1. The POSTGRES Access Methods

This section describes the POSTGRES access methods in detail. The major concepts covered here are

- the relation descriptor and its contents,
- the differences between heap and index relations,
- scan keys, scan descriptors, and the scan interface, and
- the POSTGRES access method interface routines.

1.1. The Relation Descriptor

The relation descriptor, or *reldesc*, is the in-memory data structure that describes an open relation. The reldesc is an argument to most of the procedures that operate on relations. Relations may be opened by name or by relation ID — either of these uniquely identifies the relation to which it corresponds. A relation's ID, or *relid*, is the object ID of the tuple in `pg_class` that describes it.

The structure that stores a reldesc is declared in the source file `utils/rel.h`. The structure definition is

```
typedef struct RelationData {
     File                   rd_fd;
     int                    rd_nblocks;
     uint16                 rd_refcnt;
     bool                   rd_ismem;
     bool                   rd_isnailed;
     AccessMethodTupleForm rd_am;
     RelationTupleForm     rd_rel;
     ObjectId               rd_id;
     Pointer                lockInfo;
     TupleDescriptorData   rd_att;
     /* VARIABLE LENGTH ARRAY AT END OF STRUCT */
} RelationData;

typedef RelationData        *Relation;
```

The meaning of the `rd_fd` entry depends on the storage manager that stores the relation. For the magnetic disk storage manager, this is the POSTGRES virtual file descriptor (VFD) for the open file that stores the relation. On the Sony jukebox, this entry is meaningless, and is simply set to a non-negative value if the jukebox relation is successfully opened.

The `rd_nblocks` entry is the number of blocks in the relation, but this number is not reliable. This field is used by the executor during scans of heap relations to avoid scanning past the end of the relation. For index relations, and during scans of heap relations, the value stored here will be wrong. It should not be used outside of the executor.

The entry `rd_refcount` reflects the number of currently active references to this reldesc inside the backend. Every backend manages a private cache of relation descriptors. When the cache is full, reldescs with `rd_refcount` equal to zero may be evicted to make room for new descriptors. If a single relation is opened by a single backend more than once, every open request returns a pointer to the same reldesc, and `rd_refcount` is incremented to keep track of each reference. When the relation is closed, the reference count is decremented.

The structure entry `rd_ismem` is unused in the current system. It is intended to support in-memory-only relations, changes for which need not be flushed to stable storage at transaction boundaries.

The private cache of relation descriptors contains several reldescs that cannot be evicted, in order to guarantee that the cache continues to work. For example, in order to instantiate the relation descriptor for a user relation `user_reln`, POSTGRES must open and scan `pg_class` and `pg_attribute` for its relation and attribute data. If the reldesc for `pg_class` is not in the cache, then no relation (including `pg_class`) can ever be instantiated in the cache. The reldescs for `pg_class`, `pg_type`, and `pg_attribute` all fall into this category, and are initialized specially when the system starts up.

To keep track of which relations may not be evicted from the private reldesc cache, every reldesc contains an `rd_isnailed` entry. If this entry is set to *true*, then the reldesc is nailed and may not be evicted from the cache, even if its `rd_refcount` drops to zero.

The `rd_am` entry stores a pointer to the access method tuple for the access method that manages this relation. Currently, POSTGRES supports a heap access method for storing user and system data, and btree and rtree indexed access methods for storing indices on data. Support for a hashed indexed access method will be added in the near future. Because these access methods have different implementations, POSTGRES must know what actual routine to call to dispatch general access method interface calls for a particular access method. For indexed access methods, this information is stored in the access method tuple. Since there is only one access method for user and system data (the heap), the `rd_am` entry is NULL for reldescs describing heap relations.

The access method tuple contents will be described in section 1.1.1.

The structure entry `rd_rel` stores a pointer to the `pg_class` tuple that describes this relation. The `pg_class` tuple is copied to a safe place in memory when the relation is entered into the reldesc cache. The tuple contains, among other things, the relation's name and the user id of its owner. The complete contents of the tuple pointed to by `rd_rel` are supplied in section 1.1.2.

The entry `rd_id` stores the relid of the open relation. The relid is the object ID of the `pg_class` tuple describing the relation. This does not appear in the tuple pointed to by the `rd_rel` entry, because that tuple includes only the ordinary attributes, and not the system attributes, from the `pg_class` tuple. Note that the `pg_class` tuple has system attributes, as it is stored in the relation. These attributes simply are not copied into memory when the reldesc is initialized.

The structure entry `lockInfo` points to an in-memory representation of the rule lock data associated with this relation. This entry is not used by the access methods. The rule system and the data stored in a rule lock are beyond the scope of this section.

The final entry in the reldesc is `rd_att`, a vector of attribute descriptor data for the relation. This vector stores one entry for every user-level (that is, non-system) attribute that the relation stores. This vector is initialized when the reldesc is loaded into the cache. The data in the vector come from a scan of `pg_attribute`. The contents of the vector are described in section 1.1.3.

1.1.1. The Access Method Tuple Form (rd_am)

The `rd_am` entry in a reldesc points at an *access method tuple form* describing the access method that manages the relation[1]. This tuple has one attribute for each of the routines that implement the standard access method interface for the particular access method. Details of the interface routines appear in section 1.4. Here, we list only the contents of the access method tuple form. This declaration appears in the source file `catalog/pg_am.h`, and describes the tuples stored in the `pg_am` system catalog.

```
typedef struct AccessMethodTupleFormD {
      NameData          amname;
      ObjectId          amowner;
      char              amkind;
      uint16            amstrategies;
      uint16            amsupport;
      RegProcedure      amgettuple;
      RegProcedure      aminsert;
      RegProcedure      amdelete;
      RegProcedure      amgetattr;
      RegProcedure      amsetlock;
      RegProcedure      amsettid;
      RegProcedure      amfreetuple;
      RegProcedure      ambeginscan;
      RegProcedure      amrescan;
      RegProcedure      amendscan;
      RegProcedure      ammarkpos;
      RegProcedure      amrestrpos;
      RegProcedure      amopen;
      RegProcedure      amclose;
      RegProcedure      ambuild;
      RegProcedure      amcreate;
      RegProcedure      amdestroy;
} AccessMethodTupleFormD;

typedef AccessMethodTupleFormD   *AccessMethodTupleForm;
```

The structure entry `amname` is the name of the access method. In the current system, this is one of *btree*, or *rtree*. Support for *hash* is forthcoming.

The `amowner` entry is the object ID of the `pg_user` tuple for the owner of this access method. For all existing access methods, this is the OID for the user *postgres*. This entry is intended to support access protection, so that, for example, only the owner of an access method could change its interface routines. Such protection is not implemented in the current system.

The `amkind` entry is unused in the current system.

[1] In POSTGRES, the word *form* is often used to name structures. In this instance, for example, the clause *access method tuple form* is used to name a structure that describes an access method tuple.

The structure entry `amstrategies` is the number of strategies (operators) that can be used in searches on this index.

- For btrees, this is 5: $<$, $\leq$, $=$, $\geq$, and $>$.
- For rtrees, this is 8, corresponding to the operators for left, left-or-overlap, overlap, right-or-overlap, right, same, contains, and contained-by.
- When hash tables are supported, the number of operators will be 1, for equality searches.

Operators that are supported by an access method are listed in `pg_amop`. This relation contains one group of operators for each operator class, or *opclass*. An opclass is an access method/type pair. For example, an opclass `int4_ops` is defined for btrees, so that `int4` values may be indexed using btrees.

The structure entry `amsupport` is the number of support routines required by this access method. For example, when inserting keys into a btree, the btree access method requires one support routine that compares two keys and returns negative, zero, or positive, depending on whether the first key is less than, equal to, or greater than the second, respectively. Similarly, rtrees require routines that compute the size, intersection, and union of two rectangles. These are not operators that are available to users for index searches, so they are stored separately from the strategies. These routines are stored in the relation `pg_amsupport`.

The rest of the `AccessMethodTupleForm` structure stores the object IDs of functions that support the standard access method interface for this access method. Those routines are summarized in the table below.

Entry name	*routine description*
`amgettuple`	Get the next tuple in a scan.
`aminsert`	Insert a new tuple into the index.
`amdelete`	Delete a particular *tid* from the index.
`amgetattr`	Get a particular attribute from the index tuple.
`amsetlock`	Unsupported.
`amsettid`	Unsupported.
`amfreetuple`	Unsupported.
`ambeginscan`	Start a scan with a qualification on the index key.
`amrescan`	Reset an active scan to the beginning.
`amendscan`	End an active scan.
`ammarkpos`	Mark the current position in an index scan.
`amrestrpos`	Restore the scan to the previously-marked position.
`amopen`	Open the index relation.
`amclose`	Close an open index relation.
`ambuild`	Define an index on an existing heap relation.
`amcreate`	Create a new, empty, index relation.
`amdestroy`	Destroy an existing index relation.

In general, the programmer need not worry about how function dispatch via the `AccessMethodTupleForm` works. This is handled properly for all indexed access methods by code in `access/common`.

