CHAPTER 32

PostgreSQL

Ioannis Alagiannis (Swisscom AG)
Renata Borovica-Gajic (University of Melbourne, AU)[1]

PostgreSQL is an open-source object-relational database management system. It is a descendant of one of the earliest such systems, the POSTGRES system developed under Professor Michael Stonebraker at the University of California, Berkeley. The name "Postgres" is derived from the name of a pioneering relational database system, Ingres. Currently, PostgreSQL offers features such as complex queries, foreign keys, triggers, views, transactional integrity, full-text searching, and limited data replication. Users can extend PostgreSQL with new data types, functions, operators, or index methods. PostgreSQL supports a variety of programming languages (including C, C++, Java, Perl, Tcl, and Python), as well as the database interfaces JDBC and ODBC.

PostgreSQL runs under virtually all Unix-like operating systems, including Linux, Microsoft Windows and Apple Macintosh OS X. PostgreSQL has been released under the BSD license, which grants permission to anyone for the use, modification, and distribution of the PostgreSQL code and documentation for any purpose without any fee. PostgreSQL has been used to implement several different research and production applications (such as the PostGIS system for geographic information) and is used an educational tool at several universities. PostgreSQL continues to evolve through the contributions of a large community of developers.

In this chapter, we explain how PostgreSQL works, starting from user interfaces and languages, and continuing into the internals of the system. The chapter would be useful for application developers who use PostgreSQL, and desire to understand and

[1]Both authors equally contributed to this work.
This chapter is an online resource of Database System Concepts, 7th edition, by Silberschatz, Korth and Sudarshan, McGraw-Hill, 2019.

This chapter is released under the Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA) 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0. For any use beyond those covered by this license, obtain permission by emailing *db-book-authors@cs.yale.edu*.

make better use of its features. It would also be particularly useful for students and developers who wish to add functionality to the PostgreSQL system, by modifying its source code.

32.1 Interacting with PostgreSQL

The standard distribution of PostgreSQL comes with command-line tools for administering the database. However, there is a wide range of commercial and open-source graphical administration and design tools that support PostgreSQL. Software developers may also access PostgreSQL through a comprehensive set of programming interfaces.

32.1.1 Interactive Terminal Interfaces

Like most database systems, PostgreSQL offers command-line tools for database administration. The psql interactive terminal client supports execution of SQL commands on the server, and viewing of the results. Additionally, it provides the user with meta-commands and shell-like features to facilitate a wide variety of operations. Some of its features are:

- **Variables.** psql provides variable substitution features, similar to common Unix command shells.
- **SQL interpolation.** The user can substitute ("interpolate") psql variables into SQL statements by placing a colon in front of the variable name.
- **Command-line editing.** psql uses the GNU readline library for convenient line editing, with tab-completion support.

32.1.2 Graphical Interfaces

The standard distribution of PostgreSQL does not contain any graphical tools. However, there are several open source as well as commercial graphical user interface tools for tasks such as SQL development, database administration, database modeling/design and report generation. Graphical tools for SQL development and database administration include pgAdmin, PgManager, and RazorSQL. Tools for database design include TOra, Power*Architect, and PostgreSQL Maestro. Moreover, PostgreSQL works with several commercial forms-design and report-generation tools such as Reportizer, db-Forge, and Database Tour.

32.1.3 Programming Language Interfaces

PostgreSQL standard distribution includes two client interfaces libpq and ECPG. The libpq library provides the C API for PostgreSQL and is the underlying engine for most programming-language bindings. The libpq library supports both synchronous

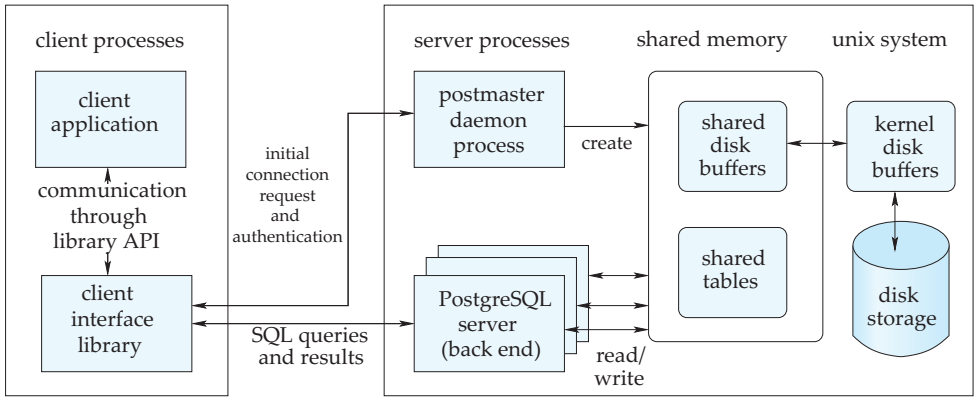

Figure 32.1 The PostgreSQL system architecture.

and asynchronous execution of SQL commands and prepared statements, through a re-entrant and thread-safe interface. The connection parameters of libpq may be configured in several flexible ways, such as setting environment variables, placing settings in a local file, or creating entries on an LDAP server. ECPG (Embedded SQL in C) is an embedded SQL preprocessor for the C language and PostgreSQL. It allows for accessing PostgreSQL using C programs with embedded SQL code in the following form: *EXEC SQL* sql-statements. ECPG provides a flexible interface for connecting to the database server, executing SQL statements, and retrieving query results, among other features.

Apart from the client interfaces included in the standard distribution of PostgreSQL, there are also client interfaces provided from external projects. These include native interfaces for ODBC and JDBC, as well as bindings for most programming languages, including C, C++, PHP, Perl, Tcl/Tk, JavaScript, Python, and Ruby.

32.2 System Architecture

The PostgreSQL system is based on a multi-process architecture, as shown in Figure 32.1. A set of one or more of databases is managed by a collection of processes. The **postmaster** is the central coordinating process and is responsible for system initialization (including allocation of shared memory and starting of background processes), and for shutting down the server. Additionally, the **postmaster** manages connections with client applications and assigns each new connecting client to a backend server process for executing the queries on behalf of the client and for returning the results to the client.

Client applications can connect to the PostgreSQL server and submit queries through one of the many database application program interfaces supported by PostgreSQL (libpq, JDBC, ODBC) that are provided as client-side libraries. An example client application is the command-line psql program, included in the standard

PostgreSQL distribution. The postmaster is responsible for handling the initial client connections. For this, it constantly listens for new connections on a known port. When it receives a connection request, the postmaster first performs initialization steps such as user authentication, and then assigns an idle backend server process (or spawns a new one if required) to handle the new client. After this initial connection, the client interacts only with the backend server process, submitting queries and receiving query results. As long as the client connection is active, the assigned backend server process is dedicated to only that client connection. Thus, PostgreSQL uses a **process-per-connection model** for the backend server.

The backend server process is responsible for executing the queries submitted by the client by performing the necessary query-execution steps, including parsing, optimization, and execution. Each backend server process can handle only a single query at a time. An application that desires to execute more than one query in parallel must maintain multiple connections to the server. At any given time there may be multiple clients connected to the system, and thus multiple backend server processes may be executing concurrently.

In addition to the *postmaster* and the *backend* server processes PostgreSQL utilizes several background worker processes to perform data management tasks, including the background writer, the statistics collector, the write-ahead log (WAL) writer and the checkpointer processes. The background writer process is responsible for periodically writing the dirty pages from the shared buffers to persistent storage. The statistics collector process continuously collects statistics information about the table accesses and the number of rows in tables. The WAL writer process periodically flushes the WAL data to persistent storage while the checkpointer process performs database checkpoints to speed up recovery. These background processes are initiated by the *postmaster* process.

When it comes to memory management in PostgreSQL, we can identify two different categories a) local memory and b) shared memory. Each backend process allocates local memory for its own tasks such as query processing (e.g., internal sort operations hash tables and temporary tables) and maintenance operations (e.g., *vacuum*, *create index*).

On the other hand, the in-memory buffer pool is placed in shared memory, so that all the processes, including backend server processes and background processes can access it. Shared memory is also used to store lock tables and other data that must be shared by server and background processes.

Due to the use of shared memory as the inter-process communication medium, a PostgreSQL server should run on a single shared-memory machine; a single-server site cannot be executed across multiple machines that do not share memory, without the assistance of third-party packages. However, it is possible to build a shared-nothing parallel database system with an instance of PostgreSQL running on each node; in fact, several commercial parallel database systems have been built with exactly this architecture.

32.3 Storage and Indexing

PostgreSQL's approach to data layout and storage has the goals of (1) a simple and clean implementation and (2) ease of administration. As a step toward these goals, PostgreSQL relies on file-system files for data storage (also referred to as "*cooked*" files), instead of handling the physical layout of data on raw disk partitions by itself. PostgreSQL maintains a list of directories in the file hierarchy to use for storage; these directories are referred to as **tablespaces**. Each PostgreSQL installation is initialized with a default tablespace, and additional tablespaces may be added at any time. When creating a table, index, or entire database, the user may specify a tablespace in which to store the related files. It is particularly useful to create multiple tablespaces if they reside on different physical devices, so that tablespaces on the faster devices may be dedicated to data that are accessed more frequently. Moreover, data that are stored on separate disks may be accessed in parallel more efficiently.

The design of the PostgreSQL storage system potentially leads to some performance limitations, due to clashes between PostgreSQL and the file system. The use of cooked file systems results in double buffering, where blocks are first fetched from disk into the file system's cache (in kernel space), and are then copied to PostgreSQL's buffer pool. Performance can also be limited by the fact that PostgreSQL stores data in 8-KB blocks, which may not match the block size used by the kernel. It is possible to change the PostgreSQL block size when the server is installed, but this may have undesired consequences: small blocks limit the ability of PostgreSQL to store large tuples efficiently, while large blocks are wasteful when a small region of a file is accessed.

On the other hand, modern enterprises increasingly use external storage systems, such as network-attached storage and storage-area networks, instead of disks attached to servers. Such storage systems are administered and tuned for performance separately from the database. PostgreSQL may directly leverage these technologies because of its reliance on "cooked" file systems. For most applications, the performance reduction due to the use of "cooked" file systems is minimal, and is justified by the ease of administration and management, and the simplicity of implementation.

32.3.1 Tables

The primary unit of storage in PostgreSQL is a table. In PostgreSQL, tuples in a table are stored in *heap files.* These files use a form of the standard *slotted-page* format (Section 13.2.2). The PostgreSQL slotted-page format is shown in Figure 32.2. In each page, the page header is followed by an array of *line pointers* (also referred to as *item identifiers*). A line pointer contains the offset (relative to the start of the page) and length of a specific tuple in the page. The actual tuples are stored from the end of the page to simplify insertions. When a new item is added in the page, if all line pointers are in use, a new line pointer is allocated at the beginning of the unallocated space (pd_lower) while the actual item is stored from the end of the unallocated space (pd_upper).

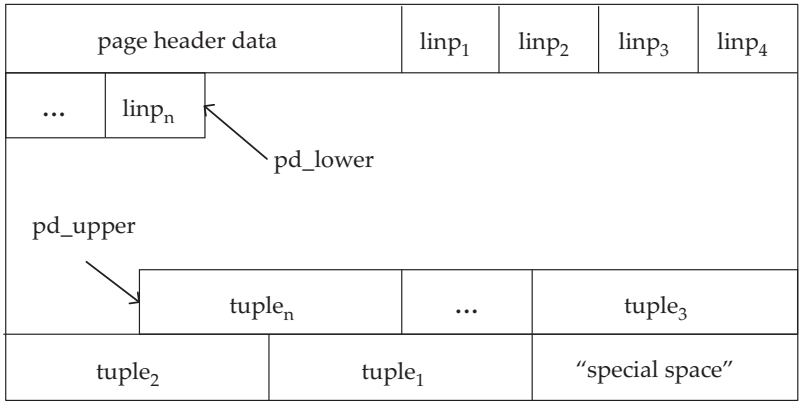

Figure 32.2 Slotted-page format for PostgreSQL tables.

A record in a heap file is identified by its **tuple ID** (**TID**). The TID consists of a 4-byte block ID which specifies the page in the file containing the tuple and a 2-byte slot ID. The slot ID is an index into the line pointer array that in turn is used to access the tuple.

Due to the multi-version concurrency control (MVCC) technique used by PostgreSQL, there may be multiple versions of a tuple, each with an associated start and end time for validity. Delete operations do not immediately delete tuples, and update operations do not directly update tuples. Instead, deletion of a tuple initially just updates the end-time for validity, while an update of a tuple create a new version of the existing tuple; the old version has its validity end-time set to just before the validity start-time of the new version.

Old versions of tuples that are no longer visible to any transaction are physically deleted later; deletion causes holes to be formed in a page. The indirection of accessing tuples through the line pointer array permits the compaction of such holes by moving the tuples, without affecting the TID of the tuple.

The length of a physical tuple is limited by the size of a data page, which makes it difficult to store very long tuples. When PostgreSQL encounters a large tuple that cannot fit in a page, it tries to "TOAST" individual large attributes, that is, compress the attribute or break it up into smaller pieces. In some cases, "toasting" an attribute may be accomplished by compressing the value. If compression does not shrink the tuple enough to fit in the page (as is often the case), the data in the toasted attribute is replaced by a reference to the attribute value; the attribute value is stored outside the page in an associated TOAST table. Large attribute values are split into smaller chunks; the chunk size is chosen such that four chunks can fit in a page. Each chunk is stored as a separate row in the associated TOAST table. An index on the combination of the identifier of a toasted attribute with the sequence number of each chunk allows efficient retrieval of the values. Only the data types with variable-length representation support

TOAST, to avoid imposing the overhead on data types that cannot produce large field values. The toasted attribute size is limited to 1 GB.

32.3.2 Indices

A PostgreSQL index is a data structure that provides a dynamic mapping from search predicates to sequences of tuple IDs from a particular table. The returned tuples are intended to match the search predicate, although in some cases the predicate must be rechecked on the actual tuples, since the index may return a superset of matching tuples. PostgreSQL supports several different index types, including indices that are based on user-extensible access methods. All the index types in PostgreSQL currently use the slotted-page format described in Section 32.3.1 to organize the data within an index page.

32.3.2.1 Index Types

PostgreSQL supports several types of indices that target different categories of workloads. In addition to the conventional B-tree,[2] PostgreSQL supports hash indices and several specialized indexing methods: the Generalized Search Tree (GiST), the Space-partitioned Generalized Search Tree (SP-GiST), the Generalized Inverted Index (GIN) and the Block Range Index (BRIN), which can be beneficial for workloads requiring full-text indexing, querying multi-value elements or being naturally clustered on some specific attribute(s).

B-tree: In PostgreSQL, the B-tree is the default index type, and the implementation is based on Lehman and Yao's B-link tree (B-link trees, described briefly in Section 18.10.2, are a variant of B^+-trees that support high concurrency of operations). B-trees can efficiently support equality and range queries on sortable data, as also certain pattern-matching operations such as some cases of **like** expressions.

