Chapter 66. Database Physical Storage

This chapter provides an overview of the physical storage format used by PostgreSQL databases.

66.1. Database File Layout

This section describes the storage format at the level of files and directories.

Traditionally, the configuration and data files used by a database cluster are stored together within the cluster's data directory, commonly referred to as PGDATA (after the name of the environment variable that can be used to define it). A common location for PGDATA is /var/lib/pgsql/data. Multiple clusters, managed by different server instances, can exist on the same machine.

The PGDATA directory contains several subdirectories and control files, as shown in Table 66.1. In addition to these required items, the cluster configuration files postgresql.conf, pg_hba.conf, and pg_ident.conf are traditionally stored in PGDATA, although it is possible to place them elsewhere.

Table 66.1. Contents of PGDATA

Item	Description
PG_VERSION	A file containing the major version number of PostgreSQL
base	Subdirectory containing per-database subdirectories
current_logfiles	File recording the log file(s) currently written to by the logging collector
global	Subdirectory containing cluster-wide tables, such as pg_database
pg_commit_ts	Subdirectory containing transaction commit timestamp data
pg_dynshmem	Subdirectory containing files used by the dynamic shared memory subsystem
pg_logical	Subdirectory containing status data for logical decoding
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN/NOTIFY status data
pg_replslot	Subdirectory containing replication slot data
pg_serial	Subdirectory containing information about committed serializable transactions
pg_snapshots	Subdirectory containing exported snapshots
pg_stat	Subdirectory containing permanent files for the statistics subsystem
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces
pg_twophase	Subdirectory containing state files for prepared transactions
pg_wal	Subdirectory containing WAL (Write Ahead Log) files
pg_xact	Subdirectory containing transaction commit status data
postgresql.auto.conf	A file used for storing configuration parameters that are set by ALTER SYSTEM
postmaster.opts	A file recording the command-line options the server was last started with

Item	Description
postmaster.pid	A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (could be empty), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)

For each database in the cluster there is a subdirectory within PGDATA/base, named after the database's OID in pg_database. This subdirectory is the default location for the database's files; in particular, its system catalogs are stored there.

Note that the following sections describe the behavior of the builtin heap table access method, and the builtin index access methods. Due to the extensible nature of PostgreSQL, other access methods might work differently.

Each table and index is stored in a separate file. For ordinary relations, these files are named after the table or index's *filenode* number, which can be found in pg_class.relfilenode. But for temporary relations, the file name is of the form t*BBB*_*FFF*, where *BBB* is the process number of the backend which created the file, and *FFF* is the filenode number. In either case, in addition to the main file (a/k/a main fork), each table and index has a *free space map* (see Section 66.3), which stores information about free space available in the relation. The free space map is stored in a file named with the filenode number plus the suffix _fsm. Tables also have a *visibility map*, stored in a fork with the suffix _vm, to track which pages are known to have no dead tuples. The visibility map is described further in Section 66.4. Unlogged tables and indexes have a third fork, known as the initialization fork, which is stored in a fork with the suffix _init (see Section 66.5).

> **Caution**
>
> Note that while a table's filenode often matches its OID, this is *not* necessarily the case; some operations, like TRUNCATE, REINDEX, CLUSTER and some forms of ALTER TABLE, can change the filenode while preserving the OID. Avoid assuming that filenode and table OID are the same. Also, for certain system catalogs including pg_class itself, pg_class.relfilenode contains zero. The actual filenode number of these catalogs is stored in a lower-level data structure, and can be obtained using the pg_relation_filenode() function.

When a table or index exceeds 1 GB, it is divided into gigabyte-sized *segments*. The first segment's file name is the same as the filenode; subsequent segments are named filenode.1, filenode.2, etc. This arrangement avoids problems on platforms that have file size limitations. (Actually, 1 GB is just the default segment size. The segment size can be adjusted using the configuration option --with-segsize when building PostgreSQL.) In principle, free space map and visibility map forks could require multiple segments as well, though this is unlikely to happen in practice.

A table that has columns with potentially large entries will have an associated *TOAST* table, which is used for out-of-line storage of field values that are too large to keep in the table rows proper. pg_class.reltoastrelid links from a table to its TOAST table, if any. See Section 66.2 for more information.

The contents of tables and indexes are discussed further in Section 66.6.

