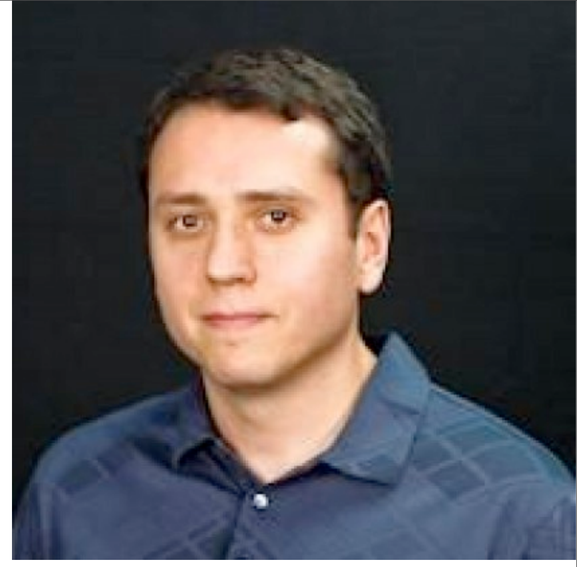
InnoDB:
A journey to the core

Jeremy Cole and Davi Arnaut

Jeremy Cole
@jeremycole

Making MySQL Awesome at Google
Worked at MySQL 2000-2004
Contributor since 3.23
14 years in the MySQL community
Code, documentation, research, bug reports
Yahoo!, Proven Scaling, Gazillion, Twitter

Davi Arnaut
@darnaut



MySQL Internals development at LinkedIn
Worked at MySQL 2007-2011
Designed and built Twitter MySQL
Long time Open Source contributor: Apache, Linux kernel, etc.

About this work...
blog.jcole.us/innodb

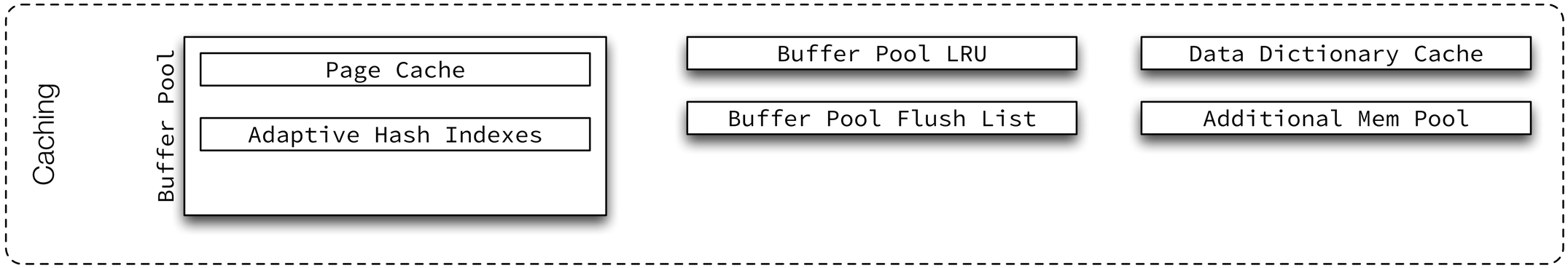
Not intended to be comprehensive
Not authoritative (it is based on research)
One of the best sources of documentation for InnoDB formats

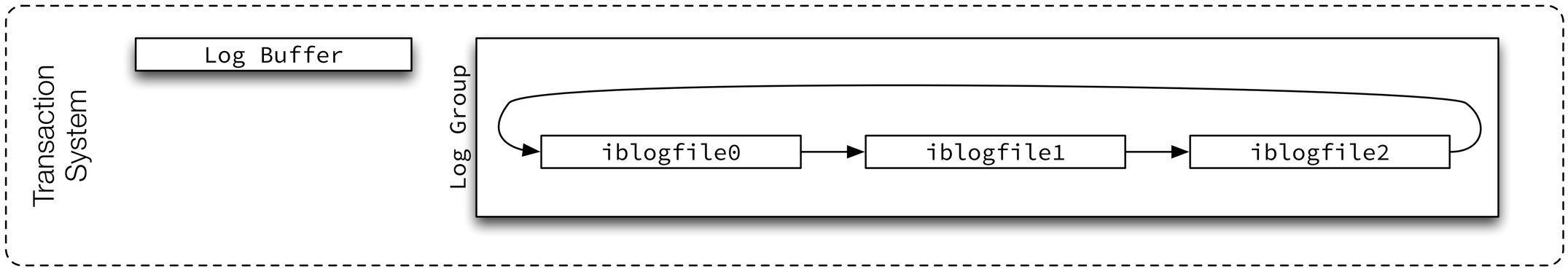
Approach:

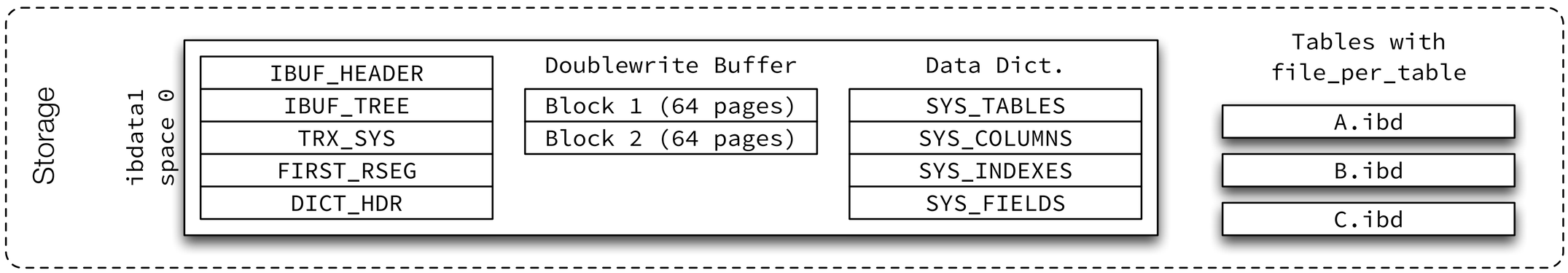
1. Read the C and C++ sources
2. Implement in Ruby
3. Refactor and correct until reasonable
4. Document!

Overview of InnoDB Storage

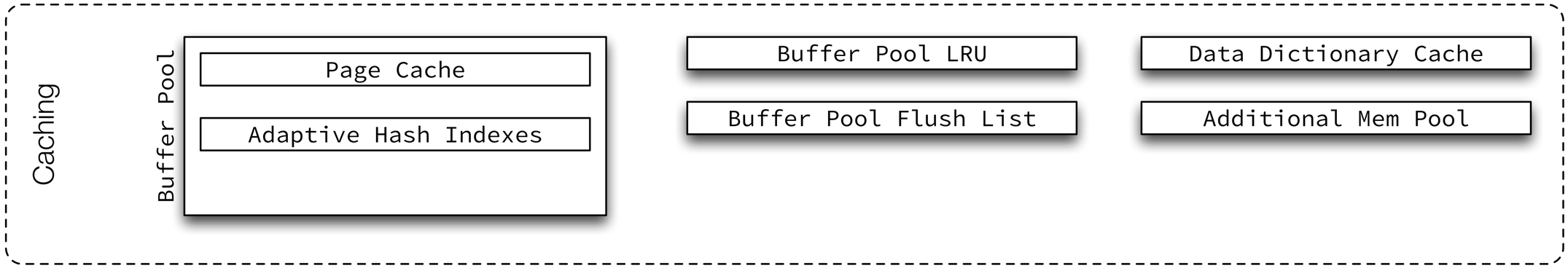
High-level Overview

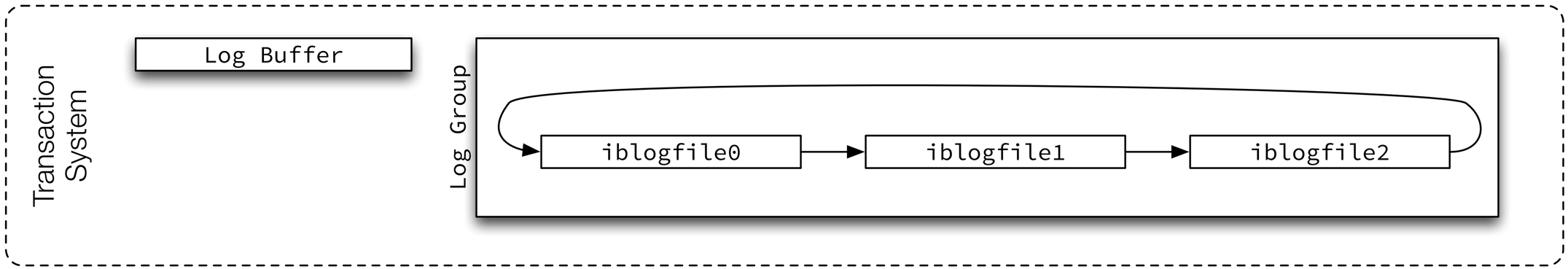


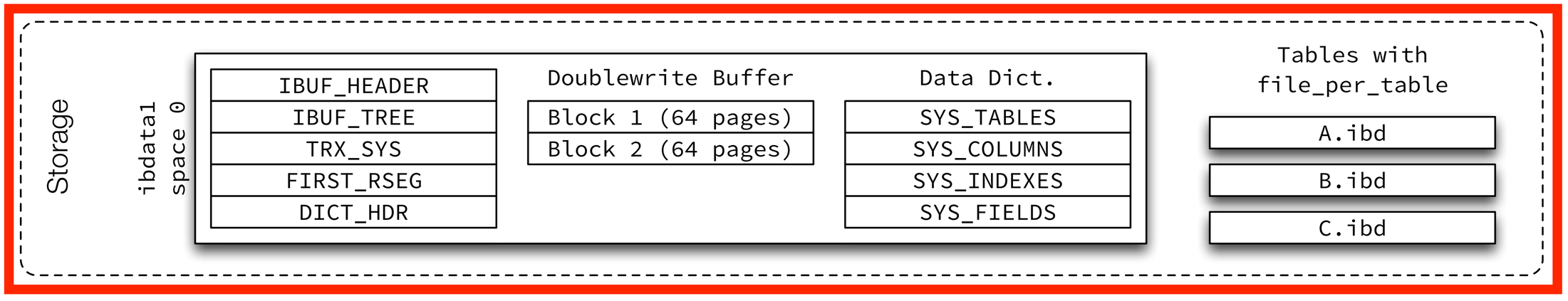




High-level Overview







Space File Structure

Space Files

Two types of space files:

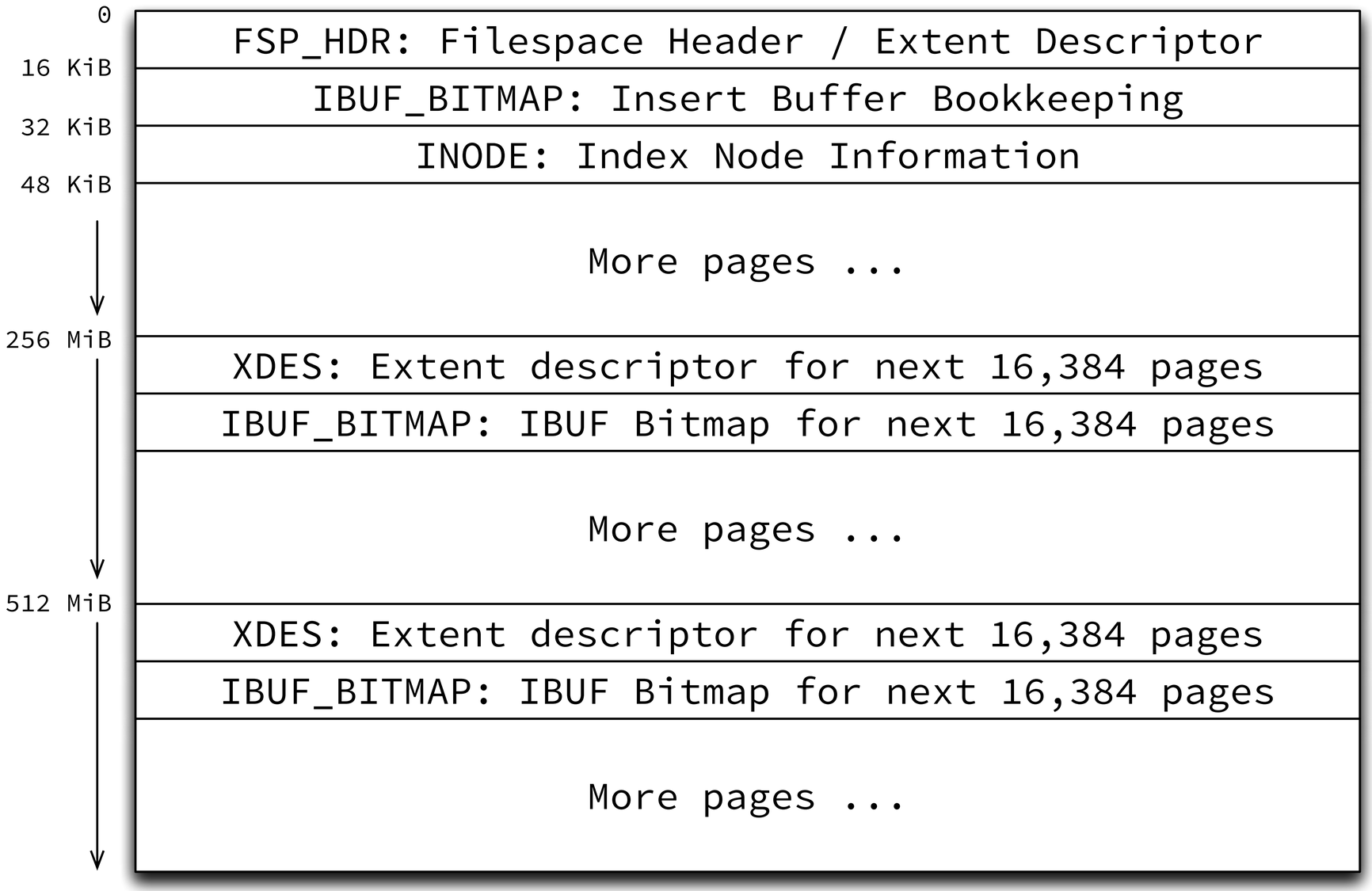
- ibdataX -- the system tablespace file(s)
- IBD -- the file-per-table tablespace files (*.ibd)

Both types have identical high-level structure

The system tablespace has additional pages for necessary structures, located at fixed positions (pages 3-7)

The file-per-table feature actually creates a *tablespace file* per table, with only a single table within each tablespace file

Space File Overview



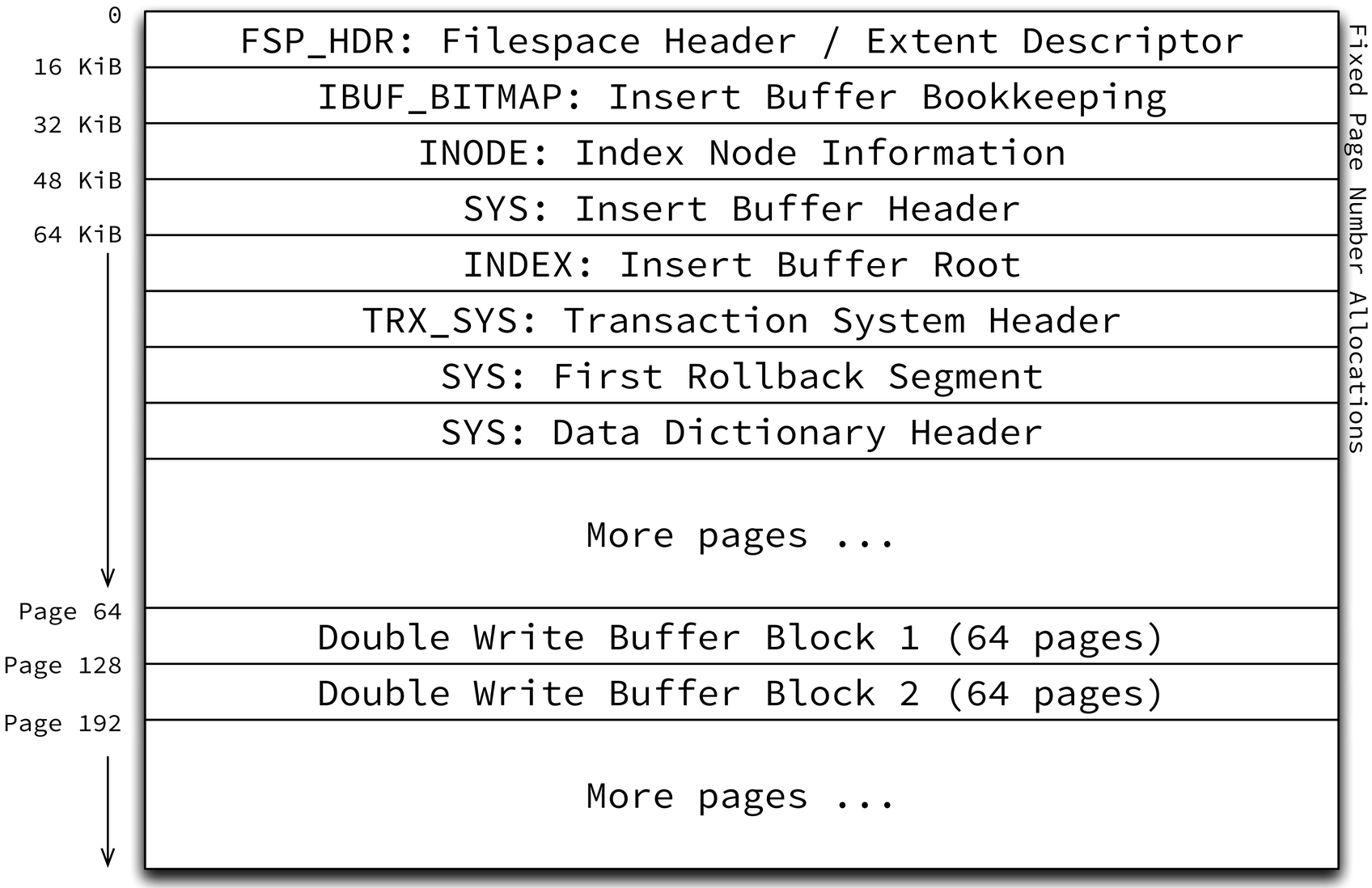
Overview of “ibdata1” system tablespace

This is the “system tablespace” and is always numbered 0
May contain user tables if created without file-per-table
Contains basic system information to bootstrap the system

System information present in “ibdata1”

Insert buffer metadata (SYS, page 3)
Insert buffer root index page (INDEX, page 4)
Transaction system metadata (TRX_SYS, page 5)
First rollback segment (SYS, page 6)
Data dictionary metadata (SYS, page 7)
Doublewrite buffer extents (pages 64-192) -- Exact locations are not exactly fixed, but stored in the TRX_SYS page.