Hash: PostgreSQL's hash indices are an implementation of linear hashing. Such indices are useful only for simple equality operations. The hash indices used by PostgreSQL had been shown to have lookup performance no better than that of B-trees while having considerably larger size and maintenance costs. The use of hash indices was generally discouraged due to the lack of write-ahead logging. However, in PostgreSQL 10 and 11, the hash index implementation has been significantly improved: hash indices now support write-ahead logging, can be replicated, and performance has improved as well.

Generalized Search Tree (GiST): The **Generalized Search Tree** (**GiST**) is an extensible indexing structure supported by PostgreSQL. The GiST index is based on a balanced tree-structure similar to a B-tree, but where several of the operations on the tree are not predefined, but instead must be specified by an access method implementor. GiST allows creation of specialized index types on top of the basic GiST template, without having to deal with the numerous internal details of a complete index implementation.

[2] As is conventional in the industry, the term B-tree is used in place of B^+-tree, and should be interpreted as referring to the B^+-tree data structure.

Examples of some indices built using GiST index include R-trees, as well as less conventional indices for multidimensional cubes and full-text search.

The GiST interface requires the access-method implementer to only implement certain operations on the data type being indexed, and specify how to arrange those data values in the GiST tree. New GiST access methods can be implemented by creating an operator class. Each GiST operator class may have a different set of strategies based on the search predicates implemented by the index. There are five support functions that an index operator class for GiST must provide, such as for testing set membership, for splitting sets of entries on page overflows, and for computing cost of inserting a new entry. GiST also allows four support functions that are optional, such as for supporting ordered scans, or to allow the index to contain a different type than the data type on which it is build. An index built on the GiST interface may be lossy, meaning that such an index might produce false matches; in that case, records fetched by the index need to have the index predicate rechecked, and some of the fetched records may fail the predicate.

PostgreSQL provides several index methods implemented using GiST such as indices for multidimensional cubes, and for storing key-value pairs. The original PostgreSQL implementation of R-trees was replaced by GiST operator classes which allowed R-trees to take advantage of the write-ahead logging and concurrency capabilities provided by the GiST index. The original R-tree implementation did not have these features, illustrating the benefits of using the GiST index template to implement specific indices.

Space-partitioned GiST (SP-GiST): Space-partitioned GiST indices leverage balanced search trees to facilitate disk-based implementations of a wide range of non-balanced data structures, such as quad-trees, k-d trees, and radix trees (tries). These data structures are designed for in-memory usage, with small node sizes and long paths in the tree, and thus cannot directly be used to implement disk-based indices. SP-GiST maps search tree nodes to disk pages in such a way that a search requires accessing only a few disk pages, even if it traverses a larger number of tree nodes. Thus, SP-GiST permits the efficient disk-based implementation of index structures originally designed for in-memory use. Similar to GiST, SP-GiST provides an interface with a high level of abstraction that allows the development of custom indices by providing appropriate access methods.

Generalized Inverted Index (GIN): The GIN index is designed for speeding up queries on multi-valued elements, such as text documents, JSON structures and arrays. A GIN stores a set of (key, posting list) pairs, where a posting list is a set of row IDs in which the key occurs. The same row ID might appear in multiple posting lists. Queries can specify multiple keys, for example with keys as words, GIN can be used to implement keyword indices.

GIN, like GiST, provides extensibility by allowing an index implementor to specify custom “strategies” for specific data types; the strategies specify, for example, how keys are extracted from indexed items and from query conditions, and how to determine whether a row that contains some of the key values in a query actually satisfies the query.

During query execution, GIN efficiently identifies index keys that overlap the search key, and computes a bitmap indicating which searched-for elements are members of the index key. To do so, GIN uses support function that extract members from a set, support functions that compare individual members. Another support function decides if the search predicate is satisfied, based on the bitmap and the original predicate. If the search predicate cannot be resolved without the full indexed attribute, the decision function must report a possible match and the predicate must be rechecked after retrieving the data item.

Each key is stored only once in a GIN, making GIN suitable for situations where many indexed items contain each key. However, updates are slower on GIN making them better for querying relatively static data, while GiST indices are preferred for workloads with frequent updates.

Block Range Index (BRIN): BRIN indices are designed for indexing such very large datasets that are naturally clustered on some specific attribute(s), with timestamps being a natural example. Many modern applications generate datasets with such characteristics. For example, an application collecting sensor measurements like temperature or humidity might have a timestamp column based on the time when each measurement was collected. In most cases, tuples for older measurements will appear earlier in the table storing the data. BRIN indices can speed up lookups and analytical queries with aggregates while requiring significantly less storage space than a typical B-tree index.

BRIN indices store some summary information for a group of pages that are physically adjacent in a table (block range). The summary data that a BRIN index will store depends on the operator class selected for each column of the index. For example, data types having a linear sort order can have operator classes that store the minimum and maximum value within each block range.

A BRIN index exploits the summary information stored for each block range to return tuples from only within the qualifying block ranges based on the query conditions. For example, in case of a table with a BRIN index on a timestamp, if the table is filtered by the timestamp, the BRIN is scanned to identify the block ranges that might have qualifying values and a list of block ranges is returned. The decision of whether a block range is to be selected or not is based on the summary information that the BRIN maintains for the given block range, such as the min and max value for timestamps. The selected block ranges might have false matches, that is the block range may contain items that do not satisfy the query condition. Therefore, the query executor re-evaluates the predicates on tuples in the selected block ranges and discards tuples that do not satisfy the predicate.

A BRIN index is typically very small, and scanning the index thus adds a very small overhead compared to a sequential scan, but may help in avoiding scanning large parts of a table that are found to not contain any matching tuples. The size of a BRIN index is determined by the size of the relation divided by the size of the block range. The smaller the block range, the larger the index, but at the same time the summary data stored can

be more precise, and thus more data blocks can potentially be skipped during an index scan.

32.3.2.2 Other Index Variations

For some of the index types described above, PostgreSQL supports more complex variations such as:

- **Multicolumn indices**: These are useful for conjuncts of predicates over multiple columns of a table. Multicolumn indices are supported for B-tree, GiST, GIN, and BRIN indices, and up to 32 columns can be specified in the index.
- **Unique indices**: Unique and primary-key constraints can be enforced by using unique indices in PostgreSQL. Only B-tree indices may be defined as being unique. PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table.
- **Covering indices**: A covering index is an index that includes additional attributes that are not part of the search key (as described earlier in Section 14.6). Such extra attributes can be added to allow a frequently used query to be answered using only the index, without accessing the underlying relation. Plans that use an index without accessing the underlying relation are called index-only plans, the implementation of index-only plans in PostgreSQL is described in Section 32.3.2.4. PostgreSQL uses the `include` clause to specify the extra attributes to be included in the index. A covering index can enhance query performance; however, the index build time increases since more data should be written and the index size becomes larger since the non-search attributes are duplicating data from the original table. Currently, only B-tree indices support covering indices.
- **Indices on expressions**: In PostgreSQL, it is possible to create indices on arbitrary scalar expressions of columns, and not just specific columns, of a table. An example is to support case-insensitive comparisons by defining an index on the expression *lower*(*column*). A query with the predicate *lower*(*column*) = 'value' cannot be efficiently evaluated using a regular B-tree index since matching records may appear at multiple locations in the index; on the other hand, an index on the expression *lower*(*column*) can be used to efficiently evaluate the query since all such records would map to the same index key.
  Indices on expressions have a higher maintenance cost (i.e., insert and update speed) but they can be useful when retrieval speed is more important.
- **Operator classes**: The specific comparison functions used to build, maintain, and use an index on a column are tied to the data type of that column. Each data type has associated with it a default *operator class* that identifies the actual operators that would normally be used for the data type. While the default operator class is sufficient for most uses, some data types might possess multiple “meaningful”

classes. For instance, in dealing with complex numbers, it might be desirable to sort a complex-number data type either by absolute value or by real part. In this case, two operator classes can be defined for the data type, and one of the operator classes can be chosen when creating the index.

- **Partial indices**: These are indices built over a subset of a table defined by a predicate. The index contains only entries for tuples that satisfy the predicate. An example of the use of a partial indices would be a case where a column contains a large number of occurrences of some of values. Such common values are not worth indexing, since index scans are not beneficial for queries that retrieve a large subset of the base table. A partial index can be built using a predicate that excludes the common values. Such an index would be smaller and would incur less storage and I/O cost than a full index. Such partial indices are also less expensive to maintain, since a large fraction of inserts and deletes will not affect the the index.

32.3.2.3 Index Construction

An index can be created using the **create index** command. For example, the following DDL statement creates a B-tree index on instructor salaries.

create index *inst_sal_idx* **on** *instructor* (*salary*);

This statement is executed by scanning the *instructor* relation to find row versions that might be visible to a future transaction, then sorting their index attributes and building the index structure. During this process, the building transaction holds a lock on the *instructor* relation that prevents concurrent **insert**, **delete**, and **update** statements. Once the process is finished, the index is ready to use and the table lock is released.

The lock acquired by the **create index** command may present a major inconvenience for some applications where it is difficult to suspend updates while the index is built. For these cases, PostgreSQL provides the **create index concurrently** variant, which allows concurrent updates during index construction. This is achieved by a more complex construction algorithm that scans the base table twice. The first table scan builds an initial version of the index, in a way similar to normal index construction described above. This index may be missing tuples if the table was concurrently updated; however, the index is well formed, so it is flagged as being ready for insertions. Finally, the algorithm scans the table a second time and inserts all tuples it finds that still need to be indexed. This scan may also miss concurrently updated tuples, but the algorithm synchronizes with other transactions to guarantee that tuples that are updated during the second scan will be added to the index by the updating transaction. Hence, the index is ready to use after the second table scan. Since this two-pass approach can be expensive, the plain **create index** command is preferred if it is easy to suspend table updates temporarily.

32.3.2.4 Index-Only Scans

An index-only scan allows for processing a query using only an index, without access the table records. Traditionally, indices in PostgreSQL are secondary indices, meaning that each index is stored separately from the table's main data (heap). In the normal index scan scenario, PostgreSQL initially uses the index to locate qualifying tuples and then accesses the heap to retrieve the tuples. Accessing the heap via an index scan might result in a large number of random accesses, since the tuples to be accessed might be anywhere in the heap; index scans can potentially have a high query execution time if a large number of records are retrieved. In contrast, an index-only scan can allow the query to be executed without fetching the tuples, provided the query accesses only index attributes. This can significantly decrease the number I/O operations required, and improve performance correspondingly.

To apply an index-only scan plan during query execution, the query must reference only attributes already stored in the index and the index should support index-only scans. As of PostgreSQL 11, index-only scans are supported with B-trees and some operator classes of GiST and SP-GiST indices. In practice, the index must physically store, or else be able to reconstruct, the original data value for each index entry. Thus, for example, a GIN index cannot support index-only scan since each index entry typically holds only part of the original data value.

In PostgreSQL, using an index-only scan plan does not guarantee that no heap pages will be accessed, due to the presence of multiple tuple versions in PostgreSQL; an index may contain entries for tuple versions that should not be visible to the transaction performing the scan. To check if the tuple is visible, PostgreSQL normally performs a heap page access, to find timestamp information for the tuple. However, PostgreSQL provides a clever optimization that can avoid heap page access in many cases. For each heap relation PostgreSQL maintains a visibility map. The visibility map tracks the pages that contain only tuples that are visible to all active transactions and therefore they do not contain any tuples that need to be vacuumed. Visibility information is stored only in heap entries and not in index entries. As a result, accessing a tuple using an index-only scan will require a heap access if the tuple has been recently modified. Otherwise the heap access can be avoided by checking the visibility map bit for the corresponding heap page. If the bit is set, the tuple is visible to all transactions, and so the data can be returned from the index. On the other hand, if the bit is not set, the heap will be accessed and the performance for this retrieval will be similar to a traditional index access.

The visibility map bit is set during vacuum, and reset whenever a tuple in the heap page is updated. Overall, the more the heap pages that have their all-visible map bits set the higher the performance benefit from an index-only scan. The visibility map is much smaller than the heap file since it requires only two bits per page; thus, very little I/O is required to access it.

In PostgreSQL index-only scans can also be performed on covering indices, which can store attributes other than the index key. Index-only scans on the covering index can

allow efficient sequential access to tuples in the key order, avoiding expensive random access that would be otherwise required by a secondary-index based access. An index-only scan can be used provided all attributes required by the query are contained in the index key or in the covering attributes.

32.3.3 Partitioning

Table partitioning in PostgreSQL allows a table to be split into smaller physical pieces, based on the value of partitioning attributes. Partitioning can be quite beneficial in certain scenarios; for example, it can improve query performance when the query includes predicates on the partitioning attributes, and the matching tuples are in a single partition or a small number of partitions. Table partitioning can also reduce the overhead of bulk loading and deletion in some cases by adding or removing partitions without modifying existing partitions. Partitioning can also make maintenance operations such as VACUUM and REINDEX faster. Further, indices on the partitions are smaller than a index on the whole table, thus it is more likely to fit into memory. Partitioning a relation is a good idea as long as most queries that access the relation include predicates on the partitioning attributes. Otherwise, the overhead of accessing multiple partitions can slow down query processing to some extent.

As of version 11, PostgreSQL comes with three types of partitioning tables:

1. **Range Partitioning**: The table is partitioned into ranges (e.g., date ranges) defined by a key column or set of columns. The range of values in each partition is assigned based on some partitioning expression. The ranges should be contiguous and non-overlapping.
2. **List Partitioning**: The table is partitioned by explicitly listing the set of discrete values that should appear in each partition.
3. **Hash Partitioning**: The tuples are distributed across different partitions according to a hash partition key. Hash partitioning is ideal for scenarios in which there is no natural partitioning key or details about data distribution.

Partitioning in PostgreSQL can be implemented manually using table inheritance. However, a simpler way of implementing partitioning is through the declarative partitioning feature of PostgreSQL. Declarative partitioning allows a user to create a partitioned table by specifying the partitioning type and the list of columns or expressions to be used as the partition key. For example, consider the *takes* relation; a clause

partition by range(*year*)

can be added at the end of the **create table** specification for the *takes* relation to specify that the relation should be partitioned by the *year* attribute.

Then, one or more partitions must be created, with each partition specifying the bounds that correspond to the partitioning method and partition key of the parent, as illustrated below:

create table *takes_till_2017* **partition of** *takes* **for values from** (1900) to (2017);
create table *takes_2017* **partition of** *takes* **for values from** (2017) to (2018);
create table *takes_2018* **partition of** *takes* **for values from** (2018) to (2019);
create table *takes_2019* **partition of** *takes* **for values from** (2019) to (2020);
create table *takes_from_2020* **partition of** *takes* **for values from** (2020) to (2100);

New tuples are routed to the proper partitions according to the selected partition key. Partition key ranges must not overlap, and there must be a partition defined for each valid key value. The query planner of PostgreSQL can exploit the partitioning information to eliminate unnecessary partition accesses during query processing.

Each partition as above is a normal PostgreSQL table, and it is possible to specify a tablespace and storage parameters for each partition separately. Partitions may have their own indexes, constraints and default values, distinct from those of other partitions of the same table. However, there is no support for foreign keys referencing partitioned tables, or for exclusion constraints[3] spanning all partitions.