Tablespaces make the scenario more complicated. Each user-defined tablespace has a symbolic link inside the PGDATA/pg_tblspc directory, which points to the physical tablespace directory (i.e., the location specified in the tablespace's CREATE TABLESPACE command). This symbolic link is named after the tablespace's OID. Inside the physical tablespace directory there is a subdirectory with a name that depends on the PostgreSQL server version, such as PG_9.0_201008051. (The reason for using this subdirectory is so that successive versions of the database can use the same CREATE TABLESPACE location value without conflicts.) Within the version-specific subdirectory, there is a subdirectory for each database that has elements in the tablespace, named after the database's OID. Tables and indexes are stored within that directory, using the filenode naming scheme. The pg_default tablespace is not accessed through pg_tblspc, but corresponds to PGDATA/base. Similarly, the pg_global tablespace is not accessed through pg_tblspc, but corresponds to PGDATA/global.

The `pg_relation_filepath()` function shows the entire path (relative to `PGDATA`) of any relation. It is often useful as a substitute for remembering many of the above rules. But keep in mind that this function just gives the name of the first segment of the main fork of the relation — you may need to append a segment number and/or `_fsm`, `_vm`, or `_init` to find all the files associated with the relation.

Temporary files (for operations such as sorting more data than can fit in memory) are created within `PGDATA/base/pgsql_tmp`, or within a `pgsql_tmp` subdirectory of a tablespace directory if a tablespace other than `pg_default` is specified for them. The name of a temporary file has the form `pgsql_tmp`*`PPP`*`.`*`NNN`*, where *`PPP`* is the PID of the owning backend and *`NNN`* distinguishes different temporary files of that backend.

66.2. TOAST

This section provides an overview of TOAST (The Oversized-Attribute Storage Technique).

PostgreSQL uses a fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages. Therefore, it is not possible to store very large field values directly. To overcome this limitation, large field values are compressed and/or broken up into multiple physical rows. This happens transparently to the user, with only small impact on most of the backend code. The technique is affectionately known as TOAST (or “the best thing since sliced bread”). The TOAST infrastructure is also used to improve handling of large data values in-memory.

Only certain data types support TOAST — there is no need to impose the overhead on data types that cannot produce large field values. To support TOAST, a data type must have a variable-length (*varlena*) representation, in which, ordinarily, the first four-byte word of any stored value contains the total length of the value in bytes (including itself). TOAST does not constrain the rest of the data type's representation. The special representations collectively called *TOASTed values* work by modifying or reinterpreting this initial length word. Therefore, the C-level functions supporting a TOAST-able data type must be careful about how they handle potentially TOASTed input values: an input might not actually consist of a four-byte length word and contents until after it's been *detoasted*. (This is normally done by invoking `PG_DETOAST_DATUM` before doing anything with an input value, but in some cases more efficient approaches are possible. See Section 36.13.1 for more detail.)

TOAST usurps two bits of the varlena length word (the high-order bits on big-endian machines, the low-order bits on little-endian machines), thereby limiting the logical size of any value of a TOAST-able data type to 1 GB (2^{30} - 1 bytes). When both bits are zero, the value is an ordinary un-TOASTed value of the data type, and the remaining bits of the length word give the total datum size (including length word) in bytes. When the highest-order or lowest-order bit is set, the value has only a single-byte header instead of the normal four-byte header, and the remaining bits of that byte give the total datum size (including length byte) in bytes. This alternative supports space-efficient storage of values shorter than 127 bytes, while still allowing the data type to grow to 1 GB at need. Values with single-byte headers aren't aligned on any particular boundary, whereas values with four-byte headers are aligned on at least a four-byte boundary; this omission of alignment padding provides additional space savings that is significant compared to short values. As a special case, if the remaining bits of a single-byte header are all zero (which would be impossible for a self-inclusive length), the value is a pointer to out-of-line data, with several possible alternatives as described below. The type and size of such a *TOAST pointer* are determined by a code stored in the second byte of the datum. Lastly, when the highest-order or lowest-order bit is clear but the adjacent bit is set, the content of the datum has been compressed and must be decompressed before use. In this case the remaining bits of the four-byte length word give the total size of the compressed datum, not the original data. Note that compression is also possible for out-of-line data but the varlena header does not tell whether it has occurred — the content of the TOAST pointer tells that, instead.