1.1.2. The Relation Tuple Form (rd_rel)

The reldesc for a relation includes a pointer to the `pg_class` tuple for the relation. When a reldesc is instantiated into the cache, it is initialized from the data for the relation from `pg_class`. The user-level (that is, non-system) attributes for the `pg_class` tuple constitute the *relation tuple form*. This structure is declared in `catalog/pg_relation.h`[2] as `Form_pg_relation`. The declaration is

```
CATALOG(pg_relation) BOOTSTRAP {
     char16      relname;
     oid         relowner;
     oid         relam;
     int4        relpages;
     int4        reltuples;
     dt          relexpires;
     dt          relpreserved;
     bool        relhasindex;
     bool        relisshared;
     char        relkind;
     char        relarch;
     int2        relnatts;
     int2        relsmgr;
     int28       relkey;
     oid8        relkeyop;
     aclitem     relacl[1];
} FormData_pg_relation;

typedef FormData_pg_relation     *Form_pg_relation;
```

The notation

```
CATALOG(pg_relation) BOOTSTRAP {
```

is turned into a structure declaration by cpp macros at compile time. This declaration is also used to produce setup files, called *bki files*, that create and populate the *template1* database.

The structure entry `relname` is the name of the relation.

The `relowner` entry is the object ID of the `pg_user` tuple describing the relation's owner. A relation is owned by the user that created it.

The `relam` entry is the object ID of the `pg_am` tuple for the access method that manages this relation. For heap relations, this is zero, an invalid object ID.

The `relpages` and `reltuples` entries are, respectively, the approximate number of pages and tuples in this relation. These numbers are set by the vacuum cleaner and when an index is defined on a heap relation, and so are wrong in general. They are not changed when

[2] The class `pg_class` was previously called `pg_relation`. In all user-visible parts of the system, the name was converted in 1990, but internally (as in the names of system header files), the name was not always changed.

tuples are inserted into or deleted from the relation. When the relation is initially created, both are zero. Both `reltuples` and `relpages` are used by the planner to estimate plan costs during query optimization. Interestingly, then, if neither `reltuples` nor `relpages` have been updated by the vacuum cleaner recently, their incorrect values can skew query optimization such that queries may run for an unexpectedly long time.

The `relexpires` entry is the amount of history that should be saved for this relation. This is not used in the current system.

The `relpreserved` entry is the date and time at which the relation was last vacuumed. This is used by the planner when it produces query plans for historical queries. If the historical query extends to before the time at which the relation was last vacuumed, then the archive must be scanned. Otherwise, it need not be.

The `relhasindex` entry is *true* if an index exists on the relation. When an index is destroyed, this flag is not changed, and so an entry of *true* may mean that there was recently an index on the relation, but that there no longer is. The vacuum cleaner restores this to the correct value when it runs, and defining an index on a heap relation always sets this entry to *true* for the heap. `Relhasindex` is always *false* for an index relation.

The `relisshared` entry is *true* for some shared system catalogs. Most relations for database *dbname* reside in the directory `$PGDATA/base/`*dbname*. However, some relations, such as `pg_user` and `pg_database`, must be visible to users of all databases simultaneously. These relations are stored in the directory `$PGDATA`, and the `relisshared` entry in the `pg_class` tuple for these relations is set to *true*.

The `relkind` entry has one of three values:

- *r* means that the relation is an ordinary heap.
- *i* means that the relation is an index.
- *u* means that the relation is an uncatalogued (that is, temporary) heap relation which will be automatically destroyed when the transaction ends.

The `relarch` entry describes the frequency with which the relation should be archived. Although three levels exist, only two are actually supported. The three levels are

- *h*, for heavy update traffic and frequent archival;
- *l*, for light update traffic and infrequent archival; and
- *n*, for no archival.

The only levels actually supported are *h* and *n*. If the `relarch` entry is *n*, then the vacuum cleaner discards all historical data from the relation when it runs.

The `relnatts` entry is the number of attributes in the relation.

The `relsmgr` entry identifies the storage manager that manages storage for this relation. Storage manager zero is magnetic disk. On the system installed at UC Berkeley, storage managers one and two are, respectively, for the Sony optical disk jukebox and main memory relations. The storage manager code is called from the POSTGRES buffer manager, and is of no concern to access method implementors.

The `relkey` entry is a vector of attribute numbers that define the unique key for this relation. This is not supported in the current system.

The structure ends with a variable-length vector of access control list information, stored in the `relacl` entry. The access control list is used to grant or deny read or write access on the relation to particular users.

1.1.3. The Tuple Descriptor Data (rd_att)

The reldesc ends with a vector of attribute data that describe the tuples stored in the relation. This is a variable-length entry; different relations have different numbers of attributes, and so their `rd_att` vectors are of different length.

There are two data structures of interest. The `TupleDescriptorData` structure, defined in `access/tupdesc.h`, is a variable-length array of `AttributeTupleFormData` entries. The `AttributeTupleFormData` structure, defined in `catalog/pg_attribute.h`, contains detailed information on a single attribute.

The data structure for `AttributeTupleFormData` is the same as a `Form_pg_attribute`, which is

```
CATALOG(pg_attribute) BOOTSTRAP {
    oid         attrelid;
    char16      attname;
    oid         atttypid;
    oid         attdefrel;
    int4        attnvals;
    oid         atttyparg;
    int2        attlen;
    int2        attnum;
    int2        attbound;
    bool        attbyval;
    bool        attcanindex;
    oid         attproc;
    int4        attnelems;
    int4        attcacheoff;
} FormData_pg_attribute;

typedef FormData_pg_attribute    *Form_pg_attribute;
```

As above, the

```
CATALOG(pg_attribute) BOOTSTRAP
```

line is converted to a struct declaration by cpp macros at compile time.

The `attrelid` entry is the object ID of the `pg_class` tuple for the relation that contains this attribute.

The `attname` entry is the name of the attribute.

The `atttypid` entry is the object ID of the `pg_type` tuple for the type of this attribute.

The `attdefrel` entry is used for object-oriented type management by POSTGRES. When a class is declared that inherits from some other class, the new class implicitly has all the attributes that the original class had. In this case, the `attdefrel` entry is the object ID of the relation from which this attribute is inherited.

The attnvals entry is intended to support query optimization. It should store the number of distinct values for this attribute that appear in the relation. This value should be computed at vacuum time, and stored in pg_attribute by the vacuum cleaner. However, the current vacuum cleaner does not compute statistics, so attnvals is always zero. If the correct value were maintained, the optimizer could use it to produce better query plans. The optimizer currently ignores attnvals.

The atttyparg entry is unused.

The attlen entry is the length of this attribute, in bytes, for fixed-length attributes, or -1, for variable-length attributes.

The attbound entry was intended to support array-valued attributes, but is not used by the existing POSTGRES array implementation.

The attbyval entry is *true* if this is a pass-by-value attribute, and *false* if it is pass-by-reference. This entry is set by looking at the typbyval attribute in the pg_type tuple for the type of this attribute.

The attcanindex entry is *true* if this entry can be indexed, and false otherwise. Array attributes cannot be indexed.

The attproc entry is unused.

The attnelems entry is used for array attributes, to record the number of elements that may be stored in the attribute.

The attcacheoff entry is never maintained on disk, but is set in memory to allow fast attribute lookups. The first time an attribute is fetched from a tuple, its offset is computed by summing the lengths of the attributes that precede it. If all of the preceding attributes are fixed-length, and none are NULL, then the computed offset may be cached for use on subsequent lookups. The offset is stored in attcacheoff.

The AttributeTupleFormData structure stores information on a single attribute. The vector that describes all of the attributes in a relation, rd_att, contains pointers to in-memory AttributeTupleFormData structures for each attribute. The vector is stored in the TupleDescriptorData structure, whose declaration is

```
typedef struct TupleDescriptorData {
      AttributeTupleForm    data[1];
      /* VARIABLE LENGTH ARRAY */
} TupleDescriptorData;

typedef TupleDescriptorData *TupleDescriptor;
```

1.1.4. The Strategy Map and Support Routines for Index Reldescs

The sections above describe the contents of a reldesc that are common to the heap and indexed access methods. The indexed access methods store some additional information at the end of the reldesc. This information is used to select operators to apply during scans, and to find the support routines that are required by particular indexed access methods. These data structures, their layout, and their purposes are obscure. They reflect poor design of the indexed access methods. This section is difficult to understand, and may be safely ignored by everyone except the maintainer of the POSTGRES access methods.

Immediately following the TupleDescriptorData vector, the indexed access methods have two additional pieces of data. The first is a pointer to the `IndexStrategyData` structure for this access method. The second is a pointer to the vector of support routines for the access method. Neither of these is a named entry of any data structure; their existence is assumed (and, at least so far, correctly maintained) by the code that manages reldescs for the indexed access methods.

The problem with this design is that it makes it impossible to name these entries from the debugger. The following cumbersome GDB statements will print out the contents of these entries:

```
print (IndexStrategy)(&reldesc->rd_att[
      reldesc->rd_rel->relnatts])
print (RegProcedure *)(((char *)
      &reldesc->rd_att[reldesc->rd_rel->relnatts])
      + sizeof(IndexStrategy))
```

The first statement prints the pointer to the vector of index strategy map data; the second prints the pointer to the vector of support procedure IDs. These pointers must be dereferenced in order to examine the contents of the vectors. Lines have been broken so that the commands fit across the page; the statements should be typed on a single line.