Turning a table into a partitioned table or vice versa is not supported; however, it is possible to add a regular or partitioned table containing data as a partition of a partitioned table, or remove a partition from a partitioned table turning it into a standalone table.

32.4 Query Processing and Optimization

When PostgreSQL receives a query, the query is first parsed into an internal representation, which goes through a series of transformations, resulting in a query plan that is used by the **executor** to process the query.

32.4.1 Query Rewrite

The first stage of a query's transformation is the **rewrite** stage, which is responsible for implementing the PostgreSQL **rules** system. In PostgreSQL, users can create **rules** that are fired on different query structures such as **update**, **delete**, **insert**, and **select** statements. A view is implemented by the system by converting a view definition into a **select** rule. When a query involving a **select** statement on the view is received, the **select** rule for the view is fired, and the query is rewritten using the definition of the view.

A rule is registered in the system using the **create rule** command, at which point information on the rule is stored in the catalog. This catalog is then used during query rewrite to uncover all candidate rules for a given query.

[3] **Exclusion constraints** in PostgreSQL allow a constraint on each row that can involve other rows; for example, such a constraint can specify that there is no other row with the same key value, or there is no other row with an overlapping range. Efficient implementation of exclusion constraints requires the availability of appropriate indices. See the PostgreSQL manuals for more details.

The rewrite phase first deals with all **update**, **delete**, and **insert** statements by firing all appropriate rules. Such statements might be complicated and contain **select** clauses. Subsequently, all the remaining rules involving only **select** statements are fired. Since the firing of a rule may cause the query to be rewritten to a form that may require another rule to be fired, the rules are repeatedly checked on each form of the rewritten query until a fixed point is reached and no more rules need to be fired.

32.4.2 Query Planning and Optimization

Once the query has been rewritten, it is subject to the planning and optimization phase. Here, each query block is treated in isolation and a plan is generated for it. This planning begins bottom-up from the rewritten query's innermost subquery, proceeding to its outermost query block.

The optimizer in PostgreSQL is, for the most part, cost based. The idea is to generate an access plan whose estimated cost is minimal. The cost model includes as parameters the I/O cost of sequential and random page fetches, as well as the CPU costs of processing heap tuples, index tuples, and simple predicates.

Optimization of a query can be done using one of two approaches:

- **Standard planner**: The standard planner uses the the bottom-up dynamic programming algorithm for join order optimization, which we saw earlier in Section 16.4.1, which is often referred to as the System R optimization algorithm.
- **Genetic query optimizer**: When the number of tables in a query block is very large, System R's dynamic programming algorithm becomes very expensive. Unlike other commercial systems that default to greedy or rule-based techniques, PostgreSQL uses a more radical approach: a genetic algorithm that was developed initially to solve traveling-salesman problems. There exists anecdotal evidence of the successful use of genetic query optimization in production systems for queries with around 45 tables.

Since the planner operates in a bottom-up fashion on query blocks, it is able to perform certain transformations on the query plan as it is being built. One example is the common subquery-to-join transformation that is present in many commercial systems (usually implemented in the rewrite phase). When PostgreSQL encounters a noncorrelated subquery (such as one caused by a query on a view), it is generally possible to pull up the planned subquery and merge it into the upper-level query block. PostgreSQL is able to decorrelate many classes of correlated subqueries, but there are other classes of queries that it is not able to decorrelate. (Decorrelation is described in more detail in Section 16.4.4.)

The query optimization phase results in a *query plan*, which is a tree of relational operators. Each operator represents a specific operation on one or more sets of tuples. The operators can be unary (for example, sort, aggregation), binary (for example, nested-loop join), or *n*-ary (for example, set union).

Crucial to the cost model is an accurate estimate of the total number of tuples that will be processed at each operator in the plan. These estimates are inferred by the optimizer on the basis of statistics that are maintained on each relation in the system. These statistics include the total number of tuples for each relation and average tuple size. PostgreSQL also maintains statistics about each column of a relation, such as the column cardinality (that is, the number of distinct values in the column), a list of most common values in the table and the number of occurrences of each common value, and a histogram that divides the column's values into groups of equal population. In addition, PostgreSQL also maintains a statistical correlation between the physical and logical row orderings of a column's values—this indicates the cost of an index scan to retrieve tuples that pass predicates on the column. The DBA must ensure that these statistics are current by running the **analyze** command periodically.

32.4.3 Query Executor

The executor module is responsible for processing a query plan produced by the optimizer. The executor is based on the demand-driven pipeline model (described in Section 15.7.2.1), where each operator implements the *iterator* interface with a set of four functions: `open()`, `next()`, `rescan()`, and `close()`). PostgreSQL iterators have an extra function, `rescan()`, which is used to reset a subplan (say for an inner loop of a join) with new values for parameters such as index keys. Some of the important categories of operators are as follows:

1. **Access methods**: Access methods are operators that are used to retrieve data from disk, and include sequential scans of heap files, index scans, and bitmap index scans.
    - **Sequential scans**: The tuples of a relation are scanned sequentially from the first to last blocks of the file. Each tuple is returned to the caller only if it is "visible" according to the transaction isolation rules (Section 32.5.1.1).
    - **Index scans**: Given a search condition such as a range or equality predicate, an index scan returns a set of matching tuples from the associated heap file. In a typical case, the operator processes one tuple at a time, starting by reading an entry from the index and then fetching the corresponding tuple from the heap file. This can result in a random page fetch for each tuple in the worst case. The cost of accessing the heap file can be alleviated if an index-only scan is used that allows for retrieving data directly from the index (see Section 32.3.2.4 for more details).
    - **Bitmap index scans**: A bitmap index scan reduces the danger of excessive random page fetches in index scans. To do so, processing of tuples is done in two phases.

a. The first phase reads all index entries and populates a bitmap that contains one bit per heap page; the tuple ID retrieved from the index scan is used to set the bit of the corresponding page.

b. The second phase fetches heap pages whose bit is set, scanning the bitmap in sequential order. This guarantees that each heap page is accessed only once, and increases the chance of sequential page fetches. Once a heap page is fetched, the index predicate is rechecked on all the tuples in the page, since a page whose bit is set may well contain tuples that do not satisfy the index predicate.

Moreover, bitmaps from multiple indexes can be merged and intersected to evaluate complex Boolean predicates before accessing the heap.

2. **Join methods**: PostgreSQL supports three join methods: sorted merge joins, nested-loop joins (including index-nested loop variants for accessing the inner relation using an index), and a hybrid hash join.

3. **Sort**: Small relations are sorted in-memory using quicksort, while larger are sorted using an external sort algorithm. Initially, the input tuples are stored in an unsorted array as long as there is available working memory for the sort operation. If all the tuples fit in memory, the array is sorted using quicksort, and the sorted tuples can be accessed by sequentially scanning the array. Otherwise, the input is divided into sorted runs by using replacement selection; replacement selection uses a priority tree implemented as a heap, and can generate sorted runs that are bigger than the available memory. The sorted runs are stored in temporary files and then merged using a polyphase merge.

4. **Aggregation**: Grouped aggregation in PostgreSQL can be either sort-based or hash-based. When the estimated number of distinct groups is very large, the former is used; otherwise, an in-memory hash-based approach is preferred.

32.4.4 Parallel Query Support

PostgreSQL can generate parallel query plans[4] to leverage multiple CPUs/cores, which can significantly improve query execution performance. Nevertheless, not all queries can benefit from parallel plans, either due to implementation limitations or because the serial query plan is still a better option. The optimizer is responsible for determining whether a parallel plan is the faster execution strategy or not. As of PostgreSQL version 11, only read-only queries can exploit parallel plans. A parallel query plan will not be generated if the query might be suspended during processing (e.g., a PL/pgSQL loop of the form **for** target **in** query **loop** .. **end loop**), if the query uses any function marked as **parallel unsafe**, if the query runs in another parallel query. Further, as of

[4] See https://www.postgresql.org/docs/current/parallel-query.html.

PostgreSQL 11, parallel query plans will not be used if the transaction isolation level is set to serializable, although this may be fixed in future versions.

When a parallel query operator is used, the master backend process coordinates the parallel execution. It is responsible for spawning the required number of workers, executing the non-parallel activity while contributing to parallel execution as one of the workers. The planner determines the number of the background workers that will be used to process the child plan of the *Gather* node.

A parallel query plan includes a *Gather* or *Gather Merge* node which has exactly one child plan. This child plan is the part of the plan that will be executed in parallel. If the root node is *Gather* or *Gather Merge*, then the whole query can be executed in parallel. The master backend executes the *Gather* or *Gather Merge* node. The *Gather* node is responsible for retrieving the tuples generated by the background workers. A *Gather Merge* node is used when the parallel part of the plan produces tuples in sorted order. The background workers and the master backend process communicate through the shared memory area.

PostgreSQL has parallel-aware flavors for the basic query operations. It supports three types of parallel scans; namely, parallel sequential scan, parallel bitmap heap scan and parallel index/index-only scan (only for B-tree indexes). PostgreSQL also supports parallel versions of nested loop, hash and merge joins. In a join operation, at least one of the tables is scanned by multiple background workers. Each background worker additionally scans the inner table of the join and then forwards the computed tuples to the master backend coordinator process. For nested-loop join and merge join the inner side of the join is always non-parallel.

PostgreSQL can also generate parallel plans for aggregation operations. In this case, the aggregation happens in two steps: (a) each background worker produces a partial result for a subset of the data, and (b) the partial results are collected to the master backend process which computes the final result using the partial results generated by the workers.

32.4.5 Triggers and Constraints

In PostgreSQL (unlike some commercial systems) active-database features such as triggers and constraints are not implemented in the rewrite phase. Instead they are implemented as part of the query executor. When the triggers and constraints are registered by the user, the details are associated with the catalog information for each appropriate relation and index. The executor processes an **update**, **delete**, and **insert** statement by repeatedly generating tuple changes for a relation. For each row modification, the executor explicitly identifies, fires, and enforces candidate triggers and constraints, before or after the change as required.

32.4.6 Just-in-Time (JIT) Compilation

The Just-in-Time compilation (JIT) functionality was introduced in PostgreSQL version 11. The physical data independence abstraction supported by relational databases pro-

vides significant flexibility by hiding many low-level details. However, that flexibility can incur a performance hit due to the interpretation overhead (e.g., function calls, unpredictable branches, high number of instructions). For example, computing the qualifying tuples for any arbitrary SQL expression requires evaluating predicates that can use any of the supported SQL datatypes (e.g.,integer,double); the predicate evaluation function must be able to handle all these data types, and also handle sub-expressions containing other operators. The evaluation function is, in effect, an interpreter that processes the execution plan. With JIT compilation, a generic interpreted program can be compiled at query execution time into a native-code program that is tailored for the specific data types used in a particular expression. The resultant compiled code can execute significantly faster than the original interpreted function.

As of PostgreSQL 11, PostgreSQL exploits JIT compilation to accelerate expression evaluation and tuple deforming (which is explained shortly). Those operations were chosen because they are executed very frequently (per tuple) and therefore have a high cost for analytics queries that process large amounts of data. PostgreSQL accelerates expression evaluation (i.e., the code path used to evaluate WHERE clause predicates, expressions in target lists, aggregates and projections) by generating tailored code to each case, depending on the data types of the attributes. Tuple deforming is the process of transforming an on-disk tuple into its in-memory representation. JIT compilation in PostgreSQL creates a transformation function specific to the table layout and the columns to be extracted. JIT compilation may be added for other operations in future releases.

JIT compilation is primarily beneficial for long-running CPU-bound queries. For short-running queries the overhead of performing JIT compilation and optimizations can be higher than the savings in execution time. PostgreSQL selects whether JIT optimizations will be applied during planning, based on whether the estimated query cost is above some threshold.

PostgreSQL uses LLVM to perform JIT compilation; LLVM allows systems to generate a device independent assembly language code, which is then optimized and compiled to machine code specific to the hardware platform. As of PostgreSQL 11, JIT compilation support is not enabled by default, and is used only when PostgreSQL is built using the `–with-llvm` option. The LLVM dependent code is loaded on-demand from a shared library.

32.5 Transaction Management in PostgreSQL

Transaction management in PostgreSQL uses both snapshot isolation (described earlier in Section 18.8), and two-phase locking. Which one of the two protocols is used depends on the type of statement being executed. For DML statements the snapshot isolation technique is used; the snapshot isolation scheme is referred to as the multiversion concurrency control (MVCC) scheme in PostgreSQL. Concurrency control for DDL statements, on the other hand, is based on standard two-phase locking.

32.5.1 Concurrency Control

Since the concurrency control protocol used by PostgreSQL depends on the *isolation level* requested by the application, we begin with an overview of the isolation levels offered by PostgreSQL. We then describe the key ideas behind the MVCC scheme, followed by a discussion of their implementation in PostgreSQL and some of the implications of MVCC. We conclude this section with an overview of locking for DDL statements and a discussion of concurrency control for indices.

32.5.1.1 Isolation Levels

The SQL standard defines three weak levels of consistency, in addition to the serializable level of consistency. The purpose of providing the weak consistency levels is to allow a higher degree of concurrency for applications that do not require the strong guarantees that serializability provides. Examples of such applications include long-running transactions that collect statistics over the database and whose results do not need to be precise.

The SQL standard defines the different isolation levels in terms of phenomena that violate serializability. The phenomena are called *dirty read*, *nonrepeatable read*, *phantom read*, and *serialization anomaly* and are defined as follows:

- **Dirty read.** The transaction reads values written by another transaction that has not committed yet.
- **Nonrepeatable read.** A transaction reads the same object twice during execution and finds a different value the second time, although the transaction has not changed the value in the meantime.
- **Phantom read.** A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed as a result of another recently committed transaction.
- **Serialization anomaly.** A successfully committed group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

Each of the above phenomena violates transaction isolation, and hence violates serializability. Figure 32.3 shows the definition of the four SQL isolation levels specified in the SQL standard—read uncommitted, read committed, repeatable read, and serializable—in terms of these phenomena. In PostgreSQL the user can select any of the four transaction isolation levels (using the command **set transaction**); however, PostgreSQL implements only three distinct isolation levels. A request to set transaction isolation level to read uncommitted is treated the same as a request to set the isolation level to read committed. The default isolation level is read committed.

Isolated level	*Dirty Read*	*Non repeatable Read*	*Phantom Read*
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeated Read	No	No	Maybe
Serializable	No	No	No

Figure 32.3 Definition of the four standard SQL isolation levels.

32.5.1.2 Concurrency Control for DML Commands

The MVCC scheme used in PostgreSQL is an implementation of the snapshot isolation protocol, which was described earlier in Section 18.8. The key idea behind MVCC is to maintain different versions of each row that correspond to instances of the row at different points in time. This allows a transaction to see a consistent **snapshot** of the data, by selecting the most recent version of each row that was committed before taking the snapshot. The MVCC protocol uses snapshots to ensure that every transaction sees a consistent view of the database: before executing a command, the transaction chooses a snapshot of the data and processes the row versions that are either in the snapshot or created by earlier commands of the same transaction. This view of the data is transaction-consistent, since the snapshot includes only committed transactions, but the snapshot is not necessarily equal to the current state of the data.

