PostgreSQL Magazine

News and stories from the Postgres Community

PostgreSQL 9.1 is out !

10 awesome new features

NoSQL : The Key Value store everyone ignored

Interview **Stephan Kaltenbrunner**

Opinion : Funding PostgreSQL features

Waiting for 9.2 : Cascading Streaming Replication

Tips & tricks **PostgreSQL in Mac OS X Lion**

May 2012

#01

About PG Mag

PostgreSQL Magazine is edited by and for the PostgreSQL Community.

Editor: Damien Clochard

Layout Editor: Cédric Gemy

Writers:
Zohaib Sibte Hassan, Marc Cousin, Simon Merrick, Joshua Drake, Hubert Lubaczewski

Reviewers:
Thom Brown, Ray O'Donnell, Osvaldo Tulini, Sivakumar, Matt Keranen, Jaime Casanova, Luca Ferrari, Nicolas Thauvin, Bob Hetrick, Josh Kupershmidt

Tools: Scribus 1.4 / Gimp 2.6

Editorial

This is it! The **first issue** of the first non-profit print media created by and for the Postgres community. Time went by since our demo issue last year! Overall we printed **2000+ hard copies** (distributed for free in events like PG Con 2011 in Ottawa or FOSDEM 2012 in Brussels). We also know that so far the online PDF version was read by **10000+ visitors**. So we took some time to analyze the feedback we received and we learned many things. Many thanks to every one that sent us feedback.

For me, the great news in all this is that we proved that we can run a print media as an open source community-driven project! We all know that the PostgreSQL community can produce robust source code and quality documentation. But could we use this strong energy to generate something else? Why not a media aimed at the PostgreSQL user base? In less than a year, what was just a theoretical question has become reality.

In this first issue, we will talk of the wonders of **PostgreSQL 9.1** and we'll get a sneak peek at the upcoming **9.2 version**. We'll take a look at **Mac OSX Lion** and we'll ask **Stephan Kaltenbrunner** a bunch of questions, one of the sysadmins behind the postgresql.org infrastructure... We really hope you will find ideas and insights in this magazine!

This time, **dozens of benevolent** contributors made this issue possible: writers, editors, reviewers, etc. Kudos to them! We try to work just like any "regular" open source project: content is edited in wiki, the source files are stored in github and we have a contributors' mailing list. If you want to join us, just check out page 35 to see how you can help.

And finally, I would like to thank our benefactors... Fotolia.com has offered us a free subscription plan to access their stock photo database. We also received funding from PostgreSQL Europe (PGEU) and Software in the Public Interest (SPI). This funding will allow us to release this issue and the next one.

For the new issue, we plan on sending more free copies to many PostgreSQL conferences and meetings around the world. So if you're organizing a PostgreSQL related event, just send us a message at contat@pgmag.org and we'll try to provide you with hard copies of this magazine.

Damien Clochard
Editor in Chief

PostgreSQL Magazine 01

TABLE OF CONTENTS

UPCOMING EVENTS

PGCon 2012
May 15-18 / Ottawa, Canada

PGCon is an annual conference for users and developers of PostgreSQL. PGCon is the place to meet, discuss, build relationships, learn valuable insights, and generally chat about the work you are doing with PostgreSQL.

http://www.pgcon.org/2012/

Postgres Open 2012
September 16-19 / Chicago, USA

Postgres Open is a non–profit, community–run conference series in the United States focused on business users, database professionals and developers of PostgreSQL. Postgres Open will promote the business of PostgreSQL as well as its use and development.

http://postgresopen.org/

PG Day France
June 7 / Lyon, France

The major French-speaking PostgreSQL Conference is back this year with some great talks : PostGIS 2.0, fuzzy logic in PostgreSQL, High-Availibility, Benchmarking, Foreign Data Wrappers and more !

http://www.pgday.fr/

PG Conference Europe 2012
October 23-26 / Prague, Czech Rep.