The strategy map is a vector of scan key entries. Scan keys are described in sections 1.3.1 and 1.3.2. The strategy map stores information on what procedure to call for a particular operator on this index. For example, a btree storing `int4` values would have the procedure ID of the `int4lt` procedure at the location corresponding to '<' in the strategy vector. The strategy map is constructed automatically by the system when the index relation is opened. The contents are read from the `pg_amop` relation, which is keyed by operator class and access method.

Finally, the vector of support procedures is simply an array of procedure IDs, or `pg_proc` tuple OIDs, for the support routines required by the access method. This vector is initialized when the index relation is opened by scanning the `pg_amsupport` relation. The tuples in `pg_amsupport` are keyed by access method and operator class. POSTGRES assigns no meaning to the entries in this vector; they are intended for use by the access method. The btree access method uses a single support routine, which compares two keys and returns less than zero, zero, and greater than zero, respectively, if the first key is less than, equal to, or greater than the second. The rtree access method uses three support routines, which compute the size of a rectangle in standard units, the intersection of two rectangles, and the union of two rectangles.

1.1.5. Summary of the Relation Descriptor Data Structure

The reldesc describes an open relation, and is the argument that is passed to most of the routines that operate on relations. POSTGRES manages a private cache of reldescs, and at most one copy of the reldesc for a given relation appears in the cache at a time. The reldesc includes pointers to the relation tuple form and a vector of information that describes the attributes that appear in tuples of the relation. For indexed access methods, the reldesc also includes a pointer to the access method tuple form, a pointer to a vector of strategy information that describes the operators supported by the access method for the operator class, and a pointer to a vector of access method-specific support routines.

1.2. Heap and Index Relations

Heap relations are the primary POSTGRES storage structure; all user relations and all the system catalog information is stored in heaps. The POSTGRES heap is logically an unordered set of tuples. Heaps consist of zero or more 8192-byte blocks, each of which contains zero or more tuples.

Index relations store key/value pairs, where the key allows fast lookup on some set of attributes from a heap, and the value is a pointer to a tuple in the heap.

1.2.1. Tuple Identifiers

Tuples are identified by *tuple identifiers*, or tids. A tid is a triple of the form (*blockno*, *pageno*, *offset*). The original design of the system allowed for more than one page to be stored on a single block of a relation, but this feature has never been used. As a result, the middle value (*pageno*) is always zero. To confuse things further, the term "page" is used interchangeably with the term "block" in the code and by programmers talking about the system. Since the original usage of "page" was never implemented, it has come to mean exactly the same thing as "block."

Thus a tid is a triple of the form (*blockno*, 0, *offset*).

The *blockno* is the number of the block in the relation; blocks are numbered sequentially from zero. When a new block is allocated to a relation, it is given the next available block number.

The *offset* stored in a tid is the offset of the tuple within the block. Blocks store zero or more tuples, and tuples are numbered sequentially from one within a block. The number of tuples that fit in a block depend on the schema of the relation that the block stores, and the number may vary for a given relation (for example, if the tuple includes variable-length attributes).

A tid uniquely identifies a single tuple in a relation. In the current version of the system, relations are not compressed when they are vacuumed, so tuples do not move around inside a relation. Thus the tid is a valid identifier for the tuple until the tuple is replaced or deleted, at which point it (logically) vanishes from the relation. Note that other tuples on the same page as a deleted tuple do not change their tids when the tuple is deleted; the available slot is reserved for subsequent reuse. In fact, slots are never reused in the current system. The vacuum cleaner and the tuple allocation code may be changed to fix this problem in the future.

1.2.2. Data Stored in Heap Relations

Every heap tuple begins with a `HeapTupleData` structure. The declaration for this structure appears in the source file `access/htup.h`, and is

```
typedef struct HeapTupleData {
      Size            t_len;
      ItemPointerData t_ctid;
      ItemPointerData t_chain;
      union {
            ItemPointerData l_ltid;
            RuleLock        l_lock;
      }               t_lock;
      ObjectId        t_oid;
      CommandId       t_cmin;
      CommandId       t_cmax;
```

```
        TransactionId    t_xmin;
        TransactionId    t_xmax;
        ABSTIME          t_tmin, t_tmax;
        AttributeNumber  t_natts;
        char             t_vtype;
        char             t_infomask;
        char             t_locktype;
        uint8            t_hoff;
        char             t_bits[MinHeapTupleBitmapSize / 8];
} HeapTupleData; /* MORE DATA FOLLOWS AT END OF STRUCT */

typedef HeapTupleData *HeapTuple;
```

This structure includes a variable-length bitmap vector at its end, after which the user-level attributes of the tuple appear.

The `t_len` entry is the length of the entire tuple, including the `HeapTupleData` structure and the user data that follows it. This is used to allocate sufficient space in memory and on disk pages for the tuple.

The `t_ctid` entry is the tid of this tuple. Storing the tid in the tuple is easier than computing it when it is needed (for example, when an index tuple is constructed to point at the heap tuple).

When a tuple is replaced, POSTGRES maintains a pointer from the old version of the tuple to the new version. This pointer is stored in the `t_chain` entry. When the new version of the tuple is successfully inserted, the page containing the old version is fetched, and the original tuple's `t_chain` entry is set to the `t_ctid` entry of the new version. This strategy, called *forward chaining*, leaves older records pointing at newer records. The advantage if forward chaining is that it allows records to be updated without changing indices that point at them if the indexed attribute has not chained.

The `t_chain` entry was originally intended to support tuple differencing, which would allow POSTGRES to store only the changed values for the new version in most cases, rather than the entire new tuple. Tuple differencing is not supported in the current system. In addition, the system always inserts a new index record when a heap tuple is modified, regardless of whether the indexed attribute has changed. This may be fixed in the future.

The `t_lock` entry stores a rule lock for the tuple. A rule lock is a tag that identifies a set of rules that should be run whenever this tuple is manipulated. The mechanism is beyond the scope of this section. The value stored in `t_lock` is a list of tids when the tuple is on disk, and is swizzled to an in-memory pointer when the tuple is copied into memory.

The `t_oid` entry of the `HeapTupleData` structure is the object ID that uniquely identifies this tuple. Although tids will currently uniquely identify a tuple for its lifetime, this may change someday, when tuple differencing and space reclamation are implemented. The OID will is guaranteed by design never to change. Furthermore, when a tuple is replaced, its tid will change, but its OID is guaranteed not to. This means that the OID is a safe, globally unique, eternal identifier for the tuple.

The `t_cmin` and `t_cmax` entries are, respectively, the command IDs that inserted and deleted this tuple. In every transaction, the user may run up to 32768 individual commands. Any of these commands may update the database. The command ID is stored in every tuple to allow the system to keep track of which command inside a single transaction made a change. The reason that this is necessary is to allow subsequent commands inside the same transaction to see the effects of changes made by earlier commands in the transaction, even though those changes are not visible to other users running concurrently in other transactions.

The `t_xmin` and `t_xmax` entries are, respectively, the transaction IDs of the transactions that inserted and deleted this tuple. These transaction IDs are used by POSTGRES to check at run-time whether the transaction that made the changes ever committed. Changes made by uncommitted transactions may be safely ignored.

The `t_tmin` and `t_tmax` entries are, respectively, the commit times of the inserting and deleting transactions that operated on this tuple. When POSTGRES inserts a tuple, it writes the inserting command and transaction IDs, but leaves the `t_tmin` entry empty. When the transaction commits, a special entry is written to the `pg_time` relation. This entry contains the time at which the transaction committed. Later, the vacuum cleaner (or, in some cases, a POSTGRES backend in normal operation) will fill in the commit time for the inserting transaction. Once this time is filled in, POSTGRES does not need to check the transaction log in order to see whether the transaction ever committed. This substantially speeds up processing, since the transaction log check generally requires disk accesses.

The `t_natts` entry is the number of attributes that are stored in this tuple. The number of attributes may not be the same as the number stored by the relation, since the relation schema may change over time as a result of the *addattr* command. Any references to attributes beyond the end of the tuple will return the special database value NULL (note that database NULL is not the same as the zero value used by C).

The `t_vtype` entry is for the version type, whose purpose has been lost to history. This is unused by the current system. The value stored here is typically zero, although the header file `access/htup.h` claims that only *i*, *r*, and *d* are supported.

The `t_infomask` entry is used to encode information about the tuple to speed up fetching of attributes. The information encoded here includes flags indicating whether the tuple contains any null values or any variable-length attributes.

The `t_locktype` entry indicates whether the value stored in the `t_lock` union is an on-disk (tid) or in-memory (pointer) lock representation. This is stored far from the union in order to keep the data structure tightly packed in memory.

