

Chapter 27: PostgreSQL

Database System Concepts, 6th Ed.

- Chapter 1: Introduction
- **Part 1: Relational databases**
 - Chapter 2: Introduction to the Relational Model
 - Chapter 3: Introduction to SQL
 - Chapter 4: Intermediate SQL
 - Chapter 5: Advanced SQL
 - Chapter 6: Formal Relational Query Languages
- **Part 2: Database Design**
 - Chapter 7: Database Design: The E-R Approach
 - Chapter 8: Relational Database Design
 - Chapter 9: Application Design
- **Part 3: Data storage and querying**
 - Chapter 10: Storage and File Structure
 - Chapter 11: Indexing and Hashing
 - Chapter 12: Query Processing
 - Chapter 13: Query Optimization
- **Part 4: Transaction management**
 - Chapter 14: Transactions
 - Chapter 15: Concurrency control
 - Chapter 16: Recovery System
- **Part 5: System Architecture**
 - Chapter 17: Database System Architectures
 - Chapter 18: Parallel Databases
 - Chapter 19: Distributed Databases
- **Part 6: Data Warehousing, Mining, and IR**
 - Chapter 20: Data Mining
 - Chapter 21: Information Retrieval
- **Part 7: Specialty Databases**
 - Chapter 22: Object-Based Databases
 - Chapter 23: XML
- **Part 8: Advanced Topics**
 - Chapter 24: Advanced Application Development
 - Chapter 25: Advanced Data Types
 - Chapter 26: Advanced Transaction Processing
- **Part 9: Case studies**
 - Chapter 27: PostgreSQL
 - Chapter 28: Oracle
 - Chapter 29: IBM DB2 Universal Database
 - Chapter 30: Microsoft SQL Server
- **Online Appendices**
 - Appendix A: Detailed University Schema
 - Appendix B: Advanced Relational Database Model
 - Appendix C: Other Relational Query Languages
 - Appendix D: Network Model
 - Appendix E: Hierarchical Model

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

Introduction

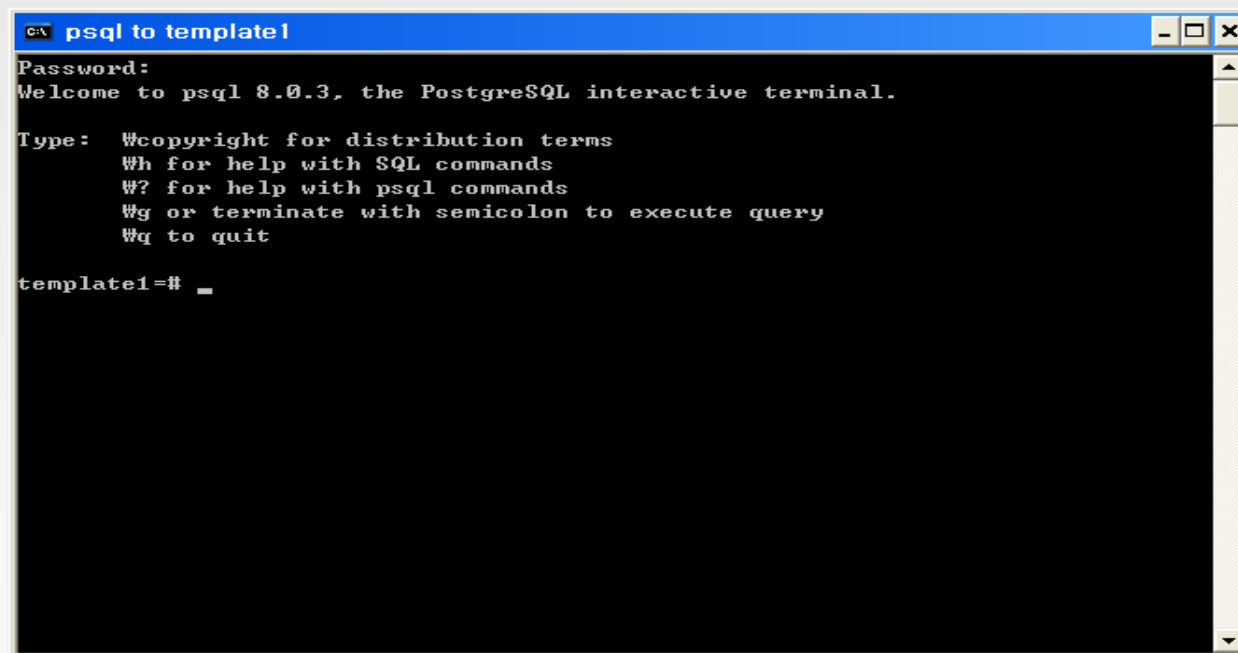
- PostgreSQL runs over virtually all Unix-like operating systems, including Linux and Apple Macintosh OS X
- PostgreSQL can be run over Microsoft Windows under the Cygwin environment, which provides Linux emulation under Windows
- This chapter surveys how the PostgreSQL works, starting from user interfaces and languages and continuing into the core of the system