This year, the major European PostgreSQL conference will take place at the Corinthia hotel in Prague. You can expect three days packed with presentations about PostgreSQL and related technologies, one day of extended trainings, and the usual hallway and social track!

http://2012.pgconf.eu/

For more information on local and international PostgreSQL user conferences, check out the wiki !

http://wiki.postgresql.org/wiki/Events

From Our Readers

Corrections, ideas and requests: this section is the place for telling us what you think of each issue of PostgreSQL Magazine!

What are your plans?

Dear people from PostgreSQL Magazine,

We (database people of the City of Vienna) have read (part of) the pilot issue and this magazine would be interesting for us.

Do you have any ideas how often you plan to publish the magazine? Will it be possible to subscribe? Do you have any estimates how much it might cost?

Even if you have no definite answers to any of the above yet, we thought it would be a good idea to give you feedback.

Yours, Laurenz Albe

The editors reply: For now we don't know how often we will publish and how much it will cost. It is not possible to subscribe to the magazine right now but we're working on this. All these questions are currently being discussed. Follow us on twitter or subscribe to our newsletter to stay tuned!

http://twitter.com/#!/pg_mag
http://pgmag.org/join_the_newsletter

Online version not universally readable...

... due to requiring some kind of proprietary plugin. How about limiting the delivery technology to some freely available formats?

I will of course download and read the PDF, but it does feel like I'm excluded. The only other online publication I pay money for is LWN, and I pay because of its *content*. In fact, given its technical audience its simplicity of design is probably an attraction.

Perhaps I'm not part of the mainstream target audience? Aiming more at the corporate crowd who need flashy colors? Dunno...

Regards,

Mark Lawrence

The webmaster replies: I think you're talking about the online PDF reader that requires a flash plugin. I know this is a proprietary format but you should be able to read it with free alternatives to the Adobe plug-in...Have you tried gnash or lightspark ?

Still the Flash format is not open. I know that too. For now, I don't know any way to display a magazine online with HTML5. I know scribd does it for slides but this is not appropriate for a magazine.

If you have any idea on how we can improve that, I'm open to your suggestions.

PostgreSQL magazine on tablets

Hi,

I am thrilled to see a magazine for PostgreSQL. I like the format and the open source experiment around the magazine. I found the announcement via the hacker news. So I am sure the news about the publication will be viral very soon.

It would be great if the magazine is made available on the iPad too. I am sure there are others who would like to read on other ereaders like kindle or nook. I use iBooks, Flipboard, and Zinio to primarily read articles and magazines on the iPad. I especially love the flipboard interface to read articles on the iPad.

I know that the PDF can be read on the iPad or other tablets. However, it’s a bit inconvenient, since I need to pull it manually. Having the magazine pushed to my device(s) would save me a lot of trouble. Pushing it to my device will also make it easy for me to access the magazine, especially when I am not particularly busy and want to catch up on my news and magazine reading.

