Enhancements to indexing

252-byte index length limit is gone

A. Brinkman

New and reworked index code is very fast and tolerant of large numbers of duplicates. The old aggregate key length limit of 252 bytes is removed. Now the limit depends on page size: the maximum size of the key in bytes is 1/4 of the page size (512 on 2048, 1024 on 4096, etc.).

A 40-bit record number is included on “non leaf-level pages” and duplicates (key entries) are sorted by this number.

Expression indexes

O. Loa, D. Yemanov, A. Karyakin

Arbitrary expressions applied to values in a row in dynamic DDL can now be indexed, allowing indexed access paths to be available for search predicates that are based on expressions.

Syntax pattern

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX <index name>
  ON <table name>
  COMPUTED BY ( <value expression> )
```

Examples

1.

```
CREATE INDEX IDX1 ON T1
   COMPUTED BY ( UPPER(COL1 COLLATE PXW_CYRL) );
COMMIT;
/**/
SELECT * FROM T1
   WHERE UPPER(COL1 COLLATE PXW_CYRL) = 'ÔÛÂÀ'
 -- PLAN (T1 INDEX (IDX1))
```

2.

```
CREATE INDEX IDX2 ON T2
   COMPUTED BY ( EXTRACT(YEAR FROM COL2) || EXTRACT(MONTH FROM COL2) );
COMMIT;
/**/
SELECT * FROM T2
   ORDER BY EXTRACT(YEAR FROM COL2) || EXTRACT(MONTH FROM COL2)
 -- PLAN (T2 ORDER IDX2)
```

Note:

1. The expression used in the predicate must match exactly the expression used in the index declaration, in order to allow the engine to choose an indexed access path. The given index will not be available for any retrieval or sorting operation if the expressions do not match.
2. Expression indices have exactly the same features and limitations as regular indices, except that, by definition, they cannot be composite (multi-segment).

back to top of page

Changes to NULL keys handling

V. Horsun, A. Brinkman

- Null keys are now bypassed for uniqueness checks. (V. Horsun)

If a new key is inserted into a unique index, the engine skips all NULL keys before starting to check for key duplication. It means a performance benefit as, from v.1.5 on, NULLs have not been considered as duplicates.

- NULLs are ignored during the index scan, when it makes sense to ignore them. (A. Brinkman).

Prevously, NULL keys were always scanned for all predicates. Starting with v.2.0, NULL keys are usually skipped before the scan begins, thus allowing faster index scans.

Note: The predicates IS NULL and IS NOT DISTINCT FROM still require scanning of NULL keys and they disable the aforementioned optimization.

Improved index compression

A. Brinkman

A full reworking of the index compression algorithm has made a manifold improvement in the performance of many queries.

Selectivity maintenance per segment

D. Yemanov, A. Brinkman

Index selectivities are now stored on a per-segment basis. This means that, for a compound index on columns (A, B, C), three selectivity values will be calculated, reflecting a full index match as well as all partial matches. That is to say, the selectivity of the multi-segment index involves those of segment A alone (as it would be if it were a single-segment index), segments A and B combined (as it would be if it were a double-segment index) and the full three-segment match (A, B, C), i.e., all the ways a compound index can be used.

This opens more opportunities to the optimizer for clever access path decisions in cases involving partial index matches.

The per-segment selectivity values are stored in the column RDB$STATISTICS of table RDB$INDEX_SEGMENTS. The column of the same name in RDB$INDICES is kept for compatibility and still represents the total index selectivity, that is used for a full index match.

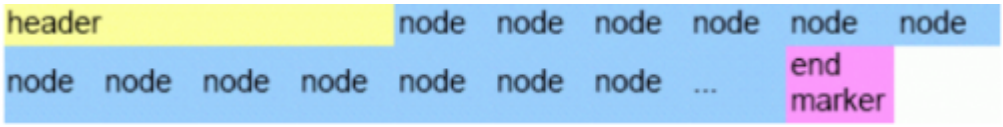
back to top of page

Firebird index structure from ODS11 onward

The aims achieved by the new structure were:

- better support for deleting an index key out of many duplicates (caused slow garbage collection),
- support for bigger record numbers than 32-bits (40 bits),
- to increase index-key size (1/4 page-size).

Figure 8.1. Existing structure (ODS10 and lower)



header =