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

■ The standard PostgreSQL

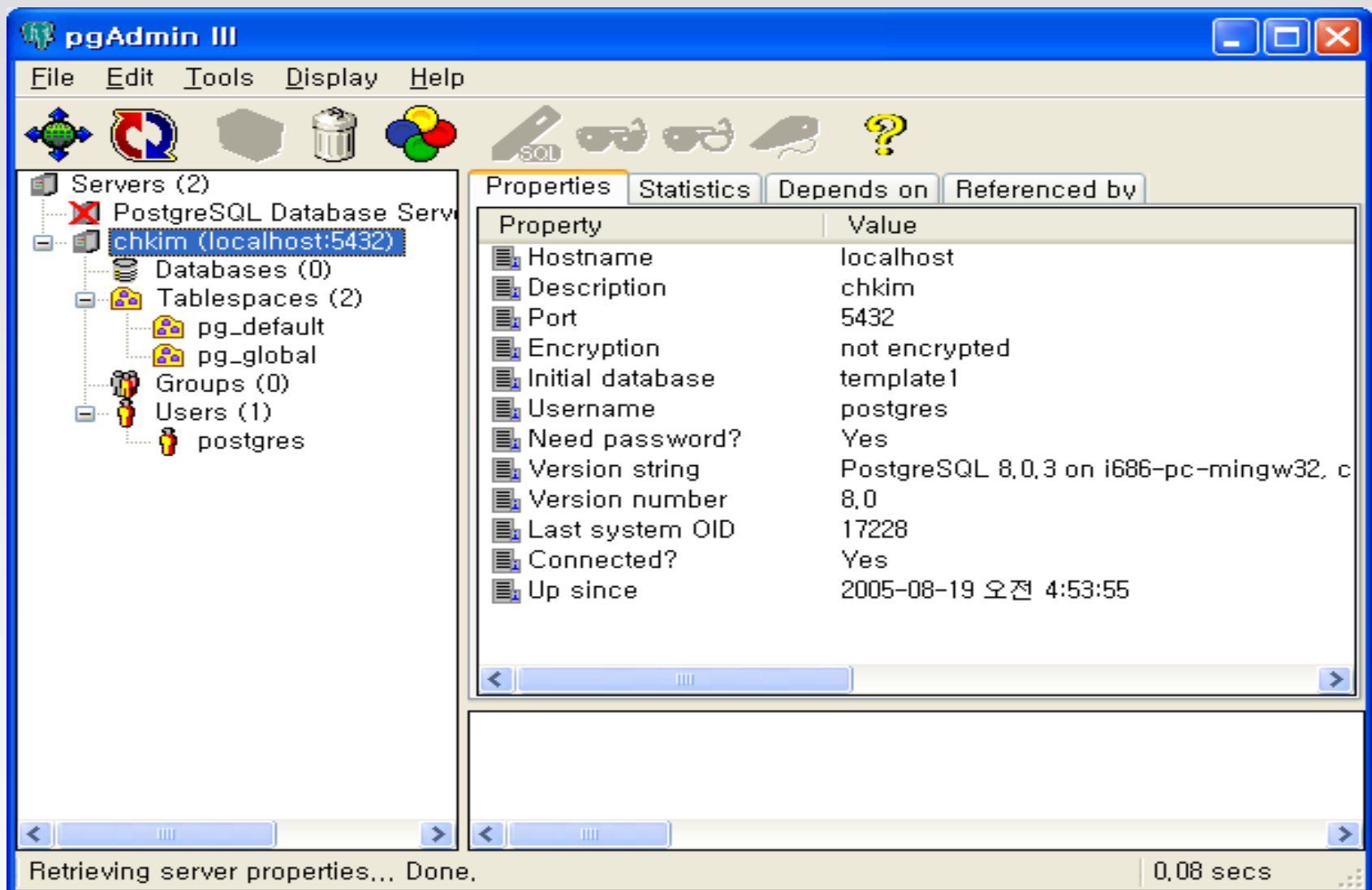
- Based on command-line tools for administrating the database
- Offers a comprehensive set of programming interfaces
- A wide range of commercial and open-source graphical tools exist



```
C:\> psql to template1
Password:
Welcome to psql 8.0.3, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

template1=# _
```



■ Interactive Terminal Interfaces

- main interactive terminal client, psql
 - ▶ modeled after the Unix shell
 - ▶ execute SQL commands on the server
 - ▶ performs several other operations
 - ▶ features
 - Variables
 - SQL interpolation
 - Command-line editing
- pgtksh, pgtclsh (versions of the Tk and Tcl include PostgreSQL bindings)
 - ▶ Tcl/Tk is a flexible scripting language for rapid prototyping

■ Graphical Interfaces

- Graphical tools for administration
 - ▶ pgAccess
 - ▶ pgAdmin
- Graphical tools for database design
 - ▶ TORA
 - ▶ Data Architect
- Works with several commercial forms-design and report-generation tools
 - ▶ Rekal
 - ▶ GNU Report Generator
 - ▶ GNU Enterprise

