~E#0003.01(1) Estim   Index/Estimate 1/2 2/4 3/5534
~E#0003.01(1) BgrNdx1 EofData  DBKeys=2  Fetches=0+0  RecsOut=0 #Bufs=2
~E#0003.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0003.01(1) Fin     Buf      DBKeys=2  Fetches=0+2  RecsOut=0
```

Observe that the sorted index estimate remains unchanged at 5534, since there is nothing the optimizer can do to improve this estimate.

The ranked index estimate is now precisely two. Estimation refinement has descended the two different branches for the key values 1 and 100,000 to find that only one row exists for each key value.

The hashed index has now been estimated, and obtained a precise estimate of 4. Estimation has performed a hash lookup for each key value in the “IN” clause and found one row for each key value.

Notice that the last index to be estimated, the hashed index, reports an I/O count of 14. Sorted index estimation does not count towards the I/O count. So it took 14 I/O’s to estimate the ranked and hashed indices. This seems somewhat excessive considering it would most likely take two I/O’s to read the two rows identified by the best index. The best index is the sorted ranked index with an estimate of two.

To prevent excessive and wasted I/O, enable the limiting rules for index estimation. Example 27 shows the same query with refinement rules in place. Again the query and strategy are the same and have been cut from the example.

In example 27 the REFINE_ESTIMATES flag has been set to the value of 111. This value corresponds to enabling all refinement rules except for the rule 5. Rule 5 is only needed when a precise estimate is requested, and no other rules are enabled. In this way the value is calculated as 1+2+4+8+32+64=111. The value 16 is missing from the calculation as it corresponds to rule 5.

It would have made no difference had the additional rule been enabled. So we could equally well have used the value of 1+2+4+8+16+32+64=127.

Example 27:

```
SQL> set flags 'refine_estimates(111),strategy,detail,execution'
SQL> select ...
~Estim  I22 Ranked: Nodes=2, Min=2, Est=2 Precise IO=6
~Estim  RLEAF Cardinality=  1.0000000E+05
~Estim  I23 Hashed: Nodes=0, Est=3 IO limit IO=6
~Estim  I21 Sorted: Split lev=4, Seps=1 Est=5534
~E#0003.01(1) Estim   Index/Estimate 1/2 2_3 3/5534
~E#0003.01(1) BgrNdx1 EofData  DBKeys=2  Fetches=0+0  RecsOut=0 #Bufs=2
~E#0003.01(1) BgrNdx2 FtchLim  DBKeys=0  Fetches=0+0  RecsOut=0
~E#0003.01(1) Fin     Buf      DBKeys=2  Fetches=0+2  RecsOut=0
```

Now the ranked index is estimated, obtaining a precise estimate of two rows. Observe that 6 I/O's were expended to obtain this estimate.

The hashed index has not been estimated because the I/O limit will be set to the minimum estimate so far which is 2 I/O's, but already 6 I/O's have been used. The underscore in the summary line shows that a new estimate was not obtained, and the "~Estim" line shows the reason is "IO limit".

The sorted index estimation is still performed because sorted indices do not obey I/O limit rules. The estimation is still 5,534 due to the high split level.

If the same query is again executed, some of the index nodes will remain in our buffers. Therefore, estimation proceeds further than before. Example 28 shows the execution trace for a second execution of the same request.

Example 28:

```
~Estim  I22 Ranked: Nodes=2, Min=2, Est=2 Precise IO=0
~Estim  RLEAF Cardinality=  1.0000000E+05
~Estim  I23 Hashed: Nodes=0, Est=3 IO limit IO=3
~Estim  I21 Sorted: Split lev=4, Seps=1 Est=5534
~E#0004.01(1) Estim   Index/Estimate 1/2 2_3 3/5534
```

Notice that the ranked index estimation needed no I/O's for estimation since the index nodes remained in our buffer pool.

The hashed index estimation performed 3 I/O's. At which point the I/O count (3) exceeded the I/O limit based on the ranked index estimate (2). The hashed index estimation was not complete, so no new estimate was obtained.

Example 29 shows a final execution of the same request where 3 more I/O's were expended estimating the hashed index, and a valid estimate was finally obtained. Thus, the refinement rules for index estimation attempt to strike a balance between the costs of the estimation process compared to that of retrieving rows, versus the benefits of estimation.

Example 29:

```
~Estim  I22 Ranked: Nodes=2, Min=2, Est=2 Precise IO=0
~Estim  RLEAF Cardinality=  1.0000000E+05
~Estim  I23 Hashed: Nodes=0, Est=3 Precise IO=3
~Estim  I21 Sorted: Split lev=4, Seps=1 Est=5534
~E#0005.01(1) Estim   Index/Estimate 1/2 2/3 3/5534
```

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 indices are formed into a background index list.

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

Bitmap 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.

Duplicity factor: The average number of rows with the same key value in an index.

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

Estim Block: An in memory data structure used to record critical information during the estimation process.

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.

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

False Interval: A range of key values that does not match the selection criteria for an index. Rows in this key range will not be selected from this index.

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

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

Index Estimation: See Estimation.

Interval: Range of key values.

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.

Mixed Interval: A range of key values only some of which may match the selection criteria for an index. Rows in this key value range may or may not be selected from this index..

Optimizer: That component of Rdb that determines the access strategy for a given query.

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

Precise Estimate: An estimate is precise when we can determine from the index structure exactly how many rows will be selected from that index.

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

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.

Refine: During index estimation, to read further down the index structure to obtain a more accurate estimate.

Request: A database request is a statement that manipulates or retrieves data. Typically a single SQL statement.

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.

Split Level: The level in an index where a selected key value range spans more than one entry in the index node.

True Interval: A range of key values that is known to match the selection criteria for an index. Rows in this key value range will be selected from this index.

With Hold: A cursor that uses the "with hold" clause can remain open between transactions.

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

Oracle Rdb
Guide to Database Design and Tuning: Predicate Estimation
August 2003

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