```
typedef struct btr {
struct pag btr_header;
   SLONG btr_sibling; // right sibling page
   SLONG btr_left_sibling; // left sibling page
   SLONG btr_prefix_total; // sum of all prefixes on page
   USHORT btr_relation; // relation id for consistency
   USHORT btr_length; // length of data in bucket
   UCHAR btr_id; // index id for consistency
   UCHAR btr_level; // index level (0 = leaf)
   struct btn btr_nodes[1];
};
```

node =

```
struct btn {
   UCHAR btn_prefix; // size of compressed prefix
   UCHAR btn_length; // length of data in node
   UCHAR btn_number[4]; // page or record number
   UCHAR btn_data[1];
```

};

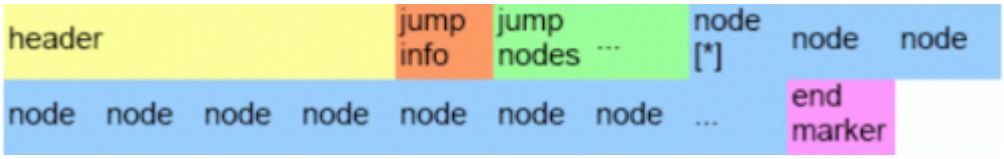
end marker = END_BUCKET or END_LEVEL

These are in place of record number for leaf nodes and in place of page number for non-leaf nodes.

If the node is a END_BUCKET marker then it should contain the same data as the first node on the next sibling page.

On an END_LEVEL marker prefix and length are zero, thus it contains no data. Also, every first node on a level (except leaf pages) contains a degeneration zero-length node.

Figure 8.2. New ODS11 structure



jump info =

```
  struct IndexJumpInfo {
     USHORT firstNodeOffset; // offset to first node in page [*]
     USHORT jumpAreaSize; // size area before a new jumpnode is made
     UCHAR jumpers; // nr of jump-nodes in page, with a maximum of 255
  };
```

jump node =