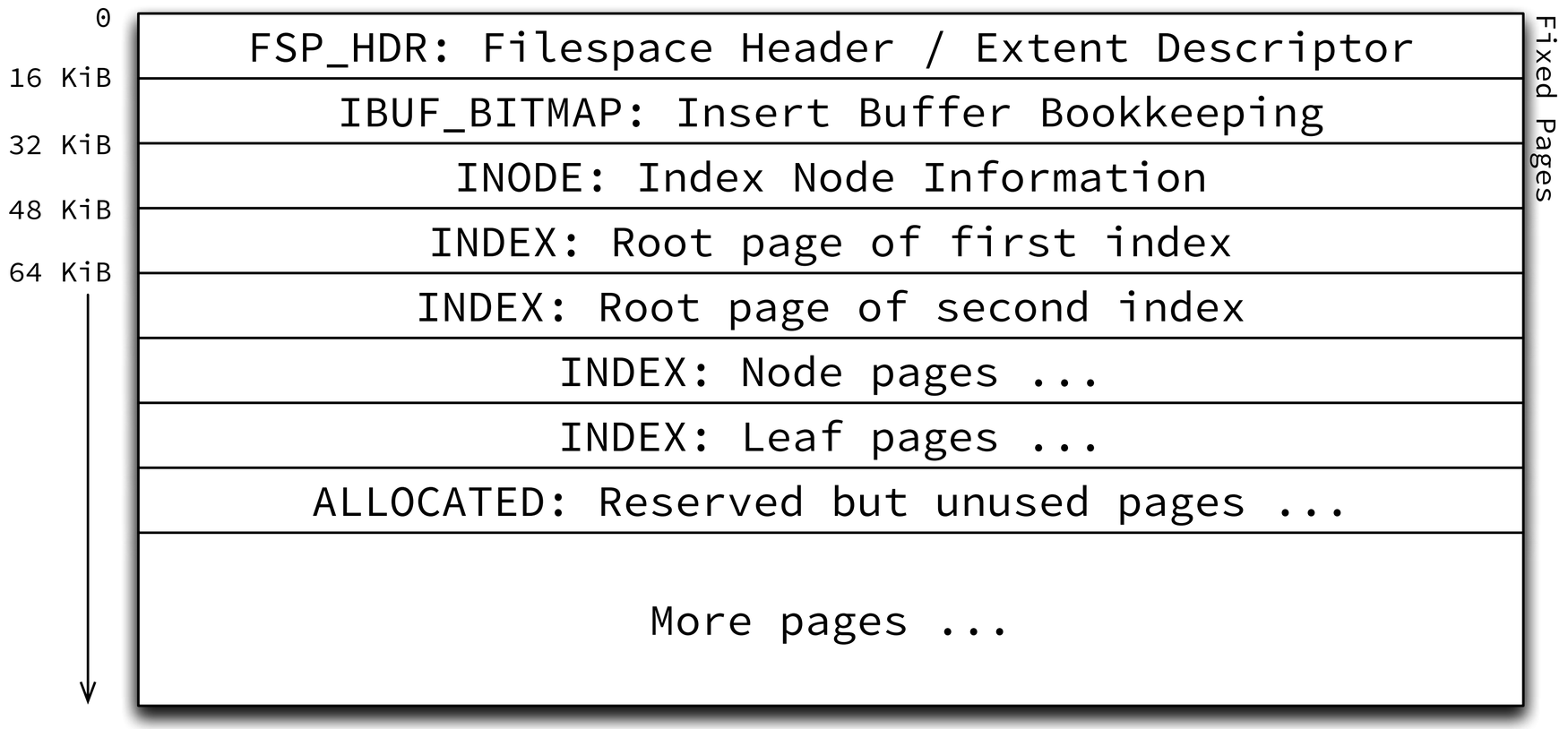
ibdata1 File Overview



Overview of IBD per-table tablespaces

Fully-fledged tablespace file
Could contain more than one table, but doesn't
Root pages for indexes allocated starting with page 3 (however in-place/online table ALTER doesn't follow that "rule")
Actual/official root page numbers are stored in data dictionary
Important parts of transaction system are still in system tablespace: rollback segments, doublewrite buffer

IBD File Overview



Aside: Everything is an “index”

InnoDB doesn’t have a separate “row data” storage structure

A B+Tree index is created for:

- the primary key: row data is stored in this index
- each secondary key: the row PKV is stored in this index

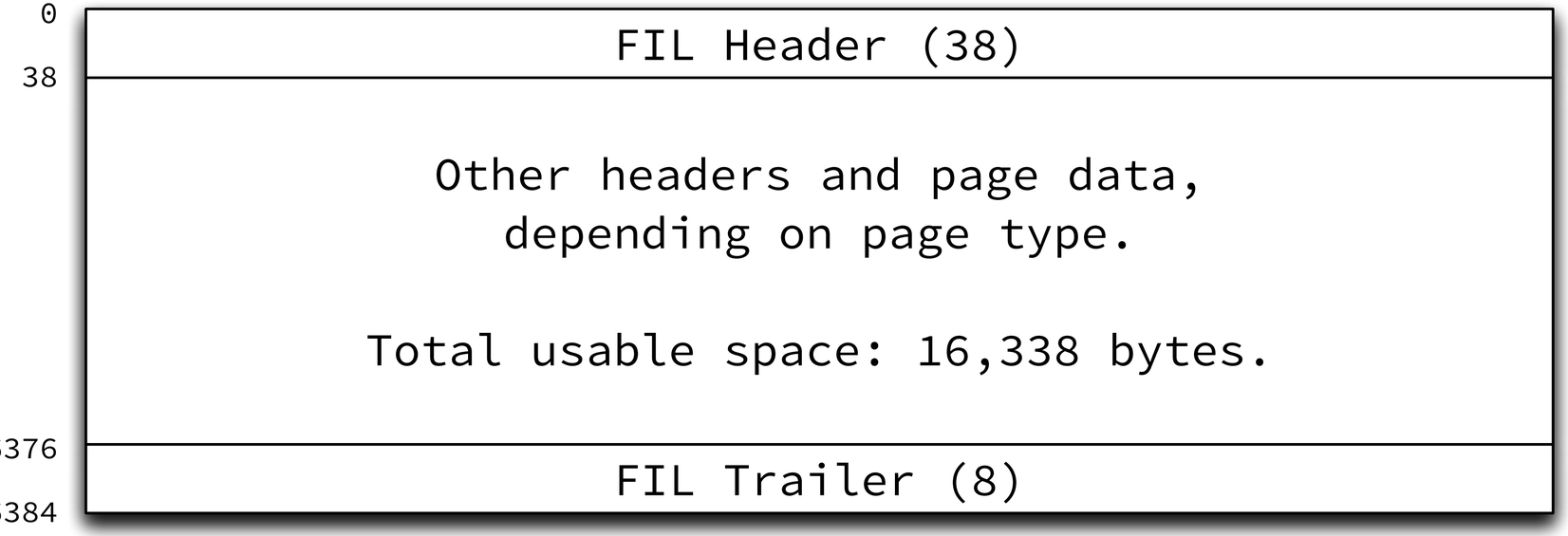
All user data is stored in pages of type “INDEX”

Page Structure

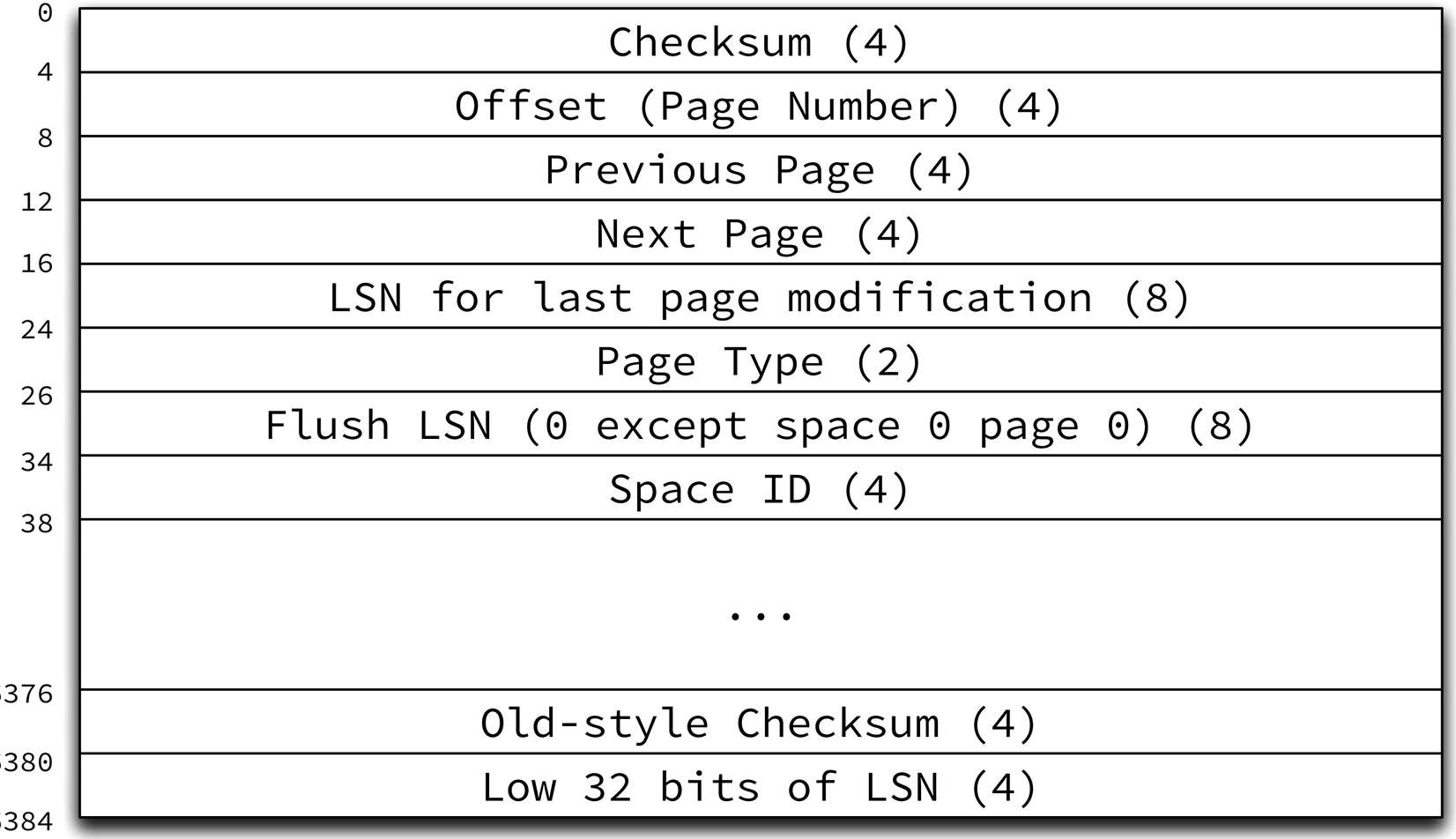
Basic page overview

Default page size is 16 KiB
Every page has a FIL header (38 bytes) and trailer (8 bytes)
The FIL header contains information to determine structure of the rest of the page
On a 16 KiB page, there are 16,338 bytes of usable space

Basic Page Overview



FIL Header/Trailer



Contents of a FIL header (irb)

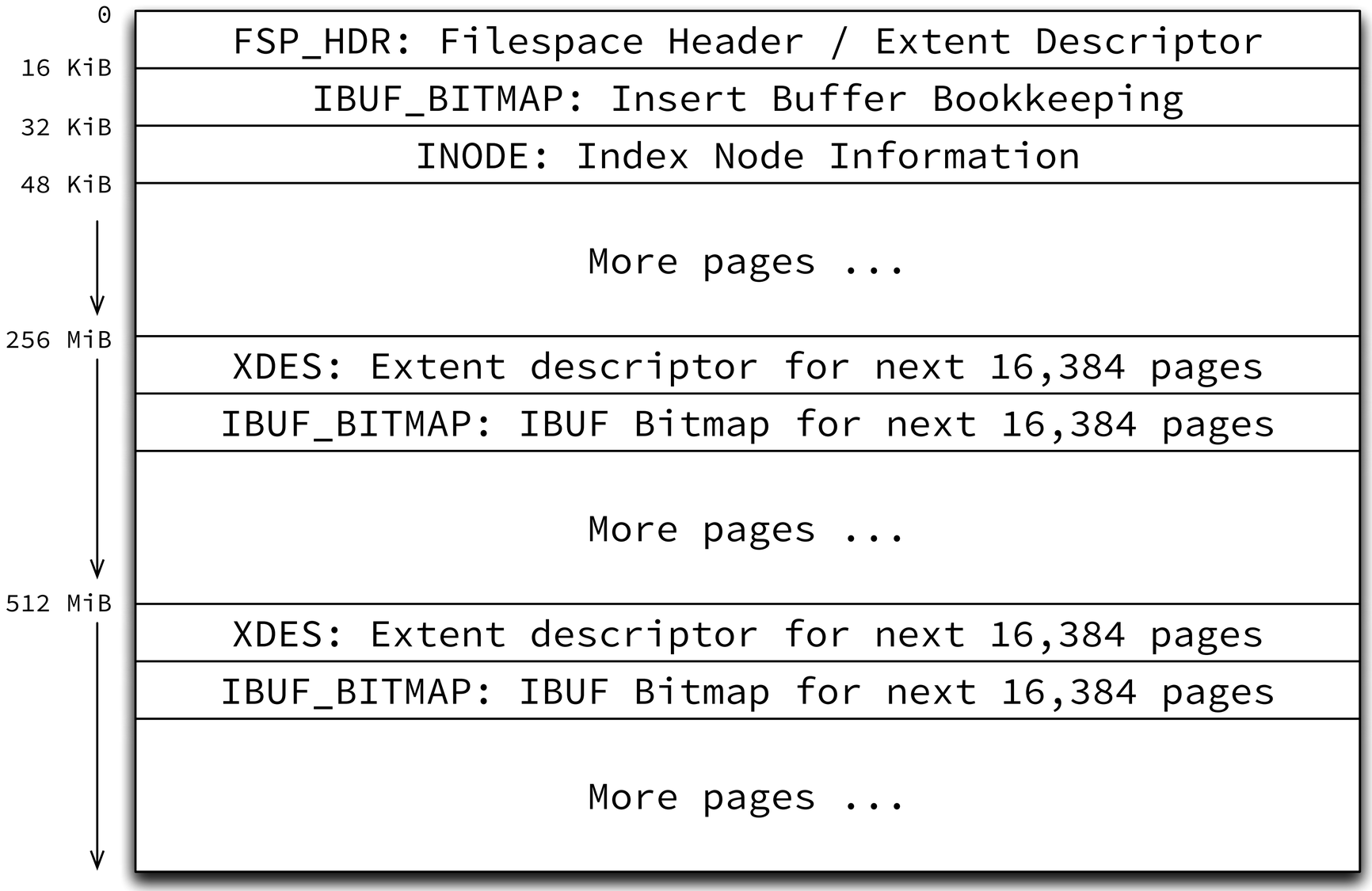
```
$ irb -rpp -rinnodb
>> space = Innodb::Space.new("ibdata1")
>> pp space.page(0).fil_header
{
  :checksum => 2067631406,
  :offset => 0,
  :prev => 0,
  :next => 0,
  :lsn => 1601269,
  :type => :FSP_HDR,
  :flush_lsn => 1603732,
  :space_id => 0
}
```

Contents of a FIL header (hex)

```
000000 = 7b3d8d2e                fil.checksum
000004 = 00000000                fil.offset
000008 = 00000000                fil.prev
000012 = 00000000                fil.next
000016 = 0000000000186ef5        fil.lsn
000024 = 0008                    fil.type
000026 = 0000000000187894        fil.flush_lsn
000034 = 00000000                fil.space_id
```

Space Management

Space File Overview



Extents and file segments (fseg)

Extents are always 1 MiB (with 16 KiB pages, 64 pages)

Each index has two “file segments”:

- leaf page file segment
- internal (non-leaf) page file segment

File segments (fseg) composed of:

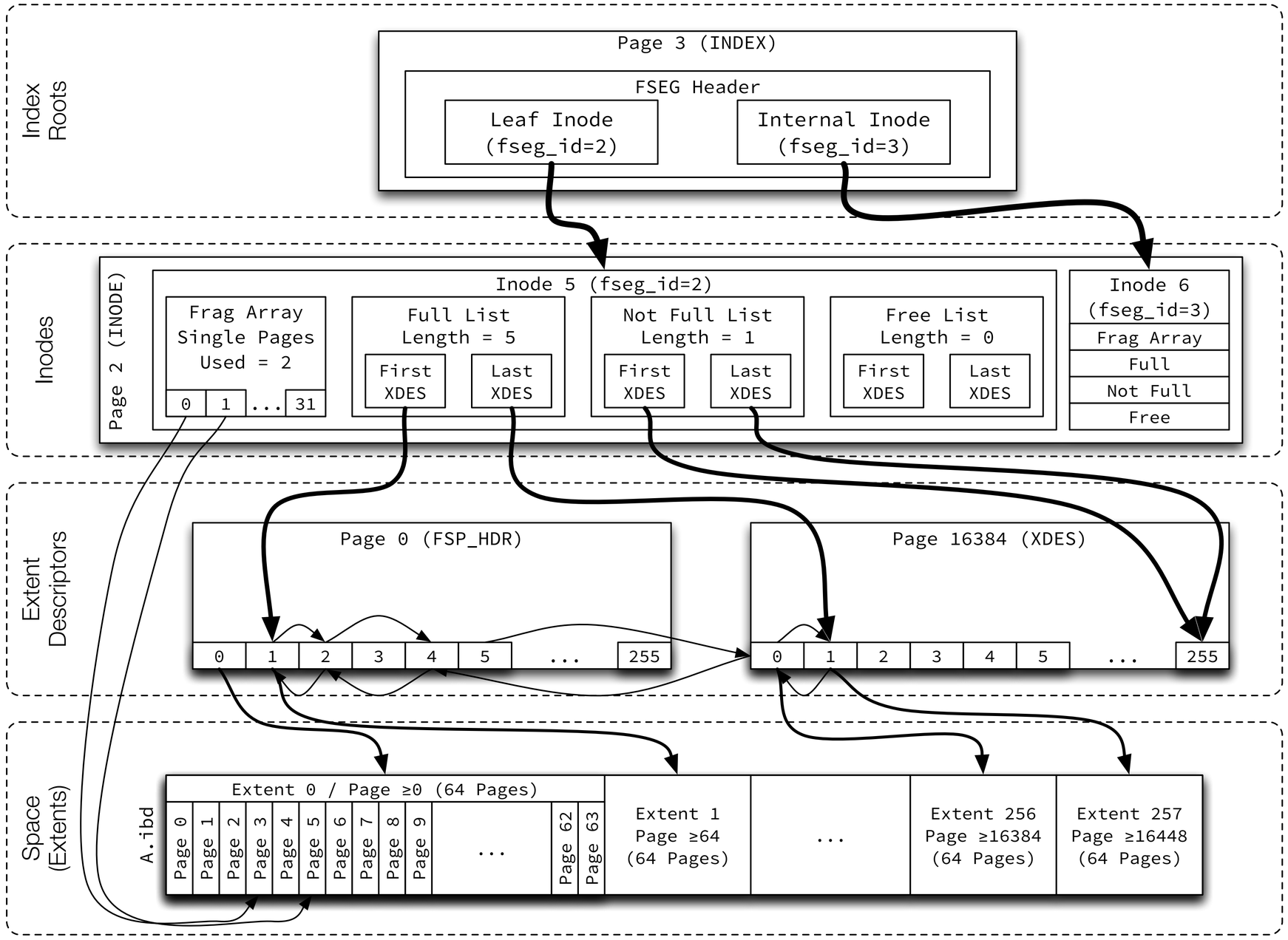
- a collection of complete extents; and
- up to 32 “fragment” pages from “fragment” extents