The `t_hoff` entry is the number of bytes occupied by the tuple header (that is, by the `HeapTupleData` structure). This may vary from tuple to tuple, because if the tuple contains no nulls values, then the bitmap of null values (stored in `t_bits`, below) is not necessary, and does not appear in the tuple. For any tuple, the address of the `HeapTupleData` structure that stores it, plus the value stored in `t_hoff`, is the address of the first byte of user data stored in the tuple.

The `t_bits` vector contains one bit for every attribute in the tuple if the tuple contains nulls. For each attribute, the corresponding bit is set if the attribute is null, and is clear otherwise. Nulls are not actually stored in the tuple; just the `t_bits` bit for the null value appear. This keeps tuples containing null values small. If the tuple contains no null values, then the `t_bits` vector is left off the end of the structure, and user data begins immediately after the `t_hoff` entry.

Collectively, the values stored in the `HeapTupleData` structure are referred to as the tuple header. These values are accessible from the query language as *system attributes*. Every heap tuple has the same system attributes, but the *user attributes* they store vary from relation to relation, and possibly even within a relation.

System attributes are all assigned attributes less than zero internally by the system. The User-defined attributes all have attribute numbers greater than zero. The names of the system attributes and their corresponding `HeapTupleData` structure elements are listed in the following table[3] .

HeapTupleData element	*System Attribute*
`t_chain`	chain
`t_cmax`	cmax
`t_cmin`	cmin
`t_ctid`	ctid
`t_oid`	oid
`t_lock`	rlock
`t_tmax`	tmax
`t_tmin`	tmin
`x_xmax`	xmax
`t_xmin`	xmin
`t_vtype`	vtype

1.2.3. Data Stored in Index Relations

All POSTGRES indices are secondary — that is, ordinary system and user data are stored in heap relations, and indices store pointers into the heaps. All of the indexed access methods store index tuples, which are much smaller than heap tuples. In general, the indexed access methods also store other data on POSTGRES pages; what data are stored depends on the indexed access method. For the rtree and btree access methods, some pages are designated as *internal* pages, and only store pointers to other pages in the same index. *Leaf* pages, on the other hand, store pointers into the heap.

A particular index is defined on a single heap relation, and all the heap tids that the index stores refer to that relation. Since indices support fast keyed lookup, the indices also store key data, which are typically either an attribute from the heap or some function of an attribute or attributes from the heap. For example, if the `EMP` class stored employee records, and included a `salary` attribute, then a btree index on `EMP.salary` would store the salaries from `EMP` together with the tids of the tuples with each particular salary.

The index tuple format is defined in `access/itup.h`, and is

```
typedef struct IndexTupleData {
     ItemPointerData            t_tid;
     unsigned short             t_info;
} IndexTupleData;     /* MORE DATA FOLLOWS */
```

[3] At one time, there existed a system attribute *anchor* which corresponded with the `HeapTupleData` element `t_anchor`. *Anchor* was the anchor point, or complete tuple, stored in a sequence of difference tuples. Neither are used in the system at this time since tuple differencing has not yet been implemented.

```
typedef IndexTupleData      *IndexTuple;
```

The `t_tid` entry is the tid of the heap tuple that corresponds to this index tuple.

The `t_info` entry encodes some information about the index tuple. This is a sixteen-bit quantity whose layout is as follows:

- Bit fifteen (the leftmost bit) is set if the index tuple contains null values, and is clear otherwise.
- Bit fourteen is set if the index tuple contains variable-length attributes, and is clear otherwise.
- Bit thirteen is set if there are rules associated with the index tuple, and is clear otherwise. Rules on index tuples are not supported by the current system.
- Bits twelve through zero are the size of the tuple, in bytes.

Immediately following the `t_info` entry is the index key. The index key is the set of all attributes from the heap tuple that form the search key for this index. The current implementations of the btree and rtree access methods support only a single index key. This may change, at least for btrees, in the near future.

1.2.4. How the Relations are Managed

When the POSTQUEL user issues an update to a heap relation (either inserting or deleting data), all of the corresponding indices are automatically updated, as well. POSTGRES stores, for every heap relation, the index relations defined on it, and for every index relation, the attributes from the heap that form the index key. These data are stored in `pg_index`.

When the user issues a POSTQUEL query against a heap relation that requires the relation to be scanned, POSTGRES checks to see whether any of the indices defined on the heap will permit the scan to be executed more quickly. For example, if a btree index is defined on the `salary` attribute of the `EMP` relation, then a query for all employees with salaries between $30,000 and $50,000 can be satisfied by scanning the btree index, and only selecting out the heap tuples with the correct salaries. This allows the scan to be completed more quickly than would a sequential scan of all of the tuples in the `EMP` relation.

1.3. Scan Keys, Scan Descriptors, and Scans

A *scan* is the abstraction used in POSTGRES to search a heap or index relation for tuples that satisfy some qualification. A scan descriptor, or *scandesc*, is the data structure that describes an open scan. The qualification is stored in the scandesc in data structures called *scan keys*. Each key consists of a procedure p, an attribute number k, and a value v. Logically, every tuple t in the relation is checked to see whether its kth attribute, t_k, passes the qualification stored in the scan key. The tuple passes the qualification if $p(t_k, v)$ returns true. In order for the tuple to satisfy the entire qualification, it must pass this test for every scan key stored in the scandesc. This means that a scandesc stores a conjunctive qualification on the relation. Disjunctive qualifications may be handled by using more than one scan, and suppressing duplicate tuples, or by using a scan with no keys, and applying the disjunctive tests manually.

Once a scan is opened, it returns tuples one at a time. Every tuple that the scan returns is guaranteed to satisfy the qualification, and the scan is guaranteed to return all qualifying tuples.

The rest of this section describes the data structures associated with scans, scandescs, and scan keys.

1.3.1. Scan Keys

A scan is a conjunction of zero or more qualifications on single attributes in the relation. Every single-attribute qualification is stored in a scan key. The data structure used to store scan keys appears in the source file `access/skey.h`. Its declaration is

```
typedef struct ScanKeyEntryData {
     bits16          flags;
     AttributeNumber attributeNumber;
     RegProcedure    procedure;
     int             (*func) ();
     int32           nargs;
     Datum           argument;
} ScanKeyEntryData;

#define CheckIfNull         0x1
#define UnaryProcedure      0x2
#define NegateResult        0x4
#define CommuteArguments    0x8

typedef ScanKeyEntryData    *ScanKeyEntry;
```

The `flags` entry is a sixteen-bit bitmask describing the function to be called and its return values. The flag values appear in the `#defines` that follow the data structure. If the `CheckIfNull` flag is set, then the scan code should check for null attributes and return null for them without calling the function. However, this behavior is not implemented in the current system. If the `UnaryProcedure` bit is set, then this procedure takes a single argument (the appropriate attribute from the tuple), rather than an attribute and a constant. If the `NegateResult` bit is set, then the scan code will negate the logical value returned by the procedure when checking for tuples that satisfy the qualification. Finally, if the `CommuteArguments` flag is set, then the scan code will call the procedure with the supplied value and the attribute from the tuple, rather than the opposite order.

The `attributeNumber` entry is the attribute number in the tuple to check.

The `procedure` entry is the object ID of the `pg_proc` tuple that describes the procedure to call. When the procedure is called for the first time, the `pg_proc` relation is scanned and the appropriate function is dynamically loaded, if necessary.

The `func` entry is a pointer to a function that must return type `bool`. This function pointer is initialized by the scan key from the `pg_proc` tuple for the desired `procedure`, and need not be set by the caller. The function pointer is cached to save repeated scans of `pg_proc`.

The `nargs` entry is the number of arguments that are required by the function. This is filled in from the `pg_proc` tuple, and need not be set by the caller. However, it should always be either one or two.

Finally, the `argument` entry is the additional argument to the function. This must be set by the caller.

1.3.2. Setting up Scan Keys

In order to properly set up a scan key, the caller must know which attributes in the tuple are of interest, the object IDs of the procedures that should be used to test the attributes, and what values they should be tested against. In order to simplify scan key setup, a number of conventions have been adopted in the POSTGRES source code.

First, the schemas for all of the system catalogs are defined in header files in the `catalog` directory. The header files are named *relname*.h, where *relname* is the name of the catalog of interest. For example, the schema of the `pg_user` relation appears in the header file `catalog/pg_user.h`. The lone exception to this rule is that the schema for the `pg_class` relation appears in the file `catalog/pg_relation.h`. This is an artifact of the terminology change (relations became classes) that swept the project in the early 1990s.

Second, the header files described above include standard `#defines` for the relations that they describe. The relation's name, as a string, is `Name_`*relname*; for example, `Name_pg_user`. The number of attributes in the relation is `Natts_`*relname* (`Natts_pg_user`). Every attribute may be referenced as `Anum_`*relname_attname* (for example, the `usename` attribute of the `pg_user` relation is `Anum_pg_user_usename`. These conventions permit programmers to use symbolic names, rather than embedded constants, to set up scans and open relations.

Similar conventions have not been adopted for user relations, so programmers who want to set up and use scans on them must already know their schemas.

Third, there exist some `#defined` constants for procedures that are frequently used in scans on the system catalogs. These procedure IDs appear in the source file `catalog/pg_proc.h`, and generally take the form *OperationOfInterest*`RegProcedure` — for example, `CharacterEqualRegProcedure` and `ObjectIdEqualRegProcedure`.

