The Internals of PostgreSQL

for database administrators and system developers

Chapter 7

Heap Only Tuple and Index-Only Scans

This chapter describes two features related to the index scan, which are the heap only tuple and index-only scans.

7.1. Heap Only Tuple (HOT)

The HOT was implemented in version 8.3 to effectively use the pages of both index and table when the updated row is stored in the same table page that stores the old row. The HOT also reduces the necessity of VACUUM processing.

Since the details of HOT are described in the README.HOT in the source code directory, this chapter briefly introduces HOT. Firstly, Section 7.1.1 describes how to update a row without HOT to clarify the issues that the resolves. Next, Section 7.1.2 describes how HOT performs.

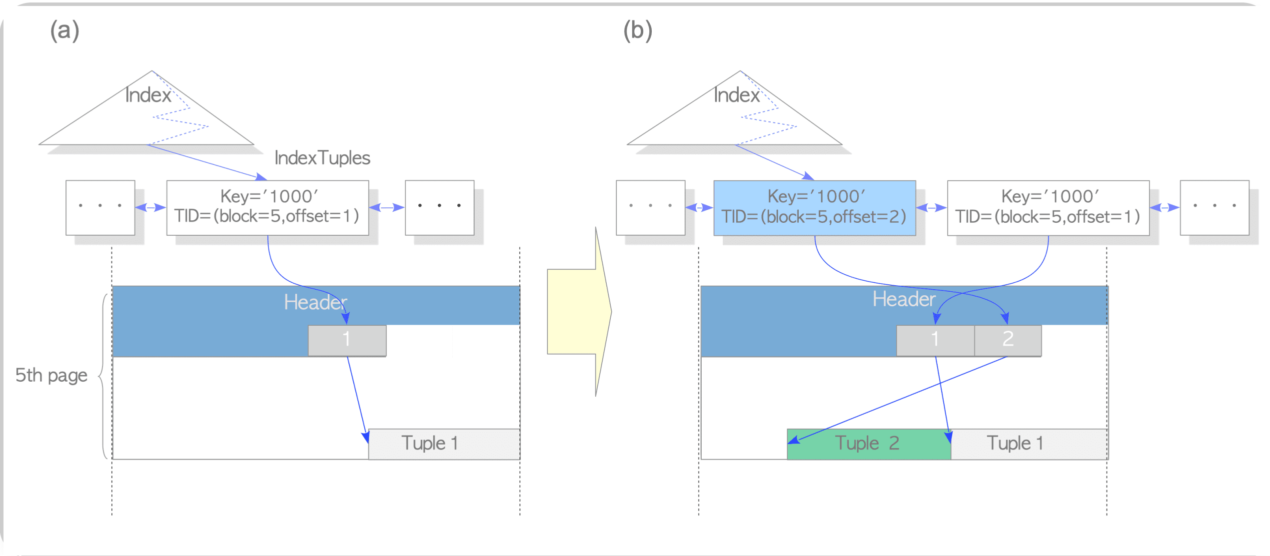
7.1.1. Update a Row Without HOT

Assume that the table 'tbl' has two columns: 'id' and 'data'; 'id' is the primary key of 'tbl'.

```
testdb=# \d tbl
                Table "public.tbl"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 data   | text    |           |          |
Indexes:
    "tbl_pkey" PRIMARY KEY, btree (id)
```

The table ‘tbl’ has 1000 tuples; the last tuple of which the id is ‘1000’ is stored in the 5th page of the table. The last tuple is pointed from the corresponding index tuple, of which the key is ‘1000’ and whose tid is ‘(5,1)’. Refer to Fig. 7.1(a).

Fig. 7.1. Update a row without HOT



We consider how the last tuple is updated without HOT.

```
testdb=# UPDATE tbl SET data = 'B' WHERE id = 1000;
```

