Multi-Versioning and Secondary Indexes

innoDB is a multi-version storage engine. It keeps information about old versions of changed rows to support transactional features such as concurrency and rollback. This information is stored in undo tablespaces in a data structure called a rollback segment. See Section 15.6.3.4, “Undo Tablespaces”. innoDB uses the information in the rollback segment to perform the undo operations needed in a transaction rollback. It also uses the information to build earlier versions of a row for a consistent read. See Section 15.7.2.3, “Consistent Nonlocking Reads”.

Internally, innoDB adds three fields to each row stored in the database:

* A 6-byte db\_trx\_id field indicates the transaction identifier for the last transaction that inserted or updated the row. Also, a deletion is treated internally as an update where a special bit in the row is set to mark it as deleted.
* A 7-byte db\_roll\_ptr field called the roll pointer. The roll pointer points to an undo log record written to the rollback segment. If the row was updated, the undo log record contains the information necessary to rebuild the content of the row before it was updated.
* A 6-byte db\_row\_id field contains a row ID that increases monotonically as new rows are inserted. If innoDB generates a clustered index automatically, the index contains row ID values. Otherwise, the db\_row\_id column does not appear in any index.

Undo logs in the rollback segment are divided into insert and update undo logs. Insert undo logs are needed only in transaction rollback and can be discarded as soon as the transaction commits. Update undo logs are used also in consistent reads, but they can be discarded only after there is no transaction present for which innoDB has assigned a snapshot that in a consistent read could require the information in the update undo log to build an earlier version of a database row. For additional information about undo logs, see Section 15.6.6, “Undo Logs”.

It is recommend that you commit transactions regularly, including transactions that issue only consistent reads. Otherwise, innoDB cannot discard data from the update undo logs, and the rollback segment may grow too big, filling up the undo tablespace in which it resides. For information about managing undo tablespaces, see Section 15.6.3.4, “Undo Tablespaces”.

The physical size of an undo log record in the rollback segment is typically smaller than the corresponding inserted or updated row. You can use this information to calculate the space needed for your rollback segment.

In the innoDB multi-versioning scheme, a row is not physically removed from the database immediately when you delete it with an SQL statement. innoDB only physically removes the corresponding row and its index records when it discards the update undo log record written for the deletion. This removal operation is called a purge, and it is quite fast, usually taking the same order of time as the SQL statement that did the deletion.

If you insert and delete rows in smallish batches at about the same rate in the table, the purge thread can start to lag behind and the table can grow bigger and bigger because of all the “dead” rows, making everything disk-bound and very slow. In such cases, throttle new row operations, and allocate more resources to the purge thread by tuning the innodb\_max\_purge\_lag system variable. For more information, see Section 15.8.9, “Purge Configuration".

Multi-Versioning and Secondary Indexes

innoDB multiversion concurrency control (MVCC) treats secondary indexes differently than clustered indexes. Records in a clustered index are updated in-place, and their hidden system columns point undo log entries from which earlier versions of records can be reconstructed. Unlike clustered index records, secondary index records do not contain hidden system columns nor are they updated in-place.

When a secondary index column is updated, old secondary index records are delete-marked, new records are inserted, and delete-marked records are eventually purged. When a secondary index

InnoDB Architecture

record is delete-marked or the secondary index page is updated by a newer transaction, InnoDB looks up the database record in the clustered index. In the clustered index, the record's db\_trx\_id is checked, and the correct version of the record is retrieved from the undo log if the record was modified after the reading transaction was initiated.

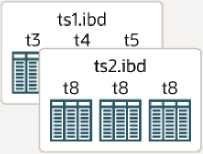
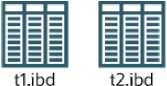
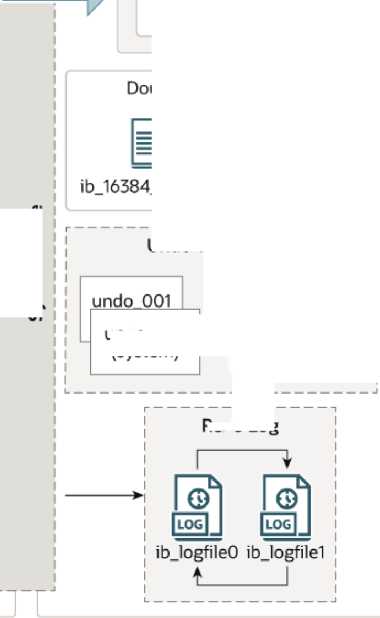
If a secondary index record is marked for deletion or the secondary index page is updated by a newer transaction, the covering index technique is not used. Instead of returning values from the index structure, InnoDB looks up the record in the clustered index.

However, if the index condition pushdown (ICP) optimization is enabled, and parts of the where condition can be evaluated using only fields from the index, the MySQL server still pushes this part of the where condition down to the storage engine where it is evaluated using the index. If no matching records are found, the clustered index lookup is avoided. If matching records are found, even among delete-marked records, InnoDB looks up the record in the clustered index.

15.4 InnoDB Architecture

The following diagram shows in-memory and on-disk structures that comprise the InnoDB storage engine architecture. For information about each structure, see [Section 15.5, “InnoDB In-Memory](#bookmark19) [Structures"](#bookmark19), and Section 15.6, “InnoDB On-Disk Structures".

**Figure 15.1 InnoDB Architecture**



**In-Memory Structures**

Buffer Pool

□口¥□■

DDDCin !