Finally, the constant values used by POSTGRES in scans are of type `Datum`. A number of macros have been defined that convert embedded constants to values of type `Datum`. These macros are defined in the source file `tmp/datum.h`, and are

convert type	*from Datum*	*to Datum*
one-byte char	DatumGetChar(*v*)	CharGetDatum(*v*)
one-byte integer	DatumGetInt8(*v*)	Int8GetDatum(*v*)
unsigned one-byte integer	DatumGetUInt8(*v*)	UInt8GetDatum(*v*)
two-byte integer	DatumGetInt16(*v*)	Int16GetDatum(*v*)
unsigned two-byte integer	DatumGetUInt16(*v*)	UInt16GetDatum(*v*)
four-byte integer	DatumGetInt32(*v*)	Int32GetDatum(*v*)
unsigned four-byte integer	DatumGetUInt32(*v*)	UInt32GetDatum(*v*)
four-byte float	DatumGetFloat32(*v*)	Float32GetDatum(*v*)
eight-byte double	DatumGetFloat64(*v*)	Float64GetDatum(*v*)
void *	DatumGetPointer(*v*)	PointerGetDatum(*v*)
pointer to struct	DatumGetStructPointer(*v*)	StructPointerGetDatum(*v*)
16-byte char string name	DatumGetName(*v*)	NameGetDatum(*v*)
object ID	DatumGetObjectId(*v*)	ObjectIdGetDatum(*v*)

In POSTGRES, four-byte floating point values are passed by reference, not by value. All other four-byte quantities are pass-by-value. A very common programming error in POSTGRES is to use values of type `Name` as if they were pointers. In fact, `Name` is a structure containing a sixteen-byte character array, and so will be passed on the stack if its address is not explicitly used. These facts should be documented elsewhere, but are worth mentioning here.

The following code fragment shows how to set up a scan key on the `pg_user` relation for all tuples that have `usename` equal to “mao” and an object ID of 1806:

```
NameData name;
ScanKeyEntryData skey[2];

bzero(&name, sizeof(name));
bcopy(name.data[0], "mao", strlen("mao"));

ScanKeyEntryInitialize(&skey[0], (bits16)0x0,
      Anum_pg_user_usename,
      (RegProcedure)NameEqualRegProcedure,
      NameGetDatum(name));
ScanKeyEntryInitialize(&skey[1], (bits16)0x0,
      ObjectIdAttributeNumber,
      (RegProcedure)ObjectIdEqualsRegProcedure,
      ObjectIdGetDatum(1806));
```

1.3.3. The Scan Descriptor

Once the scan keys are properly initialized, they may be used to open a scan on a relation. An open scan is described by a scandesc. The data structures that describe open scans are `HeapScans` and `IndexScans`, and are declared in the source file `access/relscan.h`.

1.3.3.1. The Heap Scan Descriptor

The declaration for `HeapScanDescData` is

```
typedef struct HeapScanDescData {
      Relation        rs_rd;
      HeapTuple       rs_ptup;
      HeapTuple       rs_ctup;
      HeapTuple       rs_ntup;
      Buffer          rs_pbuf;
      Buffer          rs_cbuf;
      Buffer          rs_nbuf;
      struct dchain   *rs_dc;
      ItemPointerData rs_mptid;
      ItemPointerData rs_mctid;
      ItemPointerData rs_mntid;
      ItemPointerData rs_mcd;
      Boolean         rs_atend;
      TimeQual        rs_tr;
      uint16          rs_cdelta;
      bool            rs_parallel_ok;
      uint16          rs_nkeys;
      ScanKeyData     rs_key;
      /* VARIABLE LENGTH ARRAY AT END OF STRUCT */
} HeapScanDescData;

typedef HeapScanDescData *HeapScanDesc;
```

The `rs_rd` entry is a pointer to the reldesc for the relation on which this scan has been opened.

The `rs_ptup`, `rs_ctup`, and `rs_ntup` entries are, respectively, the previous tuple, current tuple, and next tuple visited in this scan. The scan code caches these because it was originally thought that scans would change direction frequently. In fact, that is not the case, and maintaining the previous and next tuple pointers has turned out to be pure overhead. Getting rid of the previous and next tuple pointers everywhere that they appear in the scandesc would speed up scan processing.

The `rs_pbuf`, `rs_cbuf`, and `rs_nbuf` entries are the buffers on which the previous, current, and next tuple are stored. These buffers are kept pinned, which guarantees that they will not be evicted from the shared buffer cache by any backend. Pinning the buffers allows the backend to reference them directly, without reacquiring a pointer to the buffer every time data on it is used. The shared buffer cache is protected by mutual exclusion. Some POSTGRES ports use System V semaphores to implement exclusion. Acquiring and releasing these semaphores is slow, so pinning the buffers during a scan improves performance significantly.

The `rs_dc` entry is intended to support tuple differencing, which is not supported in the current version of the system.

The `rs_mptid`, `rs_mctid`, and `rs_mntid` entries support marking of positions in scans. While processing some join strategies, the executor marks a location so that it can return to it later. When such a mark is made, the tids of the rs_ptup, rs_ctup, and rs_ntup tuples are copied to the rs_mptid, rs_mctid, and rs_mntid entries.

The rs_mcd entry is intended to support tuple differencing, and is not used in the current system.

The rs_atend entry indicates whether the scan should begin at the end of the relation. Some attempt is made to store a sensible value here, but it is ignored by much of the code and should not be relied on.

The rs_tr entry is a time range or snapshot time qualification that indicates what historical tuples are of interest. The special constant value NowTimeQual indicates that only current data is of interest.

The entry rs_cdelta is intended to support tuple differencing, and is not used at present.

The structure entry rs_parallel_ok was added to support parallelization of POSTGRES for shared-memory architectures by Wei Hong, whose doctoral dissertation was on that topic. This entry is no longer used.

The entry rs_nkeys stores the number of ScanKeyEntryData structures that are stored in the rs_key entry that follows.

The rs_key entry stores a variable-length vector of ScanKeyEntryData structures, one per scan key that are to be applied for the scan.

1.3.3.2. The Index Scan Descriptor

The declaration for IndexScanDescData is

```
typedef struct IndexScanDescData {
      Relation         relation;
      Pointer          opaque;
      ItemPointerData  previousItemData;
      ItemPointerData  currentItemData;
      ItemPointerData  nextItemData;
      MarkData         previousMarkData;
      MarkData         currentMarkData;
      MarkData         nextMarkData;
      uint8            flags;
      Boolean          scanFromEnd;
      uint16           numberOfKeys;
      ScanKeyData      keyData;
      /* VARIABLE LENGTH ARRAY AT END OF STRUCT */
} IndexScanDescData;

typedef IndexScanDescData   *IndexScanDesc;
```

The relation entry points at the reldesc for the relation being scanned.

The opaque entry is for use by the indexed access method, and is assigned no meaning by higher-level scan processing code. The btree code uses it to store a pointer to a BTScanOpaque structure, defined in access/nbtree.h. The BTScanOpaque structure stores, among other things, the current buffer in use by the scan. This is equivalent to the rs_cbuf entry of the

HeapScanDescData structure.

The previousItemData, currentItemData, and nextItemData entries store the tids of the previous, current, and next index tuples (*not* heap tuples!) returned by the scan. As was the case for heap scans, it was originally believed that index scans would change direction frequently, making it useful to cache these values. In fact, this never happens, and the previous and next tuple tids could be removed from this structure.

The previousMarkData, currentMarkData, and nextMarkData entries serve the same purpose as rs_mptid, rs_mctid, and rs_mntid in the HeapScanDescData structure.

The flags entry is used to encode the direction of the scan (forwards, backwards, or no movement). The constants for these three values are declared in access/sdir.h, and are BackwardScanDirection, NoMovementScanDirection, and ForwardScanDirection.

The scanFromEnd flag is *true* if the scan should begin at an endpoint, and false otherwise. For example, in the btree code, a scan for all values greater than 500 could begin at 500 and move forward, or at the end of the relation and move backward.

The numberOfKeys entry records the number of keys stored in the keyData entry that follows. Currently, numberOfKeys is hardcoded to 1 in many places since multi-key indexing is not supported at this time.

The keyData entry stores a vector of ScanKeyEntryData structures that describe the scan key in use on the index.

1.4. The POSTGRES Access Method Interface

This section describes the programmatic interface to the POSTGRES access methods. The previous sections described the data structures that are important when using the access methods. Here, the structures are filled in and used to do real work.

The prototypes for the functions described in this section are not well-managed in the current version of the system. Some routines are not prototyped at all, and the prototypes that do exist are not ANSI. The header files in the directory access are where the prototypes appear.

The basic operations covered here are

- opening and closing relations,
- using scans to fetch tuples,
- fetching particular attributes of a tuple, and
- inserting, deleting, and replacing tuples.

There are two classes of interfaces. The routines whose names begin with heap_ operate on heap relations. The routines whose names begin with index_ work on index relations. The index_ routines use the pg_am tuple for the access method to call the routine that does the work for a particular access method. This dispatch happens transparently to the user.