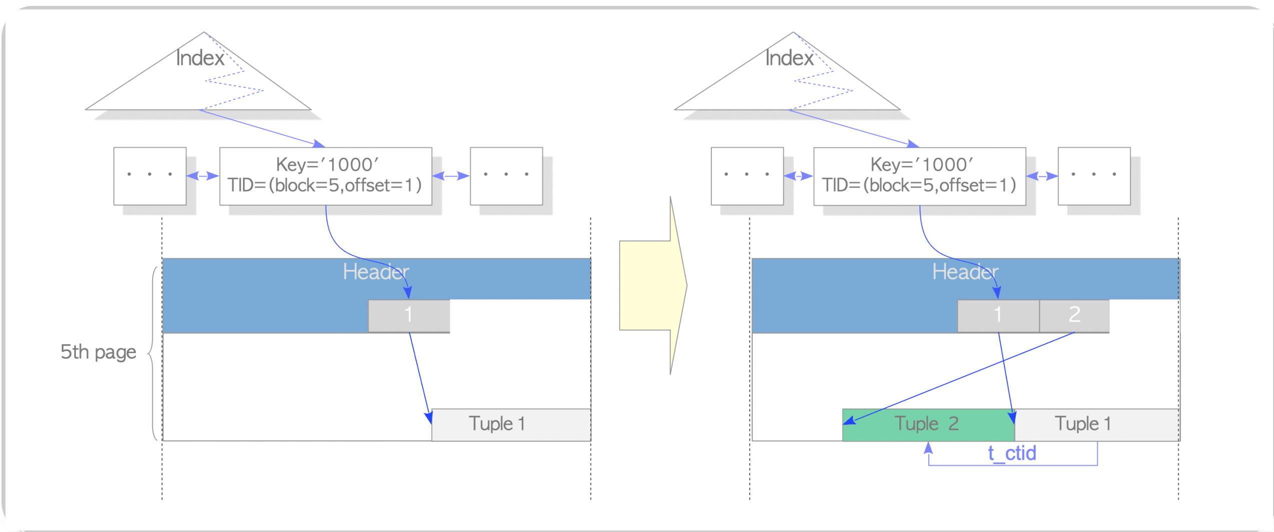
In this case, PostgreSQL inserts not only the new table tuple but also the new index tuple in the index page. Refer to Fig. 7.1(b).

The inserting of the index tuples consumes the index page space, and both the inserting and vacuuming costs of the index tuples are high. The HOT reduces the impact of these issues.

7.1.2. How HOT Performs

When a row is updated with HOT, if the updated row will be stored in the same table page that stores the old row, PostgreSQL does not insert the corresponding index tuple and sets the HEAP_HOT_UPDATED bit and the HEAP_ONLY_TUPLE bit to the t_informask2 fields of the old tuple and the new tuple, respectively. Refer to Figs. 7.2 and 7.3.

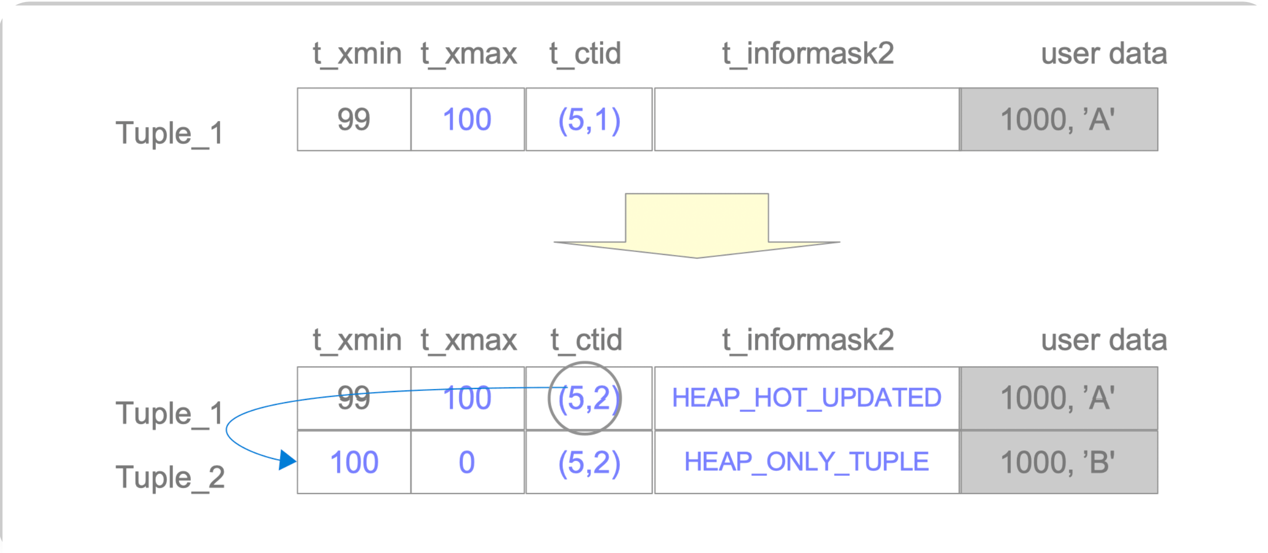
Fig. 7.2. Update a row with HOT



For example, in this case, 'Tuple_1' and 'Tuple_2' are set to the HEAP_HOT_UPDATED bit and the HEAP_ONLY_TUPLE bit, respectively.

