Jeremy Cole

Geek, electronics nerd, database nerd, aviation nerd, father of three.

The physical structure of InnoDB index pages

In On learning InnoDB: A journey to the core, I introduced the innodb_diagrams project to document the InnoDB internals, which provides the diagrams used in this post. (Note that each image below is linked to a higher resolution version of the same image.)

The basic structure of the space and each page was described in The basics of InnoDB space file layout, and we'll now take a deeper look into how `INDEX` pages are physically structured. This will lay the ground work to discuss indexes at a logical (and much higher) level.

Everything is an index in InnoDB

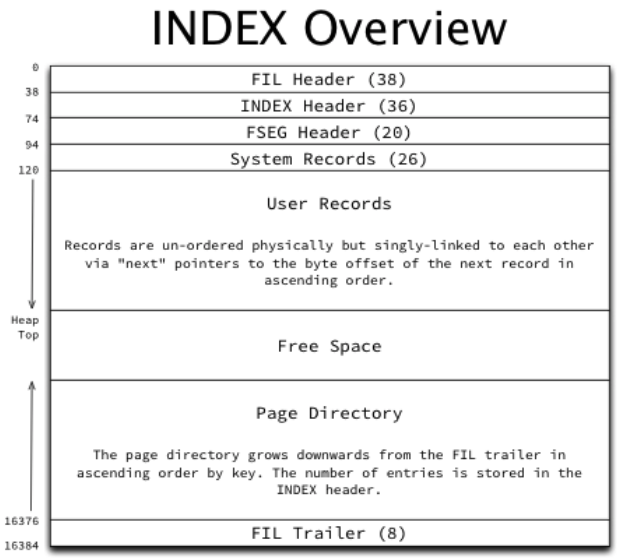
Before diving into physical structures, it's critical to understand that in InnoDB, everything is an index. What does that mean to the physical structure?

1. Every table *has* a primary key; if the `CREATE TABLE` does not specify one, the first non-`NULL` unique key is used, and failing that, a 48-bit hidden "Row ID" field is automatically added to the table structure and used as the primary key. *Always* add a primary key yourself. The hidden one is useless to you but still costs 6 bytes per row.
2. The "row data" (non-`PRIMARY KEY` fields) are stored in the `PRIMARY KEY` index structure, which is also called the "clustered key". This index structure is keyed on the `PRIMARY KEY` fields, and the row data is the value attached to that key (as well as some extra fields for MVCC).
3. Secondary keys are stored in an identical index structure, but they are keyed on the `KEY` fields, and the primary key value (PKV) is attached to that key.

When discussing "indexes" in InnoDB (as in this post), this actually means both *tables* and *indexes* as the DBA may think of them.

Overview of `INDEX` page structure

Every index page has an overall structure as follows:

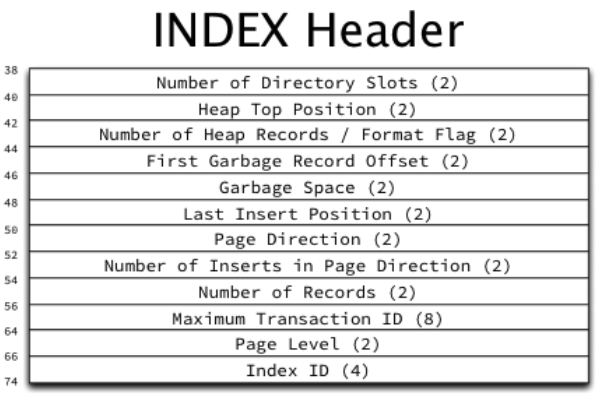


The major sections of the page structure are (not in order):

- The `FIL` header and trailer: This is typical and common to all page types. One aspect somewhat unique to `INDEX` pages is that the "previous page" and "next page" pointers in the `FIL` header point to the previous and next pages *at the same level* of the index, and in ascending order based on the index's key. This forms a doubly-linked list of all pages at each level. This will be further described in the logical index structure.
- The `FSEG` header: As described in Page management in InnoDB space files, the index root page's `FSEG` header contains pointers to the file segments used by this index. All other index pages' `FSEG` headers are unused and zero-filled.
- The `INDEX` header: Contains many fields related to `INDEX` pages and record management. Fully described below.
- System records: InnoDB has two system records in each page called `infimum` and `supremum`. These records are stored in a fixed location in the page so that they can always be found directly based on byte offset in the page.
- User records: The actual data. Every record has a variable-width header and the actual column data itself. The header contains a "next record" pointer which stores the offset to the next record within the page in ascending order, forming a singly-linked list. Details of user record structure will be described in a future post.
- The page directory: The page directory grows downwards from the "top" of the page starting at the `FIL` trailer and contains pointers to some of the records in the page (every 4th to 8th record). Details of the page directory logical structure and its purpose will be described in a future post.