■ Programming Language Interfaces

- Provides native interfaces for
 - ▶ ODBC
 - ▶ JDBC
 - ▶ Bindings for most programming languages
 - C / C++ / PHP / Perl / Tcl / Tk / ECPG / Python / Ruby
- C API for PostgreSQL: libpq
 - ▶ supports synchronous/asynchronous execution of SQL commands and prepared statements
 - ▶ re-entrant
 - ▶ Thread-safe
 - ▶ Environment variables for certain parameters
 - ▶ Optional password file for connections

Overview

- 26.1 Introduction
- 26.2 User Interfaces
- 26.3 SQL Variations and Extensions
- 26.4 Transaction Management in PostgreSQL
- 26.5 Storage and Indexing
- 26.6 Query Processing and Optimization
- 26.7 System Architecture
- Summary & Bibliographical Notes

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

PostgreSQL Variations and Extensions

■ PostgreSQL

- ANSI SQL compliant
- supports almost all entry-level SQL92 features and many of the intermediate- and full-level features
- supports several of the SQL:1999 features
- Data can be easily loaded into OLAP servers

PostgreSQL Standard and NonStandard Types

■ The PostgreSQL Type System

- Base types
- Composite types
- Domains
- Pseudotypes
- Polymorphic types

■ Nonstandard Types

- data type for complex numbers
- type for ISBN/ISSN
- geometric data type
 - ▶ point, line, lseg, box, polygon, path, circle
- data types to store network addresses
 - ▶ cidr(IPv4), inet(IPv6), macaddr(MAC)

PostgreSQL Rules and Triggers [1/4]

■ Rules and Other Active-Database Features

- PostgreSQL supports
 - ▶ SQL constraints
 - check constraints
 - not-null constraints
 - primary key constraints
 - foreign key constraints
 - restricting and cascading deletes
 - ▶ Triggers
 - Useful for nontrivial constraints and consistency checking or enforcement
 - ▶ Query-rewriting rules

PostgreSQL Rules and Triggers [2/4]

■ PostgreSQL rules system

- allows to define query-rewrite rules on the DB server
- Intervenes between the query parser and the planner
- modifies queries on the basis of the set of rules
- General syntax for declaring rules

create rule *rule_name* **as**

on { **select** | **insert** | **update** | **delete** }

to *table* [**where** *rule_qualification*]

do [**instead**] { **nothing** | *command* | (*command*; *command*...) }

PostgreSQL Rules and Triggers [3/4]

■ Rules

- PostgreSQL uses rules to implement **views**

create view *myview* **as select * from** *mytab*;

→ **create table** *myview* (same column list as *mytab*)

create rule *return* **as on select to** *myview* **do instead**
select * from *mytab*;

- **create view** syntax is more concise and it also prevents creation of views that reference each other
- **rules** can be used to define update actions on views explicitly, but view can not

■ Example (when the user wants to audit table updates)

create rule *salary_audit* **as on update to** *employee*

where *new.salary* < > *old.salary*

do insert into *salary_audit*

values (*current_timestamp*, *current_user*,

new.emp_name, *old.salary*, *new.salary*);

PostgreSQL Rules and Triggers [4/4]

- Example(more complicated insert/update rule)
 - salary increases stored in *salary_increases* (*emp_name*, *increase*)
 - Define dummy table *approved_increases* with the same fields

```
create rule approved_increases_insert
as on insert to approved_increases
do instead
update employee
    set salary = salary + new.increase
    where emp_name = new.emp_name;
```

- Then the following query will update all salaries in *employee* table at once

```
Insert into approved_increases select * from salary_increases;
```

PostgreSQL Extensibility

■ Extensibility

- RDB systems are extended
 - ▶ by changing the source code or
 - ▶ by loading extension modules
- PostgreSQL stores information about system in system catalogs
- Extended easily through extension of system catalogs
- Also incorporate user-written code by dynamic loading of shared objects
- **contrib module** includes
 - ▶ user functions(array iterators, fuzzy string matching, cryptographic functions)
 - ▶ base types(encrypted passwords, ISBN/ISSNs, n-dimensional cubes)
 - ▶ index extensions(RD-trees, full-text indexing)

PostgreSQL Type Extension

- PostgreSQL allows to define **composite types** and **extend the available base types**

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

- Example of adding base type to PostgreSQL

```
typedef struct Complex{
```

```
    double x;
```

```
    double y;
```

```
} Complex;
```

- User must define functions to read or write values of the new data type in the text format
- Assume that the I/O functions are *complex_in* and *complex_out*
- New data type can be registered like that:

create type *complex*{

internallength = 16,

input = *complex_in*,

output = *complex_out*,

alignment = **double**

};

PostgreSQL Function Extension [1/3]

- Can be stored and executed on the server
- PostgreSQL supports [function overloading](#)
- Can be written as statements using SQL or several procedural languages
- PostgreSQL has an API for adding functions written in C
- Declaration to register the user-defined function on the server is

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

PostgreSQL Function Extension [2/3]

- Example for the text output of *complex* values

```
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);
```

```
}
```

PostgreSQL Function Extension [3/3]

■ Aggregate functions

- operate by updating a state value via a [state transition function](#)
- final function can be called to compute the return value
- extending example for sum aggregate function

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

- PostgreSQL call the appropriate function, on the basis of the actual type of its argument
- *basetype*: argument type
- *stype*: state value type

PostgreSQL Index Extension [1/2]

■ Index Extensions

- PostgreSQL supports **B-tree**, **hash**, **R-tree** and **Gist** indices
- All indices can be easily extended
- Need definition **operator class** to encapsulate the followings two:
 - ▶ Index-method strategies
 - operators that can be used as qualifiers in where clause
 - depends on the index type
 - B-tree indices: <, <=, =, >=, >
 - Hash indices: =
 - R-tree indices: contained, to-the-left, etc
 - ▶ Index-method support routines
 - functions to support the operation of indices

PostgreSQL Index Extension [2/2]

■ Index Extensions(Cont.)