■□□□□；

□ □□□□

r—— — — — — — — —

Change Buffer

!□□□□■!

System Tablespace (ibdatal)

**On-Disk Structures**

Log Buffer

Change Buffer

Undo Tablespaces

undcLOOS^ibu.丄

I 厂— 31

j undo\_004.ibu i '(user-defined) •

*I* 」I

a a

.O.dblwr ib\_16384\_1.dblwr

\ \ ^emQ libt\_Ln

I [J 2Jbt *1 n |* 打 temp^Jbt ；[

I (session)丨；

OIP8 EoltJAS Dou 专」wdo

undo\_002  
(system)

Redo Log

File-Per-Table Tablespaces innodb\_file\_per\_table=ON

General Tablespaces

Temporary Tablespaces

ibtmpl  
(global)

**■ I**

15.5 InnoDB In-Memory Structures

This section describes InnoDB in-memory structures and related topics.

Buffer Pool

15.5.1 Buffer Pool

The buffer pool is an area in main memory where innoDB caches table and index data as it is accessed. The buffer pool permits frequently used data to be accessed directly from memory, which speeds up processing. On dedicated servers, up to 80% of physical memory is often assigned to the buffer pool.

For efficiency of high-volume read operations, the buffer pool is divided into pages that can potentially hold multiple rows. For efficiency of cache management, the buffer pool is implemented as a linked list of pages; data that is rarely used is aged out of the cache using a variation of the least recently used (LRU) algorithm.

Knowing how to take advantage of the buffer pool to keep frequently accessed data in memory is an important aspect of MySQL tuning.

**Buffer Pool LRU Algorithm**

The buffer pool is managed as a list using a variation of the LRU algorithm. When room is needed to add a new page to the buffer pool, the least recently used page is evicted and a new page is added to the middle of the list. This midpoint insertion strategy treats the list as two sublists:

* At the head, a sublist of new ("young”)pages that were accessed recently
* At the tail, a sublist of old pages that were accessed less recently

New Sublist -

5/8

Midpoint

insertion

Old Sub list

3/8

■—I'

Evicted paga^

>£gmr&d Egml/>

\_r-3csa pages



Buffer Pool

The algorithm keeps frequently used pages in the new sublist. The old sublist contains less frequently used pages; these pages are candidates for eviction.

By default, the algorithm operates as follows:

* 3/8 of the buffer pool is devoted to the old sublist.
* The midpoint of the list is the boundary where the tail of the new sublist meets the head of the old sublist.
* When innoDB reads a page into the buffer pool, it initially inserts it at the midpoint (the head of the old sublist). A page can be read because it is required for a user-initiated operation such as an SQL query, or as part of a read-ahead operation performed automatically by innoDB.
* Accessing a page in the old sublist makes it “young", moving it to the head of the new sublist. If the page was read because it was required by a user-initiated operation, the first access occurs immediately and the page is made young. If the page was read due to a read-ahead operation, the first access does not occur immediately and might not occur at all before the page is evicted.
* As the database operates, pages in the buffer pool that are not accessed “age" by moving toward the tail of the list. Pages in both the new and old sublists age as other pages are made new. Pages in the old sublist also age as pages are inserted at the midpoint. Eventually, a page that remains unused reaches the tail of the old sublist and is evicted.

By default, pages read by queries are immediately moved into the new sublist, meaning they stay in the buffer pool longer. A table scan, performed for a mysqldump operation or a select statement with no where clause, for example, can bring a large amount of data into the buffer pool and evict an equivalent amount of older data, even if the new data is never used again. Similarly, pages that are loaded by the read-ahead background thread and accessed only once are moved to the head of the new list. These situations can push frequently used pages to the old sublist where they become subject to eviction. For information about optimizing this behavior, see Section 15.8.3.3, “Making the Buffer Pool Scan Resistant", and Section 15.8.3.4, “Configuring InnoDB Buffer Pool Prefetching (Read- Ahead)".

innoDB Standard Monitor output contains several fields in the buffer pool and memory section regarding operation of the buffer pool LRU algorithm. For details, see Monitoring the Buffer Pool Using the InnoDB Standard Monitor.

Buffer Pool Configuration

You can configure the various aspects of the buffer pool to improve performance.

* Ideally, you set the size of the buffer pool to as large a value as practical, leaving enough memory for other processes on the server to run without excessive paging. The larger the buffer pool, the more innoDB acts like an in-memory database, reading data from disk once and then accessing the data from memory during subsequent reads. See Section 15.8.3.1, “Configuring InnoDB Buffer Pool Size".
* On 64-bit systems with sufficient memory, you can split the buffer pool into multiple parts to minimize contention for memory structures among concurrent operations. For details, see Section 15.8.3.2, “Configuring Multiple Buffer Pool Instances".
* You can keep frequently accessed data in memory regardless of sudden spikes of activity from operations that would bring large amounts of infrequently accessed data into the buffer pool. For details, see Section 15.8.3.3, “Making the Buffer Pool Scan Resistant".
* You can control how and when to perform read-ahead requests to prefetch pages into the buffer pool asynchronously in anticipation of impending need for them. For details, see Section 15.8.3.4, “Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)".
* You can control when background flushing occurs and whether or not the rate of flushing is dynamically adjusted based on workload. For details, see Section 15.8.3.5, “Configuring Buffer Pool Flushing".