covered in this talk

- installation
- extensions
- updates and upgrades
- configuration
- connections & security
- backups
- replication
- VACUUM
not covered

- older versions (< 9.1)
- schema design
- query troubleshooting
- query rewriting
- indexes
- testing
- application stuff
“You know servers, right? You're in charge of the database now.”
“Efficiency”
“DevOps”
“Cloud”
“Growth Opportunity”
“We're not going to hire a DBA”
Y U no DBA?

1. limited budgets
2. shortage of operational staff
3. cheaper OSS databases

... you are the DBA now.
Oh My God, We’re All Going To Die.
Don't Panic
Linux: use packages!

- version not important?
- use the ones that come with your distro
  - Red Hat, Centos, SciLinux
  - Debian, Ubuntu
  - SuSE
Linux: use packages!

- need the latest version?
- alternate packages
  - Red Hat: yum.postgresql.org
  - Ubuntu: apt.postgresql.org
  - SuSE: build service
  - Debian: apt.postgresql.org
Windows/OSX
Windows/OSX

- use the graphical installer
  - from EnterpriseDB
  - <link here>
  - “wizard” GUI
- also installs optional components
  - pgAdmin
  - Extensions
Windows/OSX

- use the graphical installer
  - from EnterpriseDB
  - <link here>
  - “wizard” GUI
- also installs optional components
  - pgAdmin
  - Extensions
other platforms

- Packages available:
  - Solaris 10/11
  - OpenSolaris/Illumos
  - FreeBSD
  - OpenBSD
  - NetBSD

- No Packages:
  - HP-UX
  - AIX
  - Solaris 9
  - Tablets
  - “Home” Windows
create data directory

- $PGDATA is where the database files live
- most packages create it
- if not, use “initdb” to create it
  - pick a suitable location!

initdb -D /db/9.2/main
```bash
Sidney-Stratton:~ josh$ psql libdata
psql (9.1.1)
Type "help" for help.

libdata=# \dt

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>books</td>
<td>table</td>
<td>libdata</td>
</tr>
<tr>
<td>public</td>
<td>branches</td>
<td>table</td>
<td>libdata</td>
</tr>
<tr>
<td>public</td>
<td>copies</td>
<td>table</td>
<td>libdata</td>
</tr>
<tr>
<td>public</td>
<td>copy_history</td>
<td>table</td>
<td>libdata</td>
</tr>
<tr>
<td>public</td>
<td>copy_status</td>
<td>table</td>
<td>libdata</td>
</tr>
</tbody>
</table>
```

the psql command line
pgAdmin
extensions

Don't Panic
extensions?

- extensions add extra functionality
  - like Python/Perl modules, Ruby Gems, etc.
- need to be installed separately
  - some come with PostgreSQL packages
  - some need to be installed from PGXN or source
- handled very differently before 9.1
# Evergreen extensions

<table>
<thead>
<tr>
<th>Extension</th>
<th>Purpose</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>plpgsql</td>
<td>Procedural language for triggers and functions. Automatically included 9.0 and later.</td>
<td>core</td>
</tr>
<tr>
<td>hstore</td>
<td>Extensible key-value store. Used for “user-defined” fields.</td>
<td>contrib</td>
</tr>
<tr>
<td>xml2</td>
<td>Extra XML functions for Postgres's XML support.</td>
<td>contrib</td>
</tr>
<tr>
<td>tablefunc</td>
<td>Database linking and crosstab reports.</td>
<td>contrib</td>
</tr>
</tbody>
</table>
installing extensions

1. install the binary files
   ● using packages, PGXN, or source
   ● installs to postgres “share” directory
   ● a few extensions don't have binaries

2. install the extension in each database where it's used

   CREATE EXTENSION hstore;
extensions pre-9.1

1. install the binary files
   - using packages, PGXN, or source
   - installs to postgres “share” directory
   - a few extensions don't have binaries