- Example(declaration for operators)

```
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);
```


PostgreSQL Procedural Extension [1/2]

■ Procedural Languages

- functions and procedures can be written in procedural languages
- PostgreSQL defines API to support any procedural languages
- programming languages are trusted or untrusted
- if untrusted, superuser privileges can allow them to access the DBMS and the file system
 - ▶ PL/pqSQL
 - trusted language
 - adds procedural programming capabilities to SQL
 - similar to Oracle's PL/SQL
 - ▶ PL/Tcl, PL/Perl, and PL/Python
 - use the power of Tcl, Perl and Python
 - have bindings that allow access the DB via language-specific interfaces

PostgreSQL Procedural Extension [1/2]

- Server Programming Interface(SPI)
 - API that allows running arbitrary SQL commands inside user-defined C functions
 - function definitions can be implemented with essential parts in C
 - functions can use the full power of RDB system engine

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

Transaction Management in PostgreSQL

■ Concurrency control

- MVCC and 2-PL is implemented in PostgreSQL
- DML statements use MVCC schema
- DDL statements is based on standard 2-PL

PostgreSQL Isolation Levels [1/2]

- The SQL standard also defines 3 weak levels of consistency
- Weak consistency levels provide a higher degree of concurrency
- 3 phenomena that violates serializability
 - Nonrepeatable read
 - ▶ Transaction reads same object twice during execution
 - ▶ Get a different value although a transaction has not changed the value
 - Dirty read
 - ▶ Transaction reads values written by other transaction that has not committed yet
 - Phantom read
 - ▶ Transaction re-executes a query returning a set of rows that satisfy search condition
 - ▶ The set has changed by other recently committed transaction

PostgreSQL Isolation Levels [2/2]

- PostgreSQL supports read committed and serializable
- Definition of the 4 SQL isolation level

Isolation level	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	Maybe	Maybe	Maybe
Repeated Read	Maybe	Maybe	Maybe
Serializable	No	No	No

PostgreSQL Concurrency Control [1/2]

■ Concurrency Control for DML Commands

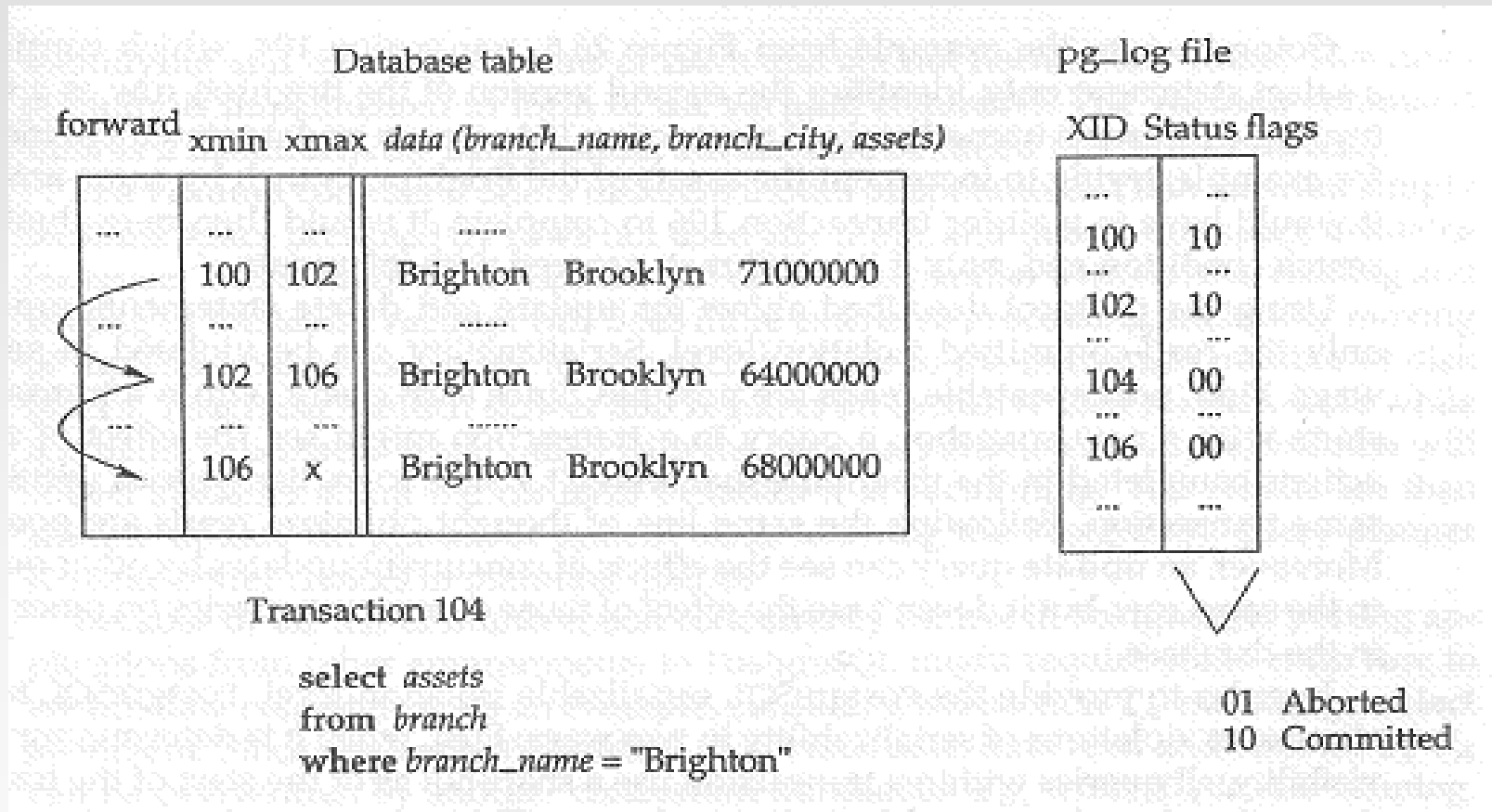
- The MVCC Protocol
 - ▶ Maintain different versions of a row at different points in time
 - ▶ guarantees all transactions see the consistent versions of data with the transaction's view
 - ▶ Each transaction sees a snapshot of the committed data at the time the transaction was started
 - ▶ The snapshot can be different from current state of data
 - ▶ Readers access the most recent version of a row from the snapshots
 - ▶ Writers create their own copy of the row to be updated
 - ▶ The only conflict that blocks transaction is occurred if two writers try to update the same row

