Postgres Plus Database Administration

Postgres Plus Advanced Server (PPAS) 9.5

Day 2 - Admin

Postgres Plus Advanced Server (PPAS) 9.5

Day 2 - 1
Transactions and Concurrency

Objectives

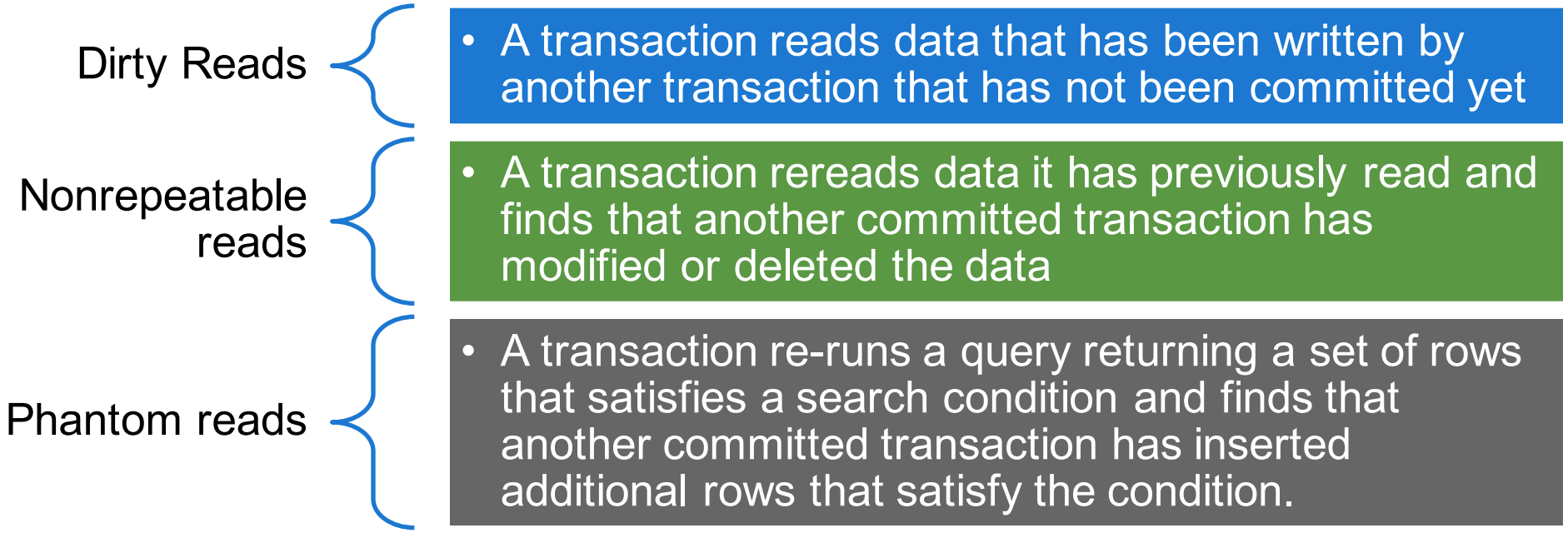
- This module will cover:
  - Transaction Definition
  - Effects of Concurrency on Transactions
  - Transaction Isolation Levels
  - Multi-Version Concurrency Control Overview (MVCC)
  - MVCC Example
  - Internal Identifiers
  - Transaction Wraparound
  - MVCC Maintenance
  - MVCC Demo

What is a Transaction?

- A transaction is set of statements bundled into a single step, all-or-nothing operation.
- A transaction must possess ACID properties:
  - An all-or-nothing operation (Atomicity).
  - Only valid data is written to the database (Consistency).
  - The intermediate states between the steps are not visible to other concurrent transactions (Isolation).
  - If some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all (Durability).

Transaction Isolation

- The ANSI/ISO SQL standard defines four levels of transaction isolation
- These isolation levels are defined in terms of three phenomena:



Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	*Possible*	*Possible*	*Possible*
Read committed	*Not possible*	*Possible*	*Possible*
Repeatable read	*Not possible*	*Not possible*	*Possible*
Serializable	*Not Possible*	*Not Possible*	*Not Possible*

Transaction Isolation Levels (Cont)

- Internally, only three distinct isolation levels are available: Read Committed, Repeatable Read, and Serializable.
- When you use Read Uncommitted level PostgreSQL will use Read Commited.
- Actual isolation level may be stricter than what you select.
- SQL Standard - the four isolation levels only define which phenomena must not happen, they do not define which phenomena must happen.
- PostgreSQL only provides three isolation levels sensibly map to MVCC model.

Transaction Isolation Lab

Lab

Consistency and Durability

- Only valid data is written to the database (Consistency).
- If some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all (Durability).

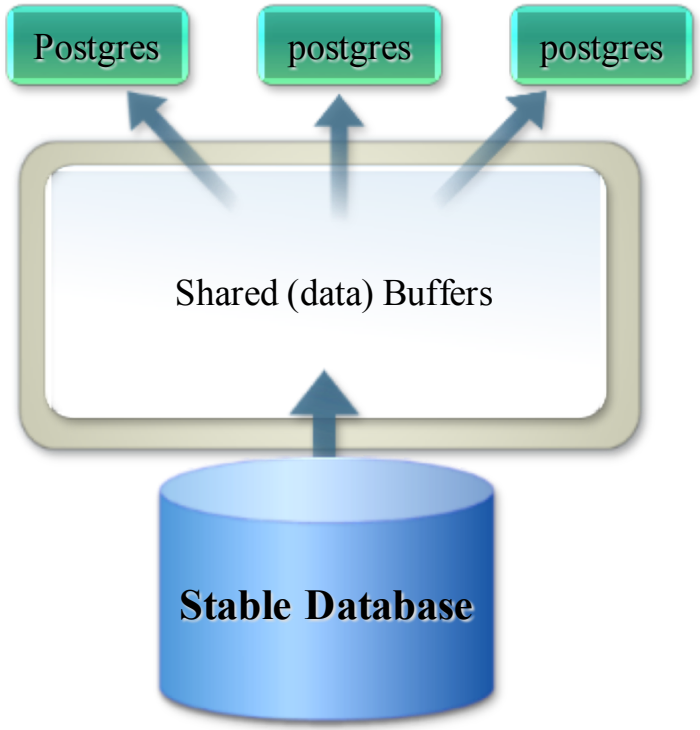
- How?
  - Transaction logging - WAL (Write Ahead Log) or Redo log in Oracle
  - Checkpoint

DB의 I/O 특성

- DB의 모든 데이터 접근은 memory를 통한다.
- Read
  - 읽으려는 블럭이 buffer cache에 있는지 확인
  - 없을 경우 메모리를 확보하고 해당 블럭을 메모리로 읽어 올린다.
  - 메모리에서 데이터를 읽는다.
- Write
  - 읽으려는 블럭이 buffer cache에 있는지 확인
  - 없을 경우 메모리를 확보하고 해당 블럭을 메모리로 읽어 올린다.
  - 변경 내용을 Log buffer에 기록한다.
  - 메모리의 데이터를 변경한다.
  - Commit시 Log buffer가 file에 완전히 기록된것을 확인한다.

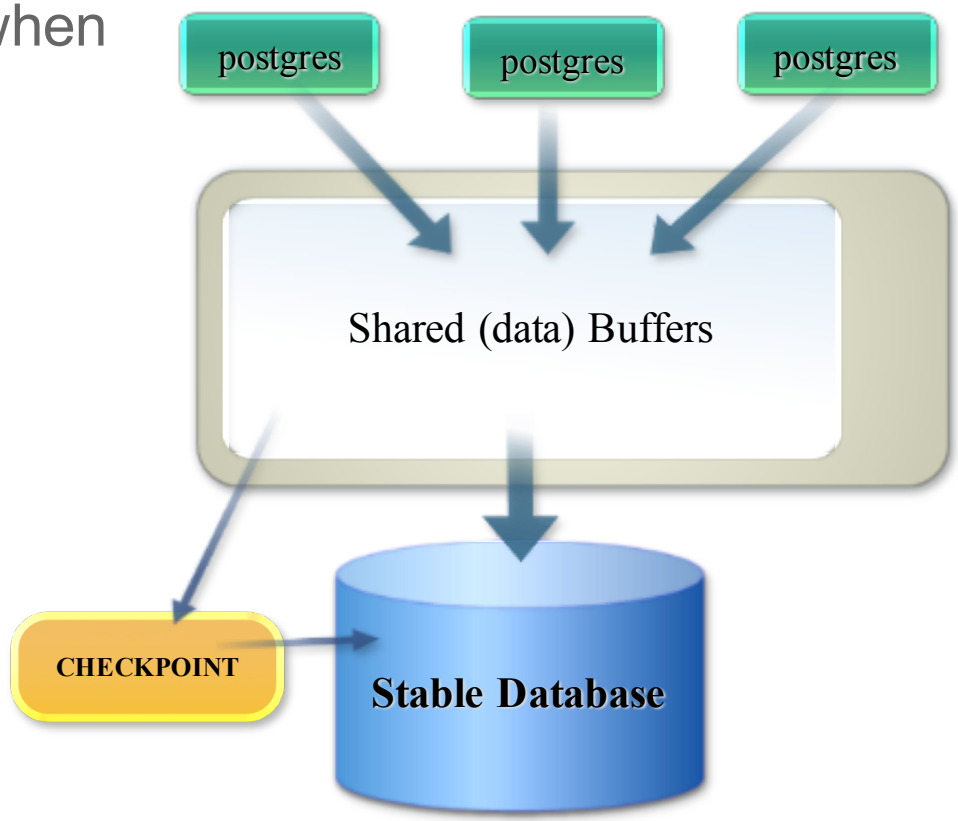
Disk Read Buffering

- PostgreSQL buffer cache (shared_buffers) reduces OS reads.
- Read the block once, then examine it many times in cache.
- There is no direct read.



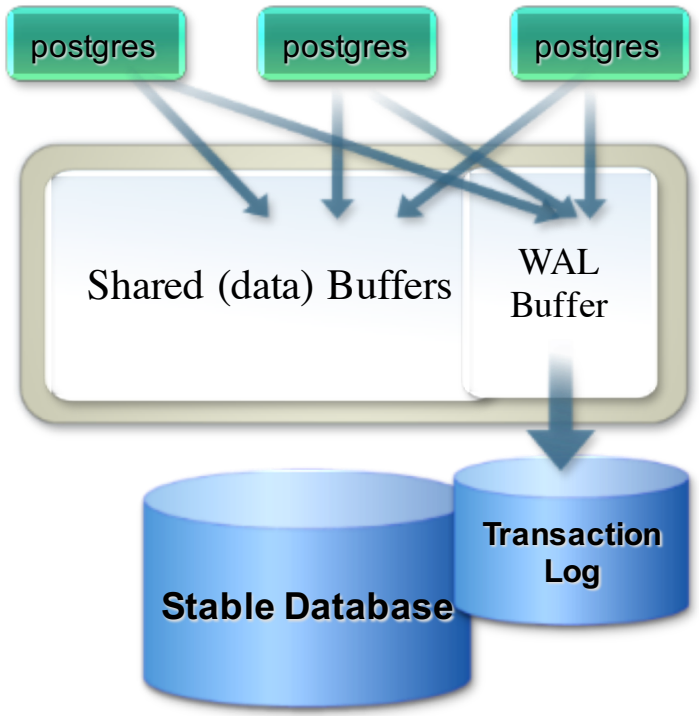
Disk Write Buffering

- Blocks are written to disk only when needed:
  - To make room for new blocks
  - At checkpoint time



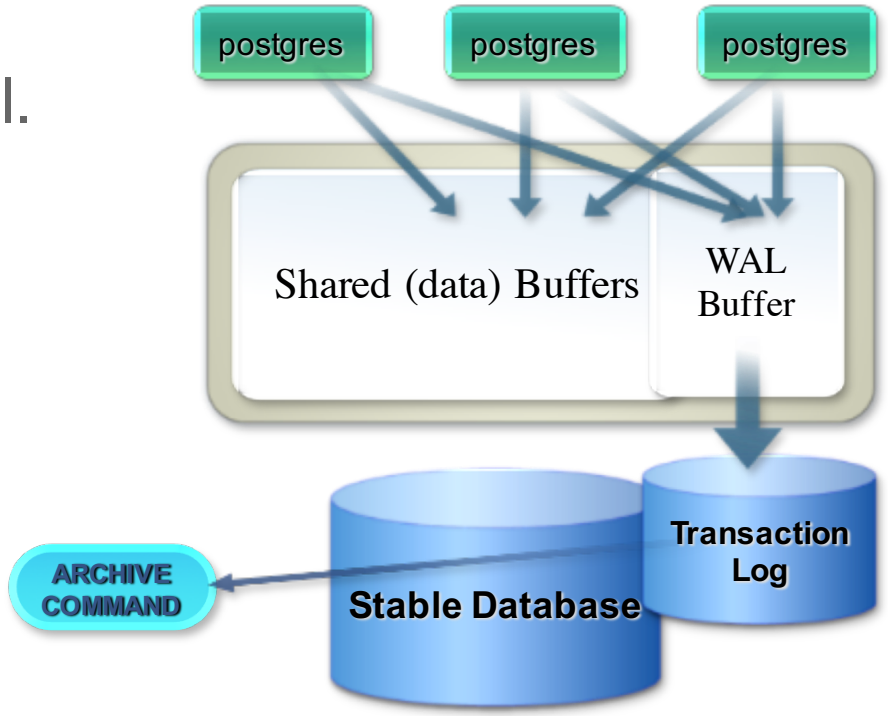
Write Ahead Logging (WAL)

- Backend write data to WAL buffers.
- Flush WAL buffers periodically (WAL writer), on commit, or when buffers are full.
- Group commit.