2. load the SQL file in each database where it's used

   psql -f hstore.sql evergreen-ils
updates & upgrades
The PostgreSQL developers released updates for all major branches of the popular open-source database system on Thursday in order to address several vulnerabilities, including a high-risk one that could allow attackers to crash the server, modify
**major vs. minor**

9.2 == a major version
- requires an *upgrade* from 9.1.8
- contains features not in 9.1
- requires testing and planned downtime

9.1.9 == a minor version
- is a minor *update* from 9.1.8.
- can (and should) be applied immediately
minor updates

- come out ~~ every 2 months
- contain only bugfixes
  - security hole patches
  - data loss prevention
  - fix server crashes
- no new or changed features
  - occasional documented breakage
update promptly
update update often
update procedure

1. schedule 5 minute downtime
2. download packages
3. shut down postgresql
4. install packages
5. restart postgresql
6. restart application
major upgrades

- come out once per year
- have many new features
  - and sometimes break stuff which used to work
- require extensive testing with your application
- require significant downtime to upgrade
upgrade procedures

- dump & reload
  - use pg_dump & pg_restore on database
  - most reliable way
  - “cleans up” database in process
  - best with small databases
  - can take a long, long time
upgrade procedures

- **pgUpgrade**
  - upgrade “in place”
  - much faster
  - does not “clean up” database
  - sometimes doesn't work
    - issues with extensions
EOL after 5 years
upgrading extensions

not possible before 9.1

1. upgrade Postgres
2. install new extension binaries
3. run upgrade script in each DB

```
ALTER EXTENSION hstore
UPDATE;
```
configuration
configuration

1. Hardware
2. OS/FS
3. PostgreSQL
use good hardware

- databases use *all* the hardware
  - RAM, CPU, IO
  - disk can be very important
    - DB larger than RAM
    - write-heavy database

- *the database cannot outperform bad hardware*
put the database on its own server (or virtual server)
cloud servers

- cloud server performance sucks
  - especially IO
- make sure you have enough RAM to cache the whole database
Linux configuration

1. turn the OOM killer off
2. turn reclaim_zone_files off
3. use XFS or Ext4 for database files
   1. use “noatime,nodiratime”
4. increase shmmmax, shmall
   1. so that you can raise shared_buffers
   2. not required for 9.3beta, yaaaay!
BSD/Solaris Config.

- Use ZFS
  - decrease block size to 8K
- increase shmmmax/shmmall on BSD
- may need to mess with ulimits
  - on very busy systems
  - Postgres will give you errors
Windows/OSX

optimization not possible
the xlog

- xlog == WAL
  - where transactions are recorded
- best on its own drive/volume
  - write-only
  - writes synchronously
  - response time paramount
- create volume, then link pg_xlog dir
postgresql.conf

parameters you care about:
10 to 18

parameters you don't care about:
206 to 214
connections
&
security
network

• local connections: UDP, if possible
  • faster than TCP/IP
• other servers: port 5432
  • make sure it's open on the firewall!
• on the cloud? use SSL
  • secure your connections
  • PITA to set up, though
max_connections

“ERROR: connection limit exceeded for non-superusers”

• postgresql.conf
• increase number of connections
• good up to about 50 + 10 x cores
• keep needing to increase it? something is wrong
connection pooling

- pgbouncer
  - event-based pooler
  - separate package
  - on DB server, or
  - app server, or
  - 3rd “bouncer” server
host-based access

“FATAL: no pg_hba.conf entry for host "192.168.0.1", user "chaos", database "chaosLRdb", SSL off”

- pg_hba.conf
- access control list:
  - database/user/host address
  - like iptables for Postgres
- change config and reload
-security

“FATAL: password authentication failed for user "wwwuser"