1.4.1. Managing the System Catalogs

Because backend code frequently scans and updates the system catalogs, and because these catalogs often have special indices defined on them, POSTGRES includes special interfaces for dealing with some system catalogs. This section describes those interfaces.

1.4.1.1. The System Cache

Each POSTGRES backend maintains, in private memory, a cache of system catalog tuples that are frequently required during query processing. Cache consistency is provided by some extremely complicated invalidation code that runs at transaction boundaries. This cache, called the *sys cache*, actually consists of a number of different caches that support fast lookup of catalog tuples by various attributes. The cache management code appears in the source files `utils/cache/syscache.c` and `utils/cache/catcache.c`, and the cache invalidation code is in `storage/ipc/sinval.c` and `utils/cache/inval.c`.

The primary interface for using the sys cache is

```
HeapTuple
SearchSysCacheTuple(int cacheid, char *key1, char *key2,
                    char *key3, char *key4)
```

The `cacheid` argument is the sys cache of interest. The system currently supports the following caches:

Cache ID	*Relation*	*Cache Key Attributes*
AMOPOPID	pg_amop	amopclaid, amopopr
AMOPSTRATEGY	pg_amop	amopid, amopclaid, amopstrategy
ATTNAME	pg_attribute	attrelid, attname
ATTNUM	pg_attribute	attrelid, attnum
INDEXRELID	pg_index	indexrelid
LANNAME	pg_language	lanname
OPRNAME	pg_operator	oprname, oprleft, oprright, oprkind
OPROID	pg_operator	OID
PRONAME	pg_proc	proname, pronargs, proargtypes
PROOID	pg_proc	OID
RELNAME	pg_class	relname
RELOID	pg_class	OID
TYPNAME	pg_type	typname
TYPOID	pg_type	OID
AMNAME	pg_am	amname
CLANAME	pg_opclass	opcname
INDRELIDKEY	pg_index	indrelid, indkey
INHRELID	pg_inherits	inhrelid, inhseqno
PRS2PLANCODE	pg_prs2	prs2ruleid, prs2planno
RULOID	pg_rewrite	OID
PRS2STUB	pg_prs2	prs2relid, prs2no
PRS2RULEID	pg_prs2	prs2name
PRS2EVENTREL	pg_prs2	OID
AGGNAME	pg_aggregate	aggname
NAMEREL	pg_naming	parent_oid, filename
LOBJREL	pg_large_object	object_oid
LISTENREL	pg_listener	relname, pid
USENAME	pg_user	usename
USESYSID	pg_user	usesysid
GRONAME	pg_group	groname
GROSYSID	pg_group	grosysid
REWRITENAME	pg_rewrite	rulename

The `USESYSID` cache, for example, allows fast lookup of users in `pg_user` by the `usesysid` attribute. To do such a lookup,

```
HeapTuple pg_user_tup;

pg_user_tup = SearchSysCacheTuple(USESYSID, 1806, 0, 0, 0);
```

If any tuple with `usesysid` equal to 1806 exists in `pg_user`, it is loaded into the cache if necessary, and a pointer to it is returned.

1.4.1.2. Maintaining System Catalog Indices

Since system catalog indices are critical to the fast and correct functioning of POSTGRES, several support routines have been defined to open, update, and close indices on a given catalog when it is updated. An example of the use of these routines appears in the file `catalog/heap.c`, in the routine `AddNewAttributeTuples`. The routines are defined in `catalog/indexing.c`. These rouintes open all the indices on some catalog relation, insert index tuples into all the indices, and close the indices. The interfaces are

```
void
CatalogOpenIndices(int nIndices, char **names,
              Relation *ind_relns)

void
CatalogIndexInsert(Relation *ind_relns, int nIndices,
              Relation heap_reln, HeapTuple htup)

void
CatalogCloseIndices(int nIndices, Relation *ind_relns)
```

For `CatalogOpenIndices`, the `nIndices` argument is the number of indices on the catalog; constants for all catalogs are defined in `catalog/indexing.h`. The `names` vector is an array of names of indices on the catalog; constants for these arrays are provided in the same header file. Finally, `ind_relns` is a vector of index reldesc pointers that is large enough to hold `nIndices` reldesc pointers. This is filled in by `CatalogOpenIndices`.

For `CatalogIndexInsert`, the `ind_relns` argument is the reldesc vector returned by `CatalogOpenIndices`. `NIndices` is the number of indices on the catalog. The `heap_reln` argument is the catalog relation being updated, and `htup` is the new tuple inserted into the catalog. When `CatalogIndexInsert` returns, all of the indices on the system catalog will have been updated with index tuples pointing at the new heap tuple.

For `CatalogCloseIndices`, `nIndices` is the number of indices to close, and `ind_relns` is the reldesc vector from `CatalogOpenIndices`.

1.4.2. Opening and Closing Relations

Relations may be opened by relid (object ID of the corresponding `pg_class` tuple) or by name. The interfaces are

```
Relation
heap_open(ObjectId relid)

Relation
heap_openr(Name relname)

Relation
index_open(ObjectId relid)
```

```
Relation
index_openr(Name relname)
```

The first two open the requested heap relation if it exists, and return the reldesc for it. The second two do the same for index relations. If the named relations do not exist, the routines will report a `NOTICE`-level `elog` message, but will not abort. Instead, they return a NULL reldesc.

In order to close a relation,

```
void
heap_close(Relation reldesc)

void
index_close(Relation reldesc)
```

1.4.3. Using Scans

Once a relation has been opened, a scan may be set up on it to find tuples of interest.

1.4.3.1. Beginning and Ending Scans

To set up a scan key,

```
void
ScanKeyEntryInitialize(ScanKeyEntry entry, bits16 flags,
                  AttributeNumber attributeNumber,
                  RegProcedure procedure, Datum argument)
```

As many scan key entries as desired may be defined for a single scan. These should be allocated as an array of `ScanKeyEntryData` structures, as shown in section 1.3.2.

Once a set of scan keys is initialized, the routines

```
HeapScanDesc
heap_beginscan(Relation reldesc, bool atend,
            TimeQual timeQual, unsigned nkeys, ScanKey key)

IndexScanDesc
index_beginscan(Relation reldesc, bool scanFromEnd,
             uint16 numberOfKeys, ScanKey key)
```

will begin scans using the keys. The keys should be in the array beginning at address `key`.

To end an open scan,

```
void
heap_endscan(HeapScanDesc scan)

void
index_endscan(IndexScanDesc scan)
```

The scans may no longer be used once the `endscan` routine has been called on them.

1.4.3.2. Fetching Qualifying Tuples

Once a scan has been initialized by the `beginscan` routines, qualifying tuples may be fetched from the relation by calling the routines

```
HeapTuple
heap_getnext(HeapScanDesc scan, int backw, Buffer *b)

RetrieveIndexResult
index_getnext(IndexScanDesc scan, ScanDirection direction)
```

In both cases, the `scan` argument is the scan to use to find qualifying tuples. For heap scans, if `backw` is non-zero, then the scan moves backwards; otherwise, it moves forwards. There is never a reason to set the `backw` flag. For index scans, `direction` may be one of `ForwardScanDirection`, `NoMovementScanDirection`, or `BackwardScanDirection`. `ForwardScanDirection` is commonly used, and the others may not work.

The purpose of the `b` argument for the heap routine is described below.

1.4.3.3. Buffer Management

POSTGRES manages a shared cache of recently-used buffers. This cache is available to all of the backends that are running concurrently. The shared cache allows some backends to take advantage of work done by others. For example, pages from the system catalog are typically moved into the cache by one backend and read by many others.

When a given backend has a pointer into one of the pages in the shared buffer cache, that buffer is *pinned*. A pinned buffer cannot be evicted from the cache. When the pointer is dropped, the buffer may be safely unpinned and evicted from the cache, and the space that it occupied may be used by another page from a database.

When scanning tuples in a relation, the user may choose to examine the tuple directly on the page, or to copy the tuple to private space and to examine the copy. The benefit to using the tuple on the page is that no extra copy is required. The down side is that the user must be more careful in managing his scan. In particular, he must unpin the buffer when he no longer plans to use the pointer into it.

The interface to fetch tuples from a heap scan is

```
HeapTuple
heap_getnext(HeapScanDesc scan, int backw, Buffer *b)
```

The `b` argument is a pointer to a value of type `Buffer`. `Buffer` is basically an integer, which is the number of the buffer in the shared buffer cache.

- If the `b` argument to `heap_getnext` is not `(Buffer *) NULL`, then the buffer number on which the returned tuple appears will be copied to the `Buffer` value that `b` points to.
- If the `b` argument is NULL, then memory is allocated (via `palloc`[4]) and the tuple is

[4] `Palloc` is the POSTGRES memory allocator. `Palloc` allocates memory from memory pools, some of which perform garbage collection at transaction end.

copied from the data page to the allocated memory.