Transaction Log Archiving

- Archiver spawns a task to copy away pg_xlog log files when full.



Transaction Log

- Sequential 하게 기록된 DB의 변경 이력
- Commit 되기 전에 무조건 file에 완전히 기록됨
- Long transaction의 경우 commit 되기 전에도 주기적으로 기록 됨
- Point-In-Time-Recovery시 또는 Roll-Forward시 이 로그에 기록된 내용을 그대로 다시 수행
- 이 로그를 별도로 backup 해 두는 것을 archive log라고 함

Commit & Checkpoint

- Before Commit
  - Uncommitted updates are in memory
- After Commit
  - Committed updates written from shared memory to disk (write-ahead log file)
- After Checkpoint
  - Modified data pages are written from shared memory to the data files

Checkpoint의 필요성

- Commit은 transaction log가 완전히 기록된 것을 보장하지만 data file의 변경이 기록되는 것을 기다리지는 않는다.
- 경우에 따라서는 data file의 변경이 영원히 기록되지 않는 것도 가능 → Transaction log만 있으면 기존 data file로 부터 복구 가능
- 문제점
  - 모든 transaction log file을 계속 보관해야 함
  - DB를 재기동 하기 위해 많은 시간이 필요함

Checkpoint

- 주기적으로 buffer cache의 변경된 block(dirty buffer)을 file에 기록하고
- 모든 변경 내용이 기록된 transaction log 파일을 release 함으로써 해당 파일을 삭제할 수 있도록 해 준다.
- 복구시에 필요한 시간을 일정 수준 이하로 유지할 수 있도록 해 준다.

Checkpoint 주기

- Checkpoint의 주기를 조절 하는 것은 DB의 성능에 큰 영향을 미친다.
- 빈번한 checkpoint
  - Checkpoint를 빈번하게 수행하면 disk I/O가 많아 지기 때문에 전체적으로는 성능을 저하 시킬 가능성이 있다.
  - 장애시에 복구에 필요한 시간을 최소화 할 수 있음
- 긴 checkpoint 주기
  - Transaction log의 기록 만으로 변경이 완료 되기 때문에 처리 성능을 높일 수 있음
  - 복구에 많은 시간이 걸리며 유지해야 하는 transaction log의 양이 많아짐
  - Dirty buffer가 많아지게 되면 새로운 buffer를 할당하기 위해 어쩔 수 없이 disk에 변경을 기록하게 된다. 이 상황이 생기면 성능에 심각한 문제가 될 수 있다.

장애 복구

- 장애가 발생하여 DB가 재기동 되었다면
  - 그 때까지 commit된 내용은 wal log에 기록이 되어 있다.
  - 진행 중이던 uncommitted 변경 사항도 wal log에 기록이 되어 있을 수 있다.
  - Checkpoint 이전까지의 변경 사항은 data file에 기록이 안 되어 있을 수 있다.
- 장애시 수행되는 복구 작업
  1. Data file에 기록된 시점 부터 Wal log에 기록된 마지막 시점까지의 변경을 wal log의 내용 대로 재현한다. - Roll forward
  2. Wal log상에 아직 commit되지 않았던 트랜젝션을 roll back한다.

Concurrency and Transactions

- Concurrency – two or more sessions accessing the same data at the same time.
- Data consistency is maintained by using a Multiversion Concurrency Control(MVCC) model
- Each transaction sees snapshot of data (database version) as it was some time ago.
- Transaction isolation - Protects transaction from viewing “inconsistent” data (currently being updated by another transaction).
- MVCC Advantage – readers don’t block writers and writers don’t block readers.

Multi-Version Concurrency Control (MVCC)

- Snapshot of data at a point in time.
- Updates, inserts and deletes cause the creation of a new row version.
- Row version stored in same page.
- MVCC uses increasing transaction IDs to achieve consistency.
- Each row has 2 transaction ids: created and expired
- Queries check:
  - creation trans id is committed and < current trans counter
  - row lacks expire trans id or expire was in process at query start

Internal System Columns

- oid
  The object identifier (object ID) of a row. This column is only present if the table was created using WITH OIDS.
- tableoid
  The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies.
- xmin
  The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)
- cmin
  The command identifier (starting at zero) within the inserting transaction.
- xmax
  The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.
- cmax
  The command identifier within the deleting transaction, or zero.
- ctid
  The physical location of the row version within its table. Note that although the ctid can be used to locate the row version very quickly, a row's ctid will change if it is updated or moved by VACUUM FULL. Therefore ctid is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

Transaction Wraparound

- XIDs used to determine which row versions are “visible”.
- XIDs have limited size (32 bits) and counter limited to about 4 billion.
- Can wraparound to zero – transsactions in past now look to be in future, and invisible!
- Data corruption.
- This problem is also called transaction id wraparound failure.

MVCC Maintenance

- MVCC creates multiple versions of a row for concurrency.
- Old row versions can cause “bloat”.
- Rows no longer needed are recovered for reuse/removed via vacuuming or autovacuum.
- To prevent transaction wraparound failure each table must be vacuumed periodically.
- PostgreSQL reserves a special XID as FrozenXID.
- This XID is always considered older than every normal XID.

MVCC Demo

PSQL Terminal Session 1

```
drop table if exists mvc;

create a table and insert rows:

create table mvc(id numeric, name
  varchar);

insert into mvc
  values(1,'A'),(2,'B');


postgres=# select*from mvc;
 id | name
----+------
  1 | A
  2 | B
(2 rows)
```

PSQL Terminal Session 2

```
postgres=# begin;
BEGIN
postgres=# update mvc set
name='C' where id=2;
```

PSQL Terminal Session 1 – Can see old version of the row through MVCC

```
postgres=# select*from mvc;
 id | name
----+------
  1 | A
  2 | B
(2 rows)
```

MVCC Lab

Lab

Summary

- In this module you learned:
  - Transaction Definition
  - Effects of Concurrency on Transactions
  - Transaction Isolation Levels
  - Multi-Version Concurrency Control Overview (MVCC)
  - MVCC Example
  - Internal Identifiers
  - Transaction Wraparound
  - MVCC Maintenance
  - MVCC Demo

Day 2 - 2
Routine Maintenance Tasks

Objectives

- Database Maintenance
- Maintenance Tools
- Optimizer Statistics
- Data Fragmentation
- Routine Vacuuming
- Vacuuming Commands
- Preventing Transaction ID Wraparound Failures
- Vacuum Freeze
- The Visibility Map
- Vacuumdb
- Autovacuuming
- Per Table Thresholds
- Routine Reindexing
- CLUSTER

Database Maintenance

- Data files become fragmented as data is modified and deleted
- Database maintenance helps reconstruct the data files
- If done on time nobody notices but when not done everyone knows
- Must be done before you need it
- Improves performance of the database
- Saves database from transaction ID wraparound failures

Maintenance Tools

- Maintenance thresholds can be configured using the PEM Client and PEM Server
- PPAS maintenance thresholds can be configured in **postgresql.conf**
- Manual scripts can be written to watch stat tables like `pg_stat_user_tables`
- Maintenance commands:
  - `ANALYZE`
  - `VACUUM`
  - `CLUSTER`
- Maintenance command `vacuumdb` can be run from OS prompt
- Autovacuum can help in automatic database maintenance

Optimizer Statistics

- Optimizer statistics play a vital role in query planning
- Not updated in real time
- Collect information for relations including size, row counts, average row size and row sampling
- Stored permanently in catalog tables
- The maintenance command `ANALYZE` updates the statistics
- Thresholds can be set using PEM Client to alert you when statistics are not collected on time

Example - Updating Statistics

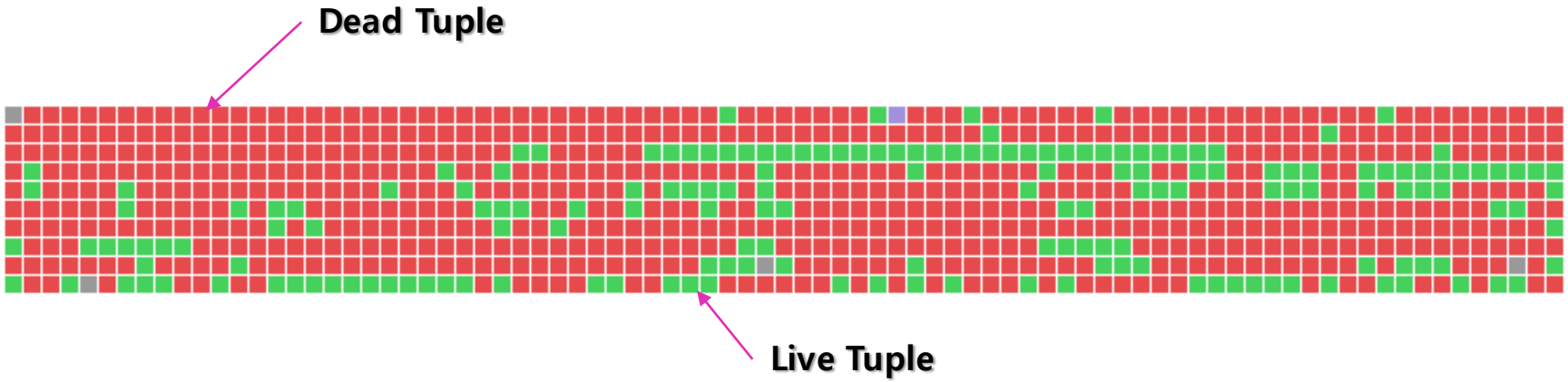
```
[postgres@localhost ~]$ psql edb
psql.bin (9.4.0.1)
Type "help" for help.
edb=# CREATE TABLE testanalyze(id number, name varchar2);
CREATE TABLE
edb=# INSERT into testanalyze values(generate_series(1,10000),'Sample');
INSERT 0 10000
edb=# SELECT relname,reltuples FROM pg_class WHERE relname='testanalyze';
   relname   | reltuples
-------------+-----------
 testanalyze |         0
(1 row)
edb=# ANALYZE testanalyze;
ANALYZE
edb=# SELECT relname,reltuples FROM pg_class WHERE relname='testanalyze';
   relname   | reltuples
-------------+-----------
 testanalyze |     10000
(1 row)
```

Data Fragmentation and Bloat

- Data is stored in data file pages
- An update or delete of a row does not immediately remove the row from the disk page
- Eventually this row space becomes obsolete and causes fragmentation and bloating
- Set PEM Alert for notifications

Routine Vacuuming

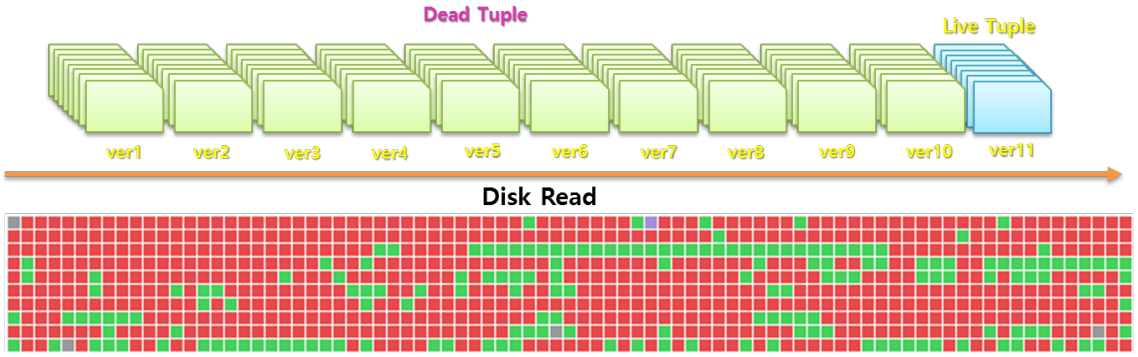
- Obsoleted rows can be removed or reused using vacuuming
- Helps in shrinking data file size when required
- Vacuuming can be automated using autovacuum
- The `VACUUM` command locks tables in access exclusive mode
- Long running transactions may block vacuuming thus it should be done during low usage times