The compression technique used for either in-line or out-of-line compressed data can be selected for each column by setting the `COMPRESSION` column option in `CREATE TABLE` or `ALTER TABLE`. The default for columns with no explicit setting is to consult the default_toast_compression parameter at the time data is inserted.

As mentioned, there are multiple types of TOAST pointer datums. The oldest and most common type is a pointer to out-of-line data stored in a *TOAST table* that is separate from, but associated with, the table containing the TOAST pointer datum itself. These *on-disk* pointer datums are created by the TOAST management code (in `access/common/toast_internals.c`) when a tuple to be stored on disk is too large to be stored as-is. Further details appear in Section 66.2.1. Alternatively, a TOAST pointer datum can contain a pointer to out-of-line data that appears elsewhere in memory. Such datums are necessarily short-lived, and will never appear on-

disk, but they are very useful for avoiding copying and redundant processing of large data values. Further details appear in Section 66.2.2.

66.2.1. Out-of-Line, On-Disk TOAST Storage

If any of the columns of a table are TOAST-able, the table will have an associated TOAST table, whose OID is stored in the table's `pg_class.reltoastrelid` entry. On-disk TOASTed values are kept in the TOAST table, as described in more detail below.

Out-of-line values are divided (after compression if used) into chunks of at most `TOAST_MAX_CHUNK_SIZE` bytes (by default this value is chosen so that four chunk rows will fit on a page, making it about 2000 bytes). Each chunk is stored as a separate row in the TOAST table belonging to the owning table. Every TOAST table has the columns `chunk_id` (an OID identifying the particular TOASTed value), `chunk_seq` (a sequence number for the chunk within its value), and `chunk_data` (the actual data of the chunk). A unique index on `chunk_id` and `chunk_seq` provides fast retrieval of the values. A pointer datum representing an out-of-line on-disk TOASTed value therefore needs to store the OID of the TOAST table in which to look and the OID of the specific value (its `chunk_id`). For convenience, pointer datums also store the logical datum size (original uncompressed data length), physical stored size (different if compression was applied), and the compression method used, if any. Allowing for the varlena header bytes, the total size of an on-disk TOAST pointer datum is therefore 18 bytes regardless of the actual size of the represented value.

The TOAST management code is triggered only when a row value to be stored in a table is wider than `TOAST_TUPLE_THRESHOLD` bytes (normally 2 kB). The TOAST code will compress and/or move field values out-of-line until the row value is shorter than `TOAST_TUPLE_TARGET` bytes (also normally 2 kB, adjustable) or no more gains can be had. During an UPDATE operation, values of unchanged fields are normally preserved as-is; so an UPDATE of a row with out-of-line values incurs no TOAST costs if none of the out-of-line values change.

The TOAST management code recognizes four different strategies for storing TOAST-able columns on disk:

- `PLAIN` prevents either compression or out-of-line storage. This is the only possible strategy for columns of non-TOAST-able data types.
- `EXTENDED` allows both compression and out-of-line storage. This is the default for most TOAST-able data types. Compression will be attempted first, then out-of-line storage if the row is still too big.
- `EXTERNAL` allows out-of-line storage but not compression. Use of `EXTERNAL` will make substring operations on wide `text` and `bytea` columns faster (at the penalty of increased storage space) because these operations are optimized to fetch only the required parts of the out-of-line value when it is not compressed.
- `MAIN` allows compression but not out-of-line storage. (Actually, out-of-line storage will still be performed for such columns, but only as a last resort when there is no other way to make the row small enough to fit on a page.)

Each TOAST-able data type specifies a default strategy for columns of that data type, but the strategy for a given table column can be altered with `ALTER TABLE ... SET STORAGE`.

`TOAST_TUPLE_TARGET` can be adjusted for each table using `ALTER TABLE ... SET (toast_tuple_target = N)`

This scheme has a number of advantages compared to a more straightforward approach such as allowing row values to span pages. Assuming that queries are usually qualified by comparisons against relatively small key values, most of the work of the executor will be done using the main row entry. The big values of TOASTed attributes will only be pulled out (if selected at all) at the time the result set is sent to the client. Thus, the main table is much smaller and more of its rows fit in the shared buffer cache than would be the case without any out-of-line storage. Sort sets shrink also, and sorts will more often be done entirely in memory. A little test showed that a table containing typical HTML pages and their URLs was stored in about half of the raw data size including the TOAST table, and that the main table contained only about 10% of the entire data (the URLs and some small HTML pages). There was no run time difference compared to an un-TOASTed comparison table, in which all the HTML pages were cut down to 7 kB to fit.