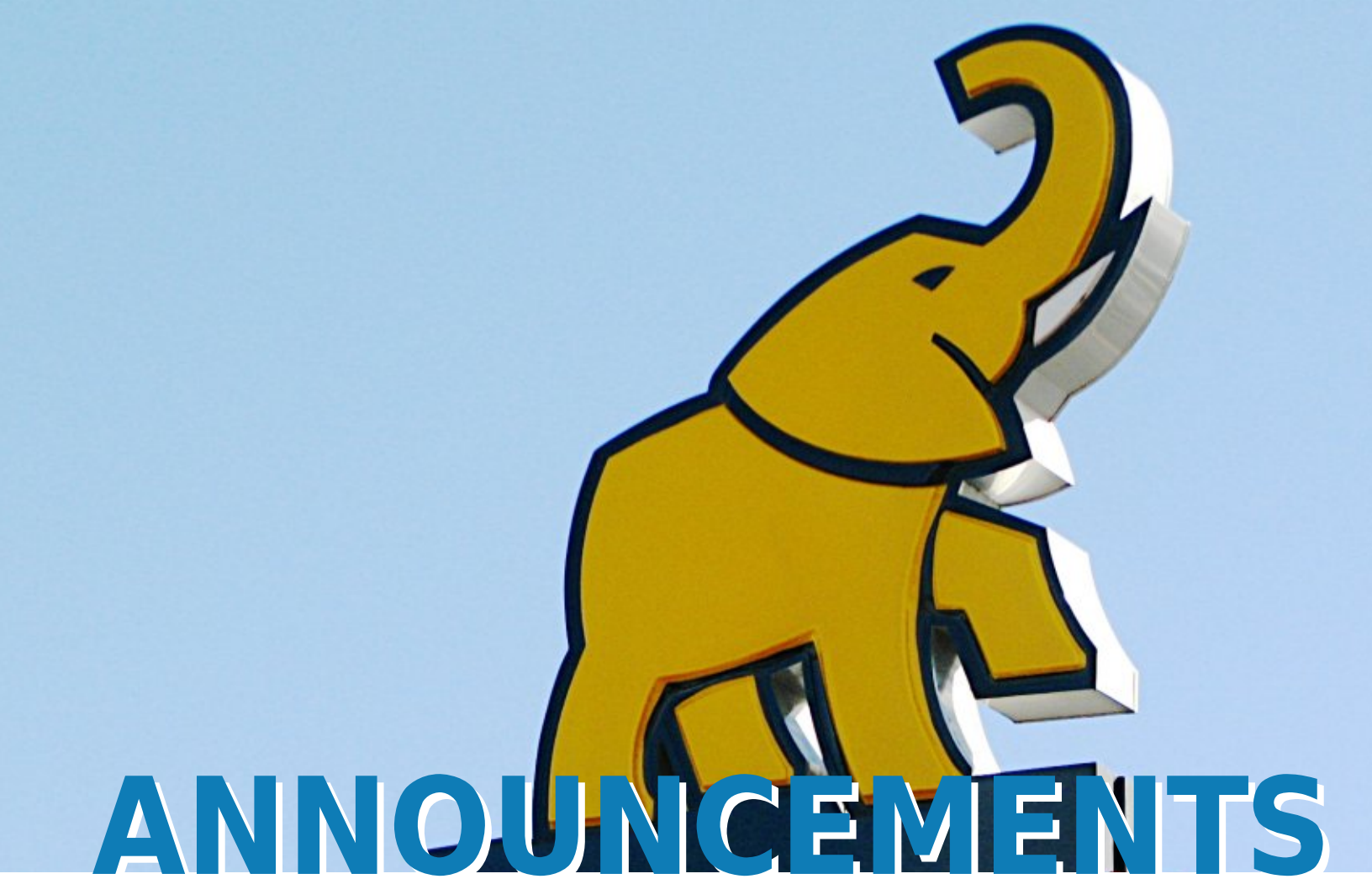
Thanks for the great work so far.

John

The editors reply: Right now we're focusing on the paper version. We're happy to provide an online version but this is not our main goal. Behind the word “tablet”, there are so many devices and formats! Releasing the magazine for each tablet would take us too long. This is why we're going to stick with the PDF format for a while.

Send us feedback !

Send your opinion about what you read in PostgreSQL Magazine to feedback@pgmag.org. Letters may be edited for length and clarity. We consider that any message we receive can be published.



ANNOUNCEMENTS

PostGIS 2.0 is out !

PostGIS, a project of the **Open Geospatial Consortium**, hit version 2.0 in April 2012. This PostgreSQL extension stores spatial data and allow users to search that data based on geographic criteria as well as on standard relational criteria.

The development process for 2.0 has been very long, but has resulted in a release with a number of exciting new features, such as: Raster data and raster/vector analysis in the database, Topological models, 3D and 4D indexing, Integration with the PostgreSQL extension system, Multi-file import support in the shapefile GUI, and more...

http://www.postgis.org/

Postgres-XC 1.0 Beta 1 released

Postgres-XC is a write-scalable multi-master symetric cluster based on PostgreSQL. This beta version is based on PostgreSQL 9.1 Beta 2 and all the fixes of PostgreSQL 9.1 stable branch will be backported in Postgres-XC 1.0 stabilized version. It is distributed under PostgreSQL license.