Segments grow automatically by adding extents

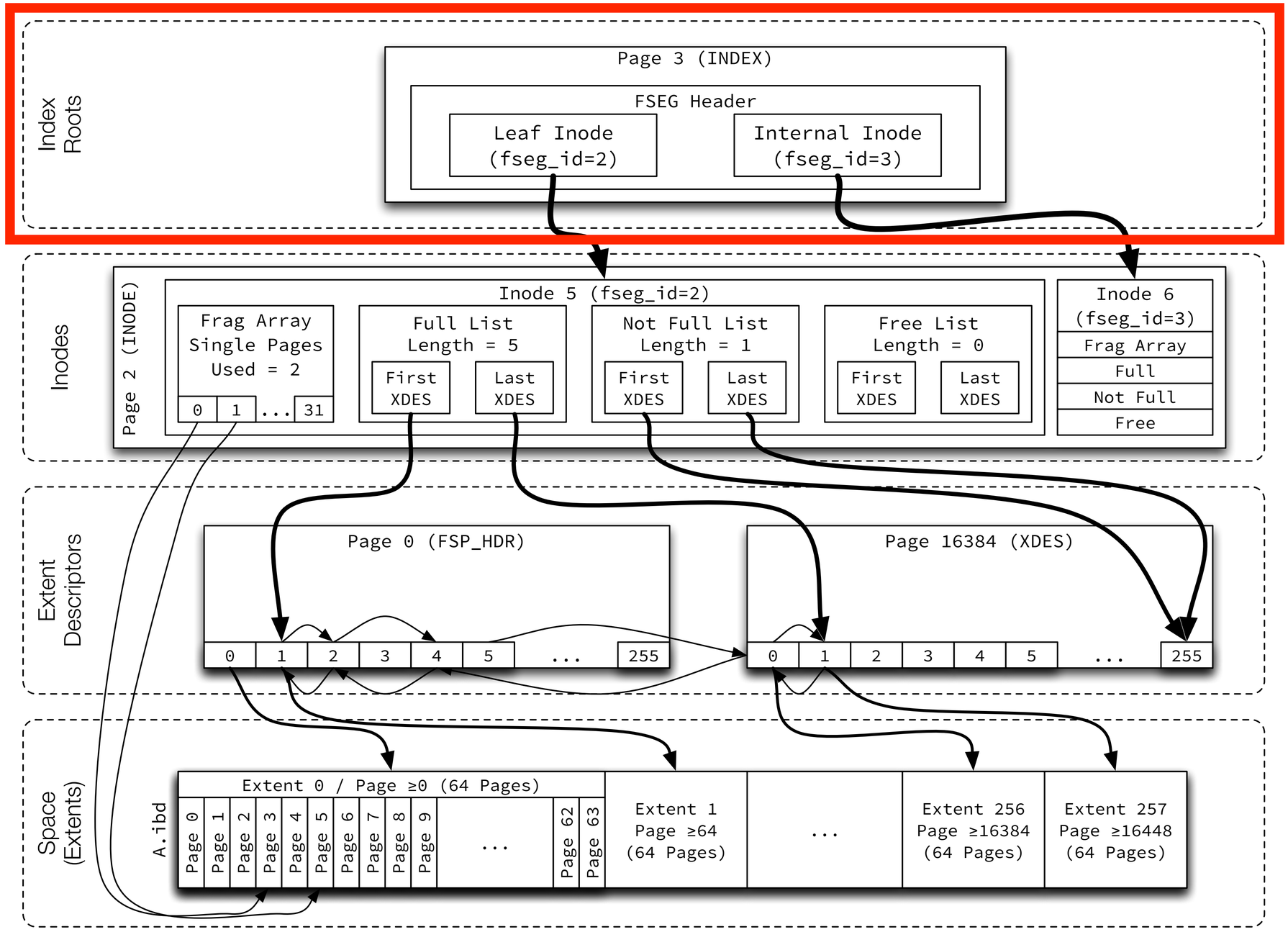
Segments can shrink, giving space back to space file

Space files never shrink to give space back to OS

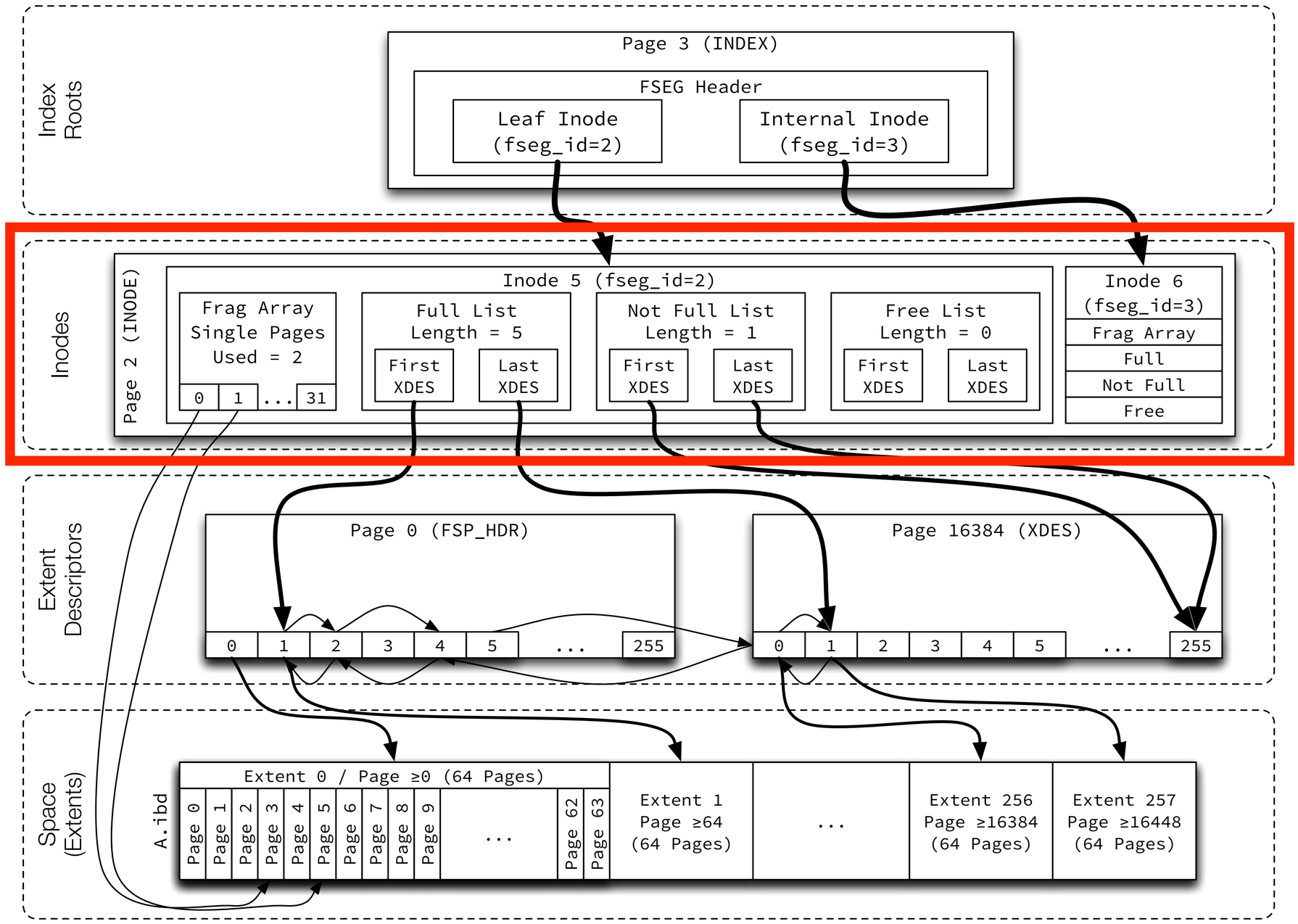
Index File Segment Structure



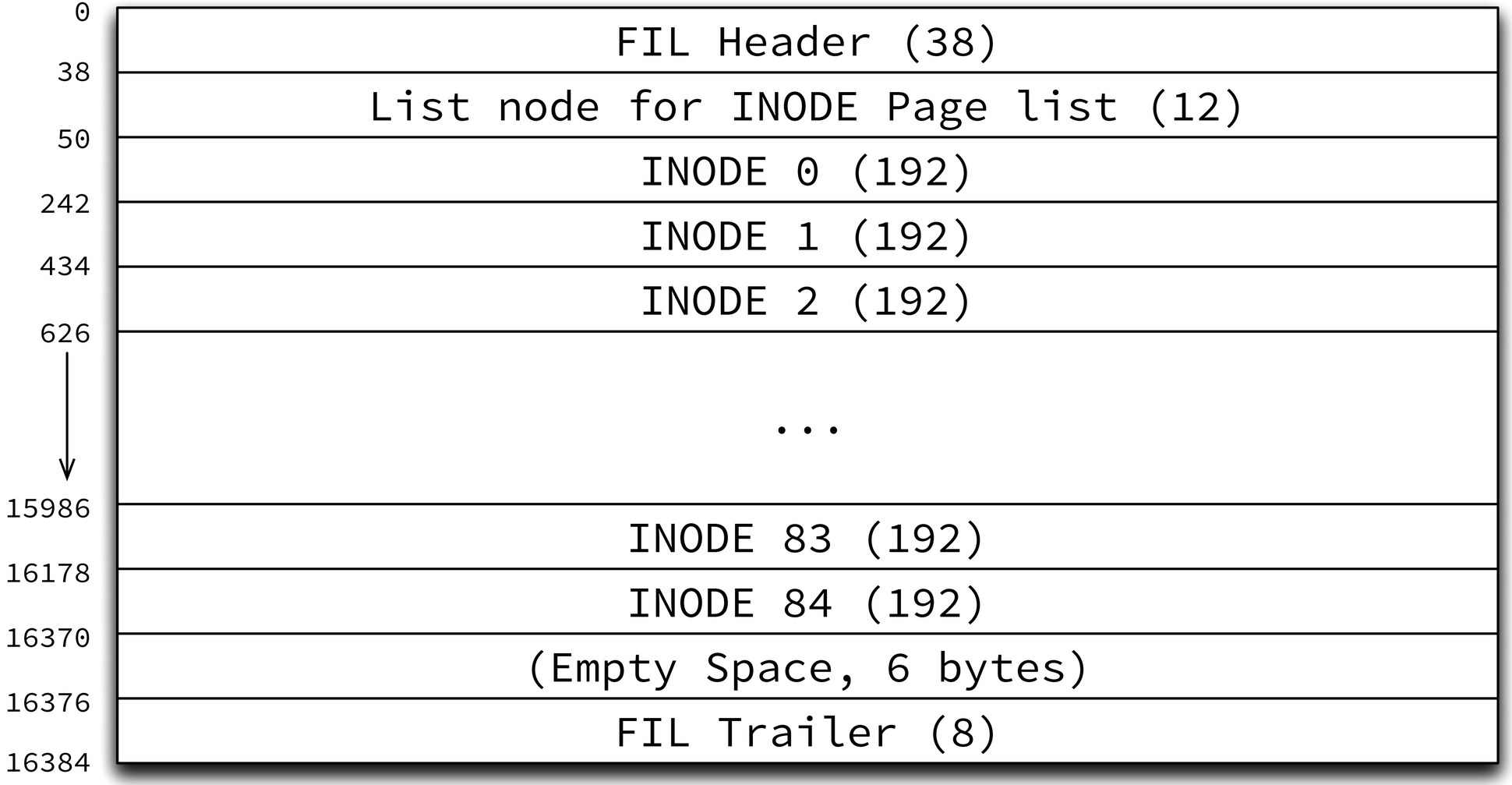
Index File Segment Structure



Index File Segment Structure



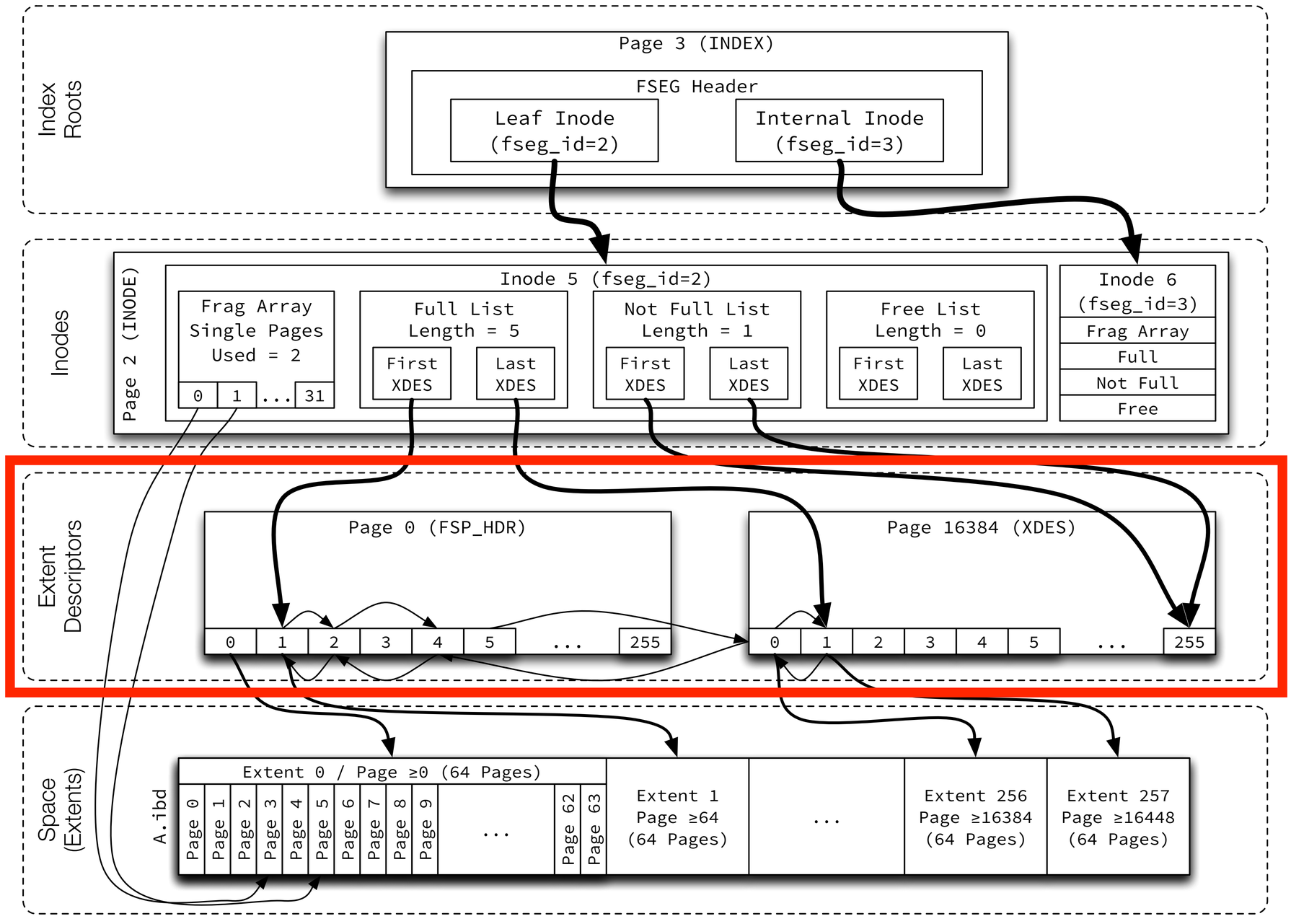
INODE Overview



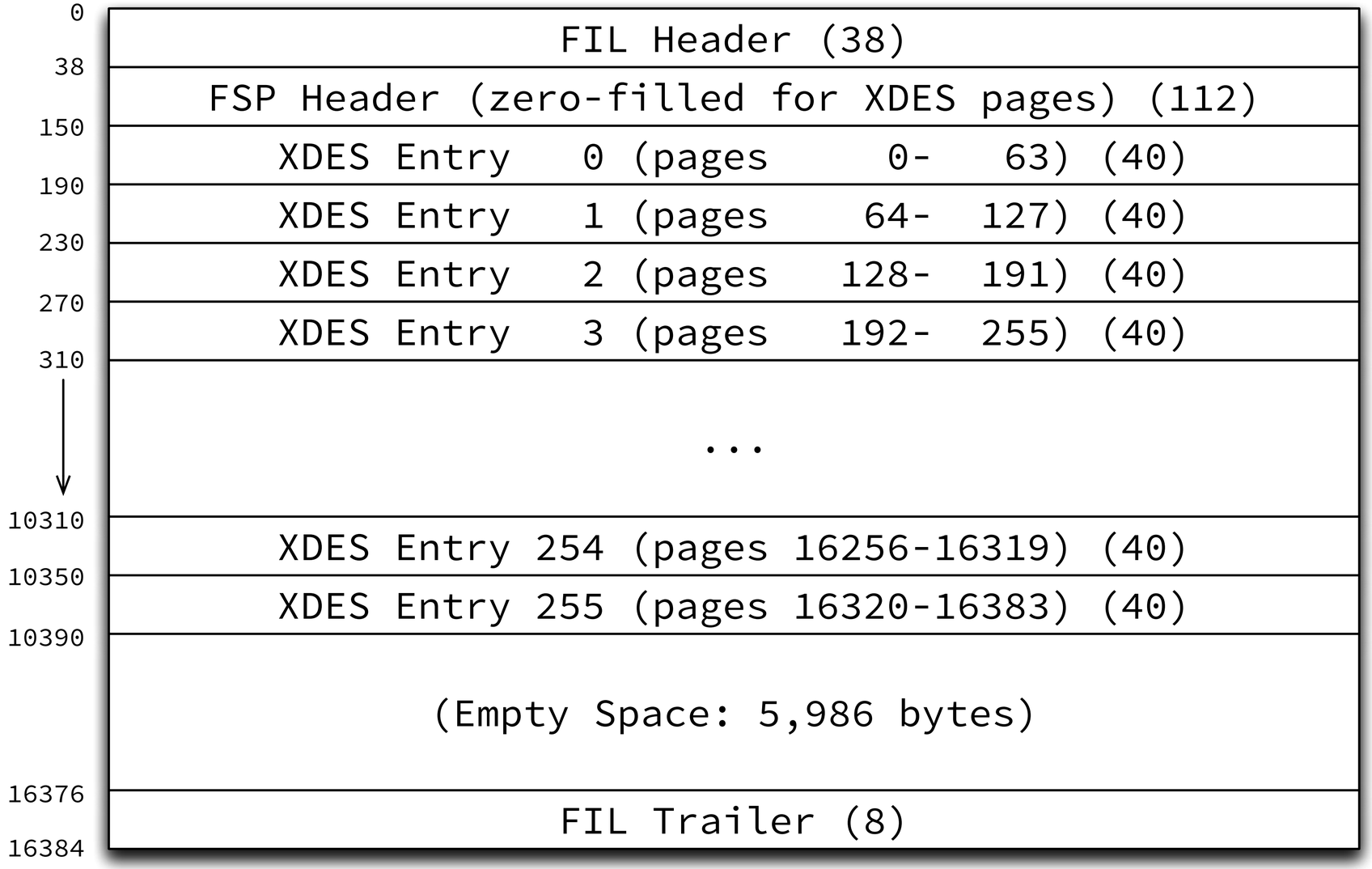
INODE Entry

N	FSEG ID (8)
N+8	Number of used pages in "NOT_FULL" list (4)
N+12	List base node for "FREE" list (16)
N+28	List base node for "NOT_FULL" list (16)
N+44	List base node for "FULL" list (16)
N+60	Magic Number = 97937874 (4)
N+64	Fragment Array Entry 0 (4)
N+68	...
N+188	Fragment Array Entry 31 (4)
N+192	