66.2.2. Out-of-Line, In-Memory TOAST Storage

TOAST pointers can point to data that is not on disk, but is elsewhere in the memory of the current server process. Such pointers obviously cannot be long-lived, but they are nonetheless useful. There are currently two sub-cases: pointers to *indirect* data and pointers to *expanded* data.

Indirect TOAST pointers simply point at a non-indirect varlena value stored somewhere in memory. This case was originally created merely as a proof of concept, but it is currently used during logical decoding to avoid possibly having to create physical tuples exceeding 1 GB (as pulling all out-of-line field values into the tuple might do). The case is of limited use since the creator of the pointer datum is entirely responsible that the referenced data survives for as long as the pointer could exist, and there is no infrastructure to help with this.

Expanded TOAST pointers are useful for complex data types whose on-disk representation is not especially suited for computational purposes. As an example, the standard varlena representation of a PostgreSQL array includes dimensionality information, a nulls bitmap if there are any null elements, then the values of all the elements in order. When the element type itself is variable-length, the only way to find the *N*'th element is to scan through all the preceding elements. This representation is appropriate for on-disk storage because of its compactness, but for computations with the array it's much nicer to have an “expanded” or “deconstructed” representation in which all the element starting locations have been identified. The TOAST pointer mechanism supports this need by allowing a pass-by-reference Datum to point to either a standard varlena value (the on-disk representation) or a TOAST pointer that points to an expanded representation somewhere in memory. The details of this expanded representation are up to the data type, though it must have a standard header and meet the other API requirements given in `src/include/utils/expandeddatum.h`. C-level functions working with the data type can choose to handle either representation. Functions that do not know about the expanded representation, but simply apply `PG_DETOAST_DATUM` to their inputs, will automatically receive the traditional varlena representation; so support for an expanded representation can be introduced incrementally, one function at a time.

TOAST pointers to expanded values are further broken down into *read-write* and *read-only* pointers. The pointed-to representation is the same either way, but a function that receives a read-write pointer is allowed to modify the referenced value in-place, whereas one that receives a read-only pointer must not; it must first create a copy if it wants to make a modified version of the value. This distinction and some associated conventions make it possible to avoid unnecessary copying of expanded values during query execution.

For all types of in-memory TOAST pointer, the TOAST management code ensures that no such pointer datum can accidentally get stored on disk. In-memory TOAST pointers are automatically expanded to normal in-line varlena values before storage — and then possibly converted to on-disk TOAST pointers, if the containing tuple would otherwise be too big.

66.3. Free Space Map

Each heap and index relation, except for hash indexes, has a Free Space Map (FSM) to keep track of available space in the relation. It's stored alongside the main relation data in a separate relation fork, named after the filenode number of the relation, plus a `_fsm` suffix. For example, if the filenode of a relation is 12345, the FSM is stored in a file called `12345_fsm`, in the same directory as the main relation file.

The Free Space Map is organized as a tree of FSM pages. The bottom level FSM pages store the free space available on each heap (or index) page, using one byte to represent each such page. The upper levels aggregate information from the lower levels.

Within each FSM page is a binary tree, stored in an array with one byte per node. Each leaf node represents a heap page, or a lower level FSM page. In each non-leaf node, the higher of its children's values is stored. The maximum value in the leaf nodes is therefore stored at the root.

See `src/backend/storage/freespace/README` for more details on how the FSM is structured, and how it's updated and searched. The pg_freespacemap module can be used to examine the information stored in free space maps.

66.4. Visibility Map

Each heap relation has a Visibility Map (VM) to keep track of which pages contain only tuples that are known to be visible to all active transactions; it also keeps track of which pages contain only frozen tuples. It's stored alongside the main relation data in a separate relation fork, named after the filenode number of the relation, plus a `_vm` suffix. For example, if the filenode of a relation is 12345, the VM is stored in a file called `12345_vm`, in the same directory as the main relation file. Note that indexes do not have VMs.

The visibility map stores two bits per heap page. The first bit, if set, indicates that the page is all-visible, or in other words that the page does not contain any tuples that need to be vacuumed. This information can also be used by *index-only scans* to answer queries using only the index tuple. The second bit, if set, means that all tuples on the page have been frozen. That means that even an anti-wraparound vacuum need not revisit the page.