Transaction Management in PostgreSQL(Cont.)

- PostgreSQL Implementation of MVCC
 - Tuple visibility defines which of the potentially many versions of a row in a table is valid
 - Tuple are visible for a transaction T_A if 2 following conditions hold
 - ▶ The tuple was created by T_B started running and was committed before started running
 - ▶ Updates were executed by T_C that either
 - Is aborted *or*
 - Started running after T_A *or*
 - Was in process at the start of T_A *or*

Transaction Management in PostgreSQL(Cont.)

- PostgreSQL Implementation of MVCC(Cont.)
 - Example



Transaction Management in PostgreSQL(Cont.)

■ PostgreSQL Implementation of MVCC(Cont.)

- 2 conditions that needs to be satisfied for a tuple to be visible
 - ▶ The creation-transaction ID in the tuple header
 - Is a committed transaction according to the *pg_log* file
 - Is less than the transaction counter stored at query start in SnapshotData
 - Was not in process at query start according to SnapshotData
 - ▶ The expire-transaction ID
 - Is blank or aborted
 - Is greater than the transaction counter stored at query start in SnapshotData
 - Was in process at query start according to SnapshotData

Transaction Management in PostgreSQL(Cont.)

■ PostgreSQL Implementation of MVCC(Cont.)

● Interaction with SQL statements

▶ insert

- no interaction with the concurrency-control protocol

▶ select

- depends on the isolation level
- If the isolation level is read committed, creates a new SnapshotData
- check the target tuples which are visible

▶ update, delete

- like as select statement
- If there is another concurrently executing transaction,
 - » update/delete can proceed if the transaction is aborted
 - » The search criteria of the update/delete must be evaluated again if the transaction commits
- update/delete includes updating the header information of the old tuple as well as creating a new tuple

Transaction Management in PostgreSQL(Cont.)

- PostgreSQL Implementation of MVCC(Cont.)
 - MVCC for update/delete provides only the read-committed isolation level
 - Serializability can be violated in several ways
 - ▶ Nonrepeatable reads
 - ▶ Phantom reads
 - ▶ Concurrent updates by other queries to the same row
 - Ways of PostgreSQL MVCC to eliminate violations of serializability
 - ▶ Queries use a snapshot as of the start of the transaction, rather than the start of the individual query
 - ▶ The way update/delete are processed is different in serializable mode compared to read-committed mode

Transaction Management in PostgreSQL(Cont.)

■ Implications of Using MVCC

- Storage manager must maintain different versions of tuples
- Developing concurrent applications takes extra care compared to systems where standard 2PL is used
- Performance depends on the characteristics of the workload running on it

■ PostgreSQL frees up space

- Storing multiple versions of row consumes excessive storage
- Periodically frees up space by identifying and deleting versions of rows that are not needed any more
- Implemented in form of the Vacuum command
- Vacuum runs as a daemon in the background or by user
- Vacuum can operate in parallel with normal reading and writing

Transaction Management in PostgreSQL(Cont.)