Index File Segment Structure



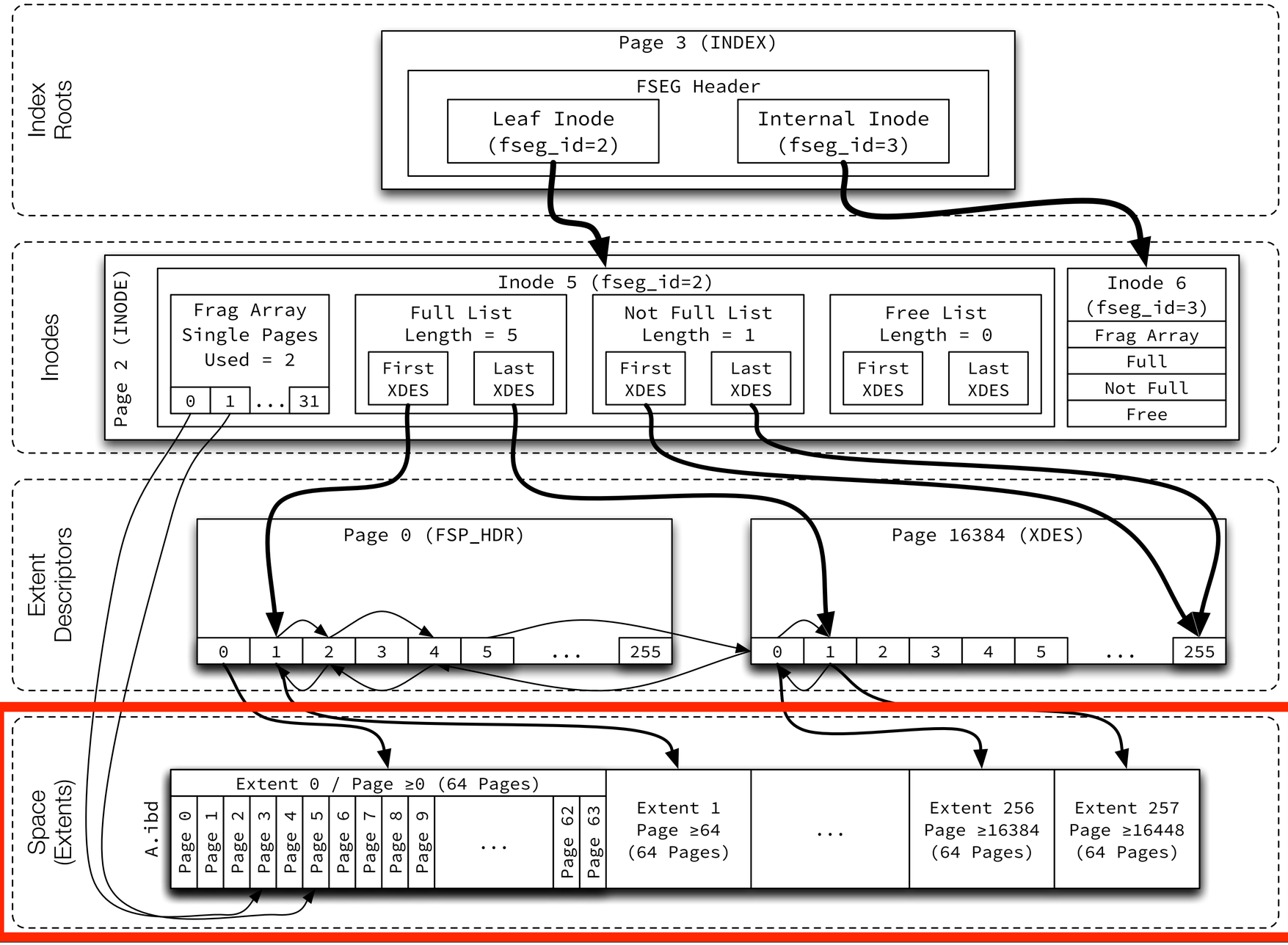
FSP_HDR/XDES Overview



FSP Header

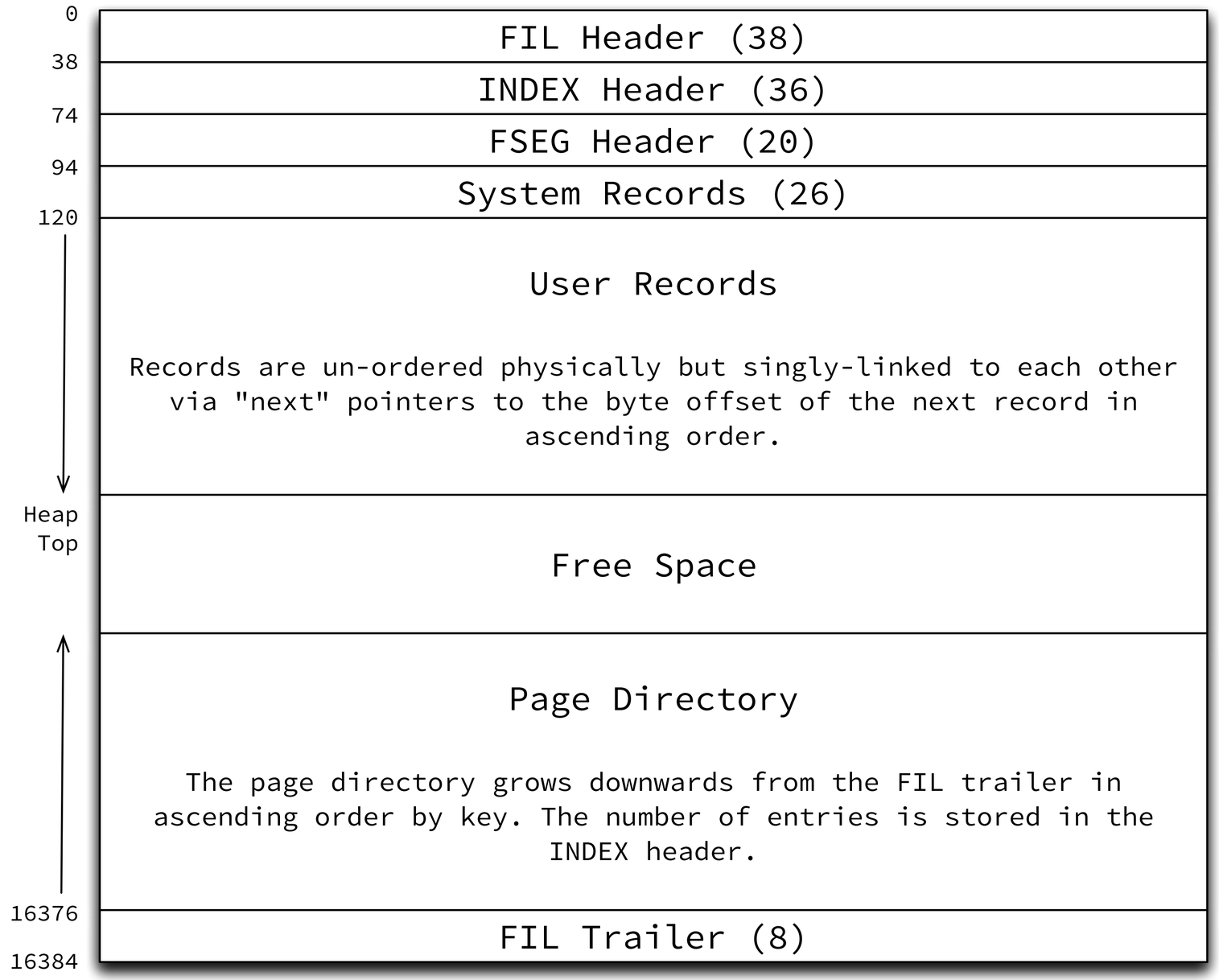
38	Space ID (4)
42	(Unused) (4)
46	Highest page number in file (size) (4)
50	Highest page number initialized (free limit) (4)
54	Flags (4)
58	Number of pages used in "FREE_FRAG" list (4)
62	List base node for "FREE" list (16)
78	List base node for "FREE_FRAG" list (16)
94	List base node for "FULL_FRAG" list (16)
110	Next Unused Segment ID (8)
118	List base node for "FULL_INODES" list (16)
134	List base node for "FREE_INODES" list (16)
150	