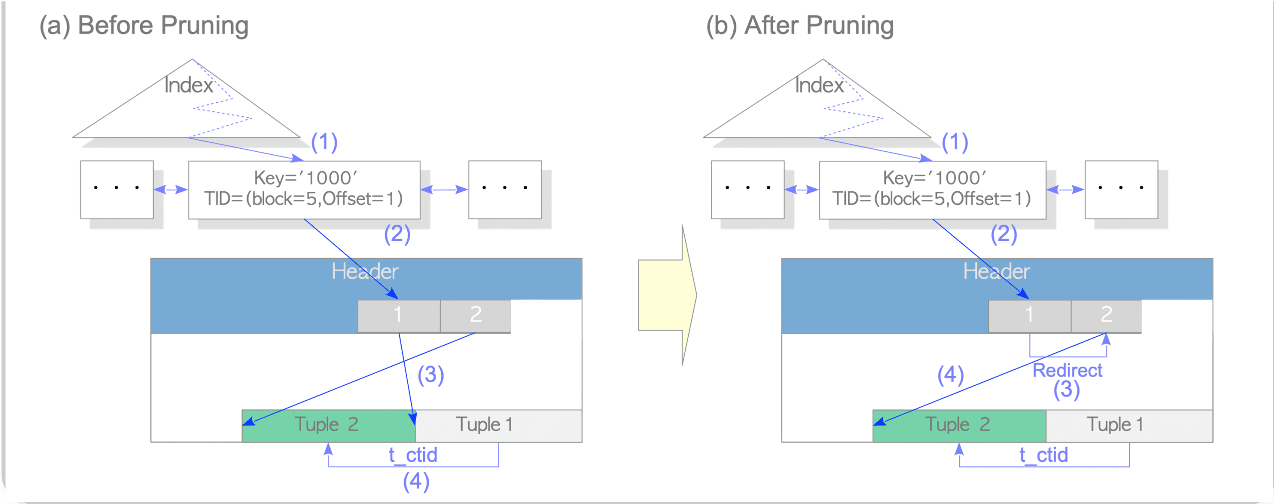
In addition, the HEAP_HOT_UPDATED and the HEAP_ONLY_TUPLE bits are used regardless of the *pruning* and the *defragmentation* processes, which are described in the following, are executed.

Fig. 7.3. HEAP_HOT_UPDATED and HEAP_ONLY_TUPLE bits



In the following, a description of how PostgreSQL accesses the updated tuples using the index scan just after updating the tuples with HOT is given. Refer to Fig. 7.4(a).

Fig. 7.4. Pruning of the line pointers



(1) Find the index tuple that points to the target tuple.
(2) Access the line pointer '[1]' that is pointed from the getting index tuple.
(3) Read 'Tuple_1'.
(4) Read 'Tuple_2' via the t_ctid of 'Tuple_1'.

In this case, PostgreSQL reads two tuples, 'Tuple_1' and 'Tuple_2', and decides which is visible using the concurrency control mechanism described in Chapter 5.

However, a problem arises if the dead tuples in the table pages are removed. For example, in Fig. 7.4(a), if 'Tuple_1' is removed since it is a dead tuple, 'Tuple_2' cannot be accessed from the index.