Vacuuming Commands

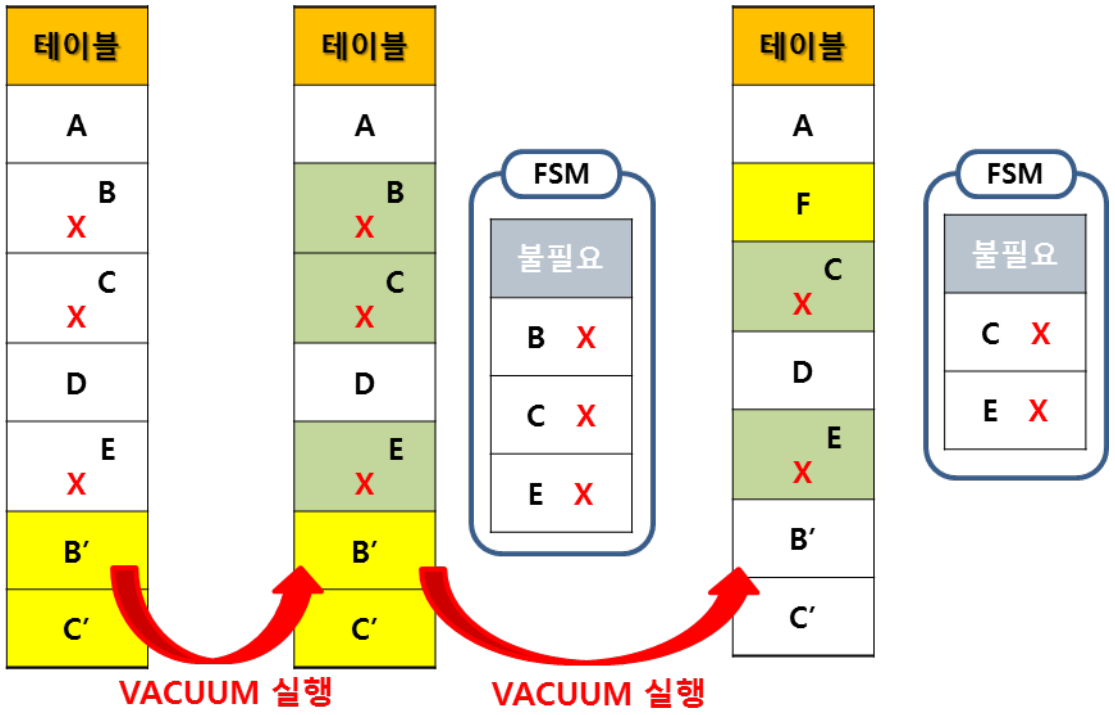
- When executed, the VACUUM command:
  - Can recover or reuse disk space occupied by obsolete rows
  - Updates data statistics
  - Updates the visibility map, which speeds up index-only scans
  - Protects against loss of very old data due to transaction ID wraparound

- The VACUUM command can be run in two modes:
  - VACUUM
  - VACUUM FULL



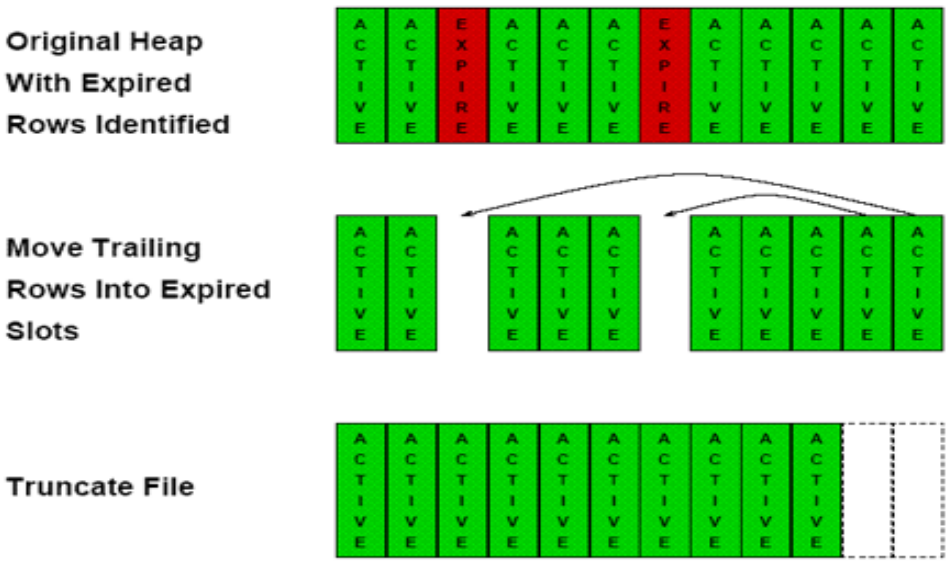
Vacuuming Commands

- VACUUM
  - Removes dead rows and marks the space available for future reuse
  - Does not return the space to the operating system
  - Space is reclaimed if obsolete rows are at the end of a table



Vacuuming Commands

- `VACUUM FULL`
  - More aggressive algorithm compared to `VACUUM`
  - Compacts tables by writing a complete new version of the table file with no dead space
  - Takes more time
  - Requires extra disk space for the new copy of the table, until the operation completes



VACUUM Syntax

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ table_name [ (column_name [, ...] ) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_name ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table_name [ (column_name [, ...] ) ] ]
```

Parameters

- **FULL**
  Selects "full" vacuum, which can reclaim more space, but takes much longer and exclusively locks the table. This method also requires extra disk space, since it writes a new copy of the table and doesn't release the old copy until the operation is complete. Usually this should only be used when a significant amount of space needs to be reclaimed from within the table.

- **FREEZE**
  Selects aggressive "freezing" of tuples. Specifying FREEZE is equivalent to performing VACUUM with the vacuum_freeze_min_age and vacuum_freeze_table_age parameters set to zero. Aggressive freezing is always performed when the table is rewritten, so this option is redundant when FULL is specified.

- **VERBOSE**
  Prints a detailed vacuum activity report for each table.

- **ANALYZE**
  Updates statistics used by the planner to determine the most efficient way to execute a query.

- ***table_name***
  The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

- ***column_name***
  The name of a specific column to analyze. Defaults to all columns. If a column list is specified, `ANALYZE` is implied.

Example - Vacuuming

```
edb=# create table testvac (id numeric, name varchar);
CREATE TABLE
edb=# insert into testvac values(generate_series(1, 10000), 'Sample');
INSERT 0 10000
edb=# select pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 440 kB
(1 row)

edb=# update testvac set name = 'Sample2';
UPDATE 10000
edb=# select pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 872 kB
(1 row)
```

Example - Vacuuming

```
edb=# vacuum verbose testvac;
INFO:  vacuuming "enterprisedb.testvac"
INFO:  "testvac": removed 10000 row versions in 55 pages
INFO:  "testvac": found 10000 removable, 10000 nonremovable row versions in 109 out of 109 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  vacuuming "pg_toast.pg_toast_16814"
INFO:  index "pg_toast_16814_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "pg_toast_16814": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

Example - Vacuuming

```
edb=# update testvac set name = 'Sample';
UPDATE 10000
edb=# select pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 872 kB
(1 row)

edb=# vacuum full verbose testvac;
INFO:  vacuuming "enterprisedb.testvac"
INFO:  "testvac": found 10000 removable, 10000 nonremovable row versions in 109 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
edb=# select pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 440 kB
(1 row)
```

Preventing Transaction ID Wraparound Failures

- MVCC depends on transaction ID numbers
- Transaction IDs have limited size (32 bits at this writing)
- A cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound
- This causes a catastrophic data loss
- To avoid this problem, every table in the database must be vacuumed at least once for every two billion transactions

Vacuum Freeze

- `VACUUM FREEZE` will mark rows as frozen
- Postgres reserves a special XID, `FrozenTransactionId`
- `FrozenTransactionId` is always considered older than every normal XID
- `VACUUM FREEZE` replaces transaction IDs with `FrozenTransactionId`, thus rows will appear to be “in the past”
- `vacuum_freeze_min_age` controls when a row will be frozen
- `VACUUM` normally skips pages without dead row versions but some rows may need `FREEZE`
- `vacuum_freeze_table_age` controls when whole table must be scanned

The Visibility Map

- Each heap relation has a Visibility Map which keeps track of which pages contain only tuples
- Stored at `<relfilenode>_vm`
- Helps vacuum to determine whether pages contain dead rows
- Can also be used by index-only scans to answer queries
- VACUUM command updates the visibility map
- The visibility map is vastly smaller, so can be cached easily

```
[postgres@edbkorea 17088]$ ls
12735      12797_vm   12856      12949      17132      17272_fsm  17418      17565      17720      17863  17991  18076
12735_fsm  12799      12857      12953      17134      17276      17420      17567      17724      17865  17993  18077
12735_vm   12800      12859      12953_fsm  17135      17278      17421      17571      17726      17867  17995  18078
12737      12801      12860      12953_vm   17137      17279      17423      17573      17727      17869  17997  18079
12737_fsm  12801_fsm  12861      12955      17137_fsm  17281      17427      17574      17729      17871  17999  18080
12737_vm   12801_vm   12862      12956      17141      17285      17429      17576      17729_fsm  17873  18001  18081
12739      12803      12862_fsm  12957      17143      17287      17429_fsm  17580      17733      17875  18003  18082
12740      12804      12862_vm   12957_fsm  17144      17287_fsm  17433      17582      17735      17877  18005  18083
```

vacuumdb

- The `VACUUM` command has a command-line executable wrapper called `vacuumdb`
- `vacuumdb` can `VACUUM` all databases using a single command

```
Usage:
  vacuumdb [OPTION]... [DBNAME]

Options:
  -a, --all                       vacuum all databases
  -d, --dbname=DBNAME             database to vacuum
  -e, --echo                      show the commands being sent to the server
  -f, --full                      do full vacuuming
  -F, --freeze                    freeze row transaction information
  -q, --quiet                     don't write any messages
  -t, --table='TABLE[(COLUMNS)]'  vacuum specific table(s) only
  -v, --verbose                   write a lot of output
  -V, --version                   output version information, then exit
  -z, --analyze                   update optimizer statistics
  -Z, --analyze-only              only update optimizer statistics
      --analyze-in-stages         only update optimizer statistics, in multiple
                                  stages for faster results
  -?, --help                      show this help, then exit
```

Autovacuuming

- Highly recommended feature of Postgres
- It automates the execution of `VACUUM`, `FREEZE` and `ANALYZE` commands
- Autovacuum consists of a launcher and many worker processes
- A maximum of `autovacuum_max_workers` worker processes are allowed
- Launcher will start one worker within each database every `autovacuum_naptime` seconds
- Workers check for inserts, updates and deletes and execute `VACUUM` and/or `ANALYZE` as needed
- `track_counts` must be set to `TRUE` as autovacuum depends on statistics
- Temporary tables cannot be accessed by autovacuum

Autovacuuming Parameters

```
psql.bin (9.4.0.1)
Type "help" for help.

edb=# select name,setting from pg_settings where name like 'autovacuum%';
                 name                  |  setting
---------------------------------------+-----------
 autovacuum                            | on
 autovacuum_analyze_scale_factor       | 0.1
 autovacuum_analyze_threshold          | 50
 autovacuum_freeze_max_age             | 200000000
 autovacuum_max_workers                | 3
 autovacuum_multixact_freeze_max_age   | 400000000
 autovacuum_naptime                    | 60
 autovacuum_vacuum_cost_delay          | 20
 autovacuum_vacuum_cost_limit          | -1
 autovacuum_vacuum_scale_factor        | 0.2
 autovacuum_vacuum_threshold           | 50
 autovacuum_work_mem                   | -1
(12 rows)
```

vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples

analyze threshold = analyze base threshold + analyze scale factor * number of tuples

Per-Table Thresholds

- Autovacuum workers are resource intensive
- Table-by-table `autovacuum` parameters can be configured for large tables
- Configure the following parameters using `ALTER TABLE` or `CREATE TABLE`:
  - `autovacuum_enabled`
  - `autovacuum_vacuum_threshold`
  - `autovacuum_vacuum_scale_factor`
  - `autovacuum_analyze_threshold`
  - `autovacuum_analyze_scale_factor`
  - `autovacuum_freeze_max_age`
  - `autovacuum_vacuum_cost_delay`
  - `autovacuum_vacuum_cost_limit`

Routine Reindexing

- Indexes are used for faster data access
- `UPDATE` and `DELETE` on a table modify underlying index entries
- Indexes are stored on data pages and become fragmented over time
- `REINDEX` rebuilds an index using the data stored in the index's table
- Time required depends on:
  - Number of indexes
  - Size of indexes
  - Load on server when running command

Routine Reindexing

- There are several reasons to use `REINDEX`:
  - An index has become "bloated", meaning it contains many empty or nearly-empty pages
  - You have altered a storage parameter (such as fillfactor) for an index
  - An index built with the `CONCURRENTLY` option failed, leaving an "invalid" index
- Syntax:

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } name [ FORCE ]
```

CONCURRENTLY

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it's done.