Index File Segment Structure

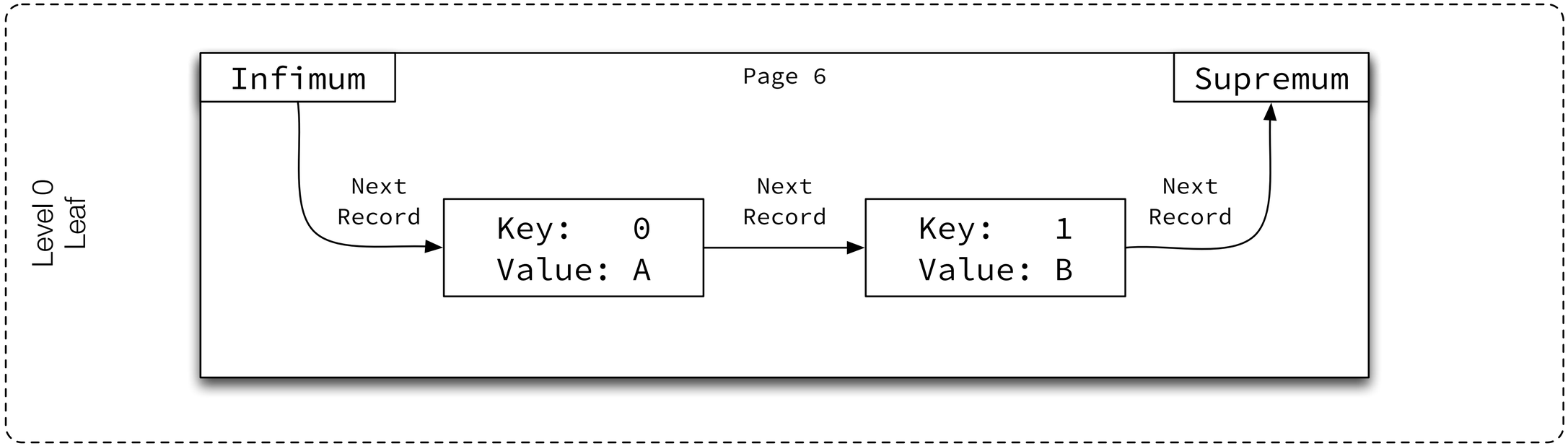


Indexes in InnoDB

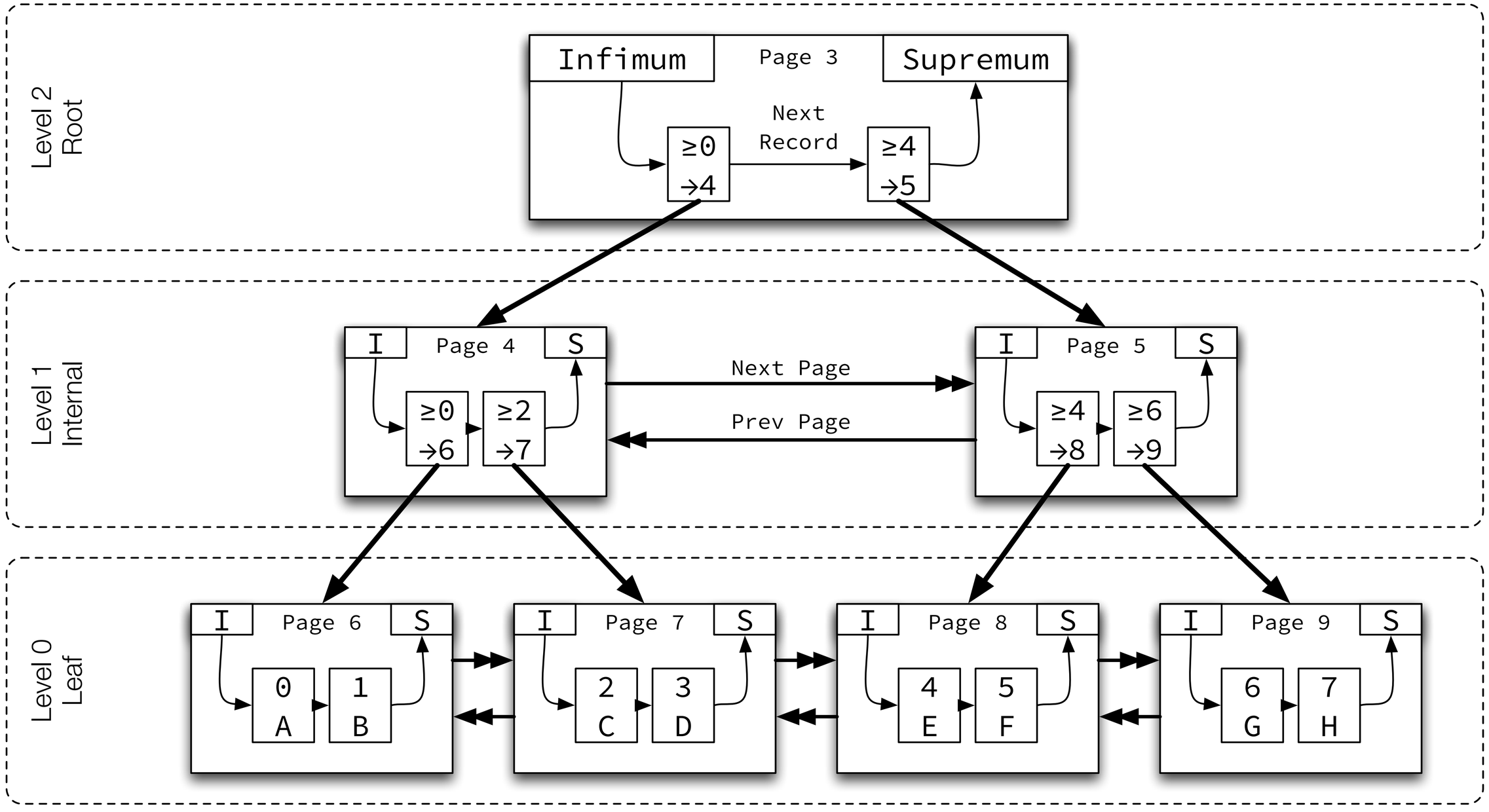
INDEX Overview



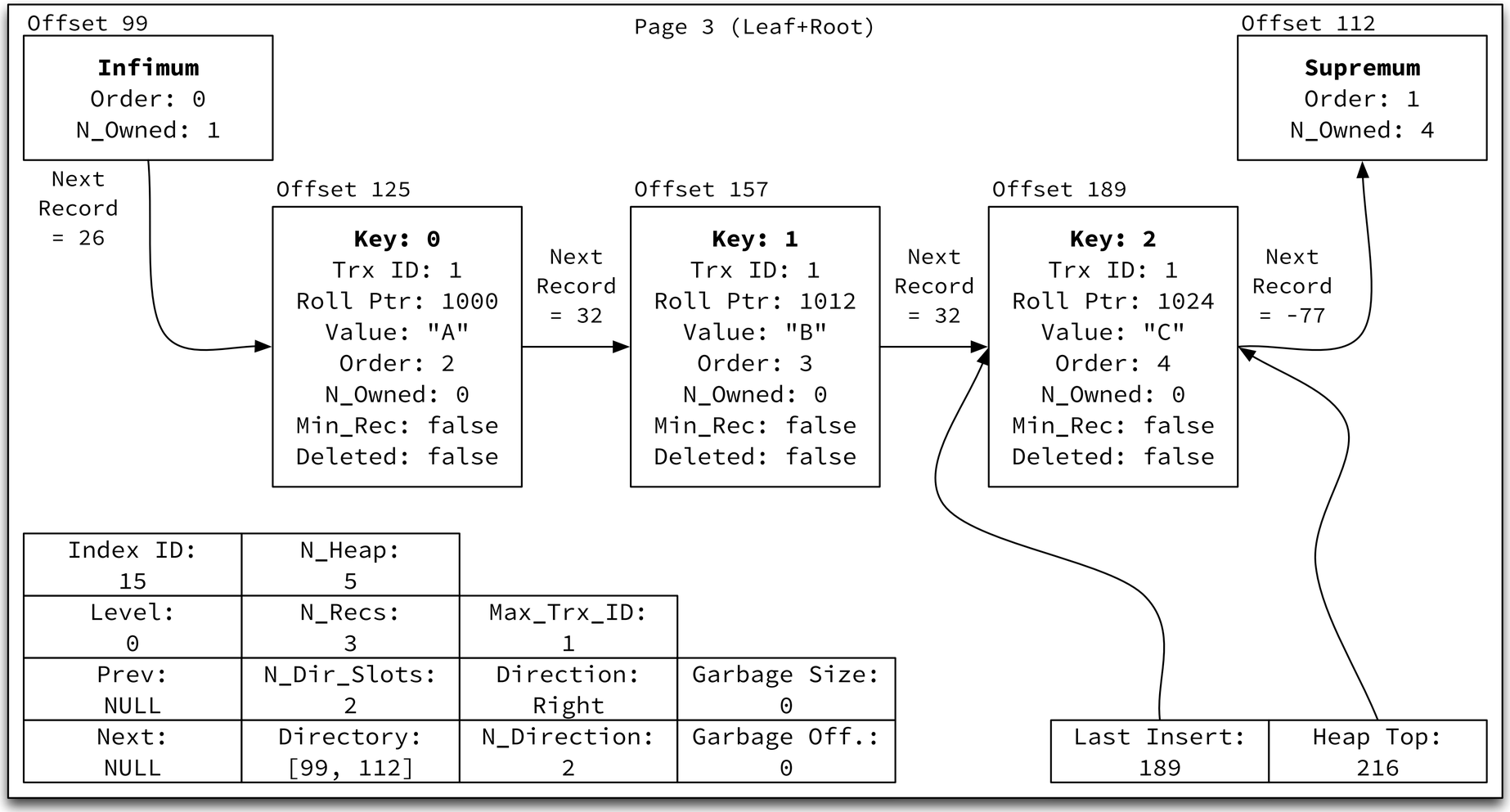
B+Tree Simplified Leaf Page



B+Tree Structure

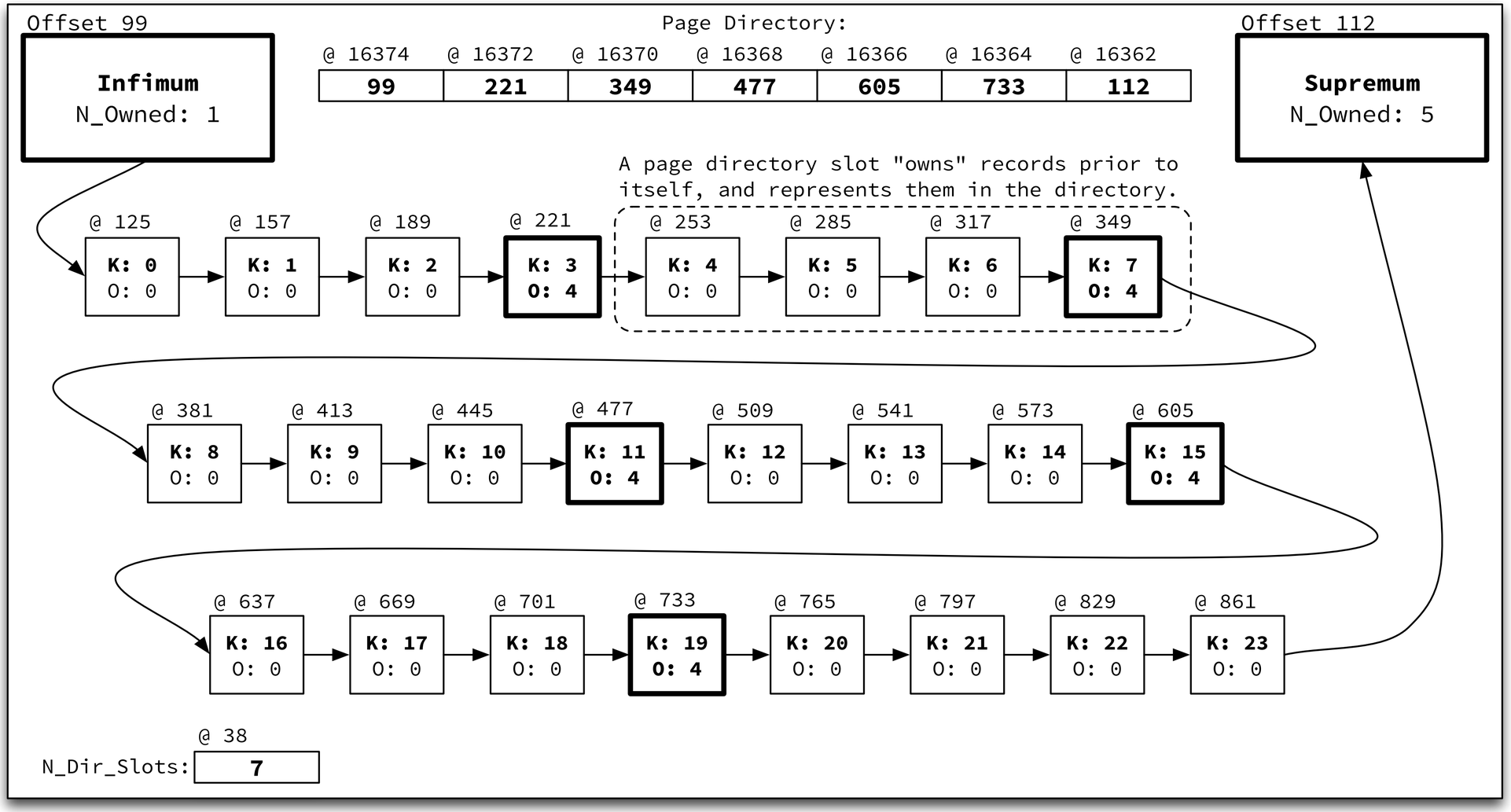


B+Tree Detailed Page Structure



InnoDB table format is Barracuda with "compact" record structure, non-compressed.
Table created with: `CREATE TABLE t (i INT NOT NULL, s CHAR(10) NOT NULL, PRIMARY KEY(i)) ENGINE=InnoDB;`
Table populated with: `INSERT INTO t (i, s) VALUES (0, "A"), (1, "B"), (2, "C");`
Record size: 5 (header) + 4 (PK) + 6 (TRX_ID) + 7 (ROLL_PTR) + 10 (non-key fields) = 32 bytes

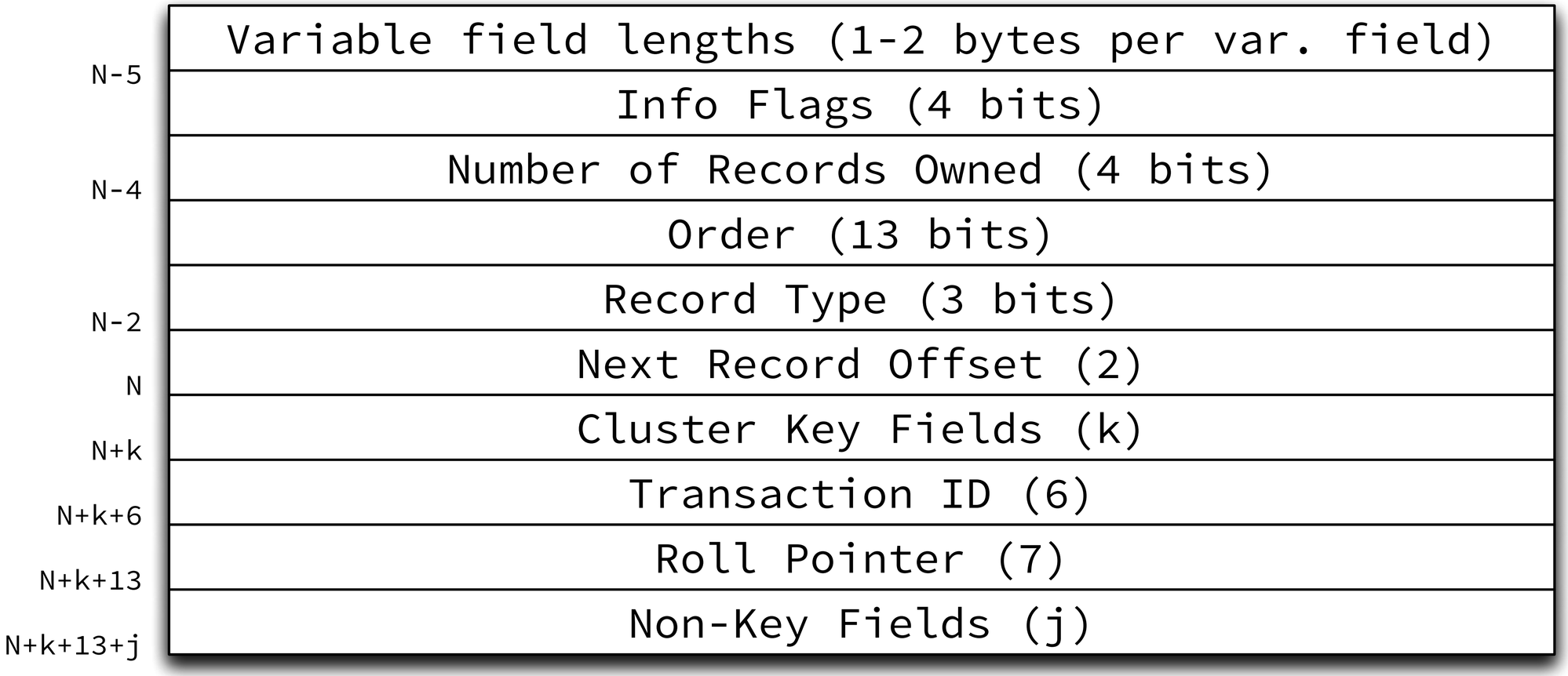
B+Tree Page Directory Structure



Infimum always owns only itself, so will always have a slot in the page directory with N_Owned = 1.
Supremum always owns the last few records in the page, and is allowed to own less than 4 records (if the page has fewer).
All directory slots will own a minimum of 4 and maximum of 8 records, except supremum, which may own fewer.
The page directory grows "downwards" from offset 16376, the beginning of the FIL trailer; the first directory entry starts at 16374.

Record Structure

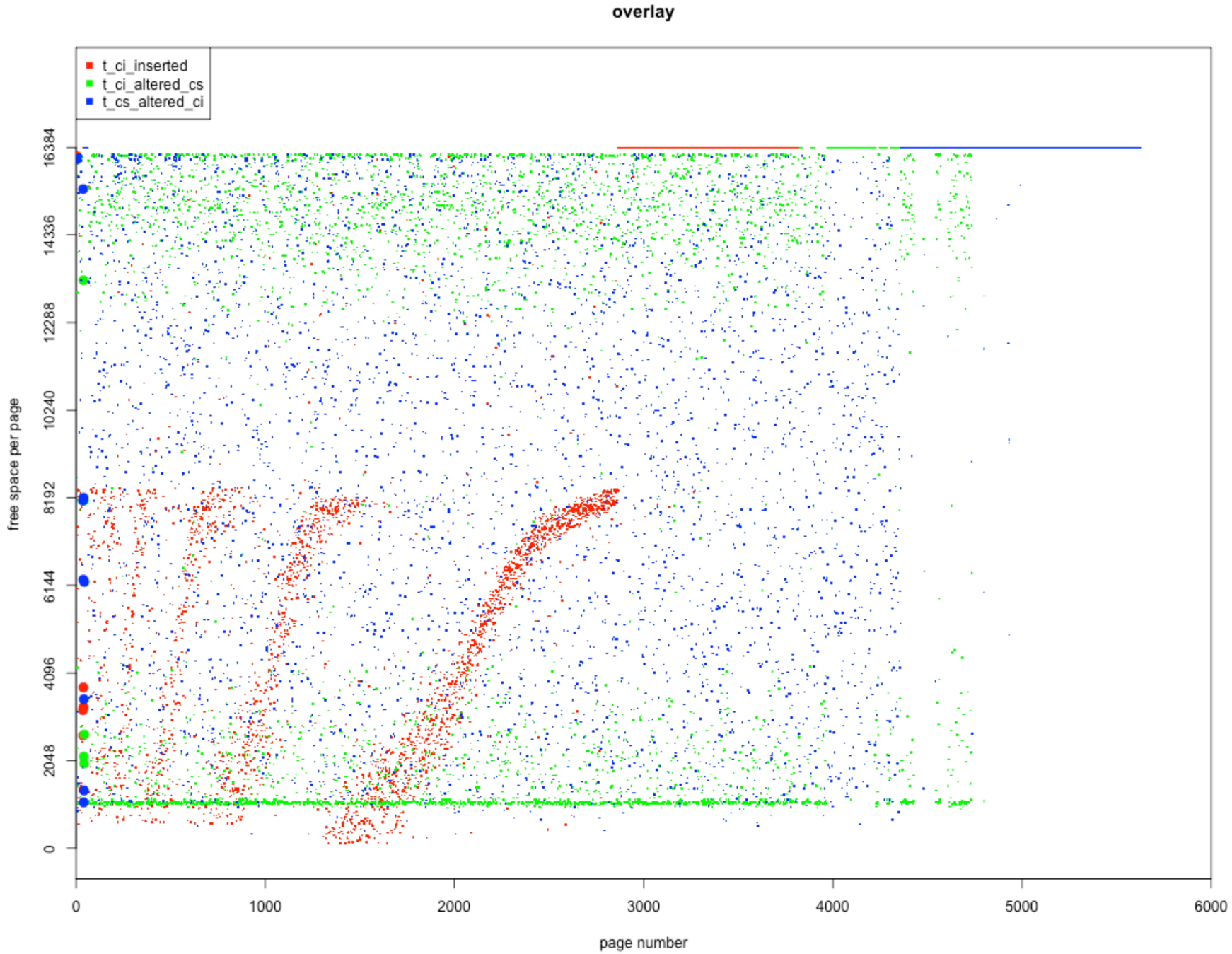
Record Format - Clustered Key - Leaf Pages

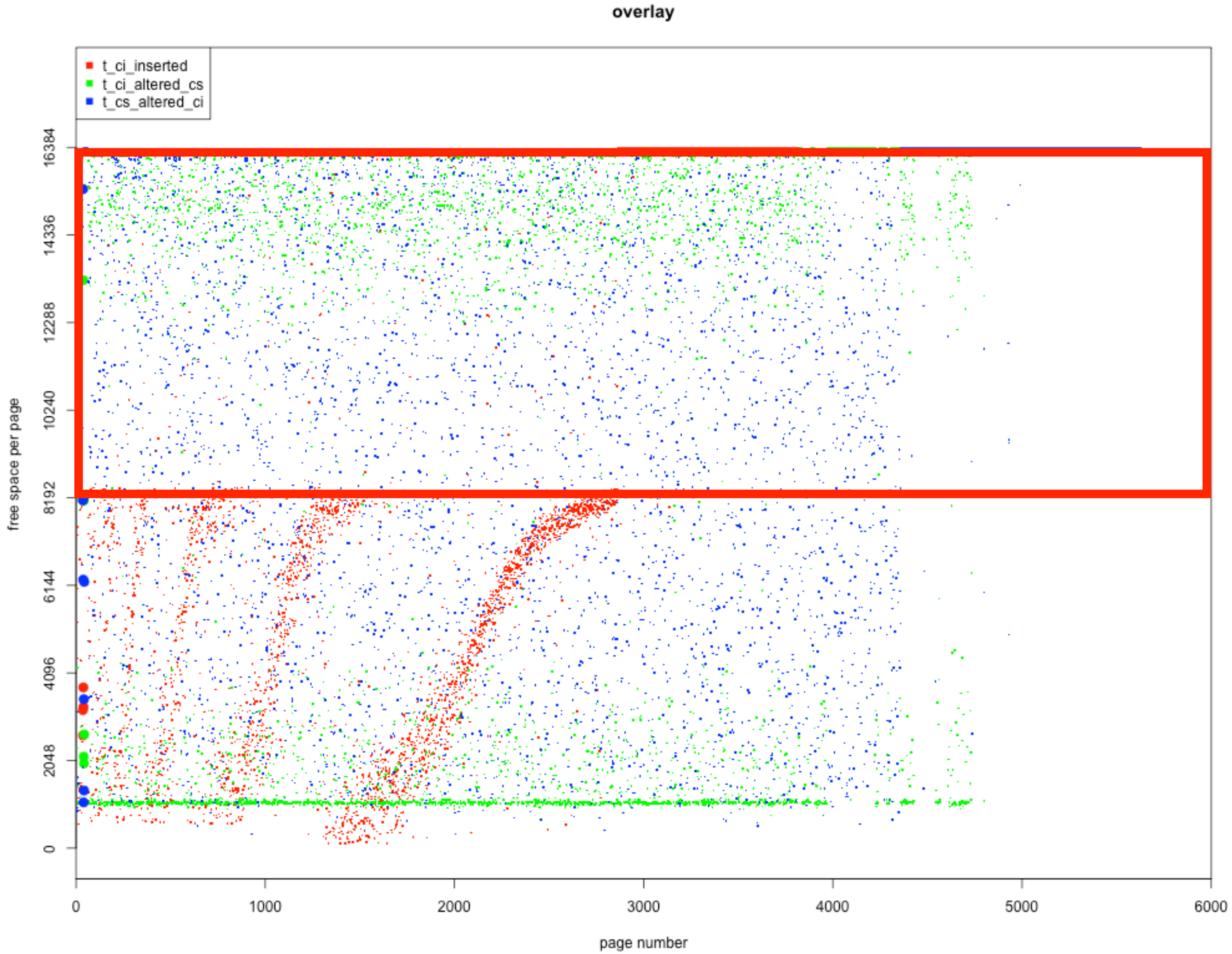


Record Format - Secondary Key - Leaf Pages

	Variable field lengths (1-2 bytes per var. field)
	Nullable field bitmap (1 bit per nullable field)
N-5	Info Flags (4 bits)
	Number of Records Owned (4 bits)
N-4	Order (13 bits)
	Record Type (3 bits)
N-2	Next Record Offset (2)
N	Secondary Key Fields (k)
N+k	Cluster Key Fields (j)
N+k+j	

Storage Inefficiencies


overlay
t_ci_inserted
t_ci_altered_cs
t_cs_altered_ci
free space per page
0
2048
4096
6144
8192
10240
12288
14336
16384
0
1000
2000
3000
4000
5000
6000
page number

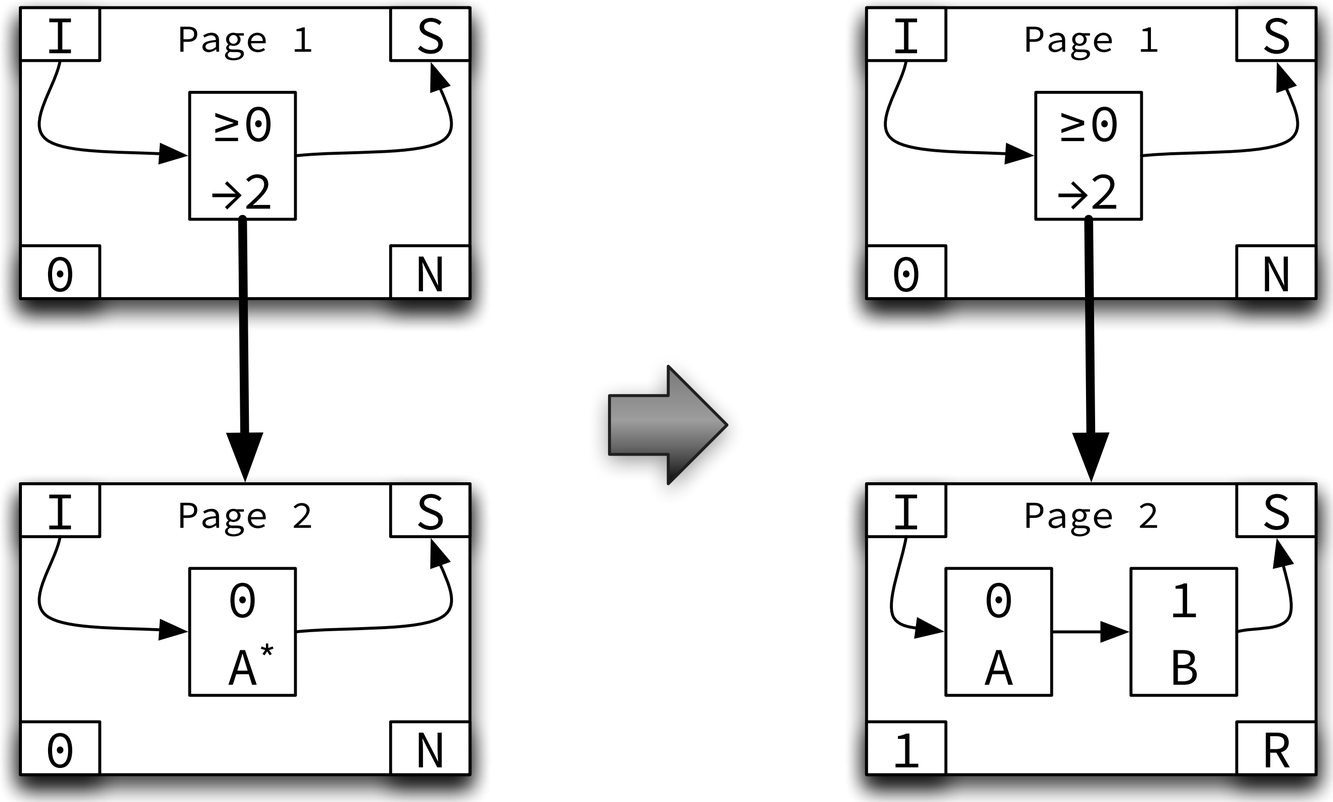

overlay
t_ci_inserted
t_ci_altered_cs
t_cs_altered_ci
free space per page
0
2048
4096
6144
8192
10240
12288
14336
16384
0
1000
2000
3000
4000
5000
6000
page number

?!