The motivation for using MVCC is that, unlike with locking, readers never block writers, and writers never block readers. Readers access the version of a row that is part of the transaction's snapshot. Writers create their own separate copy of the row to be updated. Section 32.5.1.3 shows that the only conflict that causes a transaction to be blocked arises if two writers try to update the same row. In contrast, under the standard two-phase locking approach, both readers and writers might be blocked, since there is only one version of each data object and both read and write operations are required to obtain a lock before accessing any data.

The basic snapshot isolation protocol has several benefits over locking, but unfortunately does not guarantee serializability, as we saw earlier in Section 18.8.3. The Serializable Snapshot Isolation (SSI), which provides the benefits of snapshot isolation, while also guaranteeing serializability, was introduced in PostgreSQL 9.1. The key ideas behind SSI are summarized in Section 18.8.3, and more details may be found in the references in bibliographic notes at the end of the chapter.

SSI retains many of the performance benefits of snapshot isolation and at the same time guarantees true serializability. SSI runs transactions using snapshot isolation, checks for conflicts between concurrent transactions at runtime, and aborts transactions when anomalies might happen.

32.5.1.3 Implementation of MVCC

At the core of PostgreSQL MVCC is the notion of *tuple visibility*. A PostgreSQL tuple refers to a version of a row. Tuple visibility defines which of the potentially many ver-

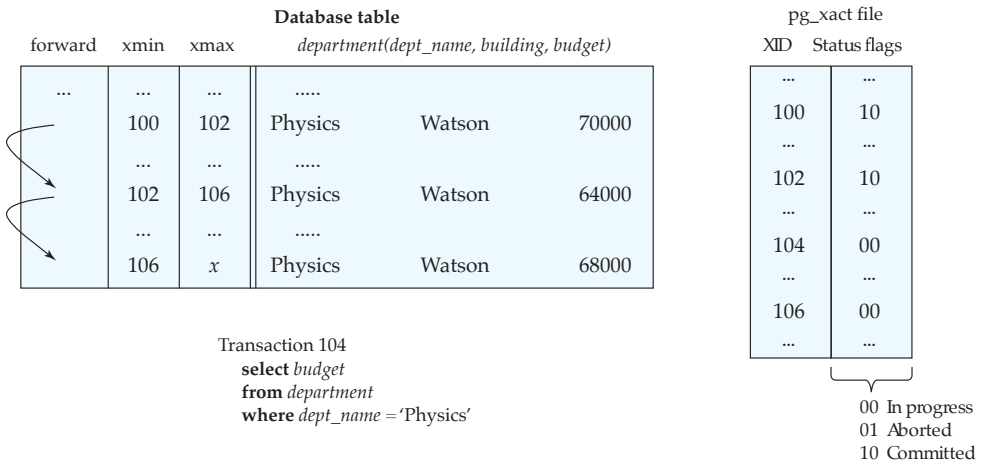

Figure 32.4 The PostgreSQL data structures used for MVCC.

sions of a row in a table is valid within the context of a given statement or transaction. A transaction determines tuple visibility based on a database snapshot that is chosen before executing a command.

A tuple is visible for a transaction T if the following two conditions hold:

1. The tuple was created by a transaction that committed before transaction T took its snapshot.
2. Updates to the tuple (if any) were executed by a transaction that is either
   - aborted, *or*
   - started running after T took its snapshot, *or*
   - was active when T took its snapshot.

To be precise, a tuple is also visible to T if it was created by T and not subsequently updated by T. We omit the details of this special case for simplicity.

The goal of the above conditions is to ensure that each transaction sees a consistent view of the data. PostgreSQL maintains the following state information to check these conditions efficiently:

- A *transaction ID*, which serves as a start timestamp, is assigned to every transaction at transaction start time. PostgreSQL uses a logical counter for assigning transaction IDs.
- A log file called *pg_xact* contains the current status of each transaction. The status can be either in progress, committed, or aborted.

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 32.5 The *department* relation.

- Each tuple in a table has a header with three fields: *xmin*, which contains the transaction ID of the transaction that created the tuple and which is therefore also called the *creation-transaction ID*; *xmax*, which contains the transaction ID of the replacing/deleting transaction (or *null* if not deleted/replaced) and which is also referred to as the *expire-transaction ID*; and a forward link to new versions of the same logical row, if there are any.
- A *SnapshotData* data structure is created either at transaction start time or at query start time, depending on the isolation level (described in more detail below). Its main purpose is to decide whether a tuple is visible to the current command. The *SnapshotData* stores information about the state of transactions at the time it is created, which includes a list of active transactions and *xmax*, a value equal to 1 + the highest ID of any transaction that has started so far. The value *xmax* serves as a "cutoff" for transactions that may be considered visible.

Figure 32.4 illustrates some of this state information through a simple example involving a database with only one table, the *department* table from Figure 32.5. The *department* table has three columns, the name of the department, the building where the department is located, and the budget of the department. Figure 32.4 shows a fragment of the *department* table containing only the (versions of) the row corresponding to the Physics department. The tuple headers indicate that the row was originally created by transaction 100, and later updated by transaction 102 and transaction 106. Figure 32.4 also shows a fragment of the corresponding *pg_xact* file. On the basis of the *pg_xact* file, transactions 100 and 102 are committed, while transactions 104 and 106 are in progress.

Given the above state information, the two conditions that need to be satisfied for a tuple to be visible can be rewritten as follows:

1. The creation-transaction ID in the tuple header
    a. is a committed transaction according to the *pg_xact* file, *and*

   b. is less than the cutoff transaction ID *xmax* recorded by *SnapshotData*, *and*
   c. is not one of the active transactions stored in *SnapshotData*.

2. The expire-transaction ID, if it exists,
   a. is an aborted transaction according to the *pg_xact* file, *or*
   b. is greater than or equal to the cutoff transaction ID *xmax* recorded by *SnapshotData*, *or*
   c. is one of the active transactions stored in *SnapshotData*.

Consider the example database in Figure 32.4 and assume that the *SnapshotData* used by transaction 104 simply uses 103 as the cutoff transaction ID *xmax* and does not show any earlier transactions to be active. In this case, the only version of the row corresponding to the Physics department that is visible to transaction 104 is the second version in the table, created by transaction 102. The first version, created by transaction 100, is not visible, since it violates condition 2: The expire-transaction ID of this tuple is 102, which corresponds to a transaction that is not aborted and that has a transaction ID less than or equal to 103. The third version of the Physics tuple is not visible, since it was created by transaction 106, which has a transaction ID larger than transaction 103, implying that this version had not been committed at the time *SnapshotData* was created. Moreover, transaction 106 is still in progress, which violates another one of the conditions. The second version of the row meets all the conditions for tuple visibility.

The details of how PostgreSQL MVCC interacts with the execution of SQL statements depends on whether the statement is an **insert**, **select**, **update**, or **delete** statement. The simplest case is an **insert** statement, which may simply create a new tuple based on the data in the statement, initialize the tuple header (the creation ID), and insert the new tuple into the table. Unlike two-phase locking, this does not require any interaction with the concurrency-control protocol unless the insertion needs to be checked for integrity conditions, such as uniqueness or foreign key constraints.

When the system executes a **select**, **update**, or **delete** statement the interaction with the MVCC protocol depends on the isolation level specified by the application. If the isolation level is read committed, the processing of a new statement begins with creating a new *SnapshotData* data structure (independent of whether the statement starts a new transaction or is part of an existing transaction). Next, the system identifies *target tuples*, that is, the tuples that are visible with respect to the *SnapshotData* and that match the search criteria of the statement. In the case of a **select** statement, the set of target tuples make up the result of the query.

In the case of an **update** or **delete** statement in read committed mode, the snapshot isolation protocol used by PostgreSQL requires an extra step after identifying the target tuples and before the actual update or delete operation can take place. The reason is that visibility of a tuple ensures only that the tuple has been created by a transaction that committed before the **update**/**delete** statement in question started. However, it is possi-

ble that, since query start, this tuple has been updated or deleted by another concurrently executing transaction. This can be detected by looking at the expire-transaction ID of the tuple. If the expire-transaction ID corresponds to a transaction that is still in progress, it is necessary to wait for the completion of this transaction first. If the transaction aborts, the **update** or **delete** statement can proceed and perform the actual modification. If the transaction commits, the search criteria of the statement need to be evaluated again, and only if the tuple still meets these criteria can the row be modified. If the row is to be deleted, the main step is to update the expire-transaction ID of the old tuple. A row update also performs this step, and additionally creates a new version of the row, sets its creation-transaction ID, and sets the forward link of the old tuple to reference the new tuple.

Going back to the example from Figure 32.4, transaction 104, which consists of a **select** statement only, identifies the second version of the Physics row as a target tuple and returns it immediately. If transaction 104 were an update statement instead, for example, trying to increment the budget of the Physics department by some amount, it would have to wait for transaction 106 to complete. It would then re-evaluate the search condition and, only if it is still met, proceed with its update.

Using the protocol described above for **update** and **delete** statements provides only the read-committed isolation level. Serializability can be violated in several ways. First, nonrepeatable reads are possible. Since each query within a transaction may see a different snapshot of the database, a query in a transaction might see the effects of an **update** command completed in the meantime that were not visible to earlier queries within the same transaction. Following the same line of thought, phantom reads are possible when a relation is modified between queries.

In order to provide the PostgreSQL serializable isolation level, PostgreSQL MVCC eliminates violations of serializability in two ways: First, when it is determining tuple visibility, all queries within a transaction use a snapshot as of the start of the transaction, rather than the start of the individual query. This way successive queries within a transaction always see the same data.

Second, the way updates and deletes are processed is different in serializable mode compared to read-committed mode. As in read-committed mode, transactions wait after identifying a visible target row that meets the search condition and is currently updated or deleted by another concurrent transaction. If the concurrent transaction that executes the update or delete aborts, the waiting transaction can proceed with its own update. However, if the concurrent transaction commits, there is no way for PostgreSQL to ensure serializability for the waiting transaction. Therefore, the waiting transaction is rolled back and returns the following error message: “could not serialize access due to read/write dependencies among transactions“. It is up to the application to handle an error message like the above appropriately, by aborting the current transaction and restarting the entire transaction from the beginning.

Further, to ensure serializability, the serializable snapshot-isolation technique (which is used when the isolation level is set to serializable) tracks read-write conflicts

between transactions, and forces rollback of transactions whenever certain patterns of conflicts are detected.

Further, to guarantee serializability, the phantom-phenomenon (Section 18.4.3) must be avoided; the problem occurs when a transaction reads a set of tuples satisfying a predicate, and a concurrent transaction performs an update that creates a new tuple satisfying the predicate, or updates a tuple in a way that results in the tuple satisfying the predicate, when it did not do so earlier.

To avoid the phantom phenomenon, the SSI implementation in PostgreSQL uses predicate locking, using ideas from the index locking technique described in Section 18.4.3, but with modifications to work correctly with SSI. Predicate locking helps to detect when a write might have an impact on the result of a predicate read by a concurrent transaction. These locks do not cause any blocking and therefore can not cause a deadlock. They are used to identify and flag dependencies among concurrent serializable transactions which in certain combinations can lead to serialization anomalies. Predicate locks show up in the pg_locks system view with a mode of SIReadLock. The particular locks acquired during execution of a query depend on the plan used by the query. A read-only transaction may be able to release its SIRead lock before completion, if it detects that no conflicts can still occur which could lead to a serialization anomaly. On the other hand, SIRead locks may need to be kept past transaction commit, until overlapping read write transactions complete.

32.5.1.4 Implications of Using MVCC

Using the PostgreSQL MVCC scheme has implications in three different areas:

1. An extra burden is placed on the storage system, since it needs to maintain different versions of tuples.
2. The development of concurrent applications takes some extra care, since PostgreSQL MVCC can lead to subtle, but important, differences in how concurrent transactions behave, compared to systems where standard two-phase locking is used.
3. The performance of MVCC depends on the characteristics of the workload running on it.

The implications of MVCC in PostgreSQL are described in more detail below.

Creating and storing multiple versions of every row can lead to excessive storage consumption. The space occupied by a version cannot be freed by the transaction that deletes the tuple or creates a new version. Instead, a tuple version can be freed only later, after checking that the tuple version cannot be visible to any active or future transactions. The task of freeing space is nontrivial, because indices may refer to the location of an old tuple version, so these references need to be deleted before reusing the space.

In general, versions of tuples are freed up by the vacuum process of PostgreSQL. The vacuum process can be initiated by a command, but PostgreSQL employs a background process to **vacuum** tables automatically. The vacuum process first scans the heap, and whenever it finds a tuple version that cannot be accessed by any current/future transaction, it marks the tuple as "dead". The vacuum process then scans all indices of the relation, and removes any entries that point to dead tuples. Finally, it rescans the heap, physically deleting tuple versions that were marked as dead earlier.

PostgreSQL also supports a more aggressive form of tuple reclaiming in cases where the creation of a version does not affect the attributes used in indices, and further the old and new tuple versions are on the same page. In this case no index entry is created for the new tuple version, but instead a link is added from the old tuple version in the heap page to the new tuple version (which is also on the same heap page). An index lookup will first find the old version, and if it determines that the version is not visible to the transaction, the version chain is followed to find the appropriate version. When the old version is no longer visible to any transaction, the space for the old version can be reclaimed in the heap page by some clever data structure tricks within the page, without touching the index.

The **vacuum** command offers two modes of operation: Plain **vacuum** simply identifies tuples that are not needed, and makes their space available for reuse. This form of the command executes as described above, and can operate in parallel with normal reading and writing of the table. **Vacuum full** does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

PostgreSQL's approach to concurrency control performs best for workloads containing many more reads than updates, since in this case there is a very low chance that two updates will conflict and force a transaction to roll back. Two-phase locking may be more efficient for some update-intensive workloads, but this depends on many factors, such as the length of transactions and the frequency of deadlocks.

32.5.1.5 DDL Concurrency Control

The MVCC mechanisms described in the previous section do not protect transactions against operations that affect entire tables, for example, transactions that drop a table or change the schema of a table. Toward this end, PostgreSQL provides explicit locks that DDL commands are required to acquire before starting their execution. These locks are always table based (rather than row based) and are acquired and released in accordance with the strict two-phase locking protocol.

Figure 32.6 lists all types of locks offered by PostgreSQL, which locks they conflict with, and some commands that use them (the **create index concurrently** command is covered in Section 32.3.2.3). The names of the lock types are often historical and do not necessarily reflect the use of the lock. For example, all the locks are table-level locks, although some contain the word "row" in the name. DML commands acquire

Lock name	*Conflicts with*	*Acquired by*
ACCESS SHARE	ACCESS EXCLUSIVE	**select** query
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	**select for update** query **select for share** query
ROW EXCLUSIVE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**update** **delete** **insert** queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**vacuum** **analyze** **create index concurrently**
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**create index**
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	---
EXCLUSIVE	All except ACCESS SHARE	---
ACCESS EXCLUSIVE	All modes	**drop table** **alter table** **vaccum full**

Figure 32.6 Table-level lock modes.

only locks of the first three types. These three lock types are compatible with each other, since MVCC takes care of protecting these operations against each other. DML commands acquire these locks only for protection against DDL commands.