```
edb=# create index testvac_idx1 on testvac(id);
CREATE INDEX
edb=# create index CONCURRENTLY testvac_idx2 on testvac(id);
CREATE INDEX
edb=# \d testvac
      Table "enterprisedb.testvac"
 Column |       Type        | Modifiers
--------+-------------------+-----------
 id     | numeric           |
 name   | character varying |
Indexes:
    "testvac_idx1" btree (id)
    "testvac_idx2" btree (id)

edb=# drop index testvac_idx1;
DROP INDEX
```

CLUSTER

- Sort data physically according to the specified index
- One time operations and changes are not clustered
- An `ACCESS EXCLUSIVE` lock is acquired
- When some data is accessed frequently and can be grouped using an index, `CLUSTER` is helpful
- `CLUSTER` lowers disk accesses and speeds up the query when accessing a range of indexed values
- Setting `maintenance_work_mem` to a large values is recommended before running `CLUSTER`
- Run `ANALYZE` afterwards

Example - CLUSTER

```
edb=# create table testcluster(id  numeric,
name varchar(10));
CREATE TABLE
edb=# insert into testcluster values (2, 'B'),
(1, 'A'), (3, 'A'), (5, 'C'), (4, 'B');
INSERT 0 5
edb=# select * from testcluster;
 id | name
----+------
  2 | B
  1 | A
  3 | A
  5 | C
  4 | B
(5 rows)
```

```
edb=# cluster testcluster using testcluster_idx1;
CLUSTER
edb=# select * from testcluster;
 id | name
----+------
  1 | A
  2 | B
  3 | A
  4 | B
  5 | C
(5 rows)
```

Lab

Summary

- Database Maintenance
- Maintenance Tools
- Optimizer Statistics
- Data Fragmentation
- Routine Vacuuming
- Vacuuming Commands
- Preventing Transaction ID Wraparound Failures
- Vacuum Freeze
- The Visibility Map
- Vacuumdb
- Autovacuuming
- Per Table Thresholds
- Routine Reindexing
- CLUSTER

Day 2 - 3
Extensions

Objectives

- pgcrypto
- pg_buffercache
- pg_freespacemap
- pageinspect
- pg_prewarm
- pg_stat_statements
- pg_repack

Extensions

- PostgreSQL은 매우 확장성이 높은 DB
- 스키마 구조 이외에도 data type, operation등 더 많은 정보가 catalog에 저장됨
  - pg_type, pg_operator, pg_opclass …
- 사용자가 동작 방식을 변경할 수 있고 type을 확장 할 수도 있음
- Extension
  - PostgreSQL의 확장 기능 package 방식
  - 표준화된 개발 환경 (PGXS)
  - 간단한 설치 및 관리
  - http://pgxn.org

pgcrypto

The pgcrypto module provides cryptographic functions for PostgreSQL.

Functionality	Built-in	With OpenSSL
MD5	Yes	Yes
SHA1	Yes	Yes
SHA224/256/384/512	Yes	Yes
Other digest algorithms	No	Yes
Blowfish	Yes	Yes
AES	Yes	Yes
DES/3DES/CAST5	No	Yes
Raw encryption	Yes	Yes
PGP Symmetric encryption	Yes	Yes
PGP Public-Key encryption	Yes	Yes

Lab

pg_buffercache

provides a means for examining what's happening in the shared buffer cache in real time

Column Name	Type	References	Description
bufferid	integer		ID, in the range 1..shared_buffers
relfilenode	oid	pg_class.relfilenode	Filenode number of the relation
reltablespace	oid	pg_tablespace.oid	Tablespace OID of the relation
reldatabase	oid	pg_database.oid	Database OID of the relation
relforknumber	smallint		Fork number within the relation; see include/storage/relfilenode.h
relblocknumber	bigint		Page number within the relation
isdirty	boolean		Is the page dirty?
usagecount	smallint		Clock-sweep access count
pinning_backends	integer		Number of backends pinning this buffer

pg_buffercache

```
regression=#  SELECT c.relname, count(*) AS buffers
              FROM pg_buffercache b INNER JOIN pg_class c
              ON b.relfilenode = pg_relation_filenode(c.oid) AND
                 b.reldatabase IN (0, (SELECT oid FROM pg_database
                                       WHERE datname = current_database()))
              GROUP BY c.relname
              ORDER BY 2 DESC
              LIMIT 10;

              relname               | buffers
------------------------------------+---------
 tenk2                             |     345
 tenk1                             |     141
 pg_proc                           |      46
 pg_class                          |      45
 pg_attribute                      |      43
 pg_class_relname_nsp_index        |      30
 pg_proc_proname_args_nsp_index    |      28
 pg_attribute_relid_attnam_index   |      26
 pg_depend                         |      22
 pg_depend_reference_index         |      20
(10 rows)
```

Lab

pg_freespacemap

provides a means for examining the free space map (FSM)

- **`pg_freespace(rel regclass IN, blkno bigint IN) returns int2`**
  - Returns the amount of free space on the page of the relation, specified by blkno, according to the FSM.
- **`pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)`**
  - Displays the amount of free space on each page of the relation, according to the FSM. A set of (blkno bigint, avail int2) tuples is returned, one tuple for each page in the relation.

pg_freespacemap

```
edb=# select * from pg_freespace('mytab');
 blkno  | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |   288
(4 rows)

edb=# select * from pg_freespace('mytab',  3);
 pg_freespace
--------------
          288
(1 row)
```

Lab

pageinspect

provides functions that allow you to inspect the contents of database pages at a low level

- **`get_raw_page(relname text, fork text, blkno int) returns bytea`**
  - get_raw_page reads the specified block of the named relation and returns a copy as a bytea value. This allows a single time-consistent copy of the block to be obtained. fork should be 'main' for the main data fork, 'fsm' for the free space map, 'vm' for the visibility map, or 'init' for the initialization fork.
- **`get_raw_page(relname text, blkno int) returns bytea`**
  - A shorthand version of get_raw_page, for reading from the main fork. Equivalent to get_raw_page(relname, 'main', blkno)

pageinspect

- **page_header(page bytea) returns record**
  - page_header shows fields that are common to all PostgreSQL heap and index pages.
  - A page image obtained with get_raw_page should be passed as argument.

```
test=# SELECT * FROM page_header(get_raw_page('pg_class',  0));
    lsn     | checksum | flags  | lower | upper | special | pagesize | version | prune_xid
------------+----------+--------+-------+-------+---------+----------+---------+-----------
 0/24A1B50  |        1 |      1 |   232 |   368 |    8192 |     8192 |       4 |         0
```

- **heap_page_items(page bytea) returns setof record**
  - heap_page_items shows all line pointers on a heap page. For those line pointers that are in use, tuple headers are also shown. All tuples are shown, whether or not the tuples were visible to an MVCC snapshot at the time the raw page was copied.
  - A heap page image obtained with get_raw_page should be passed as argument. For example:

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class',  0));
```

pageinspect

B-tree index inspection

- **`bt_metap(relname text) returns record`**
  - bt_metap returns information about a B-tree index's metapage.
- **`bt_page_stats(relname text, blkno int) returns record`**
  - bt_page_stats returns summary information about single pages of B-tree indexes.
- **`bt_page_stats(relname text, blkno int) returns record`**
  - bt_page_stats returns summary information about single pages of B-tree indexes
- **`bt_page_items(relname text, blkno int) returns setof record`**
  - bt_page_items returns detailed information about all of the items on a B-tree index page.

pageinspect

BRIN index inspection

- **`brin_page_type(page bytea) returns text`**
  - brin_page_type returns the page type of the given BRIN index page, or throws an error if the page is not a valid BRIN page.
- **`brin_metapage_info(page bytea) returns record`**
  - brin_metapage_info returns assorted information about a BRIN index metapage.
- **`brin_revmap_data(page bytea) returns setof tid`**
  - brin_revmap_data returns the list of tuple identifiers in a BRIN index range map page.
- **`brin_page_items(page bytea, index oid) returns setof record`**
  - brin_page_items returns the data stored in the BRIN data page.

pageinspect

GIN index and FSM inspection

- **`gin_metapage_info(page bytea) returns record`**
  - gin_metapage_info returns information about a GIN index metapage.
- **`gin_page_opaque_info(page bytea) returns record`**
  - gin_page_opaque_info returns information about a GIN index opaque area, like the page type.
- **`gin_leafpage_items(page bytea) returns setof record`**
  - gin_leafpage_items returns information about the data stored in a GIN leaf page.
- **`fsm_page_contents(page bytea) returns text`**
  - fsm_page_contents shows the internal node structure of a FSM page.
  - The output is a multiline string, with one line per node in the binary tree within the page. Only those nodes that are not zero are printed. The so-called "next" pointer, which points to the next slot to be returned from the page, is also printed.

Lab

pg_prewarm

provides a convenient way to load relation data into buffer cache.

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8
```

- **mode** : prewarming method to be used
  - **prefetch** : makes kernel, asynchronously, read the file. I.e. in background. This will put content of the file in kernel cache for disk, and it works in background. But – it doesn't work on all platforms, so it can error out.
  - **read** : end result is as with prefetch, but it's done synchronously, and will work everywhere.
  - **buffer** : this will actually make PG load the data to shared_buffers – that is to the closest (to PG) level of cache. That's also the default.

pg_prewarm

```
edb=# select pg_prewarm('testvac');
 pg_prewarm
------------
        109
(1 row)

edb=# SELECT c.relname, count(*) AS buffers
edb-#               FROM pg_buffercache b INNER JOIN pg_class c
edb-#               ON b.relfilenode = pg_relation_filenode(c.oid) AND
edb-#                  b.reldatabase IN (0, (SELECT oid FROM pg_database
edb(#                                        WHERE datname = current_database()))
edb-#               GROUP BY c.relname
edb-#               ORDER BY 2 DESC
edb-#               LIMIT 10;
             relname                | buffers
------------------------------------+---------
 testvac                            |     109
 ...
 ...
(10 rows)
```

Lab

pg_stat_statements

provides a means for tracking execution statistics of all SQL statements executed by a server.

- Must be loaded by adding pg_stat_statements to shared_preload_libraries in postgresql.conf, because it requires additional shared memory.
  - `shared_preload_libraries = '…,$libdir/pg_stat_statements'`
- Tracks statistics across all databases of the server.
- For security reasons, non-superusers are not allowed to see the SQL text or queryid of queries executed by other users
  - They can see the statistics, however, if the view has been installed in their database.
- Additional shared memory proportional to pg_stat_statements.max.
  - Note that this memory is consumed whenever the module is loaded, even if pg_stat_statements.track is set to none.

pg_stat_statements - view

Name	Type	Description
userid	oid	OID of user who executed the statement
dbid	oid	OID of database in which the statement was executed
queryid	bigint	Internal hash code, computed from the statement's parse tree
query	text	Text of a representative statement
calls	bigint	Number of times executed
total_time	double precision	Total time spent in the statement, in milliseconds
min_time	double precision	Minimum time spent in the statement, in milliseconds
max_time	double precision	Maximum time spent in the statement, in milliseconds
mean_time	double precision	Mean time spent in the statement, in milliseconds
stddev_time	double precision	Population standard deviation of time spent in the statement, in milliseconds
rows	bigint	Total number of rows retrieved or affected by the statement
shared_blks_hit	bigint	Total number of shared block cache hits by the statement
shared_blks_read	bigint	Total number of shared blocks read by the statement
shared_blks_dirtied	bigint	Total number of shared blocks dirtied by the statement
shared_blks_written	bigint	Total number of shared blocks written by the statement
local_blks_hit	bigint	Total number of local block cache hits by the statement
local_blks_read	bigint	Total number of local blocks read by the statement
local_blks_dirtied	bigint	Total number of local blocks dirtied by the statement
local_blks_written	bigint	Total number of local blocks written by the statement
temp_blks_read	bigint	Total number of temp blocks read by the statement
temp_blks_written	bigint	Total number of temp blocks written by the statement
blk_read_time	double precision	Total time the statement spent reading blocks, in milliseconds (if track_io_timing is enabled, otherwise zero)
blk_write_time	double precision	Total time the statement spent writing blocks, in milliseconds (if track_io_timing is enabled, otherwise zero)

pg_stat_statements

- **`pg_stat_statements_reset() returns void`**
  - pg_stat_statements_reset discards all statistics gathered so far by pg_stat_statements. By default, this function can only be executed by superusers.

pg_stat_statements

- **`pg_stat_statements.max`** (integer): default is 5000
  - Maximum number of statements tracked by the module.
- **`pg_stat_statements.track`** (enum): default is top
  - Controls which statements are counted by the module.
  - top : to track top-level statements (those issued directly by clients)
  - all : to also track nested statements (such as statements invoked within functions)
  - none : to disable statement statistics collection.
- **`pg_stat_statements.track_utility`** (boolean): default is on
  - Controls whether utility commands are tracked by the module. Utility commands are all those other than SELECT, INSERT, UPDATE and DELETE.
- **`pg_stat_statements.save`** (boolean): default is on
  - Specifies whether to save statement statistics across server shutdowns. If it is off then statistics are not saved at shutdown nor reloaded at server start. The default value is on.

pg_stat_statements

```
bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 3;
-[ RECORD 1 ]---------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + ? WHERE bid = ?;
calls       | 3000
total_time  | 9609.00100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]---------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + ? WHERE tid = ?;
calls       | 3000
total_time  | 8015.156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]---------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
total_time  | 310.624
rows        | 100000
hit_percent | 0.303951367781155015 20
```

Lab

pg_repack

CLUSTER and VACUUM FULL without exclusive lock

- lets you remove bloat from tables and indexes, and optionally restore the physical order of clustered indexes.
- Unlike CLUSTER and VACUUM FULL it works online, without holding an exclusive lock on the processed tables during processing.
  - Online CLUSTER (ordered by cluster index)
  - Ordered by specified columns
  - Online VACUUM FULL (packing rows only)
  - Rebuild or relocate only the indexes of a table
- Notice
  - Only superusers can use the utility.
  - Target table must have a PRIMARY KEY, or at least a UNIQUE total index on a NOT NULL column.

pg_repack

Installation

- pg_repack can be built with make on UNIX or Linux. The PGXS build framework is used automatically. Before building, you might need to add the directory containing pg_config to your $PATH. Then you can run:

```
$ cd pg_repack
$ make
$ sudo make install
$ psql -c "CREATE EXTENSION pg_repack" -d your_database
$ psql -f "$(pg_config --sharedir)/contrib/pg_repack.sql" -d
your_database
```

pg_repack

- Perform an online CLUSTER of all the clustered tables in the database test, and perform an online VACUUM FULL of all the non-clustered tables:

```
$ pg_repack test
```

- Perform an online VACUUM FULL on the tables foo and bar in the database test (an eventual cluster index is ignored):

```
$ pg_repack --no-order --table foo --table bar test
```

- Move all indexes of table foo to tablespace tbs:

```
$ pg_repack -d test --table foo --only-indexes --tablespace tbs
```

- Move the specified index to tablespace tbs:

```
$ pg_repack -d test --index idx --tablespace tbs
```

Lab

Summary

- Extension
- pgcrypto
- pg_buffercache
- pg_freespacemap
- pageinspect
- pg_prewarm
- pg_stat_statements
- pg_repack

Day 2 - 4
Monitoring

Module Objectives

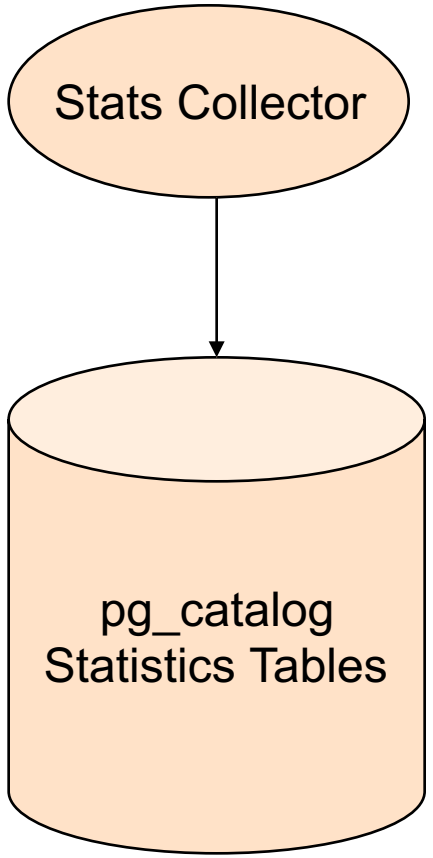
- Database Monitoring
- Database Statistics
- The Statistics Collector
- Database Statistic Tables
- Operating System Process Monitoring (Unix)
- Server Processes – Linux
- Current Sessions and Locks
- Logging Slow Running Queries
- Disk Usage
- Postgres Enterprise Manager (PEM)
- PEM - Features
- PEM - Architecture

Database Monitoring

- Database monitoring consists of capturing and recording the database events.
- This information helps DBAs to detect, identify and fix potential database performance issues.
- Database monitoring statistics make it easy for DBAs to monitor the health and performance of PPAS databases.
- Several tools are available for monitoring database activity and analyzing performance.

Database Statistics

- Database statistics catalog tables store database activity information:
  - Current running sessions
  - Running SQL
  - Locks
  - DML counts
  - Row counts
  - Index usage



The Statistics Collector

- Stats Collector is a process that collects and reports information about database activities
- Stats Collector adds some overhead to query execution
- Stats Parameters:
  - `track_counts` - controls table and index statistics
  - `track_activities` - enables monitoring of the current command
- The stats collector uses temp files stored in subdirectory `pg_stat_tmp`
- Permanent statistics are stored in the `pg_catalog` schema in the global subdirectory
- **Performance Tip:** parameter `stats_temp_directory` can be pointed at a RAM-based file system

Database Statistic Tables

- `pg_catalog` schema contains a set of tables, views and functions which store and report database statistics
- `pg_class` and `pg_stats` catalog tables store all statistics
- `pg_stat_database` view can be used to view stats information about a database
- `pg_stat_bgwriter` shows background writer stats
- `pg_stat_user_tables` shows information about activities on a table like inserts, updates, deletes, vacuum, autovacuum etc.
- `pg_stat_user_indexes` shows information about index usage for all user tables

Operating System Process Monitoring (Unix)

Unix

- ps – Information about current processes
  - `ps waux | grep post` (for Linux and OS/X)
  - `ps -ef | grep post` (other Unix)

- netstat – Information about current network connections
  - `netstat -an | grep LISTEN`

Server Processes - Linux

```
edbkorea:[/opt/PostgresPlus/9.4AS ]ps -ef |grep postgres
501  2106     1  0 08:32 ?     00:00:00 /opt/PostgresPlus/9.4AS/bin/edb-postgres -D /opt/PostgresPlus/9.4AS/data
501  2108  2106  0 08:32 ?     00:00:00 postgres: logger process
501  2110  2106  0 08:32 ?     00:00:01 postgres: checkpointer process
501  2111  2106  0 08:32 ?     00:00:00 postgres: writer process
501  2112  2106  0 08:32 ?     00:00:00 postgres: wal writer process
501  2113  2106  0 08:32 ?     00:00:00 postgres: autovacuum launcher process
501  2114  2106  0 08:32 ?     00:00:00 postgres: archiver process   last was 000000020000000000000013
501  2115  2106  0 08:32 ?     00:00:01 postgres: stats collector process
501  5043  2106  0 09:13 ?     00:00:00 postgres: edbstore edbstore [local] idle
501  5942  2106  0 11:08 ?     00:00:00 postgres: enterprisedb edb 192.168.236.1[63290] idle
501  5943  2106  0 11:08 ?     00:00:02 postgres: enterprisedb edbstore 192.168.236.1[63291] idle
501  7242  2106  0 13:32 ?     00:00:00 postgres: enterprisedb edb [local] idle
501  7790  2420  0 14:24 pts/0 00:00:00 grep postgres
```

Current Sessions and Locks

- `pg_locks` - system table that stores information about outstanding locks in lock manager
- `pg_stat_activity` - shows the process id, database, user, query string and start time of currently executing query
  - Run the following SQL command:
    - `SELECT * FROM pg_stat_activity;`
- Terminate a session gracefully using the `pg_terminate_backend` function
- Rollback a transaction gracefully using function `pg_cancel_backend`

Logging Slow Running Queries

- `log_min_duration_statement`
  - Sets a minimum statement execution time (in milliseconds) that causes a statement to be logged. All SQL statements that run for the time specified or longer will be logged with their duration.
  - Setting this to `0` will print all queries and their durations.
  - A setting of `-1` (the default) disables the feature.
  - Enabling this option can be useful in tracking down non-optimized queries in your applications.

99

Log Monitoring

- data/pg_log 디렉토리 밑에 날짜별로 생성 됨.
- DB 재기동시 다시 생성됨
- Log 형식과 파일명, 생성 위치등은 *postgresql.conf* 파일에서 변경 가능
- 상황에 따라 크기가 매우 커질 수 있으므로 관리에 주의를 기울여야 함