The following code fragments show how to use the b argument in both cases. The first example uses the tuple directly on the data page.

```
HeapTuple htup;
Buffer b;

...
htup = heap_getnext(hscan, 0, &b);

/* ... process the tuple ... */

/* all done */
ReleaseBuffer(b);
```

When `heap_getnext` returns, `b` is the buffer number on which the tuple appears, and the *pin count* on `b` has been incremented to account for this new reference. After the call to `ReleaseBuffer`, the tuple pointed to by `htup` may no longer be used. Some of the most insidious bugs in the project's history were from users who unpinned buffers, but continued to use tuples on them. The buffer would be paged out at some later time by another process, and the user would have a pointer into some random location of another data page.

The second example shows how to use a copy of the tuple that does not reside on the page.

```
HeapTuple htup;

...
htup = heap_getnext(hscan, 0, (Buffer *) NULL);

/* ... process the tuple ... */

/* all done */
pfree(htup);
```

In this case, the user should release the memory allocated to the tuple before returning. Allocated memory is automatically freed at transaction boundaries, but relying on this feature causes memory leaks that make POSTGRES run slowly and consume large amounts of system memory.

1.4.3.4. Using an Index Scan

Buffer management is not required for index scans. Data of interest are copied to a special structure, called a `RetrieveIndexResult`. A `RetrieveIndexResult` includes the tid of the index tuple and the tid of the heap tuple that it references. To use these,

```
RetrieveIndexResult res;
Relation heap_reln, index_reln;
HeapTuple htup;
ItemPointer heap_tid;
Buffer b;
```

```
...
indscan = index_beginscan(index_reln, ...);
res = index_getnext(indscan, ForwardScanDirection);
if (res) {
      heap_tid = RetrieveIndexResultGetHeapItemPointer(res);
      htup = heap_fetch(heap_reln, NowTimeQual, heap_tid, &b);
      if (HeapTupleIsValid(htup)) {

            /* ... process the heap tuple ... */

            /* all done */
            ReleaseBuffer(b);
      }
}
```

The index relation requires no explicit buffer management. The heap tid of interest is extracted from the `RetrieveIndexResult`, and is used to fetch the heap tuple itself. The `heap_fetch` interface fetches a heap tuple by tid; its declaration is

```
HeapTuple
heap_fetch(Relation relation, TimeQual timeQual,
            ItemPointer tid, Buffer *b)
```

The description of buffer management for heap scans applies to the `heap_fetch` interface, as well.

If the resulting heap tuple does not satisfy the time qualification, then `heap_fetch` will return NULL. Since the index may contain old data until the vacuum cleaner is run, this is possible, so the user must check the return value from `heap_fetch`.

If there are no more index tuples that match the qualification, `index_getnext` will return NULL. At that point, the scan must be ended using `index_endscan`. If `index_endscan` is not called, any subsequent calls to `index_getnext` on the scan will reset it to the beginning of the qualifying index tuples, and they will all be returned again. This is not intended as a feature, so should not be used.

1.4.4. Fetching Attributes

Once a tuple of interest has been fetched from the heap, attributes of interest may be extracted from it and used. The interfaces are

```
char *
heap_getattr(HeapTuple tup, Buffer b, int attnum,
              TupleDescriptor tupdesc, bool *isnull)

Pointer
index_getattr(IndexTuple tup, int attnum,
              TupleDescriptor tupdesc, bool *isnull)
```

In general, users (and even POSTGRES implementors) need not extract index tuple values; only access method implementors need to do that. The rest of this section concentrates on the heap_getattr interface.

The tup argument points at the tuple of interest. If b is non-null, then it is the buffer on which the tuple resides. If the user is doing explicit buffer management via heap_fetch or heap_getnext, then the returned buffer may be passed along to heap_getattr.

The tuple descriptor is actually the rd_att entry from the reldesc for the relation that stores the tuple. To extract the tuple descriptor,

```
Relation reln;
TupleDescriptor tupdesc;

...
tupdesc = RelationGetTupleDescriptor(reln);
```

The isnull argument is a pointer to a value of type bool. If the corresponding attribute has the database value null, then isnull will be *true* and the return value should be ignored.

Although it is declared to return type char *, heap_getattr actually returns type Datum, so the return value should always be coerced to that type. This is an implementation mistake. The DatumGet*Type* macros described in section 1.3.2 convert values from Datum to the type required by the user.

1.4.5. Inserting New Tuples

When a user submits a POSTQUEL query to insert new tuples into a heap relation, the POSTGRES executor automatically updates all associated indices with pointers to the new tuple after it has been inserted. There is no simple interface for doing this in general on user relations inside the POSTGRES backend. Section 1.4.1.2 describes a simple way for doing this on system catalogs, but the user must explicitly code updates to indices on user relations when new values are inserted into the heap.

The code for the routine CopyFrom in the source file commands/copy.c contains the sample code that was used to produce this system. That file is a reasonable reference for how to manage tuple insertions correctly.

1.4.5.1. Forming a Heap Tuple

The routine

```
HeapTuple
heap_formtuple(AttributeNumber natts, TupleDescriptor tupdesc,
               Datum *values, char *nulls)
```

creates a heap tuple from the supplied vector of Datum values. The natts argument is the number of attributes in the relation (the rd_rel->relnatts entry of the reldesc). The tupdesc is the rd_att entry of the reldesc, which may be fetched via RelationGetTupleDescriptor.

Both values and nulls are arrays of length natts. Values contains a datum for every non-null attribute in the tuple. The *Type*GetDatum macros can be used to initialize the entries in this array. If an attribute is null, its space is ignored in the array. For example, if the third attribute is null, then the value stored in the third location of the `values` array will be ignored by `heap_formtuple`.

The `nulls` array is a vector of characters, one per attribute in the tuple. If the corresponding attribute has the database value null, then the array entry is the character *n*. If the corresponding attribute is non-null, then the array entry is the ASCII blank character.

The heap tuple returned by `heap_formtuple` consumes `palloc`ed memory, and should be `pfree`d when the heap and index relations have all been updated.

1.4.5.2. Inserting the Heap Tuple

To insert the tuple returned by `heap_formtuple`,

```
ObjectId
heap_insert(Relation reln, HeapTuple tup, double *off)
```

On return, the `tup->t_ctid` entry is the tid of the tuple in the heap. The `off` argument is unnecessary and should never be used. Its purpose is unclear. This argument may safely be C-language NULL. `heap_insert` returns the object ID of the new tuple in the heap (the `t_oid` entry of `tup` is also correctly set).

1.4.5.3. Finding the Indices

The procedure `GetIndexRelations` finds all of the indices defined on a heap relation and initializes an array with the reldescs for all of the indices. The interface is

```
GetIndexRelations(ObjectId heap_relid, int *n_indices,
                  Relation **index_reldescs);
```

The `heap_relid` argument is the `rd_id` entry of the reldesc for the heap. `N_indices` is set, on return, to the number of indices for the heap relation. On return, `index_reldescs` is a pointer to an array of reldescs, one for each index defined on the heap.

1.4.5.4. Updating the Indices

A POSTGRES index may be

- *functional*, meaning that the index is on the return value of some function of the heap tuple's attributes, rather than on the attributes themselves;
- *partial*, meaning that a predicate controls whether or not particular heap tuples should appear in the index; or
- *regular*, meaning that the index is on one or more attributes of all tuples in the heap.

Functional indices are used for some system catalogs, since the current version of POSTGRES does not support multi-column btree indices. Partial indices are very seldom used. Normal indices are by far the most common. The rest of this section ignores functional and partial indices, and concentrates on normal ones. For sample code that updates these two kinds of indices, see the routine `CopyFrom` in `commands/copy.c`. Special interfaces, described in section 1.4.1.2, have been defined to update the functional and normal indices on system catalogs, so the user need not manage these by hand.

After calling GetIndexRelations, the user has the number of indices and all of the index reldescs for the indices defined on the heap. Updating the indices requires forming an index tuple for each index and inserting it. The following code fragment gives an example.

```
Relation heap_reln;
Relation *index_relns;
int n_indices;
Datum idatum;
HeapTuple htup;
IndexTuple itup;
HeapTuple pg_indextup;
Form_pg_index pgiform;
TupleDescriptor tupdesc;
TupleDescriptor itupdesc;
char *nulls;
int i;

...
tupdesc = RelationGetTupleDescriptor(heap_reln);

/* get space for null flags -- this is more than we need */
nulls = palloc(heap_reln->rd_rel->relnatts);
for (i = 0; i < heap_reln->rd_rel->relnatts; i++)
      nulls[i] = ' ';

/* ... insert the heap tuple, which sets htup->t_ctid ... */

/* update each index */
for (i = 0; i < n_indices; i++) {
      /* get the pg_index tuple for this index */
      pg_indextup = SearchSysCacheTuple(INDEXRELID,
                                        index_relns[i]->rd_id,
                                        NULL, NULL, NULL);
      pgiform = (Form_pg_index) GETSTRUCT(pg_indextup);

      /* form a datum to use for the index tuple */
      FormIndexDatum(index_relns[i]->rd_rel->relnatts,
                 (AttributeNumber *) &pgiform->indkey.data[0],
                 htup, tupdesc, InvalidBuffer,
                 &idatum, nulls, NULL);

      /* form the index tuple */
      itupdesc = RelationGetTupleDescriptor(index_relns[i]);
      itup = index_formtuple(1, itupdesc, &idatum, nulls);

      /* make it point at the heap tuple */
      itup->t_tid = htup->t_ctid;
```

```
        /* insert it */
        (void) index_insert(index_relns[i], itup, NULL, NULL);

        pfree(itup);
    }

    pfree(nulls);
```

The loop iterates over all the indices defined on the heap, updating each in turn. For each index, it fetches the corresponding `pg_index` tuple in the cache of system tuples, extracts the `Form_pg_index` structure from each, and then forms a `Datum` with the appropriate value for the index tuple. Finally, an index tuple is formed and inserted into the index.