Bug #67718

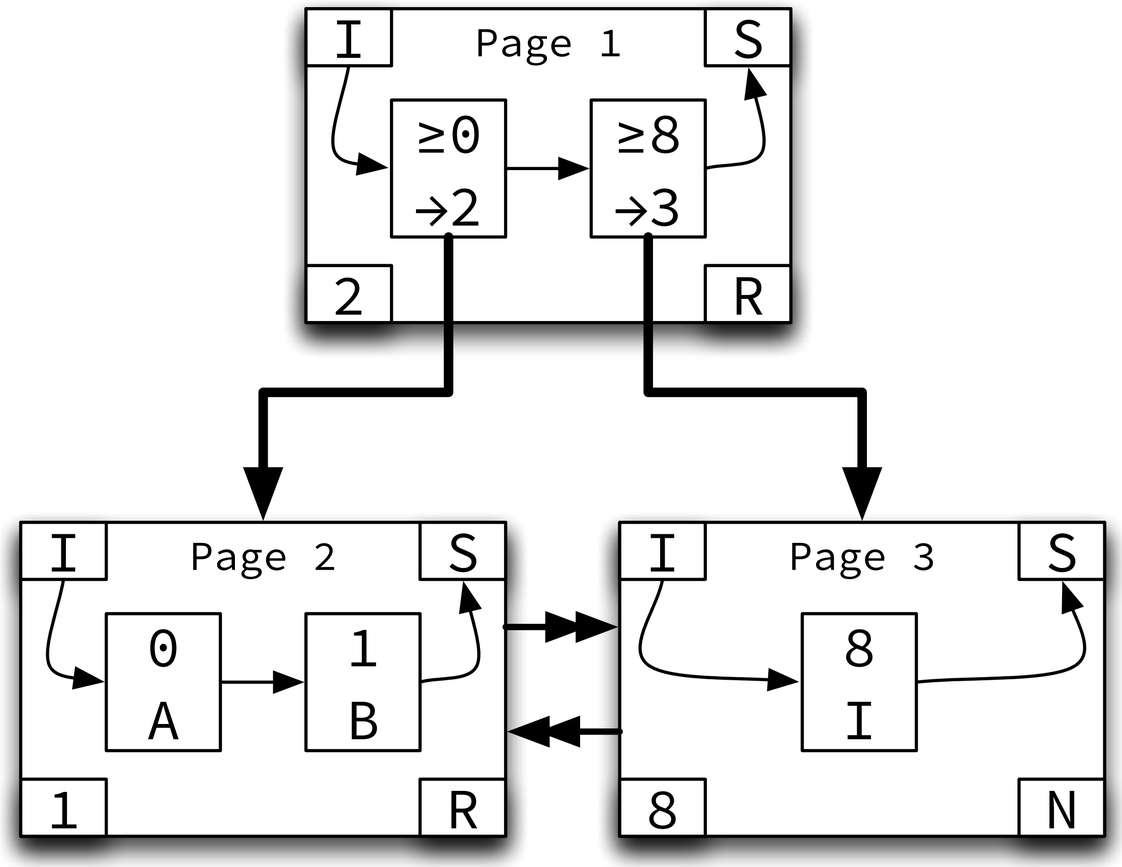
InnoDB drastically under-fills pages in certain conditions

1. Insert sets the insert direction in page



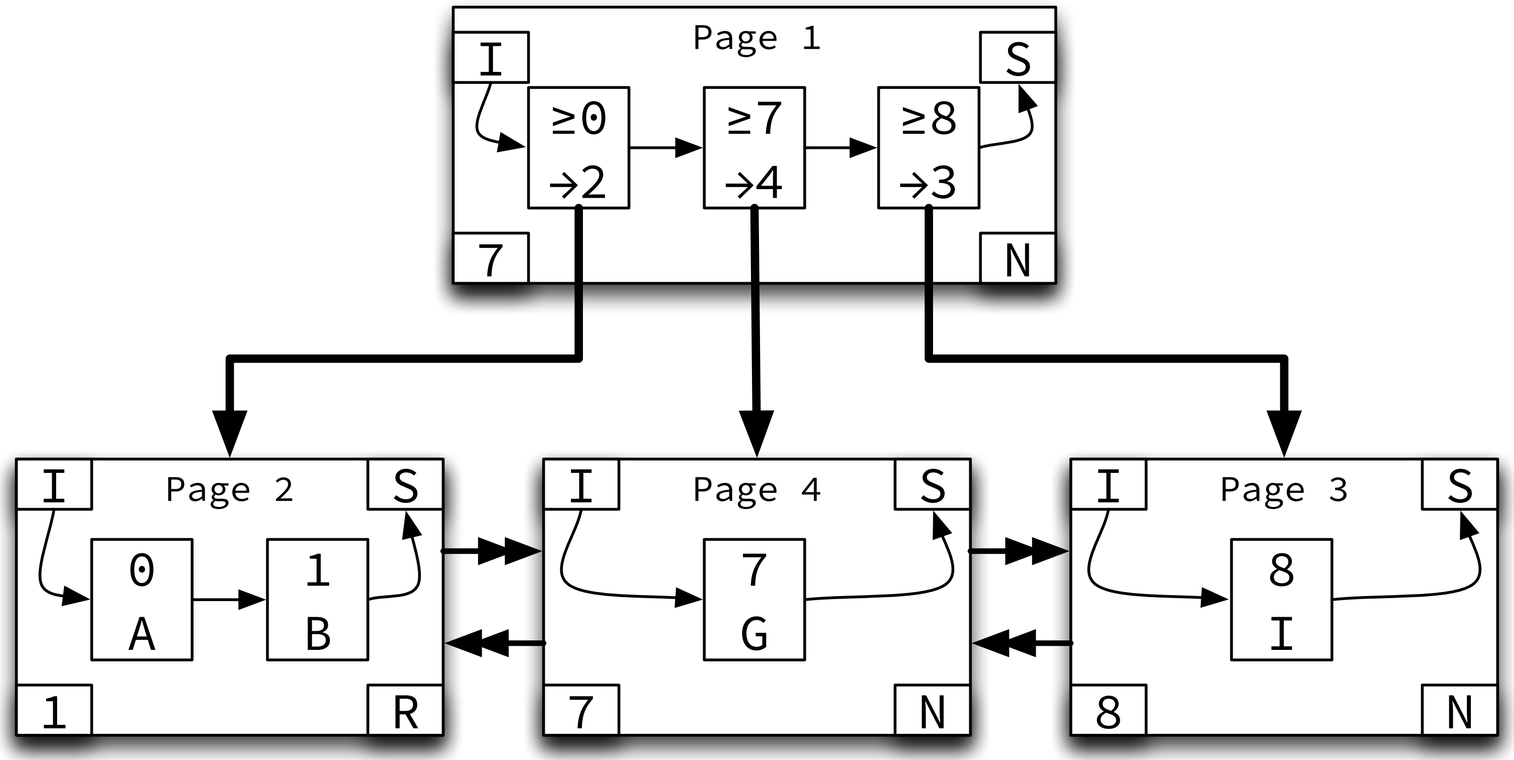
Caused by optimization to reduce page splits for inserts in sequential and increasing key values. Insertion order: page insertion point and direction.

2. Future page split behavior is affected



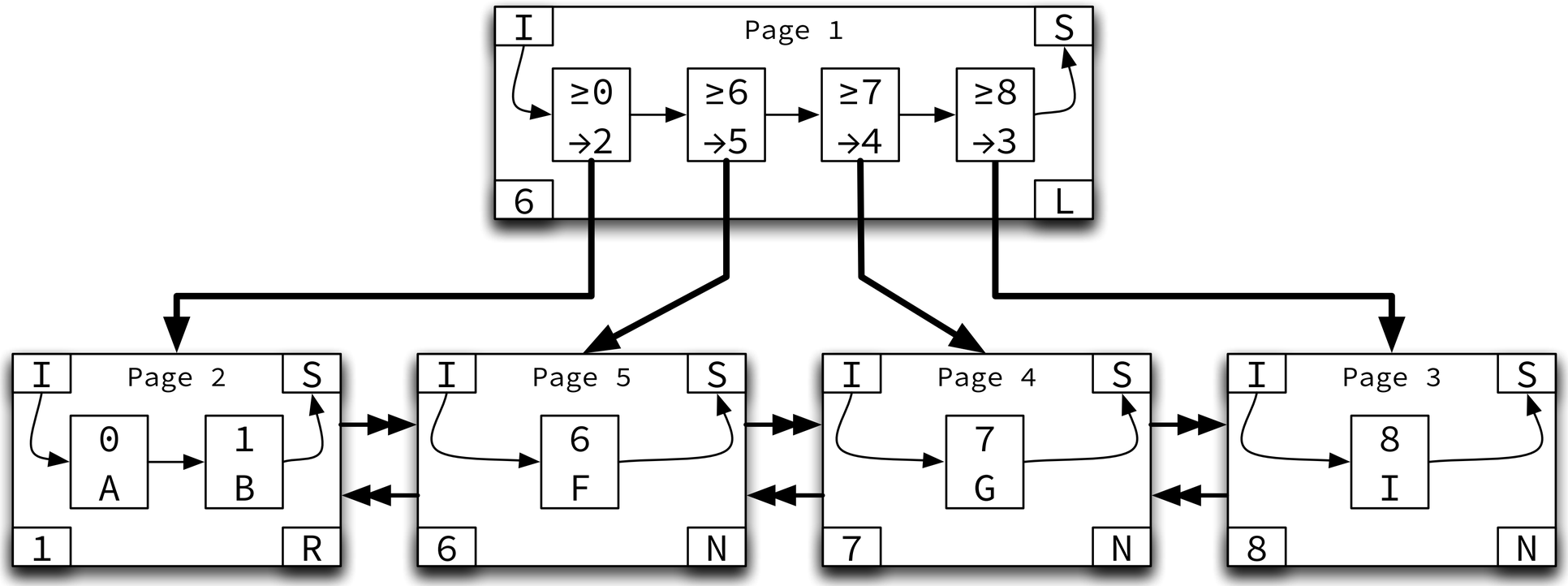
If page is full and an ascending order has been established at the page level, a new page is allocated instead of splitting the page.

3. Pages may be split unnecessarily



Triggered whenever inserting a key value that is an immediate successor or predecessor to the last inserted key value in the page.

4. Tablespace has many under-filled pages



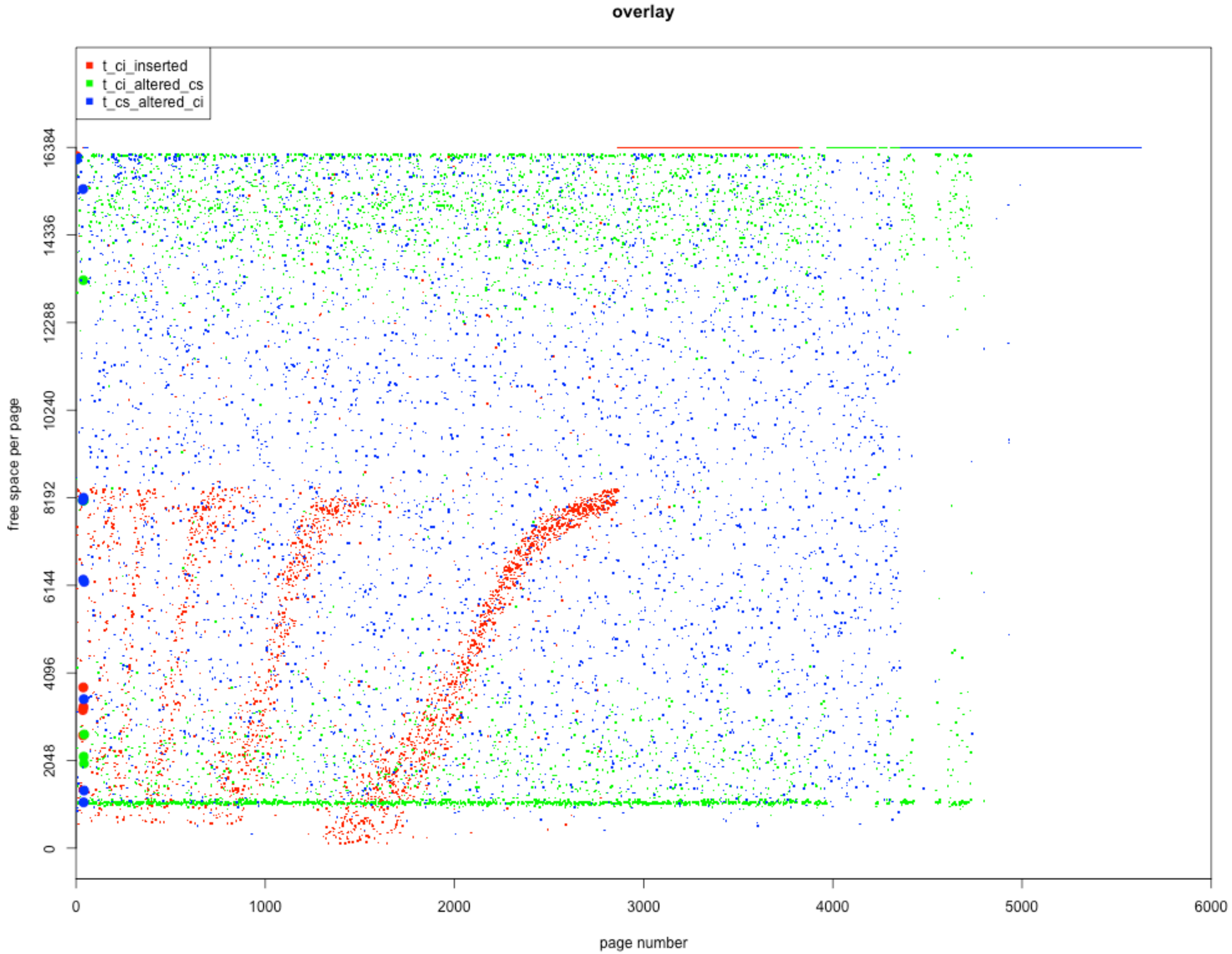
Inserts into the page with a key value that is greater than any key value already in the page, but lower than the minimum key value of its sibling page, will lead to a series of drastically under-filled pages.

Simple test case to reproduce problem

```
CREATE TABLE t1 (a BIGINT PRIMARY KEY, b VARCHAR(4096)) ENGINE=InnoDB;

# Build b-tree so that the leftmost leaf page has records 0, 1 and 2.
INSERT INTO t1 VALUES (0, REPEAT('a', 4096));
INSERT INTO t1 VALUES (1000, REPEAT('a', 4096));
INSERT INTO t1 VALUES (1001, REPEAT('a', 4096));
INSERT INTO t1 VALUES (1002, REPEAT('a', 4096));
INSERT INTO t1 VALUES (1, REPEAT('a', 4096));
INSERT INTO t1 VALUES (2, REPEAT('a', 4096));

# Each insert/record down from 999 will end up on its own page.
INSERT INTO t1 VALUES (999, REPEAT('a', 4096));
INSERT INTO t1 VALUES (998, REPEAT('a', 4096));
```


overlay
t_ci_inserted
t_ci_altered_cs
t_cs_altered_ci
free space per page
0
2048
4096
6144
8192
10240
12288
14336
16384
0
1000
2000
3000
4000
5000
6000
page number

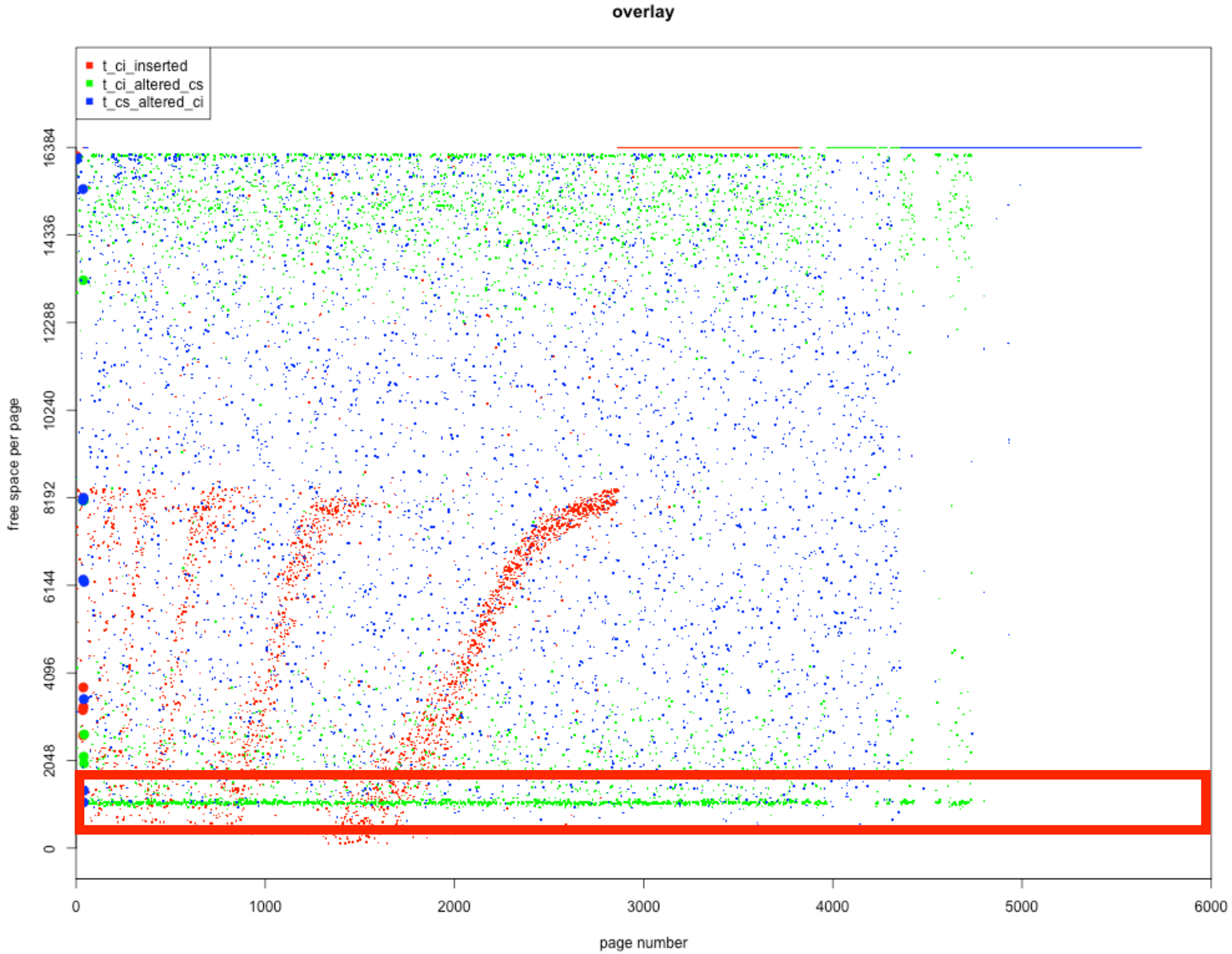
Segment fill factor

Facebook discussed this last year


overlay
t_ci_inserted
t_ci_altered_cs
t_cs_altered_ci
free space per page
0
2048
4096
6144
8192
10240
12288
14336
16384
0
1000
2000
3000
4000
5000
6000
page number