http://postgres-xc.sourceforge.net

EnterpriseDB launches Cloud Database

EnterpriseDB announced the general availability of PostgreSQL 9.1 and Postgres Plus® Advanced Server 9.0 on the Amazon Web Services platform.

http://cloud.enterprisedb.com

PGXN Client 1.0

The PGXN Client is a command line tool designed to interact with the PostgreSQL Extension Network allowing searching, compiling, installing, and removing extensions in a PostgreSQL installation or database.

http://pgxnclient.projects.postgresql.org/

Skytools 3.0

Skytools is developed by Skype for replication and failover. It includes a generic queuing framework (PgQ) and easy-to-use replication implementation (Londiste). The major new features compared to 2.1 are : Cascaded queuing, Parallel copy, Multi-database ticker and Londiste handler modules for custom processing.

http://wiki.postgresql.org/wiki/SkyTools

VMware Expands vFabric with PostgreSQL

VMware created vFabric Postgres as a closed-source fork of PostgreSQL created by VMware, with significant improvements for the vSphere environment. vFabric Postgres' “elastic database memory” dynamically adapts to changing workloads to achieve greater memory efficiency and higher consolidation ratios. vFabric Postgres is also available on http://cloudfoundry.com/.

PostgreSQL in Google Summer of Code

Google has once again selected the PostgreSQL project to take part in Summer of Code, during 2012. Google will be funding several students to work with mentors from our project in order to hack PostgreSQL code for the summer.

http://www.postgresql.org/developer/

Scalr 2.5 supports PostgreSQL

Open-source project Scalr is very pleased to announce its support for PostgreSQL. From now on, Scalr and PostgreSQL users can enjoy low maintenance on the cloud — including automatic backups, recovery, and ease of administering, along with added auto-scaling.

http://scalr.net/features/databases/postgres

Heroku Launches PostgreSQL Standalone Service

Since 2007, PostgreSQL has been available as module inside the Heroku Platform. But now Postgres is also available as a standalone service. Heroku is probably the world largest PostgreSQL hosting service with more than 150 000 PostgreSQL databases up and running.

https://postgres.heroku.com/

Npgsql 2.0.12 beta3 released

Npgsql is a .Net data provider written 100% in C# which allows .Net programs to talk to PostgreSQL backends. Npgsql is licensed under BSD.

http://www.npgsql.org

Postgres Enterprise Manager v2.1

EnterpriseDB released a new version of their management tool with SNMP and email alerts, an Audit Manager and user access controls. Postgres Enterprise Manager (PEM) 2.1 adds a series of highly configurable dashboards, new features for scheduling and organizing management tasks and enhanced alerting capabilities.

INTERVIEW
The admin behind the scenes

You probably don't know his name, even if you're a PostgreSQL zealot. But Stephan Kaltenbrunner (also known as "Mastermind") is a key member of the sysadmin team that builds and maintains the postgresql.org infrastructure. This tall and quiet Austrian guy has been involved in the web platform for years now, so we've decided to ask him about the evolution of the project through the last decade and the future of PostgreSQL's web presence: the new PGXN network, the death of pgFoundry and the recent upgrade of the front website….

PostgreSQL Mag: Who do you work for?

Stefan Kaltenbrunner: I work for Conova Communications GmbH in Salzburg, Austria as the team leader for the Engineering & Consulting department. Conova is a highly IT-Service company focusing on providing high-quality services out of its own large datacenter in Salzburg, and almost any internal or customer facing-project is running PostgreSQL in the backend.

PGM: How are you involved in the PostgreSQL project?

SK: Aside from occasionally giving talks, probably the most visible (or invisible depending on how you look at it) thing is my involvement in the postgresql.org sysadmin and web teams. PostgreSQL is a huge, well established and well respected open source project, and as part of being a real open source project we also run our own infrastructure, fully community managed and operated.

PGM: Is it hard to be an admin of postgresql.org? How many hours per week do you spend on this?

SK: Heh, “how hard is it?” — that's a really good question. Back in the '90s postgresql.org used to run on a single FreeBSD server sponsored and operated by hub.org. This was later expanded to 3 or 4 servers but it was still a small environment that was adequate to the needs of the projects 10-15 years ago.