While their main purpose is providing PostgreSQL internal concurrency control for DDL commands, all locks in Figure 32.6 can also be acquired explicitly by PostgreSQL applications through the **lock table** command.

Locks are recorded in a lock table that is implemented as a shared-memory hash table keyed by a signature that identifies the object being locked. If a transaction wants to acquire a lock on an object that is held by another transaction in a conflicting mode, it needs to wait until the lock is released. Lock waits are implemented through semaphores, each of which is associated with a unique transaction. When waiting for a lock, a transaction actually waits on the semaphore associated with the transaction holding the lock. Once the lock holder releases the lock, it will signal the waiting trans-

action(s) through the semaphore. By implementing lock waits on a per-lock-holder basis, rather than on a per-object basis, PostgreSQL requires at most one semaphore per concurrent transaction, rather than one semaphore per lockable object.

Deadlock detection is triggered if a transaction has been waiting for a lock for more than 1 second. The deadlock-detection algorithm constructs a wait-for graph based on the information in the lock table and searches this graph for circular dependencies. If it finds any, meaning a deadlock was detected, the transaction that triggered the deadlock detection aborts and returns an error to the user. If no cycle is detected, the transaction continues waiting on the lock. Unlike some commercial systems, PostgreSQL does not tune the lock time-out parameter dynamically, but it allows the administrator to tune it manually. Ideally, this parameter should be chosen on the order of a transaction lifetime, in order to optimize the trade-off between the time it takes to detect a deadlock and the work wasted for running the deadlock detection algorithm when there is no deadlock.

32.5.1.6 Locking and Indices

Indices in PostgreSQL allow for concurrent access by multiple transactions. B-tree, GIN, GiST and SP-GiST indices use short-term share/exclusive page-level locks for read/write access which are released immediately after each index row is fetched or inserted. On the other hand, hash indices use share/exclusive hash-bucket-level locks for read/write access, and the locks are released after the whole bucket is processed. This might cause deadlock since the locks are held longer than one index operation. Indices that support predicate locking acquire SIRead locks on index pages that are accessed when searching for tuples that satisfy the predicate used in the predicate read.

32.5.2 Recovery

PostgreSQL employs write-ahead log (WAL) based recovery to ensure atomicity and durability. The approach is similar to the standard recovery techniques; however, recovery in PostgreSQL is simplified in some ways because of the MVCC protocol.

Under PostgreSQL, recovery does not have to undo the effects of aborted transactions: an aborting transaction makes an entry in the *pg_xact* file, recording the fact that it is aborting. Consequently, all versions of rows it leaves behind will never be visible to any other transactions. The only case where this approach could potentially lead to problems is when a transaction aborts because of a crash of the corresponding PostgreSQL process and the PostgreSQL process does not have a chance to create the *pg_xact* entry before the crash. PostgreSQL handles this as follows: Before checking the status of a transaction in the *pg_xact* file, PostgreSQL checks whether the transaction is running on any of the PostgreSQL processes. If no PostgreSQL process is currently running the transaction, but the *pg_xact* file shows the transaction as still running, it is safe to assume that the transaction crashed and the transaction's *pg_xact* entry is updated to "aborted".

Additionally, recovery is simplified by the fact that PostgreSQL MVCC already keeps track of some of the information required by write-ahead logging. More precisely, there is no need for logging the start, commit, and abort of transactions, since MVCC logs the status of every transaction in the *pg_xact*.

PostgreSQL provides support for two-phase commit; two-phase commit is described in more detail in Section 23.2.1. The **prepare transaction** command brings a transaction to the prepared-state of two-phase commit by persisting its state on disk. When the coordinator decides on whether the transaction should be committed or aborted, PostgreSQL can do so, even if there is a system crash between reaching the prepared state and the commit/abort decision. Leaving transactions in the prepared state for a long time is not recommended since locks continue to be held by the prepared transaction, affecting concurrency and interfering with the ability of VACUUM to reclaim storage.

32.5.3 High Availability and Replication

PostgreSQL provides support for high availability (HA). To achieve HA, all updates performed on the primary database system must be replicated to a secondary (backup/standby) database system, which is referred to as a replica. In case the primary database fails, the secondary can take over transaction processing.

Updates performed at a primary database can be replicated to more than one secondary database. Reads can then be performed at the replicas, as long as the reader does not mind reading a potentially slightly outdated state of the database.

Replication in a HA cluster is done by log shipping. The primary severs (servers that can modify data) operate in continuous archiving mode while the secondary servers operate in recovery mode, continuously reading the WAL records from the primary. By default, the log is created in units of segments, which are files that are (by default) 16MB in size. PostgreSQL implements file-based log shipping by transferring WAL records one file (WAL segment) at a time. Log shipping comes with low performance overhead on master servers; however, there is a potential windows for data loss. Log shipping is asynchronous, which means that WAL records are shipped after transaction commit. Thus, if the primary server crashes then the transactions not yet shipped will be lost. PostgreSQL has a configuration parameter (archive_timeout) that can help to limit the potential window of data lost in file-based log shipping by forcing the primary server to switch to new WAL segment file periodically.

Streaming replication solution allows for secondary servers to be more up-to-date than the file-based log shipping approach, thereby decreasing the potential window of data loss. In this case, the primary servers stream WAL records to the secondary servers as the WAL records are generated. without having to wait for the WAL file to be filled. Streaming replication is asynchronous by default and thus, it is still possible to experience a delay between the time a transaction is committed on the primary server and the moment is becomes visible in the secondary server. Nevertheless, this window

of potential data loss is shorter (typically below a second) than the one in file-based log shipping.

PostgreSQL can also operate using synchronous replication. In this case, each commit of a write transaction waits until confirmation is received that the commit has been written to the WAL on disk of both the primary and secondary server. Even though this approach increases the confidence that the data of a transaction commit will be available, commit processing is slower. Further, data loss is still possible if both primary and secondary servers crash at the same time. For read-only transactions and transaction rollbacks, there is no need to wait for the response from the secondary servers.

In addition to physical replication, PostgreSQL also supports **logical replication**. Logical replication allows for fine-grained control over data replication by replicating logical data modifications from the WAL based on a replication identity (usually a primary key). Physical replication, on the other hand, is based on exact block addresses and byte-by-byte replication. The logical replication can be enabled by setting the *wal_level* configuration parameter to *logical*.

Logical replication is implemented using a publish and subscribe model in which one or more subscribers subscribing to one or more publications (changes generated from a table or a group of tables). The server responsible for sending the changes is called a publisher while the server that subscribes to the changes is called a subscriber. When logical replication is enabled, the subscriber receives a snapshot of the data on the publisher database. Then, each change that happens on the publisher is identified and sent to the subscriber using streaming replication. The subscriber is responsible for applying the change in the same order as the publisher, to guarantee consistency. Typical use-cases for logical replication include replicating data between different platforms or different major versions of PostgreSQL, sharing a subset of the database between different groups of users, sending incremental changes in a single database, consolidating multiple databases into a single one, among other use-cases.

32.6 SQL Variations and Extensions

The current version of PostgreSQL supports almost all entry-level SQL-92 features, as well as many of the intermediate- and full-level features. It also supports many SQL:1999 and SQL:2003 features, including most object-relational features and the SQL/XML features for parsed XML. In fact, some features of the current SQL standard (such as arrays, functions, and inheritance) were pioneered by PostgreSQL.

32.6.1 PostgreSQL Types

PostgreSQL has support for several nonstandard types, useful for specific application domains. Furthermore, users can define new types with the **create type** command. This includes new low-level base types, typically written in C (see Section 32.6.2.1).

32.6.1.1 The PostgreSQL Type System

PostgreSQL types fall into the following categories:

- **Base types**: Base types are also known as **abstract data types**; that is, modules that encapsulate both state and a set of operations. These are implemented below the SQL level, typically in a language such as C (see Section 32.6.2.1). Examples are **int4** (already included in PostgreSQL) or **complex** (included as an optional extension type). A base type may represent either an individual scalar value or a variable-length array of values. For each scalar type that exists in a database, PostgreSQL automatically creates an array type that holds values of the same scalar type.
- **Composite types**: These correspond to table rows; that is, they are a list of field names and their respective types. A composite type is created implicitly whenever a table is created, but users may also construct them explicitly.
- **Domains**: A domain type is defined by coupling a base type with a constraint that values of the type must satisfy. Values of the domain type and the associated base type may be used interchangeably, provided that the constraint is satisfied. A domain may also have an optional default value, whose meaning is similar to the default value of a table column.
- **Enumerated types**: These are similar to enum types used in programming languages such as C and Java. An enumerated type is essentially a fixed list of named values. In PostgreSQL, enumerated types may be converted to the textual representation of their name, but this conversion must be specified explicitly in some cases to ensure type safety. For instance, values of different enumerated types may not be compared without explicit conversion to compatible types.
- **Pseudotypes**: Currently, PostgreSQL supports the following pseudotypes: *any*, *anyarray*, *anyelement*, *anyenum*, *anynonarray cstring*, *internal*, *opaque*, *language_handler*, *record*, *trigger*, and *void*. These cannot be used in composite types (and thus cannot be used for table columns), but can be used as argument and return types of user-defined functions.
- **Polymorphic types.** Four of the pseudotypes *anyelement*, *anyarray*, *anynonarray*, and *anyenum* are collectively known as **polymorphic**. Functions with arguments of these types (correspondingly called **polymorphic functions**) may operate on any actual type. PostgreSQL has a simple type-resolution scheme that requires that: (1) in any particular invocation of a polymorphic function, all occurrences of a polymorphic type must be bound to the same actual type (that is, a function defined as *f*(*anyelement*, *anyelement*) may operate only on pairs of the same actual type), and (2) if the return type is polymorphic, then at least one of the arguments must be of the same polymorphic type.

32.6.1.2 Nonstandard Types

The types described in this section are included in the standard distribution. Furthermore, thanks to the open nature of PostgreSQL, there are several contributed extension types, such as complex numbers, and ISBN/ISSNs (see Section 32.6.2).

Geometric data types (*point*, *line*, *lseg*, *box*, *polygon*, *path*, *circle*) are used in geographic information systems to represent two-dimensional spatial objects such as points, line segments, polygons, paths, and circles. Numerous functions and operators are available in PostgreSQL to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

Full-text searching is performed in PostgreSQL using the *tsvector* type that represents a document and the *tsquery* type that represents a full-text query. A *tsvector* stores the distinct words in a document, after converting variants of each word to a common normal form (for example, removing word stems). PostgreSQL provides functions to convert raw text to a *tsvector* and concatenate documents. A *tsquery* specifies words to search for in candidate documents, with multiple words connected by Boolean operators. For example, the query `'index & !(tree | hash)'` finds documents that contain "index" without using the words "tree" or "hash." PostgreSQL natively supports operations on full-text types, including language features and indexed search.

PostgreSQL offers data types to store network addresses. These data types allow network-management applications to use a PostgreSQL database as their data store. For those familiar with computer networking, we provide a brief summary of this feature here. Separate types exist for IPv4, IPv6, and Media Access Control (MAC) addresses (*cidr*, *inet* and *macaddr*, respectively). Both *inet* and *cidr* types can store IPv4 and IPv6 addresses, with optional subnet masks. Their main difference is in input/output formatting, as well as the restriction that classless Internet domain routing (CIDR) addresses do not accept values with nonzero bits to the right of the netmask. The *macaddr* type is used to store MAC addresses (typically, Ethernet card hardware addresses). PostgreSQL supports indexing and sorting on these types, as well as a set of operations (including subnet testing, and mapping MAC addresses to hardware manufacturer names). Furthermore, these types offer input-error checking. Thus, they are preferable over plain text fields.

The PostgreSQL *bit* type can store both fixed- and variable-length strings of 1s and 0s. PostgreSQL supports bit-logical operators and string-manipulation functions for these values.

PostgreSQL offers data types to store XML and JSON data. Both XML and JSON data can be stored as text. However, the specialized data types offer additional functionality. For example, the XML data type allows for checking the input values for well-formedness while there are support functions to perform type-safe operations on it. Similarly, the JSON type has the advantage of enforcing that each stored value is valid according to the JSON rules. There are two JSON data types: *json* and *jsonb* and both accept almost identical sets of values as input. However, the *json* data type stores an exact copy of the input text, which processing functions must re-parse on each execu-

tion; while *jsonb* data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process while supporting indexing capabilities.

32.6.2 Extensibility

Like most relational database systems, PostgreSQL stores information about databases, tables, columns, and so forth, in what are commonly known as **system catalogs**, which appear to the user as normal tables. Other relational database systems are typically extended by changing hard-coded procedures in the source code or by loading special extension modules written by the vendor.

Unlike most relational database systems, PostgreSQL goes one step further and stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. Therefore, PostgreSQL makes it easy for users to extend and facilitates rapid prototyping of new applications and storage structures. PostgreSQL can also incorporate user-written code into the server, through dynamic loading of shared objects. This provides an alternative approach to writing extensions that can be used when catalog-based extensions are not sufficient.

Furthermore, the contrib module of the PostgreSQL distribution includes numerous user functions (for example, array iterators, fuzzy string matching, cryptographic functions), base types (for example, encrypted passwords, ISBN/ISSNs, *n*-dimensional cubes) and index extensions (for example, RD-trees,[5] indexing for hierarchical labels). Thanks to the open nature of PostgreSQL, there is a large community of PostgreSQL professionals and enthusiasts who also actively extend PostgreSQL. Extension types are identical in functionality to the built-in types; the latter are simply already linked into the server and preregistered in the system catalog. Similarly, this is the only difference between built-in and extension functions.

32.6.2.1 Types

PostgreSQL allows users to define composite types, enumeration types, and even new base types. A composite-type definition is similar to a table definition (in fact, the latter implicitly does the former). Stand-alone composite types are typically useful for function arguments. For example, the definition:

create type *city_t* **as** (*name* **varchar**(80), *state* **char**(2));

allows functions to accept and return *city_t* tuples, even if there is no table that explicitly contains rows of this type.

[5] RD-trees are designed to index sets of items, and support set containment queries such as finding all sets that contain a given query set.

Adding base types to PostgreSQL is straightforward; an example can be found in complex.sql and complex.c in the tutorials of the PostgreSQL distribution. The base type can be declared in C, for example:

```
typedef struct Complex {
    double x;
    double y;
} Complex;
```

The next step is to define functions to read and write values of the new type in text format. Subsequently, the new type can be registered using the statement:

```
create type complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    alignment = double
);
```

assuming the text I/O functions have been registered as *complex_in* and *complex_out*. The user has the option of defining binary I/O functions as well (for more efficient data dumping). Extension types can be used like the existing base types of PostgreSQL. In fact, their only difference is that the extension types are dynamically loaded and linked into the server. Furthermore, indices may be extended easily to handle new base types; see Section 32.6.2.3.