To resolve this problem, at an appropriate time, PostgreSQL redirects the line pointer that points to the old tuple to the line pointer that points to the new tuple. In PostgreSQL, this processing is called

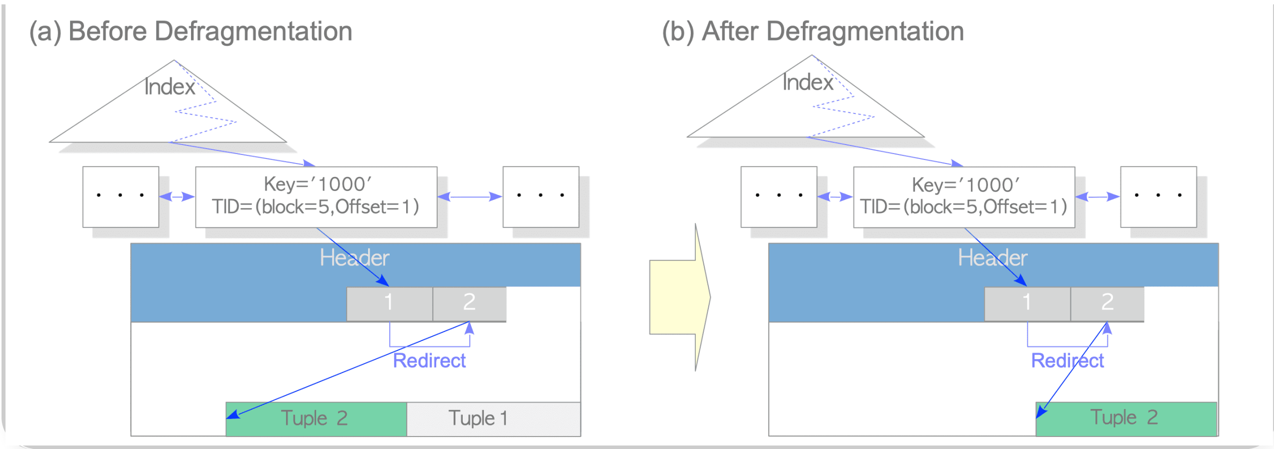
pruning. Fig. 7.4(b) despicts how PostgreSQL accesses the updated tuples after pruning.

(1) Find the index tuple.
(2) Access the line pointer '[1]' that is pointed from the getting index tuple.
(3) Access the line pointer '[2]' that points to 'Tuple_2' via the redirected line pointer.
(4) Read 'Tuple_2' that is pointed from the line pointer '[2]'.

The pruning processing will be executed, if possible, when a SQL command is executed such as SELECT, UPDATE, INSERT and DELETE. The exact execution timing is not described in this chapter because it is very complicated. The details are described in the README.HOT file.

PostgreSQL removes dead tuples if possible, as in the pruning process, at an appropriate time. In the document of PostgreSQL, this processing is called **defragmentation**. Fig. 7.5 despicts the defragmentation by HOT.

Fig. 7.5. Defragmentation of the dead tuples



Note that the cost of defragmentation is less than the cost of normal VACUUM processing because defragmentation does not involve removing the index tuples.

Thus, using HOT reduces the consumption of both indexes and tables of pages; this also reduces the number of tuples that the VACUUM processing has to process. Therefore, HOT has a good influence on performance because it eventually reduces the number of insertions of the index tuples by updating and the necessity of VACUUM processing.

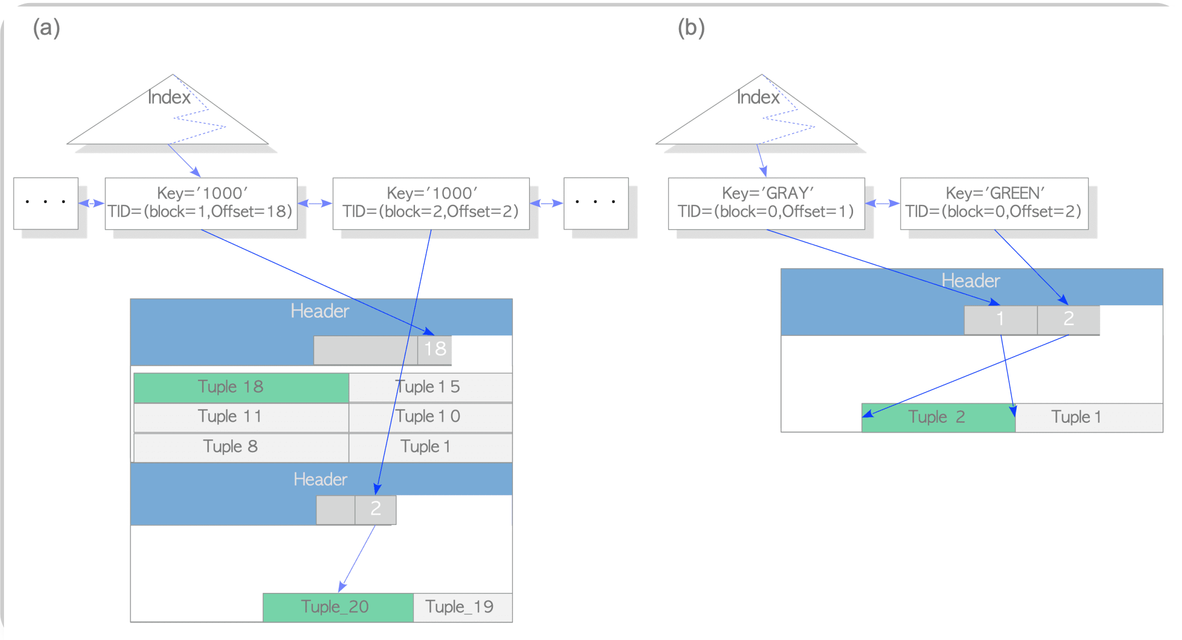
ℹ The Cases in which HOT is not available