Index fill factor

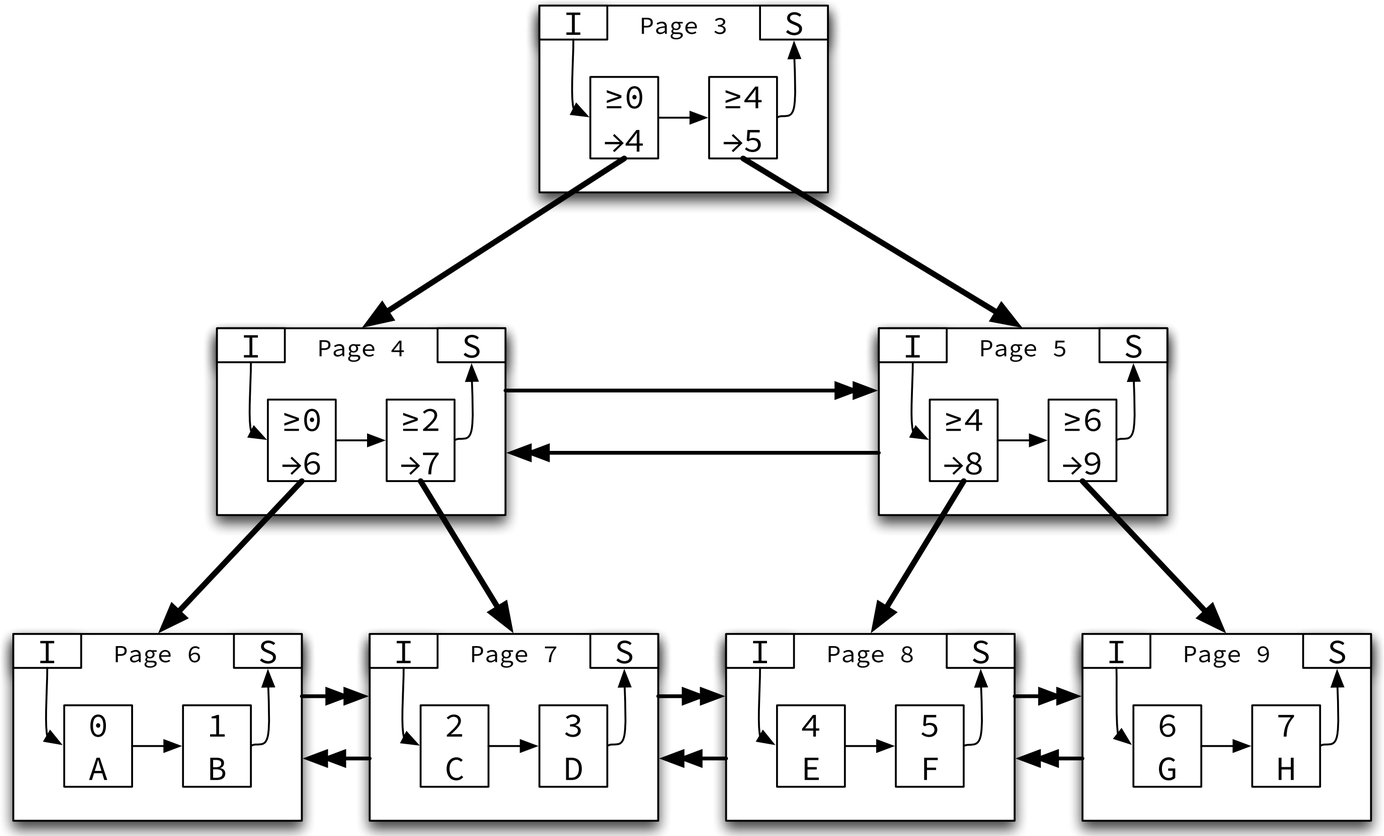
Also accounts for a lot of disk space


overlay
t_ci_inserted
t_ci_altered_cs
t_cs_altered_ci
free space per page
0
2048
4096
6144
8192
10240
12288
14336
16384
0
1000
2000
3000
4000
5000
6000
page number

Bug#68501

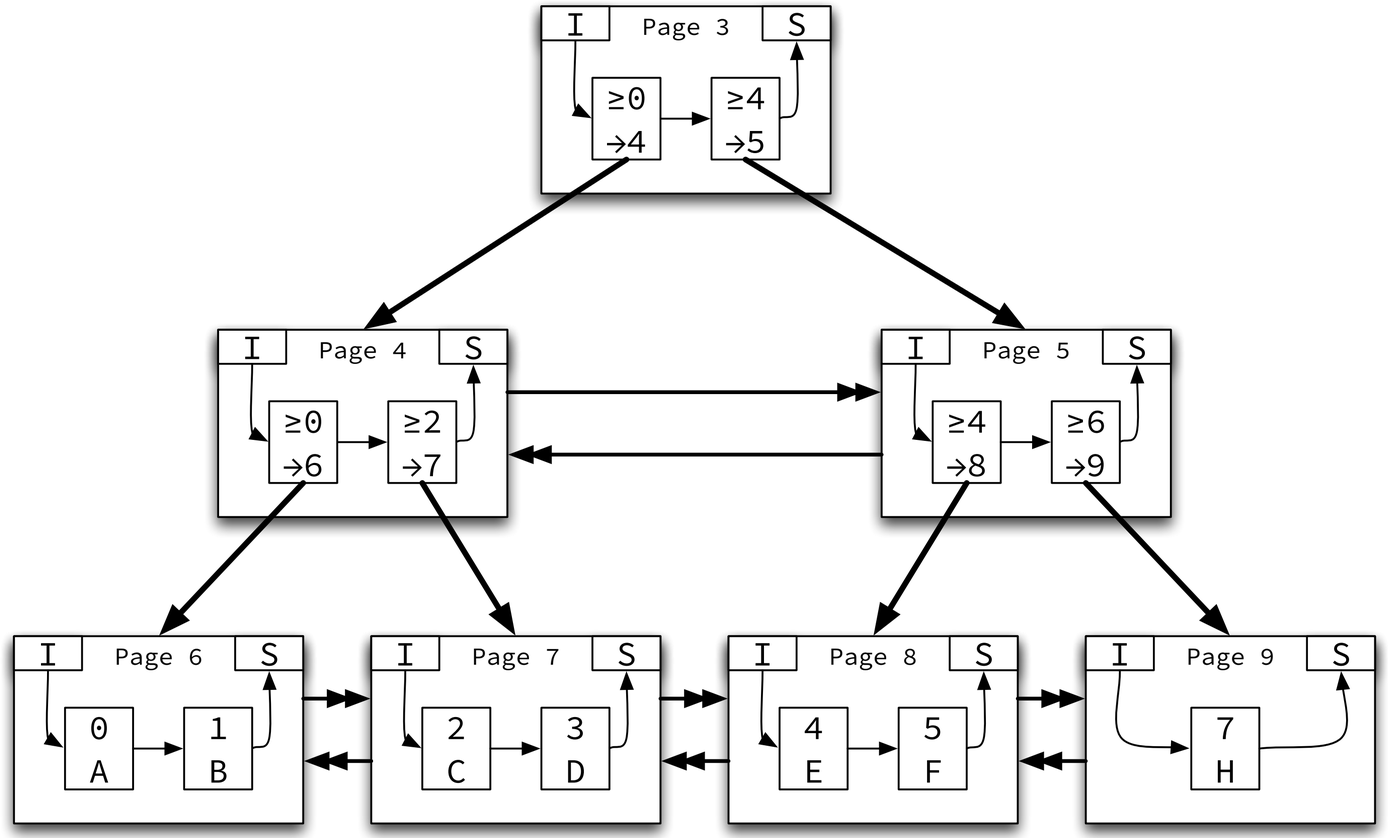
InnoDB fails to merge under-filled pages depending on deletion order

1. Pages are full or close to it



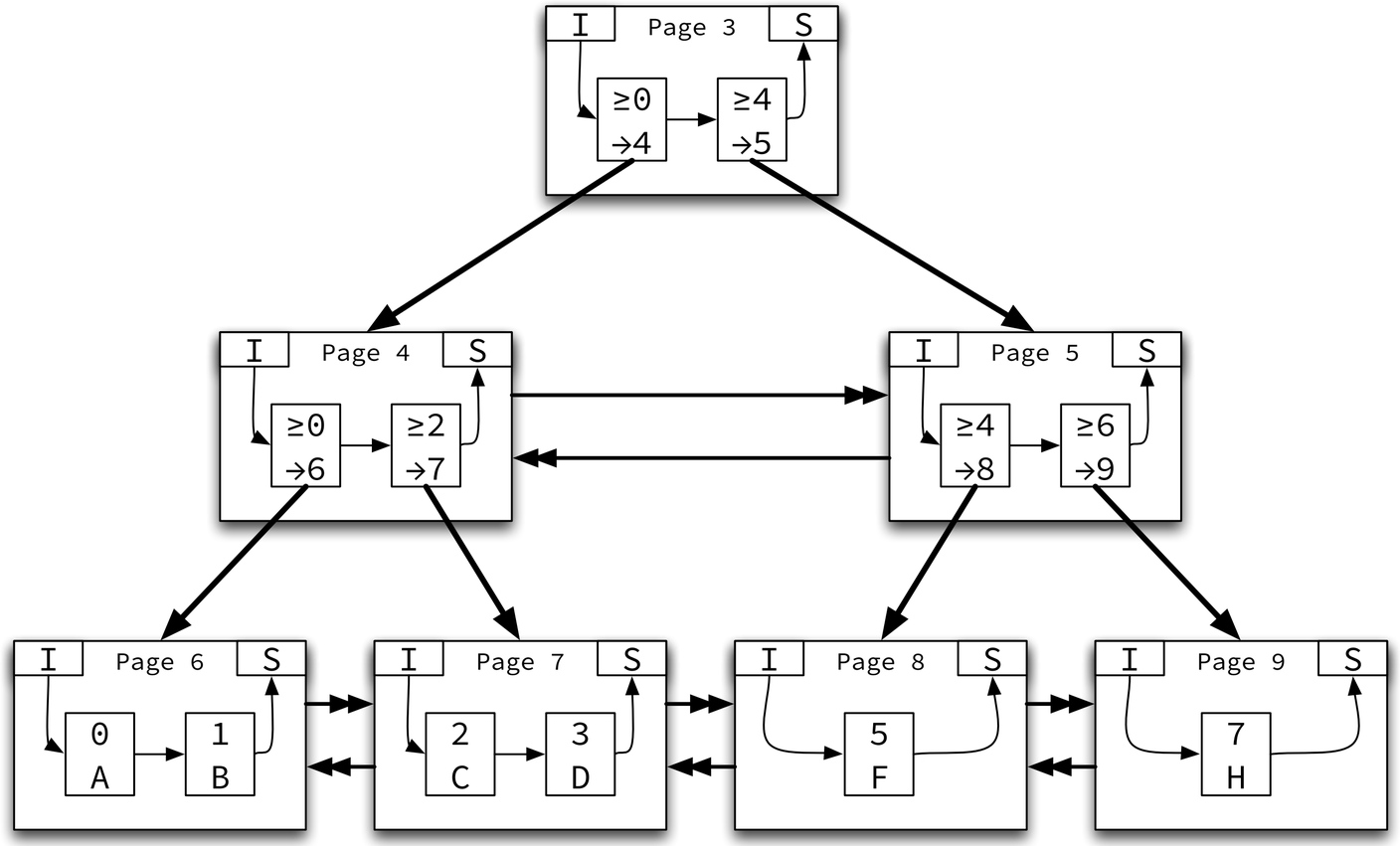
Depending on the order that records are deleted from pages, InnoDB may not merge multiple adjacent under-filled pages together, wasting disk space.

2. Many rows are deleted



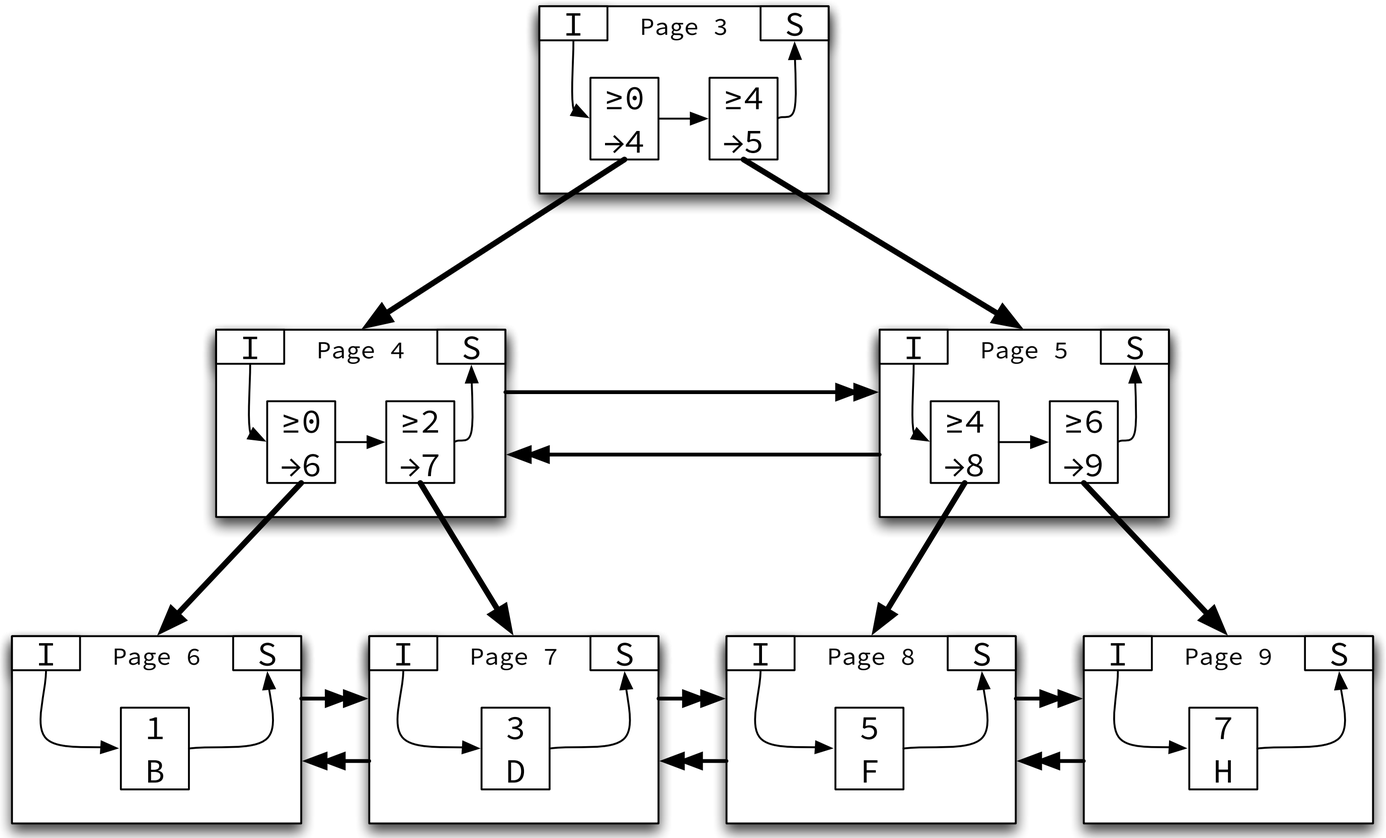
For example, deletes in descending key order.

3. Adjacent pages aren't merged



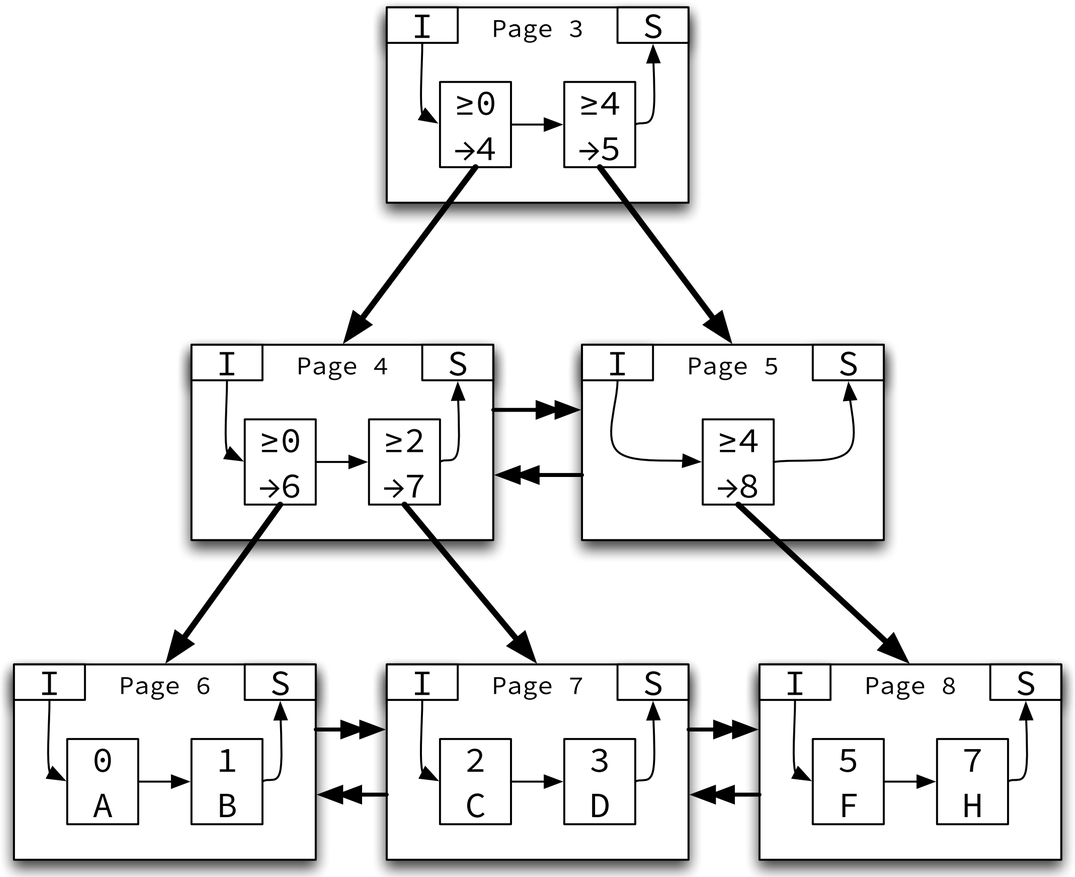
InnoDB only attempts to merge a less than half full page with its left sibling, failing to check if the sibling page to the right has enough space.

4. Many pages are underfilled



Worst-case scenario.

(What it should look like...)