```
 struct IndexJumpNode {
    UCHAR* nodePointer; // pointer to where this node can be read from the
page
    USHORT prefix;      // length of prefix against previous jump node
    USHORT length;      // length of data in jump node (together with prefix
this
                        // is prefix for pointing node)
    USHORT offset;      // offset to node in page
    UCHAR* data;        // Data can be read from here
 };
```

back to top of page

New flag for the new index structure

New flags are added to the header → pag_flags.

The flag btr_large_keys (32) is for storing compressed length/prefix and record-number. This meant also that length and prefix can be up to 1/4 of page-size (1024 for 4096 page-size) and is easy extensible in the future without changing disk-structure again.

Also the record-number can be easy extended to for example 40 bits. Those numbers are stored per 7-bits with 1 bit (highest) as marker (variable length encoding). Every new byte that needs to be stored is shifted by 7.

Examples

25 is stored as 1 byte 0x19, 130 = 2 bytes 0x82 0x01, 65535 = 3 bytes 0xFF 0xFF 0x03.

Duplicate nodes

A new flag is also added for storing the record number on every node (non-leaf pages). This speeds up index retrieval on many duplicates. The flag is btr_all_recordnumber (16).

With this added information, key lookup on inserts/deletes with many duplicates (NULLs in foreign keys, for example) becomes much faster (such as the garbage collection!).

Beside that duplicate nodes (length = 0) don't store their length information, 3 bits from the first stored byte are used to determine if this nodes is a duplicate.

Beside the ZERO_LENGTH (4) there is also END_LEVEL (1), END_BUCKET (2), ZERO_PREFIX_ZERO_LENGTH (3) and ONE_LENGTH (5) marker. Number 6 and 7 are reserved for future use.

back to top of page

Jump nodes

A jump node is a reference to a node somewhere in the page.

It contains offset information about the specific node and the prefix data from the referenced node, but prefix compression is also done on the jump nodes themselves.

Ideally a new jump node is generated after the first node that is found after every jumpAreaSize, but that's only the case on deactivate/active an index or inserting nodes in the same order as they will be stored in the index.

If nodes are inserted between two jump node references only the offsets are updated, but only if the offsets don't exceed a specific threshold (+/-10 %).

When a node is deleted only offsets are updated or a jump node is removed. This means a little hole can exist between the last jump node and the first node, so we don't waste time on generating new jump nodes.

The prefix and length are also stored by variable length encoding.

Figure 8.3. Example data ((x) = size in x bytes)

header (34)				
52 (2)	256 (2)	2 (1)	30 (2)	0 (1)
2 (1)	260 (2)	FI (2)	1 (1)	1 (1)
514 (2)	U (1)	0 (1)	1 (1)	0 (1)
A (1)	...			
2 (1)	6 (1)	21386 (3)	REBIRD (6)	...
2 (1)	2 (1)	1294 (2)	EL (2)	...

Pointer after fixed header = 0x22

Pointer after jump info = 0x29

Pointer to first jump node = 0x29 + 6 (jump node 1) + 5 (jump node 2) = 0x34

Jump node 1 is referencing to the node that represents FIREBIRD as data, because this node has a prefix of 2 the first 2 characters FI are stored also on the jump node.

Our next jump node points to a node that represents FUEL with also a prefix of 2. Thus jump node 2 should contain FU, but our previous node already contained the F so, due to prefix compression, this one is ignored and only U is stored.

back to top of page

NULL state

The data that needs to be stored is determined in the procedure compress() in btr.cpp.

For ASC (ascending) indexes no data will be stored (key is zero length). This will automatically put them as first entry in the index and thus correct order (For single field index node length and prefix is zero).

DESC (descending) indexes will store a single byte with the value 0xFF (255). To distinguish between a value (empty string can be 255) and an NULL state we insert a byte of 0xFE (254) at the front of the data. This is only done for values that begin with 0xFF (255) or 0xFE (254), so we keep the right order.

Figure 8.4. Examples

nodes ASC index, 1 segment			
prefix	length	stored data	real value/state
0	0		NULL
0	0		NULL
0	1	x65 (A)	A
1	1	x65 (A)	AA
...	...	...	...

nodes DESC index, 1 segment			
prefix	length	stored data	real value/state
...	...	...	...
0	2	xFE xFE (þ) x4A (J)	0xFE 0x4A
1	1	xFF (ÿ)	0xFF
0	1	xFF	NULL
1	0	xFF	NULL
			END_LEVEL

nodes ASC index, 3 segment			
prefix	length	stored data	real value/state
0	0		NULL, NULL, NULL
0	10	x01(1) x70(F) x73(I) x82(R) x69(E) x01(1) x66(B) x73(I) x82(R) x68(D)	NULL, NULL, FIREBIRD
0	10	x02(2) x70(F) x73(I) x82(R) x69(E) x02(2) x66(B) x73(I) x82(R) x68(D)	NULL, FIREBIRD, NULL
0	10	x03(3) x70(F) x73(I) x82(R) x69(E) x03(3) x66(B) x73(I) x82(R) x68(D)	FIREBIRD, NULL, NULL
3	9	x00(0) x00(0) x02(2) x65(A) x00(0) x00(0) x00(0) x01(1) x66(B)	FI, A, B
...	...	...	...

nodes DESC index, 3 segment			
prefix	length	stored data	real value/state
0	12	xFC xB9 xB6 xFF xFF xFD xBE xFF xFF xFF xFE xBD	FI, A, B
3	17	xAD xBA xFC xBD xB6 xAD xBB xFD xFF xFF xFF xFF xFE xFF xFF xFF xFF	FIREBIRD, NULL, NULL
1	19	xFF xFF xFF xFF xFD xB9 xB6 xAD xBA xFD xBD xB6 xAD xBB xFE xFF xFF xFF xFF	NULL, FIREBIRD, NULL
6	14	xFF xFF xFF xFF xFE xB9 xB6 xAD xBA xFE xBD xB6 xAD xBB	NULL, NULL, FIREBIRD
11	4	xFF xFF xFF xFF	NULL, NULL, NULL
			END_LEVEL