To clearly understand how HOT performs, the cases in which HOT is not available are described.

When the updated tuple is stored in the other page, which does not store the old tuple, the index tuple that points to the tuple is also inserted in the index page. Refer to Fig. 7.6(a).

When the key value of the index tuple is updated, the new index tuple is inserted in the index page. Refer to Fig. 7.6(b).

Fig. 7.6. The Cases in which HOT is not available



The statistics related to the HOT

The pg_stat_all_tables view provides a statistics value for each table. See also this extension.

7.2. Index-Only Scans

To reduce the I/O (input/output) cost, index-only scans (often called index-only access) directly use the index key without accessing the corresponding table pages when all of the target entries of the SELECT statement are included in the index key. This technique is provided by almost all commercial RDBMS, such as DB2 and Oracle. PostgreSQL has introduced this option since version 9.2.

In the following, using a specific example, a description of how index-only scans in PostgreSQL perform is given.

The assumptions of the example areexplained below:

- Table definition
  We have a table 'tbl' of which the definition is shown below:

```
testdb=# \d tbl
      Table "public.tbl"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 name   | text    |
```

```
  data      | text      |
 Indexes:
     "tbl_idx" btree (id, name)
```

- Index
  The table 'tbl' has an index 'tbl_idx', which is composed of two columns: 'id' and 'name'.
- Tuples
  'tbl' has already inserted tuples.
  'Tuple_18', of which the id is '18' and name is 'Queen', is stored in the 0th page.
  'Tuple_19', of which the id is '19' and name is 'BOSTON', is stored in the 1st page.
- Visibility
  All tuples in the 0th page are always visible; the tuples in the 1st page are not always visible. Note that the visibility of each page is stored in the corresponding visibility map, and the visibility map is described in Section 6.2.

Let us explore how PostgreSQL reads tuples when the following SELECT command is executed.

```
testdb=# SELECT id, name FROM tbl WHERE id BE
TWEEN 18 and 19;
 id |  name
----+--------
 18 | Queen
 19 | Boston
(2 rows)
```

This query gets data from two columns of the table: 'id' and 'name', and the index 'tbl_idx' is composed of these columns. Thus, when using index scan, it seems at first glance that accessing the table pages is not required because the index tuples contain the necessary data. However, in fact, PostgreSQL has to check the visibility of the tuples in principle, and the index tuples do not have any information about transactions such as the t_xmin and t_xmax of the heap tuples, which are described in Section 5.2. Therefore, PostgreSQL has to access the table data to check the visibility of the data in the index tuples. This is like putting the cart before the horse.

To avoid this dilemma, PostgreSQL uses the visibility map of the target table. If all tuples stored in a page are visible, PostgreSQL uses the key of the index tuple and does not access the table page that is pointed at from the index tuple to check its visibility; otherwise, PostgreSQL reads the table tuple that is pointed at from the index tuple and checks the visibility of the tuple, which is the ordinary process.

In this example, 'Tuple_18' need not be accessed because the 0th page that stores 'Tuple_18' is visible, that is, all tuples including Tuple_18 in the 0th page are visible. In contrast, 'Tuple_19' needs to be accessed to treat the concurrency control because the visibility of the 1st page is not visible. Refer to Fig. 7.7.

Fig. 7.7. How Index-Only Scans performs