However over the past 5-10 years with PostgreSQL getting rapidly developed, improved and attracting a larger community — both in terms of developers, contributors and users — the postgresql.org infrastructure grew. The community got new servers sponsored by companies other than hub.org; still inheriting a lot of the FreeBSD past but it helped to get new people involved with the topic.

Over the last few years the sysadmin team grew with even more serious server sponsors and a significant increase in what people wanted the infrastructure to deliver. We started working on a completely new concept for how we as a team could successfully design and run the postgresql.org infrastructure for the next 5-10 years, looking at new technologies and learning from past experience.

The end result is, in my opinion, a nice reflection on the rapid evolution and the technical excellence of PostgreSQL itself. We now run almost all services using cutting edge, KVM-based full virtualisation technology and we also have a very sophisticated custom deployment and management framework that allows us to do most stuff in minutes. The system as a whole has a very high level of automation. Newly installed VMs or even services added to existing ones are fully functional and automatically backed up, and monitoring and patching for security issues is now a matter of seconds which makes managing our 40+ VMs as well as the VM-Hosts much easier and more sustainable.

PGM: How many other sysadmins work on the infrastructure?

SK: The “core infrastructure team” — rather loosely defined here as the people with root on all VMs and the VM-Hosts and actively working on them — consists of four people: Magnus Hagander, Dave Page, Alvaro Herrera and me.

However there are a larger number of people with elevated permissions on specific VMs, or with commit access to parts of the underlying code — a quick list of the people involved can be found on http://wiki.postgresql.org/wiki/Infrastructure_team. Work on the infrastructure is — similiar to most of the PostgreSQL ecosystem — coordinated through mailinglists, IRC and for emergencies IM.

PGM: How many services are running?

SK: It depends on how one defines a service. Most of the publicly facing services we provide which present the “face” of PostgreSQL, like the website, are actually composed of a number of different servers running dozens of services. Right at this moment we have exactly 571 services monitored in nagios on a total of 61 hosts not including all the stuff we have in place for long-term capacity planning and trending that is not directly alerting which adds another 250-350 services. It is important to mention that almost all of these are fully deployed automatically — so if somebody adds a PostgreSQL server instance on an existing box, monitoring and service checking will be fully automatically enabled.

“At this moment we have exactly 571 services monitored in nagios on a total of 61 hosts”

PGM: How many servers are used to run those services?

SK: We try keeping an up-to-date list of physical servers on http://www.postgresql.org/about/servers/, so at the time of writing we had ten VM-Hosts in five different datacenters in the US and Europe. In addition to that we have a number of additional servers, not listed there on purpose, used for very specific tasks like running our monitoring system or our centralised backup system.

PGM: Who donates these servers?

SK: Companies using PostgreSQL or providing Services based on it basically. Years ago we had to rely on very few companies giving us gear they no longer needed. These days we actually evaluate the hardware we are getting offered seriously; we look at the kind of datacenter the server might be hosted at and what kind of support we can get from that place in terms of remote hands, response times and bandwidth as well as the hardware itself. In recent times we are also starting to look into buying hardware ourselves to provide an even better user experience and to improve the reliability of our services by having less dependency on somebody else to provide us with spare parts or a replacement in short time.

> **“we have outgrown what pgFoundry can really do for us and I think we should move on”**

PGM: How do you monitor this?

SK: In two words: Nagios and Munin. In some more, we use Munin for trending, Nagios for alerting and monitoring externally through NRPE. We also have a custom-developed configuration file tracking system for keeping critical configuration files across the infrastructure properly version controlled. The alerting is usually happening through email directly to the individual sysadmin members.

PGM: You seem to know Nagios very well. How would you compare it to other monitoring software such as Zabbix or Hyperic?

SK: Indeed I know Nagios pretty well, though in recent times I have seen a lot of movement to look into Icinga, and also into using some of the alternative user interfaces provided by some projects. My personal favourite there is Thruk, which is maybe a bit boring for the Web 2.0 generation but for me it has the right balance of simplicity and clarity like the original Nagios CGI, while providing some very useful capabilities on top - like very powerful filtering & view capabilities that are extremly handy in installations having a few thousand or tens of thousands of services.