The `INDEX` header

The `INDEX` header in each `INDEX` page is fixed-width and has the following structure:



The fields stored in this structure are (not in order):

- Index ID: The ID of the index this page belongs to.
- Format Flag: The format of the records in this page, stored in the high bit (`0x8000`) of the "Number of Heap Records" field. Two values are possible: `COMPACT` and `REDUNDANT`. Described fully below.
- Maximum Transaction ID: The maximum transaction ID of any modification to any record in this page.
- Number of Heap Records: The total number of records in the page, including the `infimum` and `supremum` system records, and garbage (deleted) records.
- Number of Records: The number of non-deleted user records in the page.
- Heap Top Position: The byte offset of the "end" of the currently used space. All space between the heap top and the end of the page directory is free space.
- First Garbage Record Offset: A pointer to the first entry in the list of garbage (deleted) records. The list is singly-linked together using the "next record" pointers in each record

header. (This is called “free” within InnoDB, but this name is somewhat confusing.)

- Garbage Space: The total number of bytes consumed by the deleted records in the garbage record list.
- Last Insert Position: The byte offset of the record that was last inserted into the page.
- Page Direction: Three values are currently used for page direction: `LEFT`, `RIGHT`, and `NO_DIRECTION`. This is an indication as to whether this page is experiencing sequential inserts (to the left [lower values] or right [higher values]) or random inserts. At each insert, the record at the last insert position is read and its key is compared to the currently inserted record’s key, in order to determine insert direction.
- Number of Inserts in Page Direction: Once the page direction is set, any following inserts that don’t reset the direction (due to their direction differing) will instead increment this value.
- Number of Directory Slots: The size of the page directory in “slots”, which are each 16-bit byte offsets.
- Page Level: The level of this page in the index. Leaf pages are at level 0, and the level increments up the B+tree from there. In a typical 3-level B+tree, the root will be level 2, some number of internal non-leaf pages will be level 1, and leaf pages will be level 0. This will be discussed in more detail in a future post as it relates to logical structure.

Record format: redundant versus compact

The `COMPACT` record format is new in the Barracuda table format, while the `REDUNDANT` record format is the original one in the Antelope table format (neither of which had a name officially until Barracuda was created). The `COMPACT` format mostly eliminated information that was redundantly stored in each record and can be obtained from the data dictionary, such as the number of fields, which fields are nullable, and which fields are dynamic length.

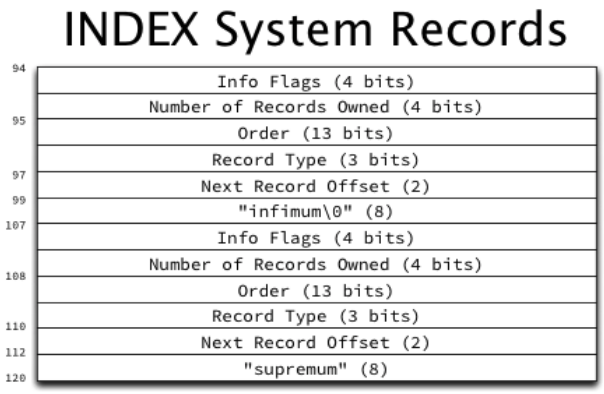
An aside on record pointers

Record pointers are used in several different places: the Last Insert Position field in the `INDEX` header, all values in the page directory, and the “next record” pointers in the system records and user records. All records contain a header (which may be variable-width) followed by the actual record data (which may also be variable-width). Record pointers point to the location of the first byte of *record data*, which is effectively “between” the header and the record data. This allows the header to be read by reading backwards from that location, and the record data to be read forward from that location.

Since the “next record” pointer in system and user records is always the first field in the header reading backwards from this pointer, this means it is also possible to very efficiently read through all records in a page without having to parse the variable-width record data at all.

System records: infimum and supremum

Every `INDEX` page contains two system records, called infimum and supremum, at fixed locations (offset 99 and offset 112 respectively) within the page, with the following structure:



The two system records have a typical record header preceding their location, and the literal strings “infimum” and “supremum” as their only data. A full description of record header fields will be provided in a future post. For now, it is important primarily to observe that the first field (working backwards from the record data, as previously described) is the “next record” pointer.

The infimum record

The infimum record represents a value lower than any possible key in the page. Its “next record” pointer points to the user record with the lowest key in the page. Infimum serves as a fixed entry point for sequentially scanning user records.