- Postgres users & passwords
  - CREATE/ALTER USER
  - "group" ROLEs
- Or: use LDAP, GSSAPI or PAM
don't expose
the postgres port
or server
to the internet
backups
invalid page header in block 311757 of relation base/26976/27977

xlog redo insert:
rel 1663/26976/27977;
tid 311757/44

startup process (PID 392) exited with exit code 1
aborting startup due to startup process failure
Our Disaster Recovery Plan Goes Something Like This...
three methods

A) pg_dump
B) PITR
C) filesystem snapshot
pg_dump

- “logical” backup
- portable
- compressed
- works for upgrades
- good for small databases
- use -Fc
  - custom binary format
PITR

- “Point-In-Time Recovery”
- “binary” and “continuous” backup
  - take snapshot of DB files
  - accumulate logfile copies
- good for large databases
- can combine with replication
DROP TABLE circulation;
PITR - PITA

• can be difficult to set up & monitor
• use tools:
  • Barman
  • OmniPITR
  • WAL-E (for AWS)
have a DR plan

- ways you can lose data
  - recovery time
  - acceptable data loss
- how to recover from lost data
  - detailed steps
  - verification
replication
use replication for ...

- **availability**: have an “always on” failover server
- **load-balancing**: offload traffic
  - especially reporting workloads
- **security**: provide users read-only access
binary replication

- Master
- Replica
- Replica

- Writes
- Reads
- Read statements
- Data pages
DRBD for PostgreSQL

(only much much faster)
also called ...

- **streaming replication**
  - refers to the ability to stream new data pages over a network connection

- **hot standby**
  - refers to the ability of standbys to run read-only queries while in standby mode
advantages

- low administration
- low overhead on master
  - not much incremental cost
- non-invasive
  - no extra tables/triggers
  - no primary key requirements
  - no limitations on statements
advantages

- low-latency
- no synchronization issues
- good for very large databases
- works with external disk tools
  - virtual machine cloning
  - filesystem snapshots
- supports disconnected replication
disadvantages

- need to replicate the whole server
  - not individual databases
  - not specific tables
- no writes of any kind on replicas
- some things not replicated
  - temp & unlogged tables
  - LISTEN/NOTIFY
- query cancel (described later)
disadvantages

- need to replicate the whole server
  - not individual databases
  - not specific tables
- no writes of any kind on replicas
- some things not replicated
  - temp & unlogged tables
  - LISTEN/NOTIFY
- query cancel (described later)
how it works
how it works
how it works

walsender

walreceiver
how it works
“recovery”

- historical term
  - because replication grew out of PITR
- used to refer to the replica being “in recovery mode”, i.e. no write queries
- all over parameter and file names
  - recovery.conf
  - recovery_target
  - pg_is_in_recovery()
replication demo
other cloning methods

- rsync
- filesystem snapshot
  - (on shared storage)
- VM cloning
  - (if rapid)
creating a snapshot

1. make sure you are replicating transaction logs to the replica
2. `pg_start_backup('replication')`
3. `rsync / snapshot / clone`
4. `pg_stop_backup()`
replica out of sync
replica out of sync

“do I gotta?”

“send me vry old logfiles pls”
replica out of sync

“scrw you! them iz gone”
log file backup
log file backup: cloud
failover vs. load-balancing

load causes replicas to fall behind
your replica can work for:
• rapid failover
• offloading work
but not both!
lag vs. query cancel

• decide what your application can accept
  • inconsistent lagged read data
  • cancel-and-retry queries

• different settings for different replicas
  • ephemeral web replicas: short delay
  • reporting replicas: long delay

• issue common to all M-S replication
other replication

- synchronous replication
- Slony-I
- Londiste
- Bucardo
monitoring

Don't Panic
use your favorite tool

ganglia, collectd, Hyperic, OpenNMS, OpenView, whatever ....