32.6.2.2 Functions

PostgreSQL allows users to define functions that are stored and executed on the server. PostgreSQL also supports function overloading (that is, functions may be declared by using the same name but with arguments of different types). Functions can be written as plain SQL statements, or in several procedural languages (covered in Section 32.6.2.4). Finally, PostgreSQL has an application programmer interface for adding functions written in C (explained in this section).

User-defined functions can be written in C (or a language with compatible calling conventions, such as C++). The actual coding conventions are essentially the same for dynamically loaded, user-defined functions, as well as for internal functions (which are statically linked into the server). Hence, the standard internal function library is a rich source of coding examples for user-defined C functions. Once the shared library containing the function has been created, a declaration such as the following registers it on the server:

```
create function complex_out(complex)
    returns cstring
    as 'shared_object_filename'
    language C immutable strict;
```

The entry point to the shared object file is assumed to be the same as the SQL function name (here, *complex_out*), unless otherwise specified.

The example here continues the one from Section 32.6.2.1. The application program interface hides most of PostgreSQL's internal details. Hence, the actual C code for the above text output function of *complex* values is quite simple:

```
PG_FUNCTION_INFO_V1(complex_out);
Datum complex_out(PG_FUNCTION_ARGS) {
    Complex *complex = (Complex *) PG_GETARG_POINTER(0);
    char *result;
    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

The first line declares the function complex_out, and the following lines implement the output function. The code uses several PostgreSQL-specific constructs, such as the palloc() function, which dynamically allocates memory controlled by PostgreSQL's memory manager.

Aggregate functions in PostgreSQL operate by updating a **state value** via a **state transition** function that is called for each tuple value in the aggregation group. For example, the state for the **avg** operator consists of the running sum and the count of values. As each tuple arrives, the transition function simply add its value to the running sum and increment the count by one. Optionally, a *final* function may be called to compute the return value based on the state information. For example, the final function for **avg** would simply divide the running sum by the count and return it.

Thus, defining a new aggregate function (referred to a **user-defined aggregate function**) is a simple as defining these functions. For the *complex* type example, if *complex_add* is a user-defined function that takes two complex arguments and returns their sum, then the **sum** aggregate operator can be extended to complex numbers using the simple declaration:

```
create aggregate sum (complex) (
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

Note the use of function overloading: PostgreSQL will call the appropriate *sum* aggregate function, on the basis of the actual type of its argument during invocation. The *stype* is the state value type. In this case, a final function is unnecessary, since the return value is the state value itself (that is, the running sum in both cases).

User-defined functions can also be invoked by using operator syntax. Beyond simple "syntactic sugar" for function invocation, operator declarations can also provide hints to the query optimizer in order to improve performance. These hints may include information about commutativity, restriction and join selectivity estimation, and various other properties related to join algorithms.

32.6.2.3 Index Extensions

The indices supported by PostgreSQL can be extended to accommodate new base types. Adding index extensions for a type requires the definition of an **operator class**, which encapsulates the following:

- **Index-method strategies**: These are a set of operators that can be used as qualifiers in **where** clauses. The particular set depends on the index type. For example, B-tree indices can retrieve ranges of objects, so the set consists of five operators (<, <=, =, >=, and >), all of which can appear in a **where** clause involving a B-tree index while a hash index allows only equality testing.
- **Index-method support routines**: The above set of operators is typically not sufficient for the operation of the index. For example, a hash index requires a function to compute the hash value for each object.

For example, if the following functions and operators are defined to compare the magnitude of *complex* numbers (see Section 32.6.2.1), then we can make such objects indexable by the following declaration:

```
create operator class complex_abs_ops
    default for type complex using btree as
            operator 1 < (complex, complex),
            operator 2 <= (complex, complex),
            operator 3 = (complex, complex),
            operator 4 >= (complex, complex),
            operator 5 > (complex, complex),
            function 1 complex_abs_cmp(complex, complex);
```

The **operator** statements define the strategy methods and the **function** statements define the support methods.

32.6.2.4 Procedural Languages

Stored functions and procedures can be written in a number of procedural languages. Furthermore, PostgreSQL defines an application programmer interface for hooking up any programming language for this purpose. Programming languages can be registered on demand and are either **trusted** or **untrusted**. The latter allow unlimited access to the DBMS and the file system, and writing stored functions in them requires superuser privileges.

- **PL/pgSQL**. This is a trusted language that adds procedural programming capabilities (for example, variables and control flow) to SQL. It is very similar to Oracle's PL/SQL. Although code cannot be transferred verbatim from one to the other, porting is usually simple.
- **PL/Tcl**, **PL/Perl**, and **PL/Python**. These leverage the power of Tcl, Perl, and Python to write stored functions and procedures on the server. The first two come in both trusted and untrusted versions (PL/Tcl, PL/Perl and PL/TclU, PL/PerlU, respectively), while PL/Python is untrusted at the time of this writing. Each of these has bindings that allow access to the database system via a language-specific interface.

32.7 Foreign Data Wrappers

Foreign data wrappers (FDW) allow a user to connect with external data sources to transparently query data that reside outside of PostgreSQL, as if the data were part of an existing table in a database. PostgreSQL implements FDWs to provide SQL/MED ("Management of External Data") functionality. SQL/MED is an extension of the ANSI SQL standard specification that defines types that allow a database management system to access external data. FDWs can be a powerful tool both for data migration and data analysis scenarios.

Today, there are a number of FDWs that enable PostgreSQL to access different remote stores, such as other relational databases supporting SQL, key-value (NoSQL) sources, and flat files; however, most of them are implemented as PostgreSQL extensions and are not officially supported. PostgreSQL provides two FDWs modules:

- file_fdw: The *file_fdw* module allow users to create foreign tables for data files in the server's file system, or to specify commands to be executed on the server and read their output. Access is read-only and the data file or command output should be in a format compatible to the **copy from** command. These include csv files, text files with one row per line, with columns separated by user-specified delimiter character, and a PostgreSQL specific binary format.
- postgres_fdw: The *postgres_fdw* module is used to access remote tables stored in external PostgreSQL servers. Using *postgres_fdw*, foreign tables are updatable as long as the the required privileges are set. When a query references a remote table,

postgres_fdw opens a transaction on the remote server that is committed or aborted when the local transaction commits or aborts. The remote transaction uses *serializable* isolation level when the local transaction has *serializable* isolation level; otherwise it uses *repeatable read* isolation level. This ensures that a query performing multiple table scans on the remote server it will get snapshot-consistent results for all the scans.

Instead of fetching all the required data from the remote database and compute the query locally, *postgres_fdw* tries to reduce the amount of data transferred from foreign servers. Queries are optimized to send the query **where** clauses that use data types, operators, and built-in functions to the remote server for execution and by retrieving only the table columns that are needed for the correct query execution. Similarly, when a join operation is performed between foreign tables on the same foreign server, *postgres_fdw* pushes down the join operation to the remote server and retrieves only the results, unless the optimizer estimates that it will be more efficient to fetch rows from each table individually.

In a typical usage scenario, the user should go through a number of prerequisites steps before accessing foreign data using the FDWs provided by PostgreSQL. Specifically, the user should a) install the desired FDW extension, b) create a foreign server object to specify connection information for the particular external data source, c) specify the credentials a user should use to access an external data resource, and d) create one or more foreign tables specifying the schema of the external data source to be accessed.

A FDW handles all the operations performed on a foreign table and is responsible for accessing the remote data source and returning it to the PostgreSQL executor. PostgreSQL internally provides a list of APIs that a FDW can implement, depending on the desired functionality. For example, if a FDW intends to support remote foreign joins instead of fetching the tables and performing the join locally, it should implement the `GetForeignJoinPaths()` callback function.

32.8 PostgreSQL Internals for Developers

This section is targeted for developers and researchers who plan to extend the PostgreSQL source code to implement any desired functionality. The section provides pointers on how to install PostgreSQL from source code, navigate the source code, and understand some basic PostgreSQL data structures and concepts, as a first step toward adding new functionality to PostgreSQL. The section pays particular focus on the region-based memory manager of PostgreSQL and the structure of nodes in a query plan, and the key functions that are invoked during the processing of a query. It also explains the organization of tuples and their internal representation in the form of `Datum` data structures used to represent values, and various data structures used to represent tuples. Finally, the section describes error handling mechanisms in PostgreSQL, and

offer advice on steps required when adding a new functionality. This section can also serve as a reference source for key concepts whose understanding is necessary when changing the source code of PostgreSQL. For more development information we encourage the readers to refer to the PostgreSQL development wiki.[6]

32.8.1 Installation From Source Code

We start with a brief walk-through of the steps required to install PostgreSQL from source code, which is a first step for the development of any new functionality. The source code can be obtained from the version control repository at: git.postgresql.org, or downloaded from: https://www.postgresql.org/download/. We describe the basic steps required for installation in this section; detailed installation instructions can be found at: https://www.postgresql.org/docs/current/installation.html.

32.8.1.1 Requirements

The following software packages are required for a successful build:

- An ISO/ANSI C compiler is required to compile the source code. Recent versions of GCC are recommended, but PostgreSQL can be built using a wide variety of compilers from different vendors.
- tar is required to unpack the source distribution, in addition to either gzip or bzip2.
- The GNU Readline library is used by default. An alternative is to disable it by specifying —without-readline option when invoking configure (discussed next). When installing PostgreSQL under a Linux distribution both readline and readline-devel packages are needed.
- The zlib compression library is used by default. This library is used by pg_dump and pg_restore.
- GNU Flex (2.5.31 or later) and Bison(1.875 or later) are needed to build from a Git checkout (which is necessary when doing any development on the server).
- Perl (5.8.3 or later) is required when doing a build from the Git checkout.

32.8.1.2 Installation Steps

Once all required software packages are installed, we can proceed with the PostgreSQL installation. The installation consists of a couple of steps discussed next:

[6] https://wiki.postgresql.org/wiki/Development_information

1. **Configuration**: The first step of the installation procedure is to configure the source tree for a given system, and specify installation options (e.g. the directory of the build). A default configuration can be invoked with:

```
./configure
```

The configure script sets up files for building the server, utilities and all clients applications, by default under /usr/local/pgsql; to specify an alternative directory you should run configure with the command line option —prefix=PATH, where PATH is the directory where you wish to install PostgreSQL.[7]).
In addition to the –prefix option, other frequently used options include –enable-debug, –enable-depend, and –enable-cassert, which enable debugging; it is important to use these options to help you debug code that you create in PostgreSQL. The –enable-debug option enables build with debugging symbols (-g), the –enable-depend option turns on automatic dependency tracking, while the –enable-cassert option enables assertion checks (used for debugging).
Further, it is recommended that you set the environment variable CFLAGS to the value -O0 (the letter "O" followed by a zero) to turn off compiler optimization entirely. This option reduces compilation time and improves debugging information. Thus, the following commands can be used to configure PostgreSQL to support debugging:

```
export CFLAGS=-O0
./configure –prefix=PATH –enable-debug –enable-depend –enable-cassert
```

where PATH is the path for installing the files. The CFLAGS variable can alternatively be set in the command line by adding the option CFLAGS='-O0' to the configure command above.

2. **Build**: To start the build, type either of the following commands:

```
make
make all
```

This will build the core of PostgreSQL. For a complete build that includes the documentation as well as all additional modules (the contrib directory) type:

```
make world
```

3. **Regression Tests**: This is an optional step to verify whether PostgreSQL runs on the current machine in the expected way. To run the regression tests type:

```
make check
```

[7] More details about the command line options of configure can be found at: https://www.postgresql.org/docs/current/install-procedure.html.

4. **Installation**: To install PostgreSQL enter:

```
make install
```

This step will install files into the default directory or the directory specified with the –prefix command line option provided in Step 1.
For a full build (including the documentation and the contribution modules) type:

```
make install-world
```

32.8.1.3 Creating a Database

Once PostgreSQL is installed, we will create our first database, using the following steps; for a more detailed list of steps we encourage the reader to refer to: https://www.postgresql.org/docs/current/creating-cluster.html.

1. Create a directory to hold the PostgreSQL data tree by executing the following commands in the bash console:

```
mkdir DATA_PATH
```

where DATA_PATH is a directory on disk where PostgreSQL will hold data.

2. Create a PostgreSQL cluster by executing:

```
PATH/bin/initdb -D DATA_PATH
```

where PATH is the installation directory (specified in the ./configure call), and DATA_PATH is the data directory path.
A database cluster is a collection of databases that are managed by a single server instance. The initdb function creates the directories in which the database data will be stored, generates the shared catalog tables (which are the tables that belong to the whole cluster rather than to any particular database), and creates template1 (a template for generating new databases) and postgres databases. The postgres database is a default database available for use by all users, and any third party applications.

3. Start up the PostgreSQL server by executing:

```
PATH/bin/postgres -D DATA_PATH >logfile 2>&1 &
```

where PATH and DATA_PATH are as described earlier.
By default, PostgreSQL uses the TCP/IP port 5432 for the postgres server to listen for connections. Since default installations frequently exist side by side with the installations from source code, not all PostgreSQL servers can be simultaneously listening on the same TCP port. The postgres server can also run on a

different port, specified by using the flag -p. This port should then be specified by all client applications (e.g. createdb, psql discussed next).
To run postgres on a different port type:

```
PATH/bin/postgres -D DATA_PATH -p PORT >logfile 2>&1 &
```

where PORT is an alternative port number between 1024 and 65535, that is not currently used by any application on your computer.
The postgres command can also be called in *single-user mode*. This mode is particularly useful for debugging or disaster recovery. When invoked in single-user mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. In the single-user mode, the session user will be set to the user with ID 1, and implicit superuser powers are granted to this user. This user does not actually have to exist, so the single-user mode can be used to manually recover from certain kinds of accidental damage to the system catalogs. To run the postgres server in the single-user mode type:

```
PATH/bin/postgres –single -D DATA_PATH DBNAME
```

4. Create a new PostgreSQL database in the cluster by executing:

```
PATH/bin/createdb -p PORT test
```

where PORT is the port on which the server is running; the port specification can be omitted if the default port (5432) is being used.
After this step, in addition to template1 and postgres databases, the database named test will be placed in the cluster as well. You can use any other name in place of test.