1.4.6. Deleting Tuples

Compared to inserting tuples, deleting tuples is simple. Users never delete tuples from indices; this is done only by the vacuum cleaner. Deleting a heap tuple requires the tid of the tuple to be deleted. The tid may be extracted from the heap tuple itself, which can be found, for example, by a scan of the heap relation. Given the tid, the interface for deleting heap tuples is

```
RuleLock
heap_delete(Relation reln, ItemPointer tid)
```

The `RuleLock` return value from `heap_delete` was originally intended to support the POSTGRES rule system, but the design of the rule system changed, so this value is not used. `Heap_delete` always returns NULL.

1.4.7. Replacing Existing Tuples

When a heap tuple in POSTGRES is replaced, it is marked as deleted and a new tuple is inserted in the relation with new values. The new tuple has the same object ID as the original tuple, but will be stored at a new location and so will have a new tid.

Index tuples are never replaced. If the heap tuples to which they point have been replaced, the vacuum cleaner deletes them from the index. No user action is required. However, if a heap tuple is replaced, all the indices defined on it must have new index tuples inserted. These index tuples must point at the new heap tuple. The code for doing this is identical to the code for inserting index tuples that point at new heap tuples, and will not be repeated here.

Replacing a tuple consists of two steps. First, a new tuple is formed, based on the old one, but with some values changed. Second, the old tuple is replaced by the new one.

Replacing a tuple requires knowledge of the schema of the relation storing the tuple, since new values for particular attributes must be supplied. The sample code fragment below assumes the existence of a relation `emp` with the schema

```
emp(name = char16, salary = int4)
```

The code sample replaces a particular `emp` tuple, assigning the employee a salary of 50,000. Note that the code fragment does not update any indices defined on `emp`; that code must be added if any such indices exist.

```
Relation empreln;
HeapTuple emptup;
HeapTuple newtup;
Buffer empbuf;
Datum value[2];
char nulls[2];
char repl[2];

...
/* by here, we have the emp tuple to replace */
value[0] = (Datum) 0;
value[1] = Int32GetDatum(50000); /* new salary */

/* no null attributes in this tuple */
nulls[0] = nulls[1] = ' ';

/* replace the second att, not the first */
repl[0] = ' ';
repl[1] = 'r';

/* get a new tuple based on the old one, with the new salary */
newtup = heap_modifytuple(emptup, empbuf, empreln, value,
                          nulls, repl);

/* replace the old tuple with the new one */
(void) heap_replace(empreln, &emptup->t_ctid, newtup);
```

The interface for `heap_modifytuple` is

```
HeapTuple
heap_modifytuple(HeapTuple tuple, Buffer buffer,
                 Relation relation, Datum *replValue,
                 char *replNull, char *repl)
```

The `tuple` argument is the tuple to replace. The `buffer` argument is the buffer in the buffer cache on which it appears, as returned by `heap_getnext` or `heap_fetch`. The `relation` argument is the relation containing the tuple.

The next three arguments control what values are stored in the new tuple. The first, `replValue`, is a vector of values of type `Datum` containing one entry for each attribute in the relation. If an attribute is not to be replaced, but rather should be copied from the old tuple, then the corresponding entry in `replValue` will be ignored.

The `replNull` argument contains one entry per attribute in the relation. If the corresponding attribute is to be replaced, and if the `replNull` entry is the character *n*, then the new value for that attribute is null. If the value is to be replaced and the `replNull` entry is an ASCII blank, then the value stored in `replValue` is consulted to get the new value for the attribute.

Finally, the `repl` argument contains one entry per attribute in the relation. If the `repl` entry for a particular attribute is the character *r*, then the value should be replaced, and the replValue and replNull entries for the attribute are used to assign a new value. If the repl entry is an ASCII blank, then the value is copied from the original tuple.

1.4.8. Creating and Destroying Relations

Creating a heap relation requires the user to supply a schema for the new relation. Creating an index relation is easier, because the schema of the index is defined in terms of the heap. The two cases are treated separately below.

1.4.8.1. Heap Relations

Creating a heap relation requires a schema for the new relation. The schema should be a TupleDescriptor, which is the same as the rd_att entry of the reldesc for the relation about to be created. Creating the tuple descriptor is straightforward; a vector is allocated with one FormData_pg_attribute structure for each attribute. The FormData_pg_attribute structures are filled in with the names, types, and other appropriate information for the new attributes. The attrelid and attdefrel entries may be left as InvalidObjectId; these will be filled in by the code to create the heap relation.

Once the tuple descriptor has been constructed, creating the relation is straightforward. The interface is

```
ObjectId
heap_create(char *relname, int archive, unsigned natts,
        unsigned smgr, TupleDescriptor tupdesc)
```

The relname argument is the name of the new relation; it must be unique, or heap_create will abort the transaction.

The archive argument, although declared to be of type int in the prototype, is actually a character; this should be one of *n* (no archiving), *l* (light archiving), or *h* (heavy archiving). Light and heavy archiving behave identically in the current system, and cause deleted and replaced tuples to be copied to a special archive by the vacuum cleaner. No archiving causes the vacuum cleaner to discard deleted and replaced tuples.

The natts argument is the number of attributes for the new relation. This should be the same as the length of the tupdesc vector.

The smgr argument is the storage manager on which the relation should be created. The set of supported storage managers varies from installed system to installed system. Storage manager number zero is always magnetic disk. Other numbers are installation-dependent.

Finally, the tupdesc argument is the tuple descriptor for the new relation.

Heap_create returns the object ID of the new relation's pg_class tuple, which is the same as the relid of the new relation.

Destroying a heap relation is even simpler:

```
void
heap_destroy(char *relname)
```

If the named relation does not exist, heap_destroy aborts the transaction. Otherwise, the relation is destroyed and all indices defined on it are removed. In addition, if the relation inherited attributes from other relations, its inheritance information is deleted from the system catalogs.

1.4.8.2. Index Relations

Creating an index relation is straightforward. The interface is

```
void
index_create(Name heap_name, Name index_name,
             FuncIndexInfo fInfo, ObjectId am_id,
             AttributeNumber natts, AttributeNumber *attnos,
             ObjectId *opclasses, uint16 paramCount,
             Datum *params, LispValue predicate)
```

The `fInfo`, `paramCount`, `params`, and `predicate` arguments are for defining functional or partial indices, and will not be covered here.

The `heap_name` argument is the name of the heap relation on which the index is to be defined. The `index_name` argument is the name of the new index relation.

The `am_id` argument is the object ID of the access method tuple for this index, from `pg_am`.

The `natts` argument is the number of attributes (keys) for this index. The current system supports only single-key indices, but this may change in a subsequent release. The `attnos` argument is a vector of attribute numbers in the heap. If the index is to be defined on the third attribute in the heap, then this vector would consist of a single entry, which would be three.

The `opclasses` argument is the object ID of the operator class tuple from `pg_amop` to use for each attribute. The operator classes map types and operators for the index to particular functions in `pg_proc`.

Sample code that defines regular, functional, and partial indices may be found in the source file `commands/defind.c`.

When `index_create` returns, the new index has been created and tuples have been inserted into it for each tuple in the heap relation.

1.4.9. Scan Position Management

Scans on index or heap relations support *marking* and *restoring* of the scan's position. A mark records the current scan location, and a restore restores the previously marked location. Only one mark may be defined on an open scan.

Marking and restoring are of limited use outside the executor, but the interfaces are included here for completeness. They are

```
void
heap_markpos(HeapScanDesc hscan)

void
heap_restrpos(HeapScanDesc hscan)

void
index_markpos(IndexScanDesc iscan)
```

```
void
index_restrpos(IndexScanDesc iscan)
```

1.4.10. Locking

Users of the access method interface routines need not worry about locking. The access method routines acquire (and, when appropriate, release) locks on relations in response to scans and updates. Heap relations are protected by two-phase relation level locks. Btree indices use Lehman-Yao[5] short-term locking for high concurrency, but since the underlying relations are locked at the relation level, index updates are serialized. Rtree indices use two-phase relation-level locking.

System catalogs in POSTGRES are not subject to two-phase locking in order to improve concurrency. This means that under some circumstances concurrent transactions that operate on the system catalogs are not serializable. This flaw is common in commercial relational systems, as well, and occurs infrequently enough that it does not matter in practice.

---

[5] Lehman, P., Yao, S., "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, 6(4), December 1981.