■ Vacuum

- Vacuum full compacts the table to minimize number of disk blocks
- Vacuum full requires an exclusive lock on each table
- Can use the optional parameter analyze to collect statistics

■ Care for MVCC of PostgreSQL

- Porting application to PostgreSQL needs extra care for data consistency
- To ensure data consistency, an application must use
 - ▶ select for update or
 - ▶ lock table to get a lock explicitly

PostgreSQL DDL Concurrency Control [1/3]

- MVCC don't protect transactions against operations that affect entire tables
- DDL commands acquire explicit locks before their execution
- These locks are table based
- Locks are acquired/released in accordance with strict 2PL protocol
- All locks can be acquired explicitly through the lock table command
- Deadlock detection is based on time-outs
 - detection is triggered if a transaction has been wait for a lock for more than 1 sec
 - detection algorithm
 - ▶ constructs a wait-for graph based on lock table
 - ▶ searches the wait-for graph for circular dependencies
 - ▶ If exists, aborts deadlock detection and returns an error
 - ▶ otherwise, continues waiting on the lock

PostgreSQL DDL Concurrency Control [2/3]

Lock name	Conflicts with	Acquired by
ACCESS SHARE	ACCESS EXCLUSIVE	Select query
ROW SHARE	EXCLUSIVE	select FOR update query
ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	update delete insert queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	VACUUM
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE ACCESS EXCLUSIVE	Create index
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE ACCESS EXCLUSIVE	—
EXCLUSIVE	ROW SHARE ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE ACCESS EXCLUSIVE	—
ACCESS EXCLUSIVE	CONFLICTS WITH LOCK OF ALL MODES	DROP table ALTER table VACUUM FULL

PostgreSQL DDL Concurrency Control [3/3]

■ Locking and indices

- Depends on the index type
- Gist and R-tree
 - ▶ used the simple index-level locks that are held for the entire duration of the command
- Hash index
 - ▶ used page-level locks that are released after the page is processed for higher concurrency
 - ▶ page-level lock for hash indices is not deadlock free
- B-tree index
 - ▶ used short-term share/exclusive page-level index locks
 - ▶ page-level lock for B-tree is deadlock free
 - ▶ released immediately after each index tuple is fetched or inserted

PostgreSQL Recovery

■ Recovery

- PostgreSQL employs WAL-based recovery after version 7.1
- Recovery need not undo
 - ▶ aborting transaction records in the pg_clog file entry
 - ▶ versions of rows left behind is invisible to all other transactions
- The only problem with this approach
 - ▶ transaction aborts before the process doesn't create pg_clog yet
 - ▶ To handle this,
 - checks the status of another transaction in pg_clog
 - finds the “in progress”
 - checks whether the transaction is running on any process
 - » If no process is currently running, decides “Abort”
- MVCC logs the status of every transaction in the pg_clog

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

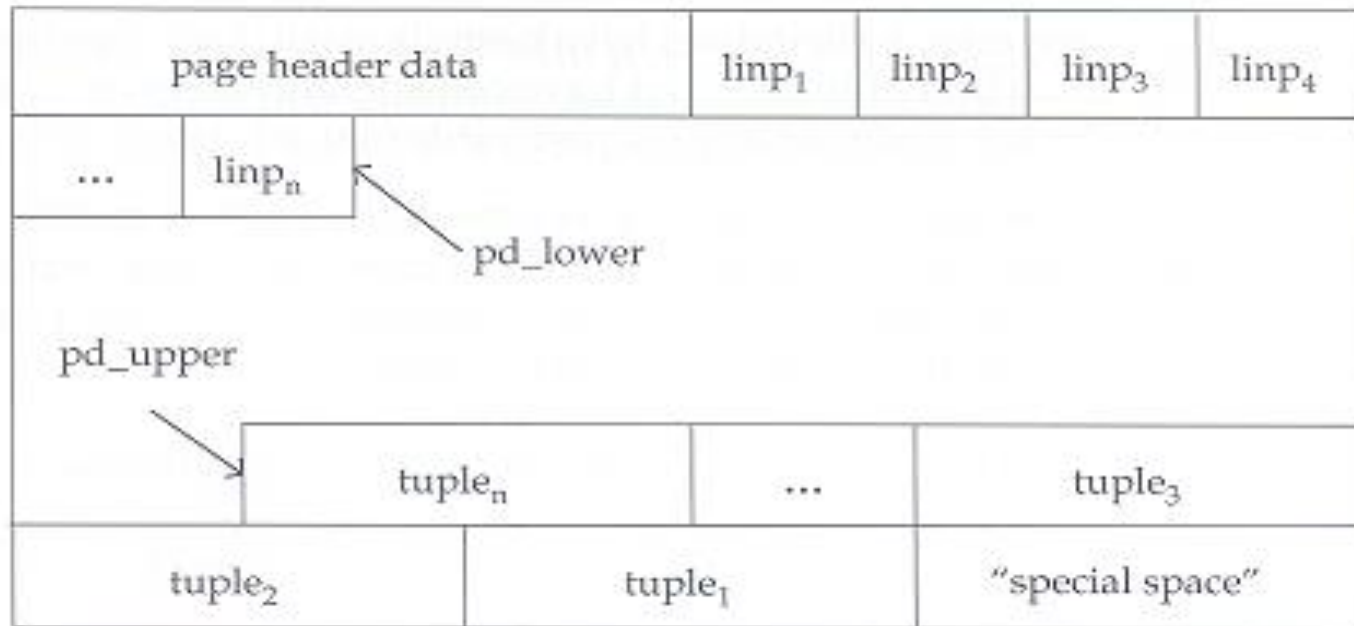
PostgreSQL Storage Structure [1/3]