The map is conservative in the sense that we make sure that whenever a bit is set, we know the condition is true, but if a bit is not set, it might or might not be true. Visibility map bits are only set by vacuum, but are cleared by any data-modifying operations on a page.

The pg_visibility module can be used to examine the information stored in the visibility map.

66.5. The Initialization Fork

Each unlogged table, and each index on an unlogged table, has an initialization fork. The initialization fork is an empty table or index of the appropriate type. When an unlogged table must be reset to empty due to a crash, the initialization fork is copied over the main fork, and any other forks are erased (they will be recreated automatically as needed).

66.6. Database Page Layout

This section provides an overview of the page format used within PostgreSQL tables and indexes.[1] Sequences and TOAST tables are formatted just like a regular table.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to an individual data value that is stored on a page. In a table, an item is a row; in an index, an item is an index entry.

Every table and index is stored as an array of *pages* of a fixed size (usually 8 kB, although a different page size can be selected when compiling the server). In a table, all the pages are logically equivalent, so a particular item (row) can be stored in any page. In indexes, the first page is generally reserved as a *metapage* holding control information, and there can be different types of pages within the index, depending on the index access method.

Table 66.2 shows the overall layout of a page. There are five parts to each page.

Table 66.2. Overall Page Layout

Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of item identifiers pointing to the actual items. Each entry is an (offset,length) pair. 4 bytes per item.
Free space	The unallocated space. New item identifiers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

[1] Actually, use of this page format is not required for either table or index access methods. The `heap` table access method always uses this format. All the existing index methods also use the basic format, but the data kept on index metapages usually doesn't follow the item layout rules.

The first 24 bytes of each page consists of a page header (`PageHeaderData`). Its format is detailed in Table 66.3. The first field tracks the most recent WAL entry related to this page. The second field contains the page checksum if `-k` are enabled. Next is a 2-byte field containing flag bits. This is followed by three 2-byte integer fields (`pd_lower`, `pd_upper`, and `pd_special`). These contain byte offsets from the page start to the start of unallocated space, to the end of unallocated space, and to the start of the special space. The next 2 bytes of the page header, `pd_pagesize_version`, store both the page size and a version indicator. Beginning with PostgreSQL 8.3 the version number is 4; PostgreSQL 8.1 and 8.2 used version number 3; PostgreSQL 8.0 used version number 2; PostgreSQL 7.3 and 7.4 used version number 1; prior releases used version number 0. (The basic page layout and header format has not changed in most of these versions, but the layout of heap row headers has.) The page size is basically only present as a cross-check; there is no support for having more than one page size in an installation. The last field is a hint that shows whether pruning the page is likely to be profitable: it tracks the oldest un-pruned XMAX on the page.

Table 66.3. PageHeaderData Layout

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none

All the details can be found in `src/include/storage/bufpage.h`.

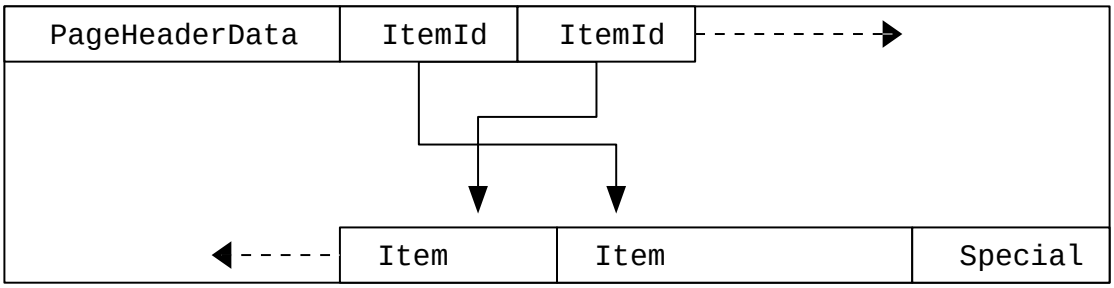
Following the page header are item identifiers (`ItemIdData`), each requiring four bytes. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a few attribute bits which affect its interpretation. New item identifiers are allocated as needed from the beginning of the unallocated space. The number of item identifiers present can be determined by looking at `pd_lower`, which is increased to allocate a new identifier. Because an item identifier is never moved until it is freed, its index can be used on a long-term basis to reference an item, even when the item itself is moved around on the page to compact free space. In fact, every pointer to an item (`ItemPointer`, also known as `CTID`) created by PostgreSQL consists of a page number and the index of an item identifier.