5. Log-in to the database using the psql command:

```
PATH/bin/psql -p PORT test
```

Now you can create tables, insert some data and run queries over this database. When debugging, it is frequently useful to run SQL commands directly from the command line or read them from a file. This can be achieved by specifying the options -c or -f. To execute a specific command you can use:

```
PATH/bin/psql -p PORT -c COMMAND test
```

where COMMAND is the command you wish to run, which is typically enclosed in double quotes.
To read and execute SQL statements from a file you can use:

```
PATH/bin/psql -p PORT -f FILENAME test
```

Directory	Description
config	Configuration system for driving the build
contrib	Source code for contribution modules (extensions)
doc	Documentation
src/backend	PostgreSQL Server (backend)
src/bin	psql, pg_dump, initdb, pg_upgrade and other front-end utilities
src/common	Code common to the front- and backends
src/fe_utils	Code useful for multiple front-end utilities
src/include	Header files for PostgreSQL
src/include/catalog	Definition of the PostgreSQL catalog tables
src/interfaces	Interfaces to PostgreSQL including libpq, ecpg
src/pl	Core Procedural Languages (plpgsql, plperl, plpython, tcl)
src/port	Platform-specific hacks
src/test	Regression tests
src/timezone	Timezone code from IANA
src/tools	Developer tools (including pgindent[8])
src/tutorial	SQL tutorial scripts

Figure 32.7 Top-level source code organization

where FILENAME is the name of the file containing SQL commands. If a file has multiple statements they need to be separated by semicolon.
When either -c or -f is specified, psql does not read commands from standard input; instead it terminates after processing all the -c and -f options in sequence.

32.8.2 Code Organization

Prior to adding any new functionality it is necessary to get familiar with the source code organization. Figure 32.7 presents the organization of PostgreSQL top level directory.

The PostgreSQL contrib tree contains porting tools, analysis utilities, and plug-in features that are not part of the core PostgreSQL system. Client (front-end) programs are placed in src/bin. The directory src/pl contains support for procedural languages (e.g. perl and python), which allows for writing PostgreSQL functions and procedures in these languages. Some of these libraries are not part of the generic build and need to be explicitly enabled (e.g. use ./configure –with-perl for perl support). The src/test directory contains a variety of regression tests, e.g. for testing authentication, concurrent behavior, locality and encodings, and recovery and replication.

Directory	Description
access	Methods for accessing different types of data (e.g., heap, hash, btree, gist/gin)
bootstrap	Routines for running PostgreSQL in a âŁžbootstrapâŁž mode (by initdb)
catalog	Routines used for modifying objects in the PostgreSQL Catalog
commands	User-level DDL/SQL commands (e.g., create, alter, vacuum, analyze, copy)
executor	Executor runs queries after they have been planned and optimized
foreign	Handles foreign data wrappers, user mappings, etc
jit	Provides independent Just-In-Time Compilation infrastructure
lib	Contains general purpose data structures used in the backend (e.g., binary heap, bloom filters, etc)
libpq	Code for the wire protocol (e.g., authentication, and encryption)
main	The main() routine determines how the PostgreSQL backend process will start and starts the right subsystem
nodes	Generalized Node structures in PostgreSQL. Contains functions to manipulate with nodes (e.g., copy, compare, print, etc)
optimizer	Optimizer implements the costing system and generates a plan for the executor
parser	Parser parses the sent queries
partitioning	Common code for declarative partitioning in PostgreSQL
po	Translations of backend messages to other languages
port	Backend-specific platform-specific hacks
postmaster	The main PostgreSQL process that always runs, answers requests, and hands off connections
regex	Henry Spencer's regex library
replication	Backend components to support replication, shipping of WAL logs, and reading them
rewrite	Query rewrite engine used with RULEs
snowball	Snowball stemming used with full-text search
statistics	Extended statistics system (CREATE STATISTICS)
storage	Storage layer handles file I/O, deals with pages and buffers
tcop	Traffic Cop gets the actual queries, and runs them
tsearch	Full-Text Search engine
utils	Various backend utility components, caching system, memory manager, etc

Figure 32.8 Source code organization of PostgreSQL backend

Since typically new functionality is added in the PostgreSQL backend directory, we further dive into the organization of this directory, which is presented in Figure 32.8.

Parser: The parser of PostgreSQL consists of two major components - the lexer and grammar. The lexer determines how the input will be tokenized. The grammar defines the grammar of the SQL and other commands that are processed by PostgreSQL, and is used for parsing commands.

The corresponding files of interest in the /backend/parser directory are: i) scan.l, which is the lexer that handles tokenization, ii) gram.y, which is the definition of the grammar, iii) parse_*.c, which contains specialized routines for parsing, and iv) analyze.c, which contains routines to transform a raw parse tree into a query tree representation.

Optimizer: The optimizer takes the query structure returned by the parser as input and produces a plan to be used by the executor as output. The /path directory contains code for exploring possible ways to join the tables (using dynamic programming), while the /plan subdirectory contains code for generating the actual execution plan. The /prep directory contains code for handling preprocessing steps for special cases. The /geqo directory contains code for a planner that uses genetic optimization algorithm to handle queries with a large number of joins; the genetic optimizer performs a semi-random search through the join tree space. The primary entry point for the optimizer is the planner() function.

Executor: The executor processes a query plan, which is a tree of plan nodes. The plan tree nodes are operators that implement a demand/pull driven pipeline of tuple processing operations (following the Volcano model). When the next() function is called on a node, it produces the next tuple in its output sequence, or NULL if no more tuples are available. If the node is not a relation-scan or index-scan, it will have 1 or more children nodes. the code implementing the operation calls next() on its children to obtain input tuples.

32.8.3 System Catalogs

PostgreSQL is an entirely catalog driven DBMS. Not only are system catalogs used to store metadata information about tables, their columns, and indices, but they are also used to store metadata information about data types, function definitions, operators and access methods. Such an approach provides an elegant way of offering extensibility: new data types or access methods can be added by simply inserting a record in the corresponding pg_* table. System catalogs are thus used to a much greater extent in PostgreSQL, compared to other relational database systems.

The list of catalog tables with their descriptions can be found at: https://www.postgresql.org/docs/current/catalogs.html. For a graphical representation of the relationships between different catalog tables, we encourage the reader to refer to: https://www.postgrescompare.com/pg_catalog/constrained_pg_catalog_organic.pdf.

Some of the widely used system catalog tables are the following:

- **pg_class**: contains one row for each table in the database.
- **pg_attribute**: contains one row for each column of each table in the database.
- **pg_index**: contains one row for each index with a reference to the table it belongs to (described in the pg_class table).
- **pg_proc**: contains one row for each defined function. Each function is described through its name, input argument types, result type, implementation language, and definition (either the text, if it is in an interpreted language, or a reference to its executable code, if it is compiled). Compiled functions can be statically linked into the server, or stored in shared libraries that are dynamically loaded on the first use.
- **pg_operator**: contains operators that can be used in expressions.
- **pg_type**: contains information about basic data types that columns in the table can have and that are accepted as inputs and outputs to functions. New data types are added by making new entries in the pg_type table.

In addition to the system catalogs, PostgreSQL provides a number of built-in system views. System views typically provide convenient access to commonly used queries on the system catalogs. For instance:

select *tablename* **from** *pg_catalog.pg_tables*;

will print the list of table names of all tables in the database. System views can be recognized in the pg_catalog schema by the plural suffix (e.g., pg_tables, or pg_indexes).

32.8.4 The Region-Based Memory Manager

PostgreSQL uses a region-based memory manager, which implements of a hierarchy of different memory contexts. Objects are created in specific memory contexts, and a single function call frees up all objects that are in a particular memory context. The memory manager routines can be found in the /backend/utils/mmgr directory. An important difference between PostgreSQL and other applications written in C is that in PostgreSQL, memory is allocated via a special routine called palloc(), as opposed to the traditional malloc() routine of C. The allocations can be freed individually with pfree() function calls, as opposed to calling the free() routine; but the main advantage of memory contexts over plain use of malloc()/free() is that the entire content of a memory context can be efficiently freed using a single call, without having to request freeing of each individual chunk within it. This is both faster and less error prone compared to per-object bookkeeping, since it prevents memory leakage and reduces chances of use-after-free bugs.

All allocations occur inside a corresponding memory context. Examples of contexts are: CurrentMemoryContext (the current context), TopMemoryContext (the backend lifetime), CurTransactionContext (the transaction lifetime), PerQueryContext (the query lifetime), PerTupleContext (the per-result-tuple lifetime). When unspecified, the default memory context is CurrentMemoryContext, which is stored as a global variable. Contexts are arranged in a tree hierarchy, expressed through a parent-child relationship. Whenever a new context gets created, the context from which the creation context routine gets invoked becomes the parent of the newly created context.

The basic operations on a memory context are:

- **MemoryContextCreate()**: to create a new context.
- **MemoryContextSwitchTo()**: to switch from the CurrentMemoryContext into a new one.
- **MemoryContextAlloc()**: to allocate a chunk of memory within a context.
- **MemoryContextDelete()**: to delete a context, which includes freeing all the memory allocated within the context.
- **MemoryContextReset()**: to reset a context, which frees all memory allocated in the context, but not the context object itself.

The MemoryContextSwitchTo() operation selects a new current context and returns the previous context, so that the caller can restore the previous context before exiting. When a memory context is reset, all allocations within the context are automatically released. Reset or deletion of a context will automatically invoke the call to reset/delete all children contexts. As a consequence, memory leaks are rare in the backend, since all memory will eventually be released. Nonetheless, a memory leak could happen when memory is allocated in a too-long-lived context, i.e., a context that lives longer than supposed. An example of such a case would be allocating resources that are needed per tuple into CurTransactionContext that lives much longer than the tuple.

32.8.5 Node Structure and Node Functions

Each query in PostgreSQL is represented as a tree of nodes; the actual type of nodes differs depending on the query stage (e.g., the parsing, optimization, or execution stage). Since C does not support inheritance at the language level (unlike C++), to represent node type inheritance, PostgreSQL uses a node structuring convention that in effect implements a simple object system with support for a single inheritance. The root of the class hierarchy is Node, presented below:

```
typedef struct {
    NodeTag type;
} Node;
```

The first field of any Node is NodeTag, which is used to determine a Node's specific type at run-time. Each node consists of a type, plus appropriate data. It is particularly important to understand the node type system when adding new features, such as new access path, or new execution operator. Important functions related to nodes are: makeNode() for creating a new node, IsA() which is a macro for run-time type testing, equal() for testing the equality of two nodes, copyObject() for a deep copy of a node (which should make a copy of the tree rooted at that node), nodeToString() to serialize a node to text (which is useful for printing the node and tree structure), and stringToNode() for deserializing a node from text.

An important thing to remember when modifying or creating a new node type is to update these functions (especially equal() and copy() that can be found in equalfuncs.c and copyfuncs.c in the /CODE/nodes/ directory). For serialization and deserialization, /nodes/outfuncs.c need to be modified as well.

32.8.5.1 Steps For Adding a New Node Type

To illustrate, suppose that we want to introduce a new node type called TestNode, the following are the steps required:

1. Add a tag (T_TestNode) to the enum NodeTag in include/nodes/nodes.h.
2. Add the structure definition to the appropriate include/nodes/*.h file. The nodes.h is used for defining node tags (NodeTag), primnodes.h for primitive nodes, parsenodes.h for parse tree nodes, pathnodes.h for path tree nodes and planner internal structures, plannodes.h for plan tree nodes, execnodes.h for executor nodes, and memnodes.h for memory nodes.
   For example, new node type can be defined as follows:

```
typedef struct TestNode
{
    NodeTag type;
    /* a list of other attributes */
} TestNode;
```

3. If we intend to use copyObject(), equal(), nodeToString() or stringToNode(), we need to add an appropriate function to copyfuncs.c, equalfuncs.c, outfuncs.c, and readfuncs.c respectively. For example:

```
static TestNode *
_copyTestNode(const TestNode *from)
{
      TestNode *newnode = makeNode(TestNode);
      /* copy remainder of node fields (if any) */
      newnode= COPY_*(from);
      return newnode;
}
```

where COPY_* is a routine to copy individual fields. Alternatively, each attribute of the TestNode can be copied individually by calling the existing copy routines, such as COPY_NODE_FIELD, COPY_SCALAR_FIELD, and COPY_POINTER_FIELD.

4. We also need to modify the functions in nodeFuncs.c to add code for handling the new node type; which of the functions needs to be modified depends on the node type added. Examples of functions in this file includes *_tree_walker() functions to traverse various types of trees, and *_tree_mutator() functions to traverse various types of trees and return a copy with specified changes to the tree.

As a general note, there may be other places in the code where we might need to inform PostgreSQL about our new node type. The safest way to make sure no place in the code has been overlooked is to search (e.g., using grep) for references to one or two similar existing node types to find all the places where they appear in the code.

32.8.5.2 Casting Pointers to Subtypes and Supertypes

To support inheritance, PostgreSQL uses a simple convention where the first field of any subtype is its parent, i.e., its supertype. Hence, casting a subtype into a supertype is trivial. Since the first field of a node of any type is guaranteed to be the NodeTag, any node can be cast into Node *. Declaring a variable to be of Node * (instead of void *) can facilitate debugging.

In the following we show examples of casting a subtype into a supertype and vice versa, by using the example of SeqScanState and PlanState. PlanState is the common abstract superclass for all PlanState-type nodes, including ScanState node. ScanState extends PlanState for node types that represent scans of an underlying relation. Its subtypes include SeqScanState, IndexScanState, and IndexOnlyScanState among others. To cast SeqScanState into PlanState we can use direct casting such as (PlanState *) SeqScanState *.

Casting a supertype into a subtype on the other hand requires a run-time call to a castNode macro, which will check whether the NodeTag of the given pointer is of the given subtype. Similarly, a supertype can be cast directly into a subtype after invoking the IsA macro at run-time, which will check whether the given node is of the requested

subtype (again by checking the NodeTag value). The following code snippet shows an example of casting a supertype into a subtype by using the castNode macro:

```
static TupleTableSlot *
ExecSeqScan(PlanState *pstate)
{
    /* Cast a PlanState (supertype) into a SeqScanState (subtype) */
    SeqScanState *node = castNode(SeqScanState, pstate);
    ...
}
```

32.8.6 Datum

Datum is a generic data type used to store the internal representation of a single value of any SQL data type that can be stored in a PostgreSQL table. It is defined in postgres.h. A Datum contains either a value of a pass-by-value type or a pointer to a value of a pass-by-reference type. The code using the Datum has to know which type it is, since the Datum itself does not contain that information. Usually, C code will work with a value in a native representation, and then convert to or from a Datum in order to pass the value through data-type-independent interfaces.

There are a number of macros to cast a Datum to and from one of the specific data types. For instance:

- **Int32GetDatum(int)**: will return a Datum representation of an Int32.
- **DatumGetInt32(Datum)**: will return Int32 from a Datum.

Similar macros exist for all other data types such as Bool (boolean), and Char (character) data types.

32.8.7 Tuple

Datums are used to extensively to represent values in tuples. A tuple comprises of a sequence of Datums. HeapTupleData (defined in include/access/htup.h) is an in-memory data structure that points to a tuple. It contains the length of a tuple, and a pointer to the tuple header. The structure definition is as follows:

```
typedef struct HeapTupleData
{
    uint32 t_len; /* length of *t_data */
    ItemPointerData t_self; /* SelfItemPointer */
    Oid t_tableOid; /* table the tuple came from */
    HeapTupleHeader t_data; /* pointer to tuple header and data */
} HeapTupleData;
```

The t_len field contains the tuple length; the value of this field should always be valid, except in the pointer-to-nothing case. The t_self pointer is a pointer to an item within a disk page of a known file. It consists of a block ID (which is a unique identifier of a block), and an offset within the block. The t_self and t_tableOid (the ID of the table the tuple belongs to) values should be valid if the HeapTupleData points to a disk buffer, or if it represents a copy of a tuple on disk. They should be explicitly set invalid in tuples that do not correspond to tables in the database.