• nagios check_postgres.pl
  • broad list of checks
  • mine it for queries and techniques
many useful checks

- disk space
- caching RAM
- response time
- connections
- idle transacts
- table growth
- waiting queries

- long queries
- database size
- table bloat
- system load
- replication lag
- XID wraparound
- execution time
OS checks

- disk space (per volume!)
- system load
- memory usage
- IO activity
- network activity
database checks

- connections: active, idle, idle xtns
- blocked queries (number, time)
- query times (pg_stat_statements)
- table size & growth
- table & index bloat
- XID wraparound
- replication lag
activity log

- connections & disconnections
- slow queries
- DB swap usage
- schema changes
- lock waits & deadlocks
Queries by type

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>48,568</td>
<td>14.25%</td>
</tr>
<tr>
<td>INSERT</td>
<td>48,578</td>
<td>14.29%</td>
</tr>
<tr>
<td>UPDATE</td>
<td>145,701</td>
<td>42.85%</td>
</tr>
<tr>
<td>DELETE</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>OTHERS</td>
<td>97,154</td>
<td>28.57%</td>
</tr>
</tbody>
</table>

Type of queries

Type of locks

Locks by type

<table>
<thead>
<tr>
<th>Type</th>
<th>Object</th>
<th>Count</th>
<th>Total Duration</th>
<th>Av. duration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExclusiveLock</td>
<td>1</td>
<td></td>
<td>1.14s</td>
<td>1.14s</td>
</tr>
<tr>
<td>tuple</td>
<td></td>
<td>1</td>
<td>1.14s</td>
<td>1.14s</td>
</tr>
<tr>
<td>ShareLock</td>
<td>20</td>
<td>25.46s</td>
<td>1.27s</td>
<td></td>
</tr>
<tr>
<td>transaction</td>
<td>20</td>
<td>25.46s</td>
<td>1.27s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>21</td>
<td>26.60s</td>
<td>1.27s</td>
<td></td>
</tr>
</tbody>
</table>
non-overwriting
garbage collection

Tuplestore

Row C: Version 1
small update
Row C: Version 2
small update
Row C: Version 3
small update
Row C: Version 4
large update
Row C: Version 5
garbage collection

Row C: Version 5
garbage collection
garbage collection
garbage collection

Tuplestore

Row C: Version 5
Autovacuum
Multiple Autovacuum
falling behind
bloat check

- `pg_stat_user_tables`: dead_rows/live_rows
- nagios bloat query
- trend table/database size
- scans very slow to return the first row
fixing bloat

- manual VACUUMs
  - even VACUUM FULL
- pg_reorg (bad cases)
- tune autovacuum
  - increase frequency for busy tables
  - increase workload size (vacuum_cost_limit)
  - more autovacuum workers
XID wraparound
preempt wraparound

- track age(datfrozenxid) for each table
  - nagios has probe for this
- VACUUM FREEZE tables preemptively
- lower vacuum_freeze_min_age
  - maybe to 250,000
bonus round!
slow queries
### pg_stat_activity

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>datid</td>
<td>16422</td>
</tr>
<tr>
<td>datname</td>
<td>libdata</td>
</tr>
<tr>
<td>procpid</td>
<td>46295</td>
</tr>
<tr>
<td>usesysid</td>
<td>10</td>
</tr>
<tr>
<td>usename</td>
<td>dataentry</td>
</tr>
<tr>
<td>application_name</td>
<td>psql</td>
</tr>
<tr>
<td>client_addr</td>
<td>192.168.101.114</td>
</tr>
<tr>
<td>client_port</td>
<td>5432</td>
</tr>
<tr>
<td>backend_start</td>
<td>2012-08-26 15:09:05.233-07</td>
</tr>
<tr>
<td>xact_start</td>
<td>2012-08-26 15:09:06.113-07</td>
</tr>
<tr>
<td>query_start</td>
<td>2012-08-26 15:11:53.521-07</td>
</tr>
<tr>
<td>waiting</td>
<td>f</td>
</tr>
<tr>
<td>current_query</td>
<td>&lt;IDLE&gt; in transaction</td>
</tr>
</tbody>
</table>
locks