- Design philosophy of data layout and storage
 - A simple and clean implementation
 - Ease of administration
- DB are partitioned into database clusters
- All data and meta data with a cluster are stored in the same directory in file system
- PostgreSQL does not support tablespces
- PostgreSQL support only cooked file systems, not raw disk partitions
- Performance limitations
 - Limits efficient using the available storage resources
 - The use of cooked file systems results in double buffering

PostgreSQL Storage Structure [2/3]

■ Tables

- primary unit of storage is table
- tables are stored in heap files
- heap files use a form of the standard slotted-page



PostgreSQL Storage Structure [3/3]

- Slotted-page format for PostgreSQL tables consists of
 - page header
 - ▶ offset
 - ▶ Length of a specific tuple in the page
 - tuples
 - ▶ stored in reverse order of line pointers from the end of the page

- The length of a tuple is limited
 - by the length of a data page
 - difficult to store very long tuples
 - If PostgreSQL encounters a large tuple,
 - ▶ tries to toast individual large columns
 - ▶ The data in toasted column is a pointer for the data outside the page

PostgreSQL Index Structure

■ Indices

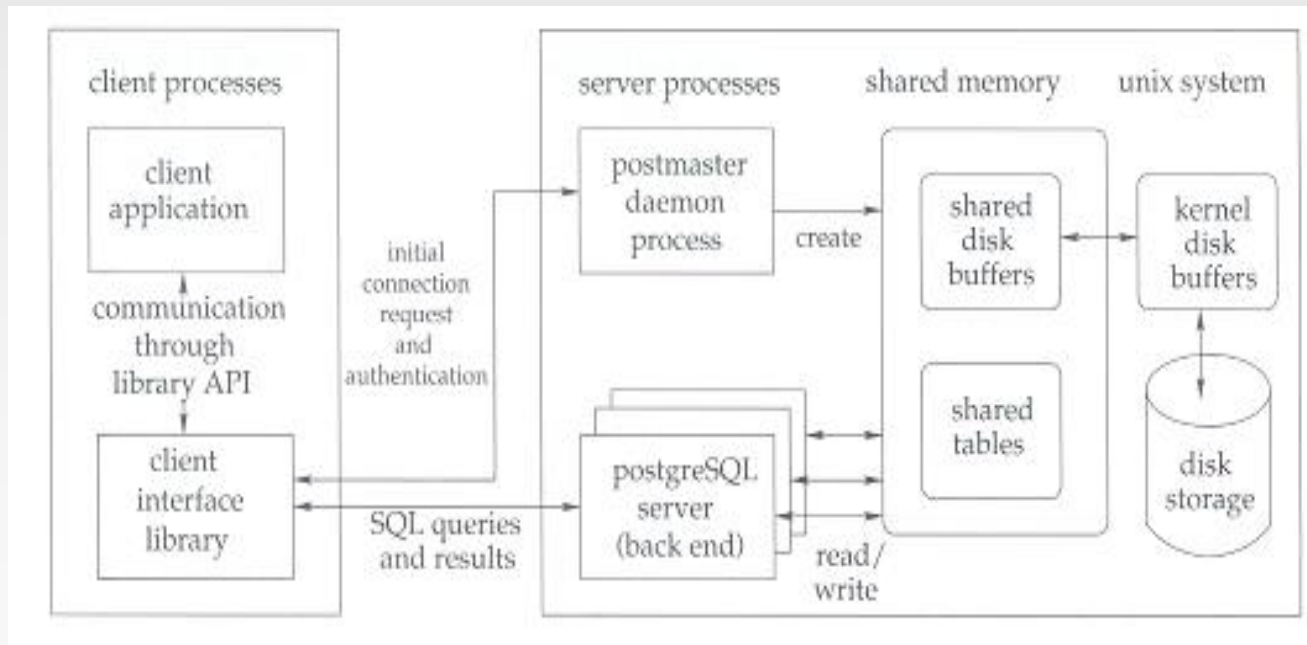
- Index types
 - ▶ B-tree
 - ▶ Hash
 - ▶ R-tree
 - ▶ Gist
- Other Index Variations
 - ▶ Multicolumn indices
 - ▶ Unique indices
 - ▶ Indices on expressions
 - ▶ Operator classes
 - ▶ Partial indices

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

Query Processing and Optimization

- Steps to process the query on PostgreSQL
 1. Receives a query
 2. Parsing
 3. Construct a query plan
 4. Execute the query plan