There are several ways in which a pointer t_data can point to a tuple:

- **Pointer to a tuple stored in a disk buffer**: which is a pointer directly to a pinned buffer page (when the page is stored in the memory buffer pool).
- **Pointer to nothing**: which points to NULL, and is frequently used as a failure indicator in functions.
- **Part of a palloc'd tuple**: the HeapTupleData struct itself and the tuple form a single palloc'd chunk. t_data points to the memory location immediately following the HeapTupleData struct.
- **Separately allocated tuple**: t_data points to a palloc'd chunk that is not adjacent to the HeapTupleData struct.
- **Separately allocated minimal tuple**: t_data points minimal tuple offset bytes before the start of a MinimalTuple.

MinimalTuple (defined in htup_details.h) is an alternative representation to HeapTuple that is used for transient tuples inside the executor, in places where transaction status information is not required, and the tuple length is known. The purpose of MinimalTuple is to save a few bytes per each tuple, which may be a worthwhile effort over a number of tuples. This representation is chosen so that tuple access routines can work with either full or minimal tuples via a HeapTupleData pointer structure introduced above. The access routines see no difference, except that they must not access fields that are not part of the MinimalTuple (such as the tuple length for instance).

PostgreSQL developers recommend that tuples (both MinimalTuple and HeapTuple) should be accessed via TupleTableSlot routines. TupleTableSlot is an abstraction used in the executor to hide details to where tuple pointers point to (e.g., buffer page, heap allocated memory, etc). The executor stores tuples in a "tuple table", which is a list of independent TupleTableSlot(s), which enables cursor-like behavior. The TupleTableSlot routines will prevent access to the system columns and thereby prevent accidental use of the nonexistent fields. Examples of TupleTableSlot routines include the following:

```
PostgresMain()
    exec_simple_query()
        pg_parse_query()
            raw_parser() – calling the parser
        pg_analyze_rewrite()
            parse_analyze() – calling the parser (analyzer)
            pg_rewrite_query()
                QueryRewrite() – calling the rewriter
                    RewriteQuery()
        pg_plan_queries()
            pg_plan_query()
                planner() – calling the optimizer
                    create_plan()
        PortalRun()
            PortalRunSelect()
                ExecutorRun()
                    ExecutePlan() – calling the executor
                        ExecProcNode()
                                – uses the demand-driven pipeline execution model
            or
            ProcessUtility() – calling utilities
```

Figure 32.9 PostgreSQL Query Execution Stack

```
void (*copyslot) (TupleTableSlot *dstslot, TupleTableSlot *srcslot);
HeapTuple (*get_heap_tuple)(TupleTableSlot *slot);
MinimalTuple (*get_minimal_tuple)(TupleTableSlot *slot);
HeapTuple (*copy_heap_tuple)(TupleTableSlot *slot);
MinimalTuple (*copy_minimal_tuple)(TupleTableSlot *slot);
```

These function pointers are redefined for different types of tuples, such as HeapTuple, MinimalTuple, BufferHeapTuple, and VirtualTuple.

32.8.8 Query Execution Stack

Figure 32.9 depicts the query execution stack through important function calls. The execution starts with the PostgresMain() routine, which is the main module of the PostgreSQL backend and can be found in /tcop/postgres.c. Execution then proceeds through the parser, rewriter, optimizer, and executor.

Parser: When a query is received, the PostgresMain() routine calls exec_simple_query(), which in turn calls first pg_parse_query() to perform the parsing of the query. This function in turn calls the function raw_parser() (which is located in /parser/parser.c). The pg_parse_query() routine returns a list of raw parse

trees – each parse tree representing a different command, since a query may contain multiple select statements separated by semicolons.

Each parse tree is then individually analyzed and rewritten. This is achieved by calling pg_analyze_rewrite() from the exec_simple_query() routine. For a given raw parse tree, the pg_analyze_rewrite() routine performs parse analysis and applies rule rewriting (combining parsing and rewriting), returning a list of Query nodes as a result (since one query can be expanded into several ones as a result of this process). The first routine that pg_analyze_rewrite() invokes is parse_analyze() (located in /parser/analyze.c) to obtain a Query node of the given raw parse tree.

Rewriter: The rule rewrite system is triggered after parser. It takes the output of the parser, one Query tree, and defined rewrite rules, and creates zero or more Query trees as result. Typical examples of rewrite rules are replacing the use of a view with its definition, or populating procedural fields. The parse_analyze() call from the parser is thus followed by pg_rewrite_query() to perform rewriting. The pg_rewrite_query() invokes the QueryRewrite() routine (located in /rewrite/rewriteHandler.c), which is the primary module of the query rewriter. This method in turn makes a recursive call of RewriteQuery() where rewrite rules are repeatedly applied, as long as some rule is applicable.

Optimizer: After pg_analyze_rewrite() finishes, producing a list of Query nodes as output, the pg_plan_queries() routine is invoked to generate plans for all the nodes from the Query list. Each Query node is optimized by calling pg_plan_query(), which in turn invokes planner() (located in /plan/planner.c), which is the main entry point for the optimizer. The planner() routine invokes the create_plan() routine to create the best query plan for a given path, returning a Plan as output. Finally, the planner routine creates a PlannedStmt node to be fed to the executor.

Executor: Once the best plan is found for each Query node, the exec_simple_query() routine calls PortalRun(). A portal, previously created in the initialization step (discussed in the next section), represents the execution state of query. PortalRun() in turn invokes ExecutorRun() through PortalRunSelect() in the case of queries, or ProcessUtility() in the case of utility functions for each individual statement. Both ExecutorRun() and ProcessUtility() accept a PlannedStmt node; the only difference is that the utility call has the commandType attribute of the node set to CMD_UTILITY.

The ExecutorRun() defined in execMain.c, which is the main routine of the executor module, invokes ExecutePlan() which processes the query plan by calling ExecProcNode() for each individual node in the plan, applying the demand-driven pipelining (iterator) model (see Section 15.7.2.1 for more details).

32.8.8.1 Memory Management and Contexts Switches

Before making any changes to PostgreSQL code, it is important to understand how different contexts get switched during the lifetime of a query. Figure 32.10 shows a sketch of the query processing control flow, with key context switch points annotated with comments.

```
CreateQueryDesc()
ExecutorStart()
    CreateExecutorState() — creates per-query context
    switch to per-query context
    InitPlan()
        ExecInitNode() — recursively scans plan tree
            CreateExprContext() — creates per-tuple context
            ExecInitExpr()
ExecutorRun()
    ExecutePlan()
        ExecProcNode() — recursively called in per-query context
            ExecEvalExpr() — called in per-tuple context
            ResetExprContext() — to free per-tuple memory
ExecutorFinish()
    ExecPostprocessPlan()
    AfterTriggerEndQuery()
ExecutorEnd()
    ExecEndPlan()
        ExecEndNode() — recursively releases resources
    FreeExecutorState() — frees per-query context and child contexts
FreeQueryDesc()
```

Figure 32.10 PostgreSQL Query Processing Control Flow

Initialization: The CreateQueryDesc() routine (defined in tcop/pquery.c), is invoked through PortalStart(), i.e. in the initialization step before calling the ExecutorRun() routine. This function allocates the query descriptor, which is then used in all key executor routines (ExecutorStart, ExecutorRun, ExecutorFinish, and ExecutorEnd). The QueryDesc encapsulates everything that the executor needs to execute the query (e.g., a PlannedStmt of the chosen plan, source text of the query, the destination for tuple output, parameter values that are passed in, and a PlanState, among other information).

ExecutorStart() (defined in executor/ExecMain.c) is invoked through PortalStart(), when starting a new portal in which a query is going to be executed. This function must be called at the beginning of execution of any query plan. ExecutorStart() in turn calls CreateExecutorState() (defined in executor/utils.c) to create a per-query context. After creating the per-query context, InitPlan() (defined in executor/execMain.c) allocates necessary memory, and calls the ExecInitNode() routine (defined in executor/execProcnode.c); this function recursively initializes all the nodes in the plan tree. Query plan nodes may invoke CreateExprContext() (defined in execUtils.c) to create a per tuple context, and ExecInitExpr() to initialize it.

Execution: After the initialization is finalized, the execution starts by invoking ExecutorRun(), which calls ExecutePlan(), which in turn invokes ExecProcNode() for each

node in the plan tree. The context is switched from the per-query context into the per-tuple context for each invocation of the ExecEvalExpr() routine.

Upon the exit from this routine, ResetExprContext() is invoked. This is a macro that invokes the MemoryContextReset() routine to release all the space allocated within the per-tuple context.

Cleanup: The ExecutorFinish() routine must be called after ExecutorRun(), and before ExecutorEnd(). This routine performs cleanup actions such as calling ExecPostprocessPlan() to allow plan nodes to execute required actions before the shutdown, and AfterTriggerEndQuery() to invoke all AFTER IMMEDIATE trigger events.

The ExecutorEnd() routine must be called at the end of execution. This routine invokes ExecEndPlan() which in turn calls ExecEndNode() to recursively release all resources. FreeExecutorState() frees up the per-query context and consequently all of its child contexts (e.g., per-tuple contexts) if they have not been released already. Finally, FreeQueryDesc() from tcop/pquery.c frees the query descriptor created by CreateQueryDesc().

This fine level of control through different contexts coupled with palloc() and pfree() routines ensures that memory leaks rarely happen in the backend.

32.8.9 Error Handling

PostgreSQL has a rich error handling mechanism, with most prominent being ereport() and elog() macros. The ereport() macro is used for user-visible errors and allows for a detailed specification through a number of fields (e.g. SQLSTATE, detail, hint, etc.). The treatment of exceptions is conceptually similar to the treatment of exceptions in other languages such as C++, but is implemented using a set of macros since C does not have an exception mechanism defined as part of the language.

The elog(ERROR) function traces up the stack to the closest error handling block, which can then either handle the error or re-throw it. The top-level error handler (if reached) aborts the current transaction and resets the transaction's memory context. Reset in turn frees all resources held by the transaction, including files, locks, allocated memory and pinned buffer pages. Assertions (i.e., the Assert function) help detect programming errors, and are frequently used in PostgreSQL code.

32.8.10 Tips For Adding New Functionality

Adding a new feature for the first time in PostgreSQL can be an overwhelming experience even for researchers and experienced developers. This section provides some guidelines and general advice on how to minimize the risk of failures.

First, one needs to isolate in which subsystem the desired feature should be implemented, for example the backend, PostgreSQL contributions (contrib), etc. Once the subsystem is identified, a general strategy is to explore how similar parts of the system function. Developers are strongly advised against copying code directly since this is a common source of errors. Instead, the developers should focus on understanding of the exposed APIs, existing function calls, and the way they are used.

Another common source of errors is re-implementing existing functionality, often due to insufficient familiarity with the code base. A general guideline to avoid potential errors and code duplication is to use existing APIs and add only necessary additions to the existing code base. For instance, PostgreSQL has very good support of data structures and algorithms that should be favored over custom made implementations. Examples include:

- Simple linked list implementation: pg_list.h, list.c
- Integrated/inline doubly- and singly- linked lists: ilist.h, ilist.c
- Binary Heap implementation: binaryheap.c
- Hopcroft-Karp maximum cardinality algorithm for bipartite graphs: bipartite_match.c
- Bloom Filter: bloomfilter.c
- Dynamic Shared Memory-Based Hash Tables: dshash.c
- HyperLogLog cardinality estimator: hyperloglog.c
- Knapsack problem solver: knapsack.c
- Pairing Heap implementation: pairingheap.c
- Red-Black binary tree: rbtree.c
- String handling: stringinfo.c

Prior to adding a desired functionality, the behavior of the feature should be discussed in depth, with a special focus on corner cases. Corner cases are frequently overlooked and result in a substantial debugging overhead after the feature has been implemented. Another important aspect is understanding the relationship between the desired feature and other parts of PostgreSQL. Typical examples would include (but are not limited to) the changes to the system catalog, or the parser.

PostgreSQL has a great community where developers can ask questions, and questions are usually answered promptly. The web page https://www.postgresql.org/developer/ provides links to a variety of resources that are useful for PostgreSQL developers. The pgsql-general@postgresql.org mailing list is targeted for developers, and database administrators (DBAs) who have a question or problem when using PostgreSQL. The pgsql-hackers@postgresql.org mailing list is targeted for developers to submit and discuss patches, or for bug reports or issues with unreleased versions (e.g. development snapshots, beta or release candidates), and for discussion about database internals. Finally, the mailing list pgsql-novice@postgresql.org is a great starting point for all new developers, with a group of people who answer even basic questions.

Bibliographical Notes

Parts of this chapter are based on a previous version of the chapter, authored by Anastasia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder, Karl Schnaitter, and Gavin Sherry, which was published in the 6th edition of this textbook,

There is extensive online documentation of PostgreSQL at www.postgresql.org. This Web site is the authoritative source for information on new releases of PostgreSQL, which occur on a frequent basis. Until PostgreSQL version 8, the only way to run PostgreSQL under Microsoft Windows was by using Cygwin. Cygwin is a Linux-like environment that allows rebuilding of Linux applications from source to run under Windows. Details are at www.cygwin.com. Books on PostgreSQL include [Schonig (2018)], [Maymala (2015)] and [Chauhan and Kumar (2017)]. Rules as used in PostgreSQL are presented in [Stonebraker et al. (1990)]. Many tools and extensions for PostgreSQL are documented by the pgFoundry at www.pgfoundry.org. These include the pgtcl library and the pgAccess administration tool mentioned in this chapter. The pgAdmin tool is described on the Web at www.pgadmin.org. Additional details regarding the database-design tools TOra and PostgreSQL Maestro can be found at tora.sourceforge.net and https://www.sqlmaestro.com/products/postgresql/maestro/, respectively.

The serializable snapshot isolation protocol used in PostgreSQL is described in [Ports and Grittner (2012)].

An open-source alternative to PostgreSQL is MySQL, which is available for noncommercial use under the GNU General Public License. MySQL may be embedded in commercial software that does not have freely distributed source code, but this requires a special license to be purchased. Comparisons between the most recent versions of the two systems are readily available on the Web.

Bibliography

[Chauhan and Kumar (2017)] C. Chauhan and D. Kumar, *PostgreSQL High Performance Cookbook*, Packt Publishing (2017).

[Maymala (2015)] J. Maymala, *PostgreSQL for data architects*, Packt Publ., Birmingham (2015).

[Ports and Grittner (2012)] D. R. K. Ports and K. Grittner, "Serializable Snapshot Isolation in PostgreSQL", *Proceedings of the VLDB Endowment*, Volume 5, Number 12 (2012), pages 1850–1861.

[Schonig (2018)] H.-J. Schonig, *Mastering PostgreSQL 11*, Packt Publishing (2018).

[Stonebraker et al. (1990)] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedure, Caching and Views in Database Systems", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 281–290.