• write queries can block on other write queries
  • as can table schema changes
  • queries can wait forever on locks

• look old "<IDLE> in transaction"
  • that's a zombie database connection
  • kill it kill it kill it
killing zombies

- **pg_cancel_backend(pid)**
  - kills running queries with sigINT
  - like CTRL-C

- **pg_terminate_backend(pid)**
  - kills bad connections, idle transactions
  - can cause DB to reload in some cases
EXPLAIN

Nested Loop  (cost=792.00..828.08 rows=1422317 width=99)
  ->  HashAggregate  (cost=792.00..792.00 rows=1 width=4)
    ->  Index Scan using
         index_player_summaries_on_player_id on player_summaries
         ps  (cost=0.00..791.80 rows=403 width=4)
             Index Cond: (player_id = 21432312)
    ->  Index Scan using index_player_summaries_on_match_id
         on player_summaries  (cost=0.00..33.98 rows=600 width=99)
             Index Cond: (match_id = ps.match_id)
Nested Loop  (cost=792.00..828.08 rows=1422317 width=99)  (actual time=9928.869..20753.723 rows=13470 loops=1)
   ->  HashAggregate  (cost=792.00..792.00 rows=1 width=4)  (actual time=9895.105..9897.096 rows=1347 loops=1)
      ->  Index Scan using index_player_summaries_on_player_id on player_summaries ps  (cost=0.00..791.80 rows=403 width=4)  (actual time=27.413..9890.887 rows=1347 loops=1)
         Index Cond: (player_id = 21432312)
      ->  Index Scan using index_player_summaries_on_match_id on player_summaries (cost=0.00..33.98 rows=600 width=99)  (actual time=7.375..8.037 rows=10 loops=1347)
         Index Cond: (match_id = ps.match_id)
Total runtime: 20764.371 ms
<table>
<thead>
<tr>
<th>exclusive</th>
<th>inclusive</th>
<th>rows x</th>
<th>rows</th>
<th>loops</th>
<th>node</th>
</tr>
</thead>
</table>
| 30.788    | 20753.723 | ↑ 105.6| 13470| 1     | Nested Loop
    (cost=792.00..828.08 rows=1422317 width=99)
    (actual time=9928.869..20753.723 rows=13470 loops=1) |
| 6.209     | 9897.096  | ↓ 1347.0| 1347 | 1     | HashAggregate (cost=792.00..792.00 rows=1 width=4)
    (actual time=9895.105..9897.096 rows=1347 loops=1) |
| 9890.887  | 9890.887  | ↓ 3.3  | 1347 | 1     | Index Scan using
    index_player_summaries_on_player_id on player_summaries ps
    (cost=0.00..791.80 rows=403 width=4)
    (actual time=27.413..9890.887 rows=1347 loops=1)
    Index Cond: (player_id = 21432312) |
| 10825.839 | 10825.839 | ↑ 60.0 | 10   | 1347  | Index Scan using
    index_player_summaries_on_match_id on player_summaries
    (cost=0.00..33.98 rows=600 width=99)
    (actual time=7.375..8.037 rows=10 loops=1347)
    Index Cond: (match_id = ps.match_id) |
what to look for

- “seq scan” on large table
  - maybe index needed
- cartesian joins
- really bad row estimates
  - ANALYZE needed?
questions?

- Josh Berkus
  - josh@pgexperts.com
  - PGX: www.pgexperts.com
  - Blog: www.databasesoup.com

- Upcoming Events
  - pgCon: Ottawa, May 21-24
  - Postgres Open: Sept. 17-19, Chicago

Copyright 2013 PostgreSQL Experts Inc. Released under the Creative Commons Attribution License. All images are the property of their respective owners. The Don’t Panic slogan and logo is property of the BBC, and the Dilbert image belongs to Scott Adams and is used here as parody.