PostgreSQL Query Rewrite

■ Query Rewrite

- Is responsible for the PostgreSQL rules system
- Rules are only defined by users and by the definition of views
- A rule is registered in the system using the create rule command
- Information on the rule is stored in the catalog
- The catalog is used during query rewrite to uncover all candidate rules for a given query
- The rewrite phase
 - ▶ At first, deals with all update, delete, insert statements by firing all appropriate rules
 - ▶ All the remaining rules involving only select statements are fired
 - ▶ The rules are repeatedly checked until no more rules need to be fired

PostgreSQL Query Planning and Optimization

- Planning begins bottom-up from the rewritten query's innermost subquery
- Optimizer in PostgreSQL
 - cost based
 - The actual process of optimization is based on one of the two:
 - ▶ Standard planner: dynamic programming algorithm
 - ▶ Genetic query optimizer
 - algorithm developed to solve TSP
 - dynamic programming algorithm are very expensive if the number of tables in a query is large
 - used on PostgreSQL
 - The query-optimization phase
 - ▶ construct a query plan that is a tree of relational operators
 - ▶ operators represent a operation on sets of tuples
 - Crucial to the cost model is
 - ▶ estimation of the total number of tuples at each operator
 - ▶ is inferred on the basis of statistics maintained on each relation
 - ▶ The DBA must ensure that these statistics are current by running the analyze command periodically

PostgreSQL Query Executor

■ Query Executor

- Access methods
 - ▶ Sequential scans
 - Scanned sequentially from the first to last blocks
 - Each tuple is returned to the caller only if it is visible
 - ▶ Index scans
 - Given an index range, returns a set of matching tuples
- Join methods
 - ▶ Supports sort merge joins, nested-loop joins and a hybrid hash join
- Sort
- Aggregation

■ Triggers and Constraints

- Not implemented in the rewrite phase
- But implemented as part of the query executor
- When registered by the user, the details are associated with the catalog information
- The executor checks for candidate triggers and constraints if tuples are changed for update, delete and insert

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

System Architecture [1/2]

- System architecture follows the process-per-transaction model
- Postmaster
 - a central coordinating process
 - manages the PostgreSQL sites
 - is responsible for
 - ▶ initializing and shutting down the server
 - ▶ handling connection request from new clients
 - ▶ constantly listens for new connections on a known port
 - ▶ assigns each new client to a back-end server process
 - ▶ once assigns a client to a back-end server, the client interacts only with the back-end server

■ Client applications

- connect to the PostgreSQL server
- submit queries through one of the DB API

■ Back-end server process

- is responsible for executing the queries by the client
- can handle only a single query at a time
- to execute in parallel, an application must maintain multiple connections to the server
- back-end server can be executing concurrently
- access DB data through the main-memory buffer pool as a shared memory to have same view of data

Overview

- 27.1 Introduction
- 27.2 User Interfaces
- 27.3 SQL Variations and Extensions
- 27.4 Transaction Management in PostgreSQL
- 27.5 Storage and Indexing
- 27.6 Query Processing and Optimization
- 27.7 System Architecture
- Summary & Bibliographical Notes

Summary

- PostgreSQL is an open-source object-relational database management system
- Currently, PostgreSQL supports SQL92 and SQL:1999 and offers features such as complex queries, foreign keys, triggers, views, transactional integrity, and multiversion concurrency control
- In addition, users can extend PostgreSQL with new data types, functions, operators or index methods
- PostgreSQL works with a variety of programming languages(including C, C++, Java, Perl, Tcl, and Python)
- PostgreSQL boasts sophisticated features such as the Multi-Version Concurrency Control (MVCC), point in time recovery, tablespaces, asynchronous replication, nested transactions (savepoints), online/hot backups, a sophisticated query planner/optimizer, and write ahead log for fault tolerance
- It supports international character sets, multibyte character encodings, Unicode, and is locale-aware for sorting, case-sensitivity, and formatting
- It is highly scalable both in sheer quantity of data it can manage and in the number of concurrent users it can accommodate.
- There are active PostgreSQL systems in production environments that manage in excess of 4 terabytes of data

Bibliographical Notes [1/2]

- **Extensive on line documentation of PostgreSQL**

<http://www.postgresql.org>

- Until PostgreSQL version 8, the only way to run PostgreSQL under Microsoft Windows was by using Cygwin, details are at

<http://www.cygwin.com>

- **Books on PostgreSQL include**

Douglas and Douglas[2003]

Stinson

- Rules as used in PostgreSQL

Stonebraker et al.[1990]

- The Gist structure is described in

Hellerstein et al.[1995]

Bibliographical Notes [2/2]

- The PostgreSQL administration tools, pgAccess and pgAdmin are described at
<http://www.pgaccess.org>
<http://www.pgadmin.org>
- The PostgreSQL database design tools, TORA and Data Architect are described at
<http://www.globecom.se/tora>
<http://www.thekompany.com/products/dataarchitect>
- The report-generation tools GNU Report Generator and GNU Enterprise are described at
<http://www.gnu.org/software/grg>
<http://www.gnuenterprise.org>
- The Mondrian OLAP server is described at
<http://mondrian.sourceforge.net>