```
edbkorea:[/opt/PostgresPlus/9.4AS/data/pg_log ]ls -al
total 68
drwxr-xr-x. 2 enterprisedb enterprisedb  4096 Sep 17 11:07 .
drwx------. 20 enterprisedb enterprisedb  4096 Sep 17 11:08 ..
-rw-------.  1 enterprisedb enterprisedb   2062 Sep 16 10:25 enterprisedb-2015-09-15_103605.log
-rw-------.  1 enterprisedb enterprisedb   1378 Sep 16 10:25 enterprisedb-2015-09-15_112241.log
-rw-------.  1 enterprisedb enterprisedb   2128 Sep 16 10:25 enterprisedb-2015-09-15_185952.log
-rw-------.  1 enterprisedb enterprisedb   1378 Sep 16 10:25 enterprisedb-2015-09-15_193935.log
-rw-------.  1 enterprisedb enterprisedb   1378 Sep 16 10:25 enterprisedb-2015-09-16_083339.log
-rw-------.  1 enterprisedb enterprisedb   2064 Sep 16 10:25 enterprisedb-2015-09-16_084034.log
-rw-------.  1 enterprisedb enterprisedb   1504 Sep 16 10:25 enterprisedb-2015-09-16_102358.log
-rw-------.  1 enterprisedb enterprisedb   1087 Sep 16 10:25 enterprisedb-2015-09-16_102439.log
-rw-------.  1 enterprisedb enterprisedb   4885 Sep 16 13:47 enterprisedb-2015-09-16_105416.log
-rw-------.  1 enterprisedb enterprisedb 14457 Sep 17 14:34 enterprisedb-2015-09-17_083210.log
-rw-rw-r--.  1 enterprisedb enterprisedb   485 Sep 17 08:32 startup.log
```

Disk Usage

- The following stat functions can be used to view database size:
  - `pg_database_size(name)`
  - `pg_tablespace_size(name)`
- The disk usage command ‘du’ is also helpful in determining the growth of disk from time to time

Postgres Enterprise Manager (PEM)

- Postgres Enterprise Manager (PEM) is an enterprise class software tool.
- Assists DBAs in administering, monitoring, and tuning PostgreSQL and EnterpriseDB Postgres Plus database servers.
- PEM is architected to manage and monitor multiple servers from a single console.

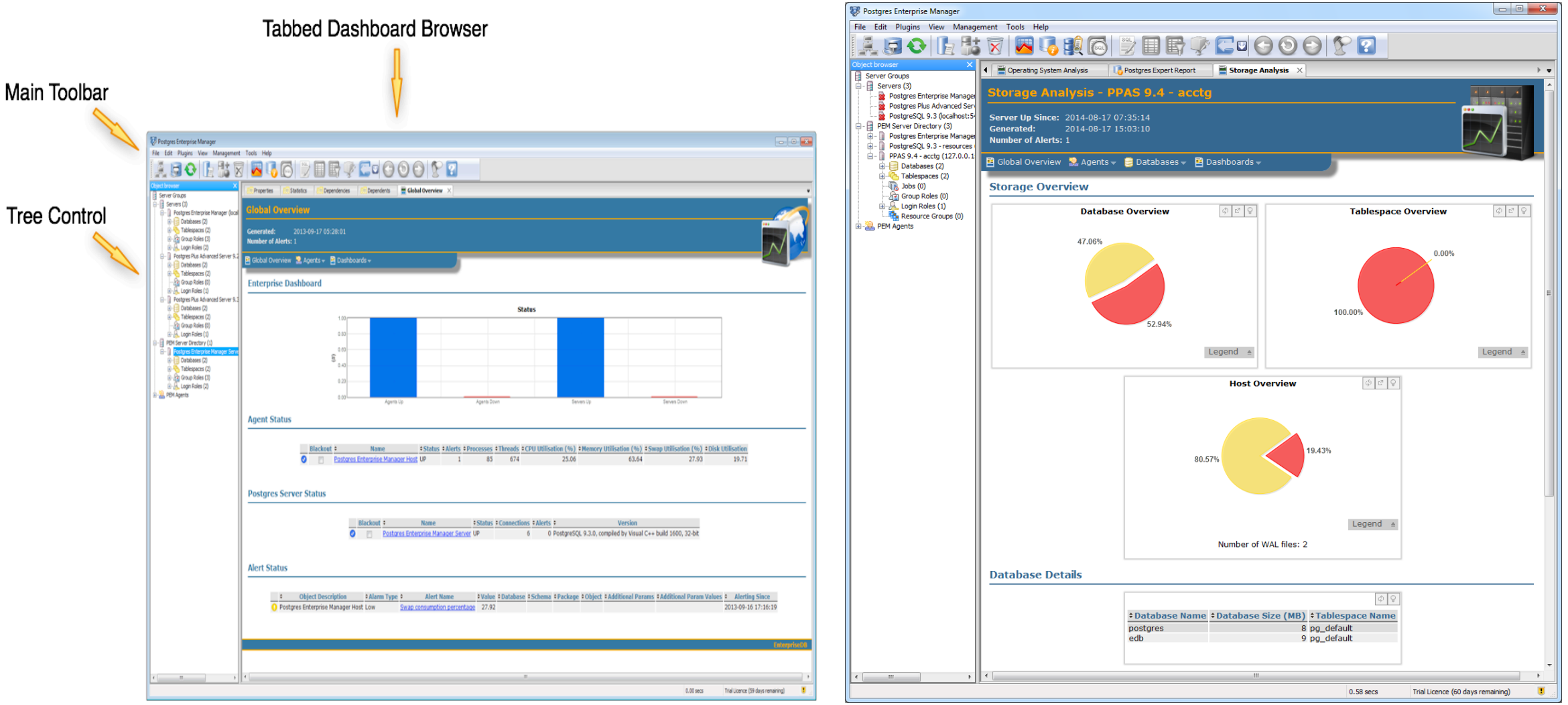
PEM - Features

- Global dashboards keep you up to date on the up/down/performance status of all your servers
- You can manage database servers regardless of the operating systems
- Database administration can be carried out via a graphical user interface
- Provides a full SQL integrated development environment (IDE)
- Lightweight and efficient agents monitor all aspects of each database server's operations

PEM - Features

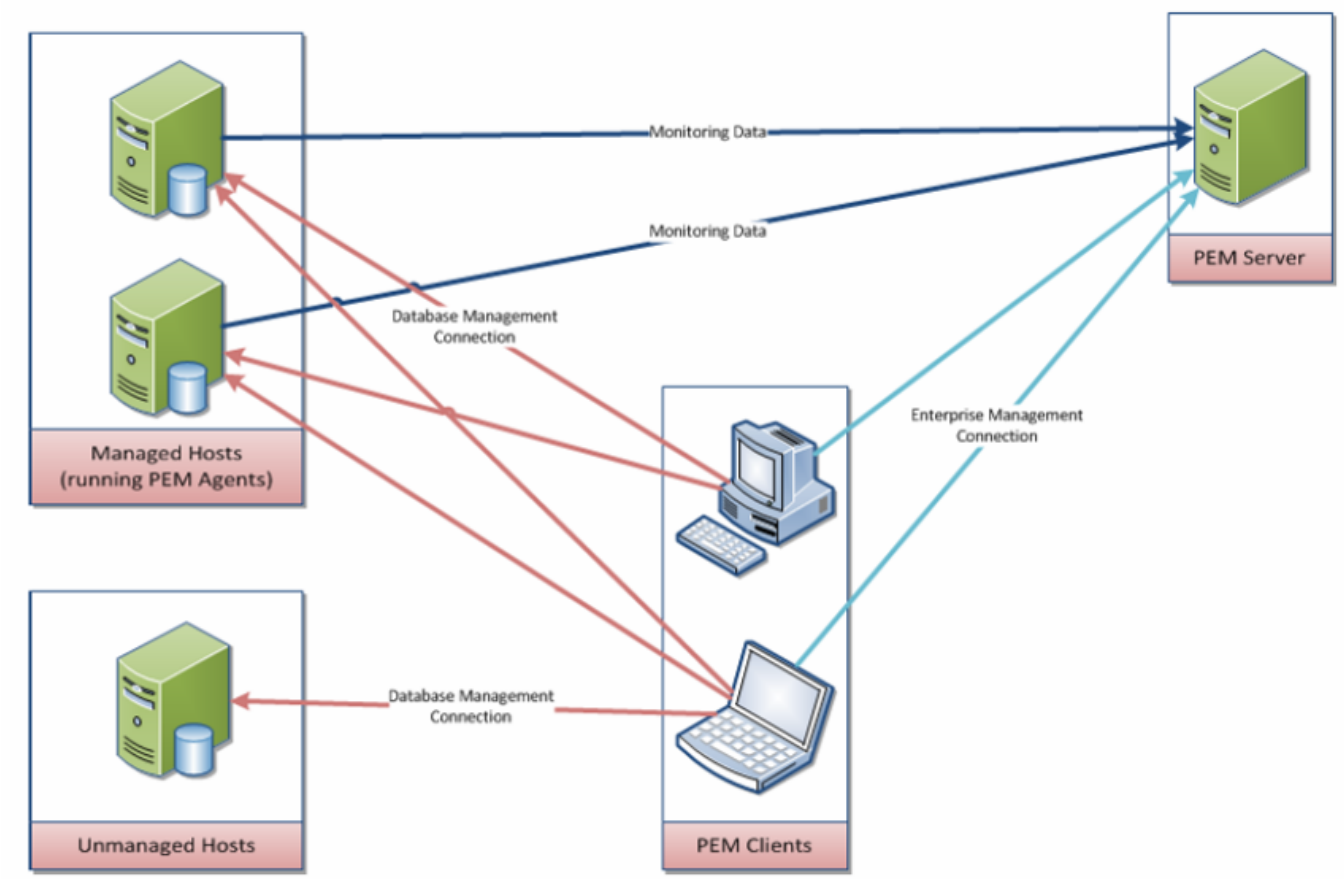
- Ability to create performance thresholds for each key metric e.g. memory, storage, etc.
- Any threshold violation results in an alert being sent to a centralized dashboard
  - Alerts can be sent via e-mail or SNMP trap
- All key performance-related statistics are automatically collected and retained
- Forecasts resource usage in the future
- Helps to identify and tune poorly running SQL statements
- Wide platform support

The PEM client window, displaying dashboard.



PEM - Architecture

- PEM is composed of three primary components
  - The PEM Server
  - The PEM Agent
  - The PEM Client



  - Note: PEM Web Client

Module Summary

- Database Monitoring
- Database Statistics
- The Statistics Collector
- Database Statistic Tables
- Operating System Process Monitoring (Unix)
- Server Processes – Linux
- Current Sessions and Locks
- Logging Slow Running Queries
- Disk Usage
- Postgres Enterprise Manager (PEM)
- PEM - Features
- PEM - Architecture

Day 2 - 4.2
The PostgreSQL Data Dictionary

Objectives

- The System Catalog Schema
- System Information Tables
- System Information Functions
- System Administration Functions
- System Information Views
- Oracle-like Dictionaries

The System Catalog Schema

- Store information about table and other objects
- Created and maintained automatically in `pg_catalog` schema
- `pg_catalog` is always effectively part of the `search_path`
- Contains:
  - System Tables like `pg_class` etc.
  - System Function like `pg_database_size()` etc.
  - System Views like `pg_stat_activity`

System Information Tables

- `\dS` in `psql` prompt will give you the list of `pg_*` tables and views
- This list is from `pg_catalog` schema
- Example:
  - `pg_tables` – list of tables
  - `pg_constraints` – list of constraints
  - `pg_indexes` – list of indexes
  - `pg_trigger` – list of triggers
  - `pg_views` – list of views

System Information Functions

- `current_database, current_schema, inet_client_addr, inet_client_port, inet_server_addr, inet_server_port, pg_postmaster_start_time, version`
- `current_user` -user - user used for permission checking, must be called without ()
- `session_user` - normally user who started the session, but superusers can change
- `current_schemas(boolean)` - returns array of schemas in the search path, optionally including implicit schemas

System Administration Functions

- `current_setting, set_config` - return or modify configuration variables
- `pg_cancel_backend` - cancel a backend's current query
- `pg_terminate_backend` – terminates backend process
- `pg_reload_conf` - reload configuration files
- `pg_rotate_logfile` - rotate the server's log file
- `pg_start_backup, pg_stop_backup` - used with Point In Time Recovery

System Administration Functions

- `pg_*_size` - disk space used by a `tablespace, database, relation or total_relation` (includes indexes and toasted data)
- `pg_column_size` - bytes used to store a particular value
- `pg_size_pretty` - convert a raw size to something more human-readable
- `pg_ls_dir`, `pg_read_file`, `pg_stat_file` - file operation functions. Restricted to superuser use and only on files in the data or log directories

System Information Views

- `pg_stat_activity` - details of open connections and running transactions
- `pg_locks` - list of current locks being held
- `pg_stat_database` - details of databases
- `pg_stat_user_*` - details of tables, indexes and functions

PPAS Oracle-Like Dictionaries

- The `sys` schema contains Oracle compatible catalog views
- PPAS provides `DBA_`, `ALL_` and `USER_` dictionary views to view database object information

View	Description
`user_*`	User's view (what is in your schema; what you own)
`all_*`	Expanded user's view (what you can access)
`dba_*`	Database administrator's view (what is in everyone's schemas)
`edb$`	Performance-related data

Module Summary

- The System Catalog Schema
- System Information Tables
- System Information Functions
- System Administration Functions
- System Information Views
- Oracle-like Dictionaries

Day 2 - 5
Moving Data

Objectives

- Loading flat files
- Import and export data using COPY
- Examples of COPY Command
- Using COPY FREEZE for performance
- Introduction to EDB*Loader for PPAS

Loading Flat Files into Database Tables

- A "flat file" is a plain text or mixed text file which usually contains one record per line
- PPAS offers the following option to load flat files into a database table:
  - `COPY` Command
- PPAS offers two options to load flat files into a database table:
  - EDB*Loader
  - COPY Command

The COPY Command

- `COPY` moves data between PPAS tables and standard file-system files
- `COPY TO` copies the contents of a table or a query to a file
- `COPY FROM` copies data from a file to a table
- The file must be accessible to the server
- `\copy` COPY with data stream to the client host

COPY TO

Data Unload / Export

- Syntax of `COPY TO` Command:

```
COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
where option can be one of:
    FORMAT format_name
    OIDS [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] ) |
    ENCODING 'encoding_name'
```

COPY FROM

Data Load / Import

- Syntax of `COPY FROM` Command:

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
where option can be one of:
    FORMAT format_name
    OIDS [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] ) |
    ENCODING 'encoding_name'
```

Example of COPY TO

```
edb=# COPY emp (empno,ename,job,sal,comm,hiredate) TO '/tmp/emp.csv' CSV
HEADER;
COPY


edb=# \! cat /tmp/emp.csv
empno,ename,job,sal,comm,hiredate
7369,SMITH,CLERK,800.00,,17-DEC-80 00:00:00
7499,ALLEN,SALESMAN,1600.00,300.00,20-FEB-81 00:00:00
7521,WARD,SALESMAN,1250.00,500.00,22-FEB-81 00:00:00
7566,JONES,MANAGER,2975.00,,02-APR-81 00:00:00
7654,MARTIN,SALESMAN,1250.00,1400.00,28-SEP-81 00:00:00
7698,BLAKE,MANAGER,2850.00,,01-MAY-81 00:00:00
7782,CLARK,MANAGER,2450.00,,09-JUN-81 00:00:00
7788,SCOTT,ANALYST,3000.00,,19-APR-87 00:00:00
7839,KING,PRESIDENT,5000.00,,17-NOV-81 00:00:00
7844,TURNER,SALESMAN,1500.00,0.00,08-SEP-81 00:00:00
7876,ADAMS,CLERK,1100.00,,23-MAY-87 00:00:00
7900,JAMES,CLERK,950.00,,03-DEC-81 00:00:00
7902,FORD,ANALYST,3000.00,,03-DEC-81 00:00:00
7934,MILLER,CLERK,1300.00,,23-JAN-82 00:00:00
```

Example of COPY FROM

```
edb=# CREATE TEMP TABLE empcsv (LIKE emp);
CREATE TABLE
edb=# COPY empcsv (empno, ename, job, sal, comm, hiredate)
edb-# FROM '/tmp/emp.csv' CSV HEADER;
COPY
edb=# SELECT * FROM empcsv;
 empno | ename  |    job    | mgr |      hiredate       |   sal   |  comm   | deptno
-------+--------+-----------+-----+---------------------+---------+---------+--------
  7369 | SMITH  | CLERK     |     | 17-DEC-80 00:00:00  |  800.00 |         |
  7499 | ALLEN  | SALESMAN  |     | 20-FEB-81 00:00:00  | 1600.00 |  300.00 |
  7521 | WARD   | SALESMAN  |     | 22-FEB-81 00:00:00  | 1250.00 |  500.00 |
  7566 | JONES  | MANAGER   |     | 02-APR-81 00:00:00  | 2975.00 |         |
  7654 | MARTIN | SALESMAN  |     | 28-SEP-81 00:00:00  | 1250.00 | 1400.00 |
  7698 | BLAKE  | MANAGER   |     | 01-MAY-81 00:00:00  | 2850.00 |         |
  7782 | CLARK  | MANAGER   |     | 09-JUN-81 00:00:00  | 2450.00 |         |
  7788 | SCOTT  | ANALYST   |     | 19-APR-87 00:00:00  | 3000.00 |         |
  7839 | KING   | PRESIDENT |     | 17-NOV-81 00:00:00  | 5000.00 |         |
  7844 | TURNER | SALESMAN  |     | 08-SEP-81 00:00:00  | 1500.00 |    0.00 |
  7876 | ADAMS  | CLERK     |     | 23-MAY-87 00:00:00  | 1100.00 |         |
  7900 | JAMES  | CLERK     |     | 03-DEC-81 00:00:00  |  950.00 |         |
  7902 | FORD   | ANALYST   |     | 03-DEC-81 00:00:00  | 3000.00 |         |
  7934 | MILLER | CLERK     |     | 23-JAN-82 00:00:00  | 1300.00 |         |
(14 rows)
```

Example - COPY Command on Remote Host

- `COPY` command on remote host using psql

```
cat emp.csv | ssh 192.168.11.128 "psql -U edbstore edbstore -c
'copy emp from stdin;' "
```

Example Cont.

```
edbstore=# create table empcsv (like emp);
CREATE TABLE
edbstore =# create table empcsv1 as select * from emp with no data;
SELECT 0
edbstore=#
edbstore=# select * from empcsv;
 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

```
edbstore=# copy emp to '/edbkr/Labs/copy/emp.csv' CSV;
COPY 14

edbstore=# \! cat /edbkr/Labs/copy/emp.csv
7369,SMITH,CLERK,7902,1980-12-17 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,1981-02-20 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,1981-02-22 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,1981-04-02 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,1981-09-28 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,1981-05-01 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,1981-06-09 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,1987-04-19 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,1981-11-17 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,1981-09-08 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,1987-05-23 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,1981-12-03 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,1981-12-03 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,1982-01-23 00:00:00,1300.00,,10
edbstore=#
```