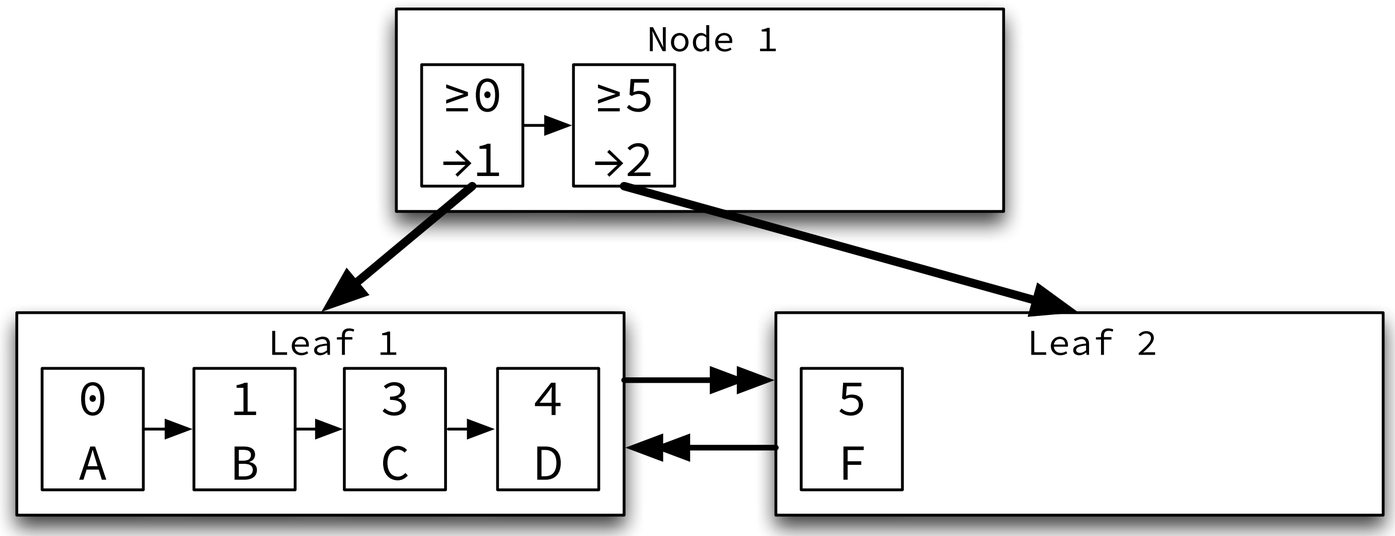


What it would look like if pages were properly merged.

Bug#68545

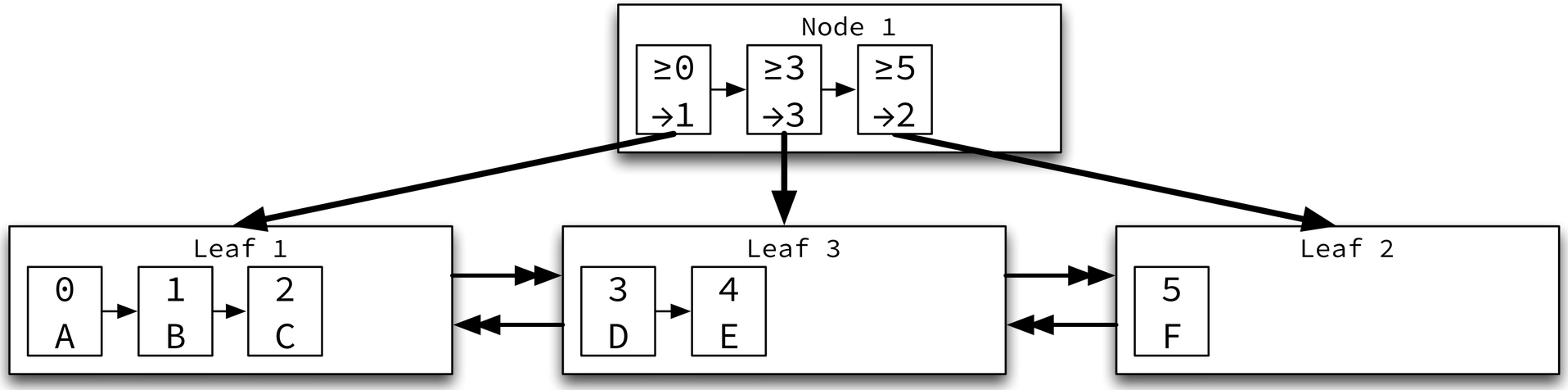
InnoDB should check left/right pages when target page is full to avoid splitting

Page is full, but record must be inserted



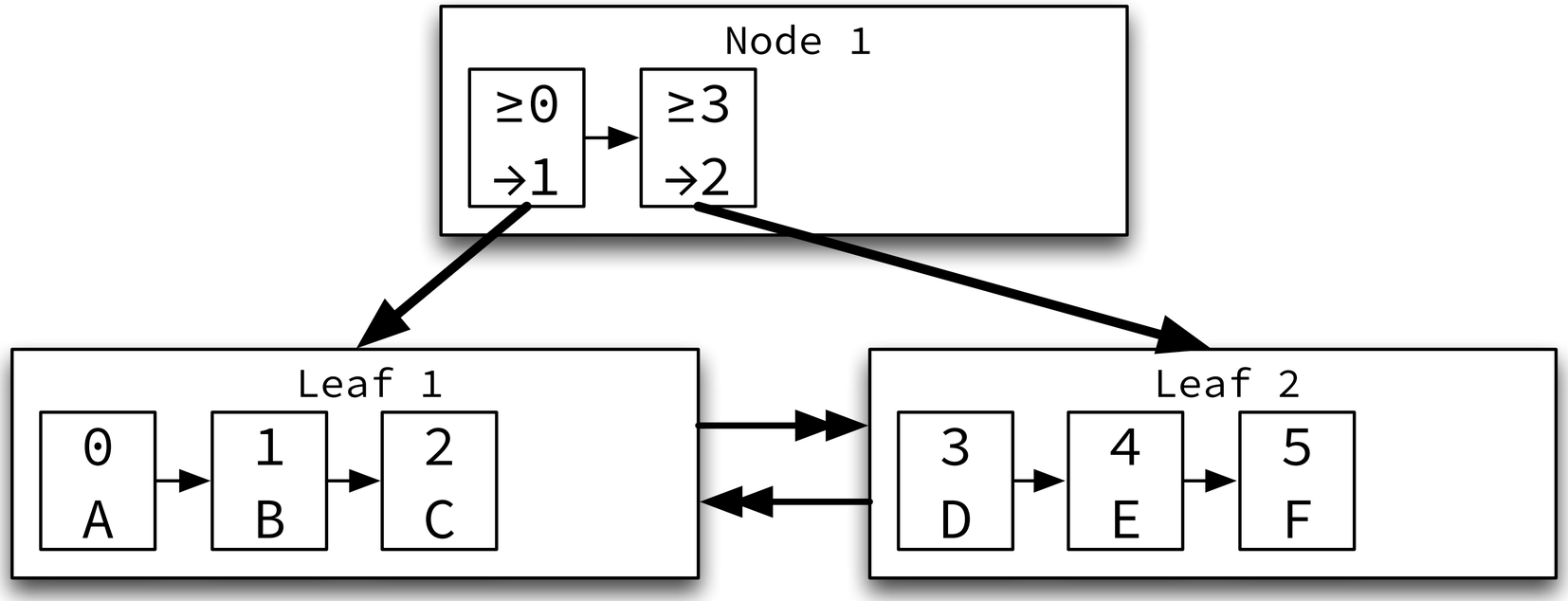
Insert a record with a key (2) that falls within a full page. A split ensues.

Currently: InnoDB always splits the page



The target page is split but one or more of its adjacent pages have free space.

Better: Rebalance records between pages

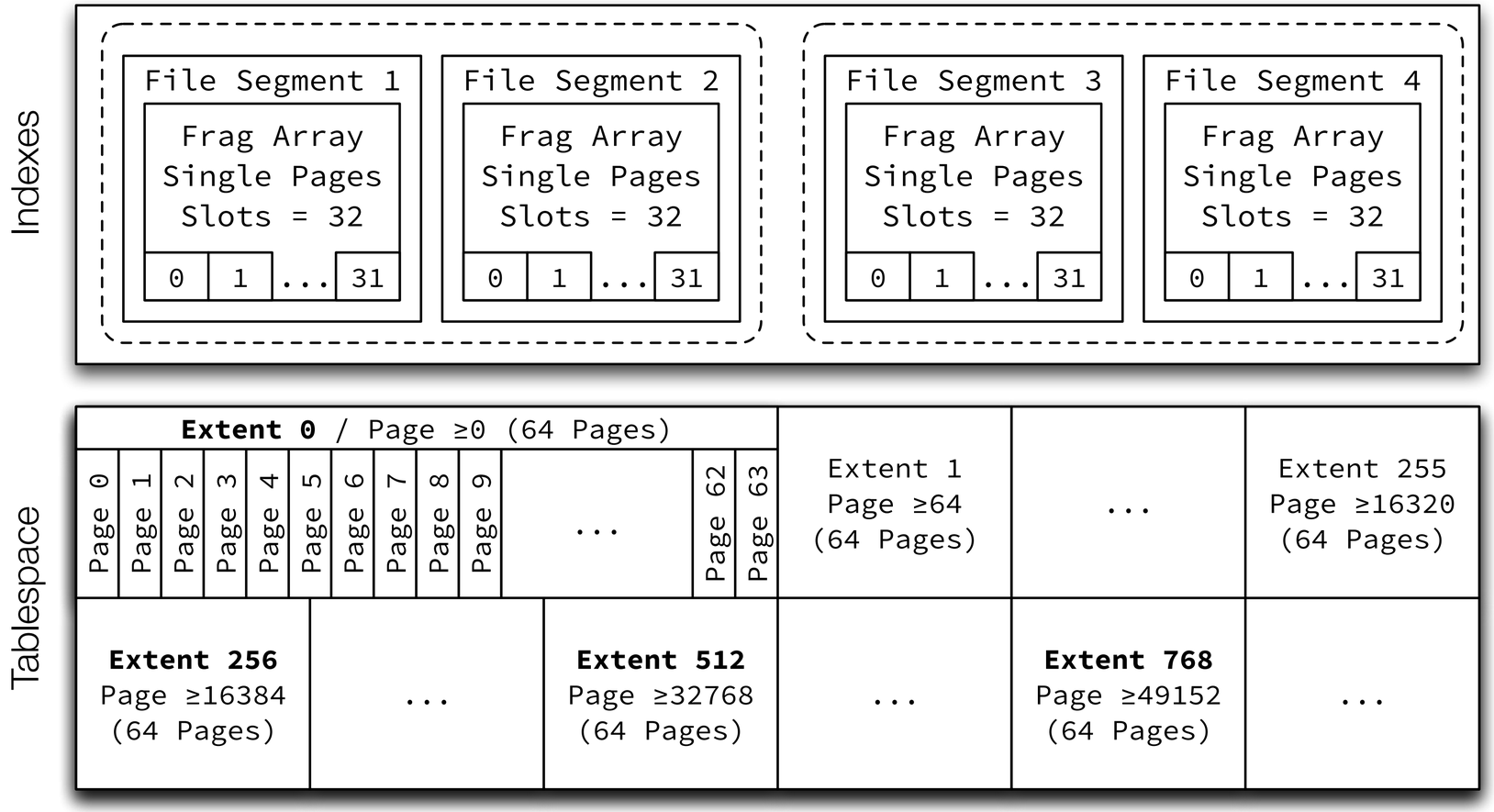


InnoDB should attempt to rebalance or merge the adjacent pages in order to make free space on the target page, rather than split the target page.

Bug #67963

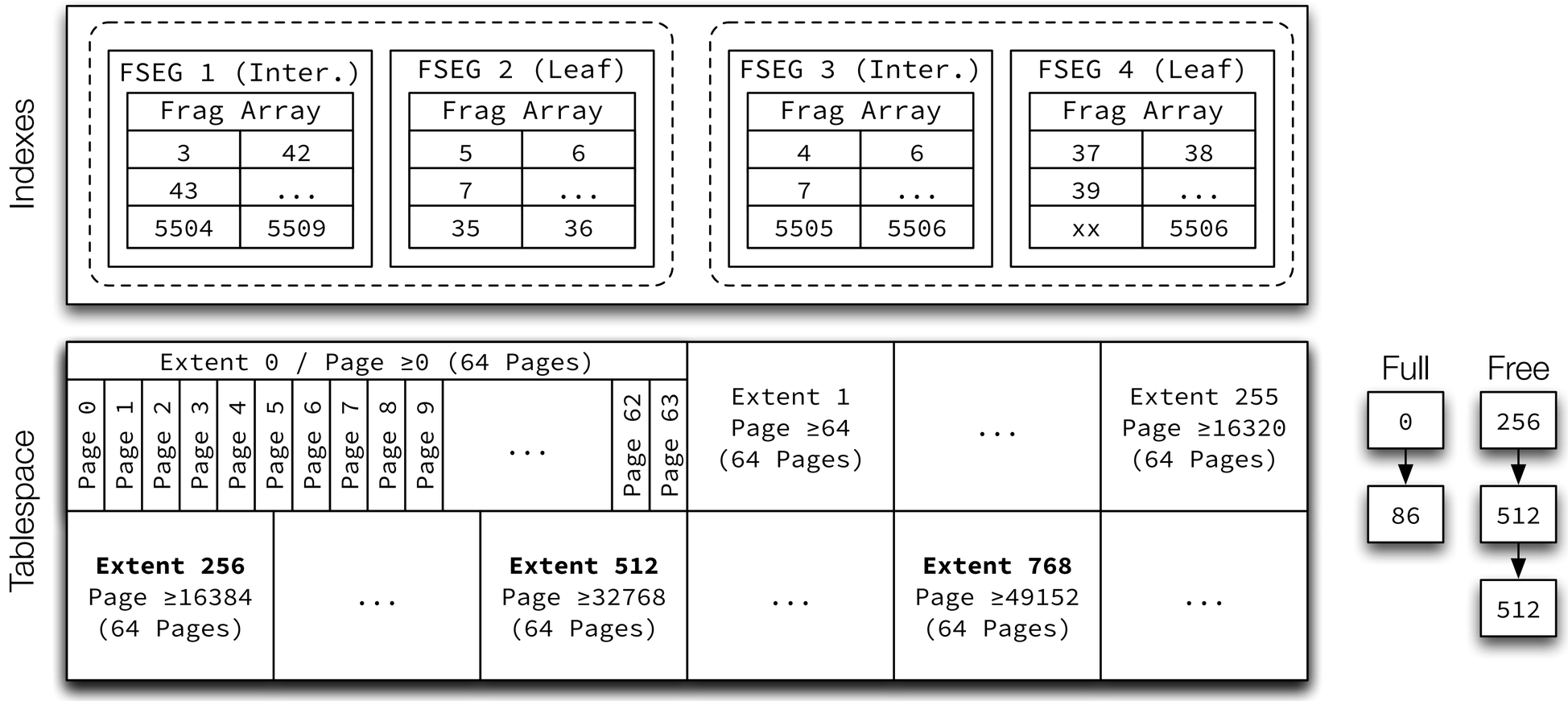
InnoDB wastes 62 out of every 16384 pages

Two pages are needed per 16,384 pages, but a full extent (64 pages) is allocated



Two bookkeeping pages for every 256 MiB of data. Remainder of the extent is used for single page (FREE_FRAG), but only 32 single pages per file segment.

As space grows, remaining 62 pages can't be used for anything



Once the file segments have allocated 32 fragment pages each, the remaining fragment pages are left unused.

Bug #68023

InnoDB reserves an excessive amount of disk space for write operations

Pre-emptive allocation for writes

InnoDB preallocates and reserves up to 1% of the total size of the table when performing operations that are likely to expand the B+Tree

That is 2 extents plus 0.5% for undo logs and 0.5% for purge

Attempts to preemptively fail operations if running out of disk space.

What's wrong with that?

Reasonable for tables smaller than a few gigabytes, but not for tables sized at tens of gigabytes. Also, the reservation approach is of dubious applicability.

- Undo logs does not apply for file per table
- Not every purge requires node pointer updates

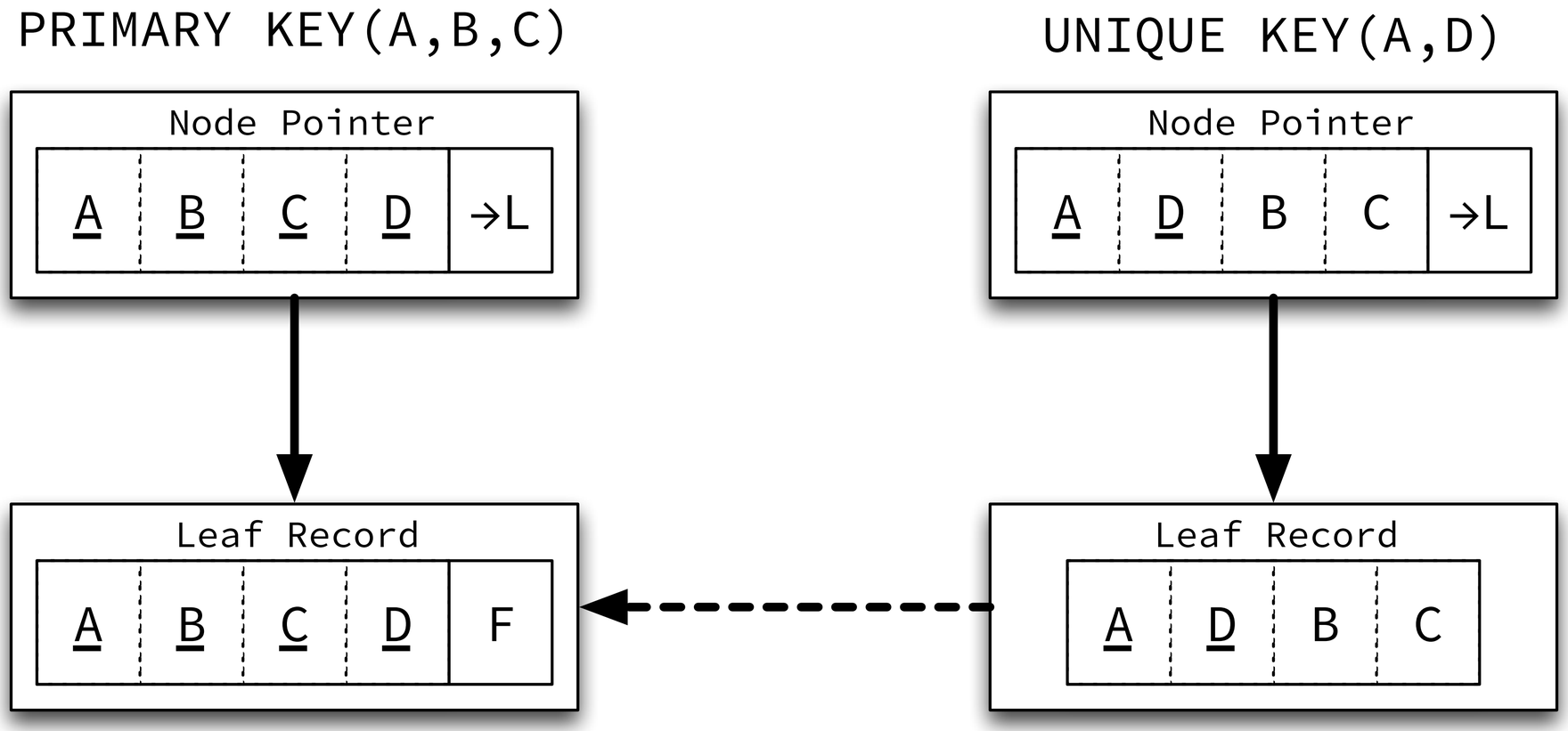
A fix: Add "reservation factor" that provides a way to either completely disable free extents reservation or to control the amount of free extents that are reserved for such operations.

How to fix it?

Bug #68546

InnoDB stores unnecessary PKV fields in unique SK non-leaf pages

Bug#68546: InnoDB stores unnecessary PKV fields in unique SK non-leaf pages



Node pointers for secondary indexes contain all fields in the internal representation of the index, not just these that can be used to uniquely determine an index entry. They are not read or updated.