The items themselves are stored in space allocated backwards from the end of unallocated space. The exact structure varies depending on what the table is to contain. Tables and sequences both use a structure named `HeapTupleHeaderData`, described below.

The final section is the “special section” which can contain anything the access method wishes to store. For example, b-tree indexes store links to the page's left and right siblings, as well as some other data relevant to the index structure. Ordinary tables do not use a special section at all (indicated by setting `pd_special` to equal the page size).

Figure 66.1 illustrates how these parts are laid out in a page.

Figure 66.1. Page Layout



66.6.1. Table Row Layout

All table rows are structured in the same way. There is a fixed-size header (occupying 23 bytes on most machines), followed by an optional null bitmap, an optional object ID field, and the user data. The header is detailed in Table 66.4. The actual user data (columns of the row) begins at the offset indicated by `t_hoff`, which must always be a multiple of the MAXALIGN distance for the platform. The null bitmap is only present if the *HEAP_HASNULL* bit is set in `t_infomask`. If it is present it begins just after the fixed header and occupies enough bytes to have one bit per data column (that is, the number of bits that equals the attribute count in `t_infomask2`). In this list of bits, a 1 bit indicates not-null, a 0 bit is a null. When the bitmap is not present, all columns are assumed not-null. The object ID is only present if the *HEAP_HASOID_OLD* bit is set in `t_infomask`. If present, it appears just before the `t_hoff` boundary. Any padding needed to make `t_hoff` a MAXALIGN multiple will appear between the null bitmap and the object ID. (This in turn ensures that the object ID is suitably aligned.)

Table 66.4. HeapTupleHeaderData Layout

Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cid	CommandId	4 bytes	insert and/or delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_infomask2	uint16	2 bytes	number of attributes, plus various flag bits
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data

All the details can be found in `src/include/access/htup_details.h`.

Interpreting the actual data can only be done with information obtained from other tables, mostly `pg_attribute`. The key values needed to identify field locations are `attlen` and `attalign`. There is no way to directly get a particular attribute, except when there are only fixed width fields and no null values. All this trickery is wrapped up in the functions *heap_getattr*, *fastgetattr* and *heap_getsysattr*.

To read the data you need to examine each attribute in turn. First check whether the field is NULL according to the null bitmap. If it is, go to the next. Then make sure you have the right alignment. If the field is a fixed width

field, then all the bytes are simply placed. If it's a variable length field (attlen = -1) then it's a bit more complicated. All variable-length data types share the common header structure `struct varlena`, which includes the total length of the stored value and some flag bits. Depending on the flags, the data can be either inline or in a TOAST table; it might be compressed, too (see Section 66.2).

66.7. Heap-Only Tuples (HOT)

To allow for high concurrency, PostgreSQL uses multiversion concurrency control (MVCC) to store rows. However, MVCC has some downsides for update queries. Specifically, updates require new versions of rows to be added to tables. This can also require new index entries for each updated row, and removal of old versions of rows and their index entries can be expensive.

To help reduce the overhead of updates, PostgreSQL has an optimization called heap-only tuples (HOT). This optimization is possible when:

- The update does not modify any columns referenced by the table's indexes, not including summarizing indexes. The only summarizing index method in the core PostgreSQL distribution is BRIN.
- There is sufficient free space on the page containing the old row for the updated row.

In such cases, heap-only tuples provide two optimizations:

- New index entries are not needed to represent updated rows, however, summary indexes may still need to be updated.
- When a row is updated multiple times, row versions other than the oldest and the newest can be completely removed during normal operation, including `SELECT`s, instead of requiring periodic vacuum operations. (Indexes always refer to the page item identifier of the original row version. The tuple data associated with that row version is removed, and its item identifier is converted to a redirect that points to the oldest version that may still be visible to some concurrent transaction. Intermediate row versions that are no longer visible to anyone are completely removed, and the associated page item identifiers are made available for reuse.)

You can increase the likelihood of sufficient page space for HOT updates by decreasing a table's `fillfactor`. If you don't, HOT updates will still happen because new rows will naturally migrate to new pages and existing pages with sufficient free space for new row versions. The system view pg_stat_all_tables allows monitoring of the occurrence of HOT and non-HOT updates.