```
[postgres@edbkorea:/opt/PostgreSQL/9.4] cat /edbkr/Labs/copy/emp.csv  | ssh 192.168.11.133  "/opt/PostgresPlus/9.4AS/bin/psql -p 5444 -d edb -U
enterprisedb -c 'copy empcsv from stdin CSV'"
COPY 14
[postgres@edbkorea:/opt/PostgreSQL/9.4] psql -h 192.168.11.133 -d edb -U enterprisedb -c "select count(*) from empcsv"
Password for user enterprisedb:
 count
-------
    14
(1 row)
```

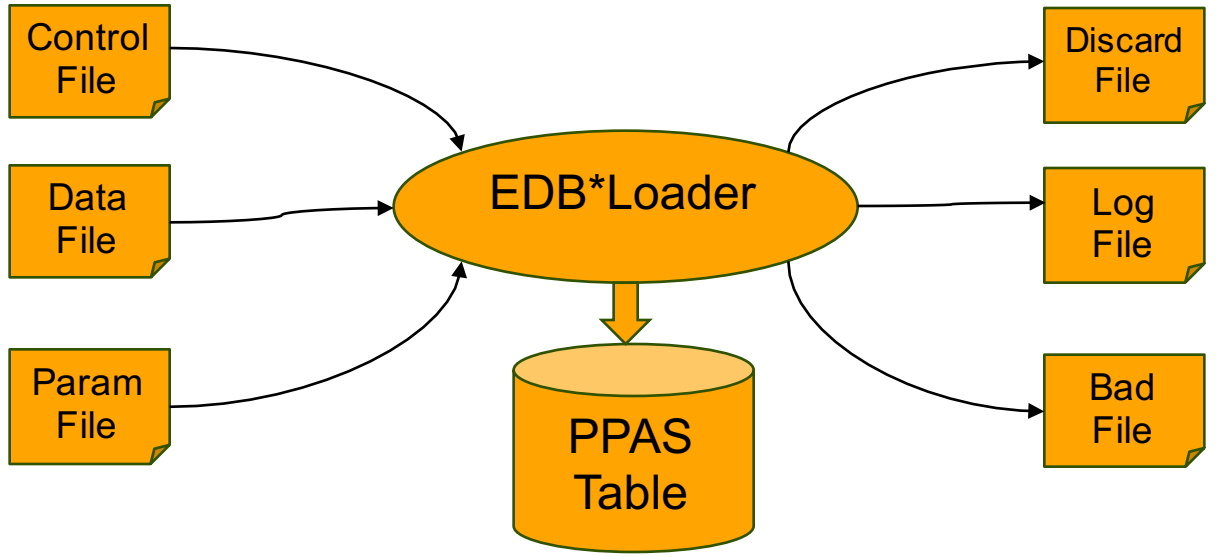
COPY FREEZE

- `FREEZE` is a new option in the `COPY` statement
  - Add rows to a newly created table and freezes them
  - Table must be created or truncated in current subtransaction
  - Improves performance of initial bulk load
  - Does violate normal rules of MVCC
- Usage:

```
COPY tablename FROM filename FREEZE;
```

EDB*Loader

- EDB*Loader is a high-performance bulk data loader
- Oracle SQL*Loader 와 같은 구문을 사용할 수 있습니다.
- Supports Oracle SQL*Loader data loading methods :
  - conventional path load
  - direct path load
  - parallel direct path load



Data Loading Methods

- Conventional path loading uses basic insert processing to add rows to the table
- Constraints, Indexes and triggers are enforced during Conventional path data loading
- Direct path loading is faster than conventional path loading, but is non-recoverable
- Direct path loading also requires removal of constraints and triggers from the table
- Conventional path data loading is slower than Direct path loading, but is fully recoverable
- A Parallel direct path load provides even greater performance

Invoking EDB*Loader

- Use the following command to invoke EBD*Loader from the command line:

```
edbldr [ -d dbname ] [ -p port ] [ -h host ]
 [ USERID={ username/password | username/ | username | / } ]
   CONTROL=control_file                    // Control 파일 이름, EDB*Loader을 수행하기 위해서는 항상 지정해 주어야 합니다
 [ BAD=bad_file ]                          // 거부된 레코드 모두를 저장하는 Bad 파일 이름을 지정 합니다
 [ DISCARD=discard_file ]                  // Load시 선택되지 않은 레코드가 저장되는 Discad 파일(선택 사항)
 [ LOG=log_file ]                          // 로그 파일 이름을 지정 합니다
 [ PARFILE=param_file ]                    // 추가 파라미터 파일을 지정 합니다
 [ SKIP=skip_count ]                       // number of initial input rows to skip during the load
 [ ERRORS=error_count ]                    // 허용하는 배드 레코드의 최대 수를 지정 합니다.
 [ SKIP_INDEX_MAINTENANCE={ FALSE | TRUE }
 [ DIRECT={ FALSE | TRUE } ]               // TRUE로 설정되면 EDB*Loader는 DIRECT PATH를 사용.
                                           //   반대의 경우는 기본 값인 CONVENTIONAL PATH를 사용 합니다
 [ PARALLEL={ FALSE | TRUE } ]             // DIRECT 로드에서만 적합한 이 파라미터는 다중 병렬 DIRECT로드가 수행되도록 지정.
```


131

The EDB*Loader Control File

- Oracle SQL*Loader compatible syntax for control file directives
- Control File은 EDB*Loader를 사용하는데 필수적인 파일의 하나로써 데이터 정의어(DDL) 지침을 포함하는 텍스트 파일입니다. 확장자는 “ctl” 입니다.
- 다음과 같은 정보를 포함한다 :
  - 로드 할 외부 파일의 위치을 지정 합니다. (INFILE)
  - Reject 되는 데이터에 대한 파일을 지정 합니다. (BADFILE)
  - 사용자가 지정한 조건에 맞지 않는 데이터를 저장 하는 파일을 지정합니다.(DISCARDFILE)
  - 테이블에 데이터를 넣는 방법 지정 합니다.(INSERT | APPEND | REPLACE | TRUNCATE)

132

Control File Syntax

- The syntax for the EDB*Loader control file is as follows:

```
LOAD DATA
   INFILE '{ data_file  | stdin }'                          /// 외부 데이터 파일 지정
   [ BADFILE 'bad_file'  ]                                  /// 거부된 레코드를 배치할 파일지정
    [ DISCARDFILE 'discard_file'  ]                         /// 폐기된 레코드를 배치할 파일지정
 [ INSERT | APPEND | REPLACE | TRUNCATE ]                   /// INSERT:빈 테이블에 삽입
                                                            /// APPEND : 현재 데이터에 추가
                                                            /// REPLACE:테이블의 기존 행을 삭제하고 삽입
                                                            /// TRUNCATE:기존 데이터를 삭제하고 삽입
 { INTO TABLE target_table                                  /// 데이터를 로드 할 테이블 지정
   [ WHEN field_condition  [ AND field_condition  ] ...]
    [ FIELDS TERMINATED BY 'termstring'                     /// 데이터 필드의 종결 문자를 지정
     [ OPTIONALLY ENCLOSED BY 'enclstring'  ] ]
   [ TRAILING NULLCOLS ]                                    /// 해당 필드에 NULL 값이 존재해도 import 한다
    (field_def  [, field_def  ] ...)
}
```

EDB*Loader Example

- Example loads data from a file named
  - */edbkr/Labs/pgbackup/edbldr_data.txt* into a table named **account**.
- The data within the input file is delimited by a '|'
- Create Control File: *data_control.ctl*

```
LOAD DATA
INFILE '/edbkr/Labs/pgbackup/edbldr_data.txt'
BADFILE '/edbkr/Labs/pgbackup/edbldr_data.bad'
APPEND
INTO TABLE account
FIELDS TERMINATED BY '|'
(id, aname)
```

```
[enterprisedb@edbkorea:/edbkr/Labs/pgbackup]cat edbldr_data.txt
1|edbkr_sample1
2|edbkr_sample2
n|edbkr_sample3
4|edbkr_sample4
5|edbkr_sample5
[enterprisedb@edbkorea:/edbkr/Labs/pgbackup]
```

```
edbstore=# \d account
        Table "edbstore.account"
 Column |       Type        | Modifiers
--------+-------------------+-----------
 id     | numeric           |
 aname  | character varying |

edbstore=#
```

- Run edbldr command:

```
$ edbldr -d edb USERID=edbstore/edbstore CONTROL=data_control.ctl DIRECT=TRUE PARALLEL=TRUE SKIP=1 ERRORS=10
```

PREPARE

- PREPARE creates a prepared statement.

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);

PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Tips for Inserting Large Amount of Data

- While inserting data using multiple inserts use BEGIN at the start and COMMIT at the end.
- Use COPY to load all the rows in one command, instead of using a series of INSERT commands.
- If you cannot use COPY, it might help to use PREPARE to create a prepared INSERT statement, and then use EXECUTE as many times as required.
- If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using COPY, then create any indexes needed for the table.
- It might be useful to drop foreign key constraints, load data, and re-create the constraints.

Tips for Inserting Large Amount of Data

- Temporarily increasing the maintenance_work_mem and checkpoint_segments configuration variables when loading large amounts of data can lead to improved performance.
- Disable WAL Archival and Streaming Replication
- Triggers and Autovacuum can also be disabled.
- Certain commands run faster if wal_level is minimal:
  - CREATE TABLE AS SELECT
  - CREATE INDEX (and variants such as ALTER TABLE ADD PRIMARY KEY)
  - ALTER TABLE SET TABLESPACE
  - CLUSTER
  - COPY FROM, when the target table has been created or truncated earlier in the same transaction

Some Notes About pg_dump

- Set appropriate (i.e., larger than normal) values for maintenance_work_mem and checkpoint_segments.
- If using WAL archiving or streaming replication, consider disabling them during the restore.
- Consider whether the whole dump should be restored as a single transaction. To do that, pass the -1 or --single-transaction command-line option to psql or pg_restore.
- If multiple CPUs are available in the database server, consider using pg_restore's --jobs option.
- Run ANALYZE afterwards.

Non-Durable Settings

- Durability guarantees the recording of committed transactions but adds significant overhead
- PostgreSQL can be configured to run without durability.
- Place the database cluster's data directory in a memory-backed file system (i.e. RAM disk).
- Turn off fsync; there is no need to flush data to disk.
- Turn off full_page_writes; there is no need to guard against partial page writes.
- Increase checkpoint_segments and checkpoint_timeout ; this reduces the frequency of checkpoints.
- Turn off synchronous_commit; there might be no need to write the WAL to disk on every commit.

Module Summary

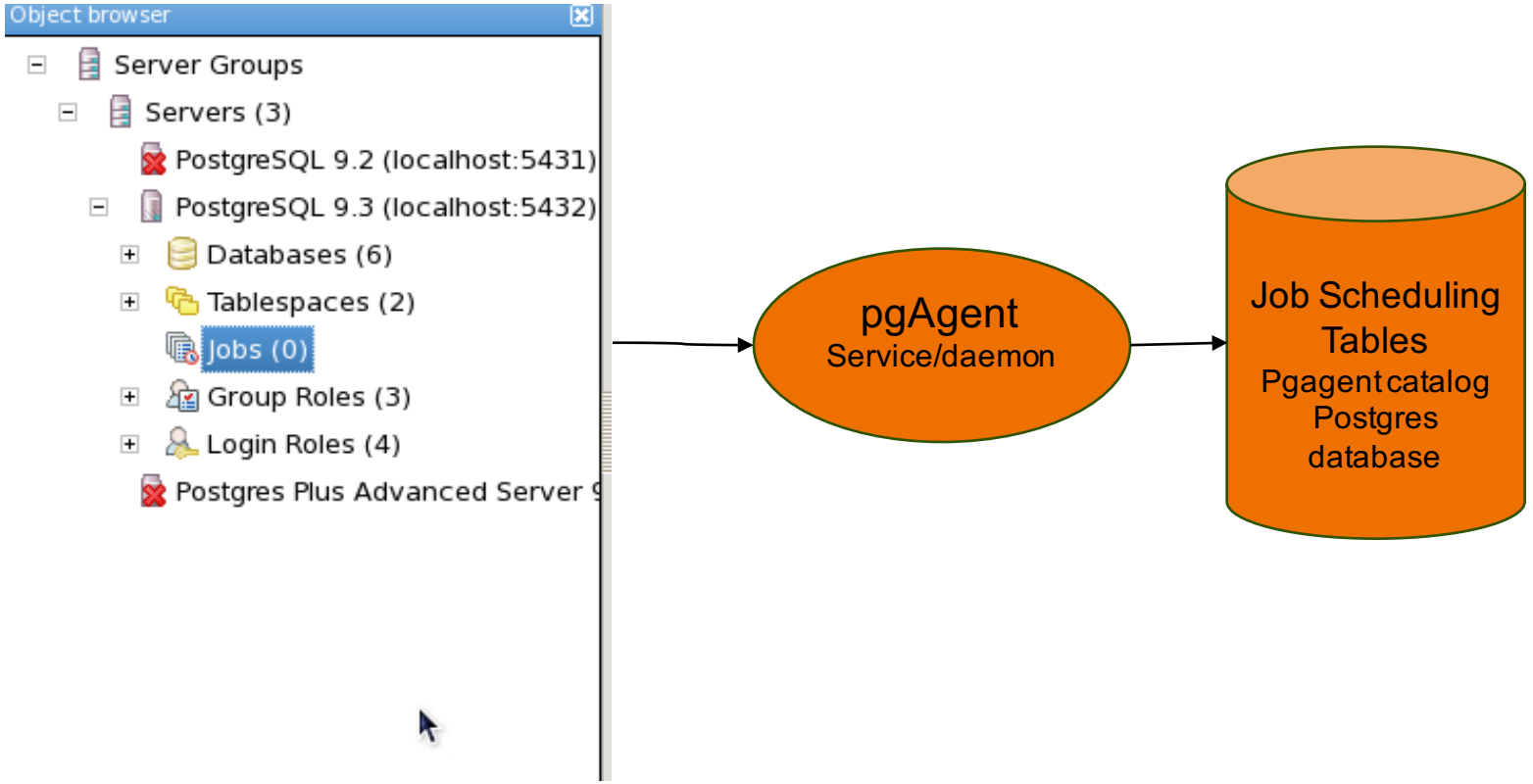
- Loading flat files
- Import and export data using COPY
- Examples of COPY Command
- Using COPY FREEZE for performance
- Introduction to EDB*Loader for PPAS

Day 2 - 6
Scheduler

pgAgent

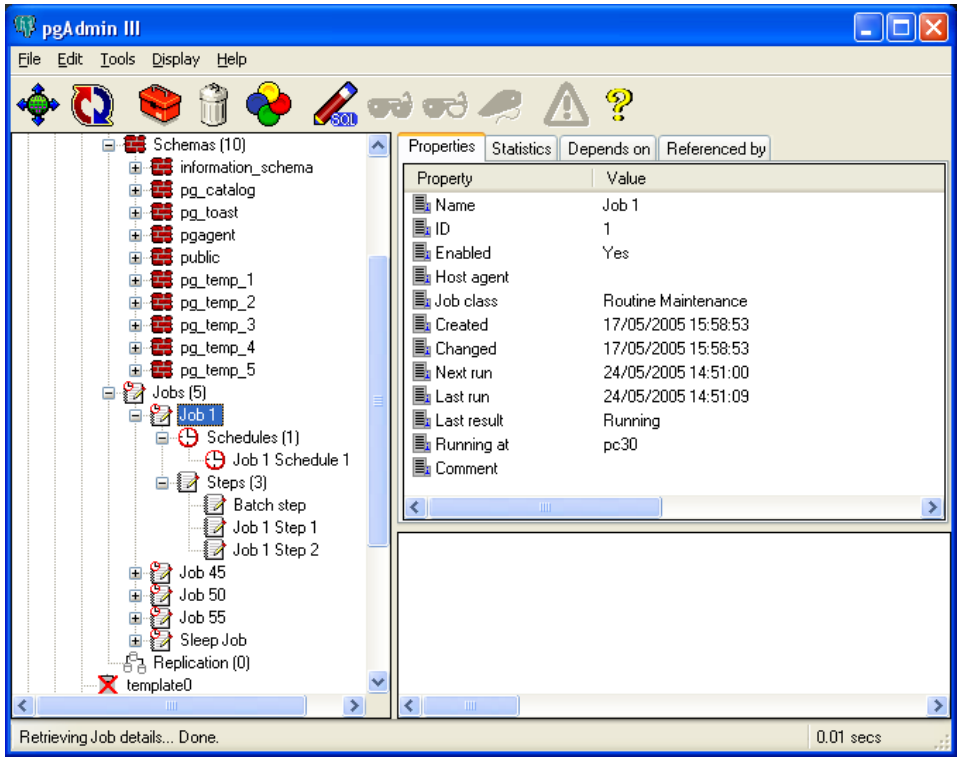
- pgAgent is a job scheduling agent for PostgreSQL.
- Working with pgAdmin III v1.4 and above or PEM client.
- Capable of running multi-step batch/shell and SQL tasks on complex schedules.

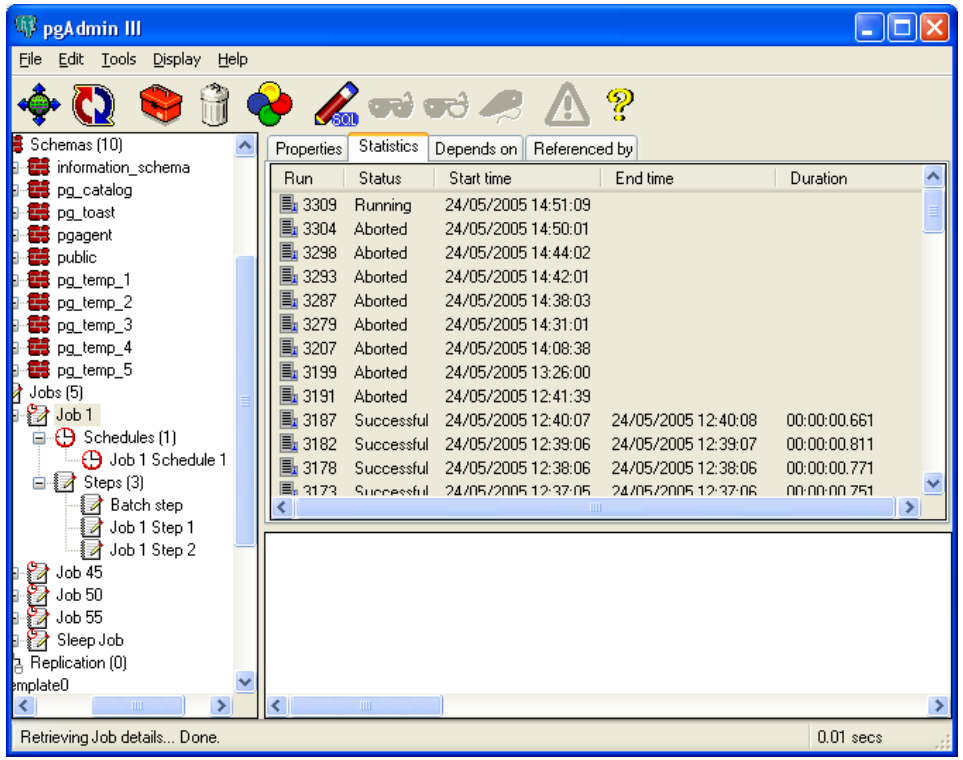
pgAgent: Concept



pgAgent Jobs

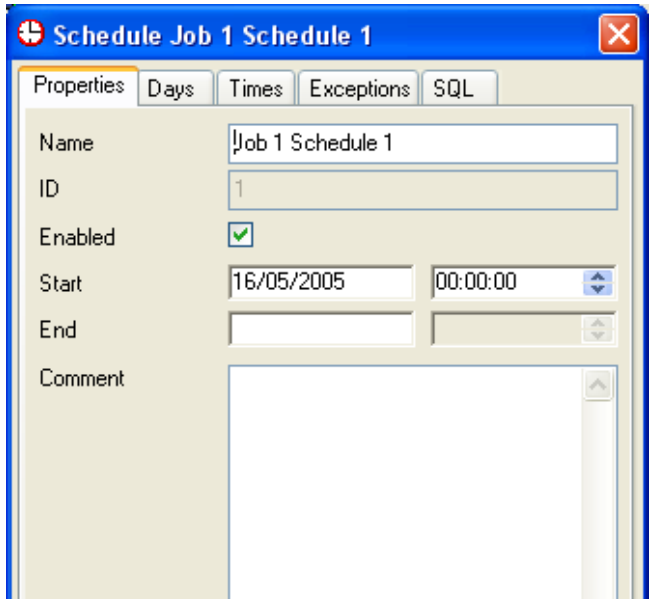
- pgAgent runs 'jobs', each of which consists of steps and schedules.

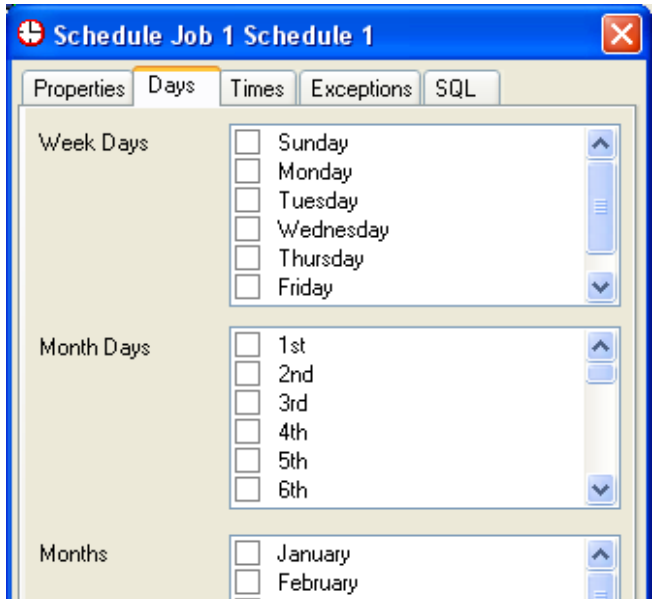


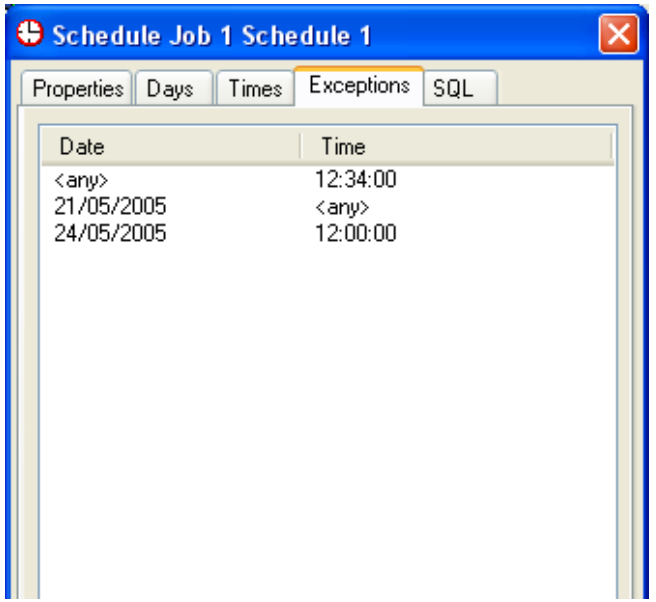


pgAgent Schedules

- Each Job is executed according to one or more schedules.
- Each time the job or any of it's schedules are altered, the next runtime of the job is re-calculated.
- If no pgAgent instance is running at the next runtime of a job, it will run as soon as pgAgent is next started.

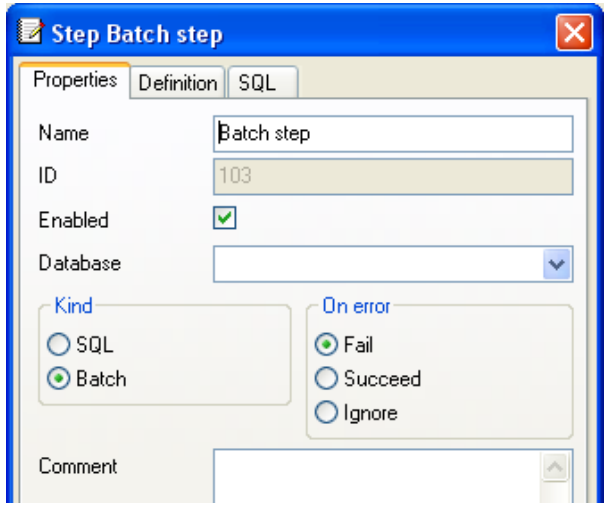






pgAgent Steps

- Each Job consists of a number of steps, each of which may be an SQL script, or an operating system batch/shell script.
- Each step in a given job is run in turn, in alphanumeric name order.
- The *Definition* tab contains a single text box into which the step script should be entered
  - For SQL steps: a series of one or more SQL statements
  - For Linux server: any shell script. A suitable interpreter is specified on the first line (e.g. *#!/bin/sh*).



Lab

Day 2 - 7
Tablespace

Tablespace

- Tablespaces is locations in the file system.
  - Symbolic links to directories on a file system.
- Can be referred to by name
- The psql's \db meta-command is useful for listing the tablespaces.
  - Be careful when use \db+ on large tablespaces.
- You can change the symbolic link after shutdown the server.
  - In PostgreSQL 9.1 and earlier you will also need to update the pg_tablespace catalog with the new locations.

Create Tablespace

```
Syntax:
CREATE TABLESPACE tablespace_name
    [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
    LOCATION 'directory'
    [ WITH ( tablespace_option = value [, ... ] ) ]
```

```
edb=# create tablespace tbs01 location '/opt/PostgresPlus/tbs/tbs01';
CREATE TABLESPACE
edb=# select oid, * from pg_tablespace where spcname = 'tbs01';
  oid  | spcname | spcowner | spcacl | spcoptions
-------+---------+----------+--------+------------
 16878 | tbs01   |       10 |        |
(1 row)
```

```
$ ls -l $PGDATA/pg_tblspc
total 0
lrwxrwxrwx. 1 enterprisedb enterprisedb 27 Feb 10 17:24 16878 -> /opt/PostgresPlus/tbs/tbs01
```

- The location must be an existing, empty directory that is owned by the PostgreSQL operating system user.
- `CREATE TABLESPACE` cannot be executed inside a transaction block.

Lab