The supremum record

The supremum record represents a key higher than any possible key in the page. Its “next record” pointer is always zero (which represents NULL, and is always an invalid position for an actual record, due to the page headers). The “next record” pointer of the user record with the highest key on the page always points to supremum.

User records

The actual on-disk format of user records will be described in a future post, as it is fairly complex and will require a lengthy explanation itself.

User records are added to the page body in the order they are inserted (and may take existing free space from previously deleted records), and are singly-linked in ascending order by key using the “next record” pointers in each record header. A singly-linked list is formed from infimum, through all user records in ascending order, and terminating at supremum. Using this list, it is trivial to scan in through all user records in a page in ascending order.

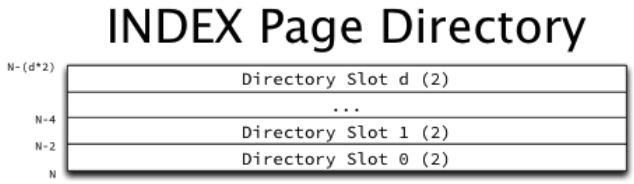
Further, using the “next page” pointer in the `INDEX` header, it’s easy to scan from page to page through the entire index in ascending order. This means an ascending-order table scan is also trivial to implement:

1. Start at the first (lowest key) page in the index. (This page is found through B+tree traversal, which will be described in a future post.)
2. Read infimum, and follow its “next record” pointer.
3. If the record is supremum, proceed to step 5. If not, read and process the record contents.
4. Follow the “next record” pointer and return to step 3.
5. If the “next page” pointer points to NULL, exit. If not, follow the “next page” pointer and return to step 2.

Since records are singly-linked rather than doubly-linked, traversing the index in descending order is not as trivial, and will be discussed in a future post.

The page directory

The page directory starts at the `FIL` trailer and grows "downwards" from there towards the user records. The page directory contains a pointer to every 4-8 records, in addition to always containing an entry for infimum and supremum.



The page directory is simply a dynamically-sized array of 16-bit offset pointers to records within the page. Its purpose will be more fully described in a future post dedicated to the page directory.

Free space

The space between the user records (which grow "upwards") and the page directory (which grows "downwards") is considered free space. Once the two sections meet in the middle, exhausting the free space (and assuming no space can be reclaimed by re-organizing to remove the garbage) the page is considered full.

What's next?

Next we'll look at the logical structure of an index, including some examples.

Related

How does InnoDB behave without a Primary Key?
In "InnoDB"

B+Tree index structures in InnoDB
In "InnoDB"

Efficiently traversing InnoDB B+Trees with the page directory
In "InnoDB"

Posted on January 7, 2013 by Jeremy Cole. This entry was posted in InnoDB, MySQL. Bookmark the permalink.

8 thoughts on "The physical structure of InnoDB index pages"

Daniël van Eeden
— January 10, 2013 at 02:50

Why are the only flags COMPACT and REDUNDANT? What about COMPRESSED? I already noticed in an earlier post that the page size is fixed at 16KiB, So I assume this is not the latest InnoDB/MySQL version?

Reply

Jeremy Cole
— January 10, 2013 at 02:58

I've just not gotten to compression. I intend to cover it at a later point, because it just expands on the non-compressed information, rather than conflicting outright with it.

(Compression is also fairly buggy and not in that wide deployment yet.)

Reply

Andy
— January 16, 2013 at 17:59

When a query requires an index, does innodb load the entire index into buffer pool? or just load part of the index? Thanks!

Reply

Jeremy Cole
— January 16, 2013 at 21:24

Andy: Pages are loaded only as-needed. The root of the index is always in cache for each open table; beyond that only what has been accessed so far will be in cache.

Reply

Calvin Sun
— January 29, 2013 at 20:17

The following statement is incorrect "The COMPACT record format is new in the Barracuda table format". In fact, the COMPACT record format has been supported since 5.0.3. Both REDUNDANT and COMPACT are part of "Antelope", while DYNAMIC and COMPRESS are new in "Barracuda". See http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_compact_row_format

Reply

Bob Wall
— February 3, 2013 at 11:22

Fantastic information – thanks much for putting it together! I was trying to figure out the details of InnoDB indices a while ago, and was having no luck. (Too lazy to go sift through the source code :)

Still trying to sort out how Multi-Version Concurrency Control fits into the picture, but this is a great start.

Reply

woonhak
— April 11, 2013 at 16:29

Great work!

I found there's one typo,

Index ID in the IDX_HDR is 8 bytes not 4 bytes.

Reply

Jeremy Cole

— April 11, 2013 at 16:34

Right you are! I will get it corrected soon (will need to regenerate the image).

Reply

What do you think?

Enter your comment here...