PGM: The sysadmin team performed an upgrade of the main www.postgresql.org platform in November 2011. Can you tell us a little more about that new version?

SK: That was a huge thing; the existing website framework was developed like 10 years ago by community members based on PHP4 that has now moved on to other things. And over the past few years we have only been making modest changes to it because people never did fully understand it, and the number of PHP-experienced developers within the community interested in hacking on it was, say, “limited”. It is mostly thanks to Magnus Hagander — who implemented most of the new framework — that we now have a pretty nice and scalable Python and Django-based framework. The new framework fully integrates Varnish as a front-end cache, Django in the backend and a myriad of what I would call “community platform services” like:

- complete community account management — providing single-sign-on for all community services
- management interfaces for news, organisations, events and training for the main website (including automatic purging of cached content on the frontend systems)
- RSS-feed generation and aggregation
- management of dynamic content like surveys
- scalability

We also completely revamped our download system in the process; this now means that we are hosting all downloads, except for the one-click installers, by ourselves on servers operated by the infrastructure team. The old and often hated page, with the enormous amount of flags for the many countries which we previously used mirrors, is now gone for good.

PGM: One word about pgFoundry, the PostgreSQL software forge.... There's been some complaints about it. We see lots of projets moving to other code sharing platforms (such as github) and some community members even think pgFoundry should be closed. What's your opinion?

SK: Once upon a time pgFoundry (and gborg for those who remember) were a very important service for the community. When those services were started, getting proper hosting was expensive and complex to get, especially for the myriad of smaller contributors in the PostgreSQL ecosystem. However, time has evolved, getting proper hosting is now reasonably easy to get in most places and there is a large set of very successful open source focused hosting systems. Contributors can choose from those that are very actively developed and improve at an amazing pace. The only real missing feature most people have with those is proper mailinglist support. The PostgreSQL community is using mailinglists as communication and coordination media in a much more intense form than other projects (except maybe the Linux kernel or the Debian project as a whole) but most of the code-sharing and hosting platforms have no or only very limited support for mailinglists. However we will be working with the projects affected by the upcoming pgFoundry shutdown on this and I expect that we will be finding a solution in most cases.

To sum this up — pgFoundry was a very important part of the infrastructure in the past in evolving and growing the community. However PostgreSQL is a very large and well established project with a rapidly growing community and we have now evolved to a point where we have outgrown what pgFoundry can really do for us. And I think we should move on and work on stuff we are really good at — running the core and mission critical infrastructure for a major open source project: PostgreSQL itself.

PGM: And what do you think of the new PostgreSQL Extension Network (PGXN)?

SK: With my community hat on I see this as a great thing to provide a lot more exposure to the great ecosystem around PostgreSQL and also see it as a showcase of one of the most powerful, but often overlooked capabilities of PostgreSQL; “Extensibility”. With my hat as a manager and team lead at Conova “on” I think that it is hard to deal with PGXN (or the concept it is loosely modeled around CPAN) in a complex production scenario usually requiring installation from binary packages or complex approval procedures for source code installations. PGXN is a clear and nice example of what the community is capable of pulling off these days, but as with a lot of services in the world wide web - only time will tell how successful it will be in the long run :)

PG Conference Europe, Amsterdam 2011

PGM: What's the biggest feature you'd want to see in the next PostgreSQL release?

SK: There are three large features actually:

- scalability — on large systems, thanks to a lot of work from Robert Haas, 9.2 will scale extremely well on modern boxes so for reads I consider this mostly done, for writes we still have ways to go
- better partitioning — the current system based on inheritence was nice to get us started and was sufficient for the time being, but going forward we probably need a radically new design embedded much deeper in the backend than what we have now
- parallel query — the industry as a whole is moving to more and more cores per socket at an amazing pace right now (to the point that commercial vendors need to update ther licencing on an almost yearly basis now), and we will have to take on the challenge of parallelising operations over multiple cores sooner rather than later

PGM: What other things are you interested in outside of open source and PostgreSQL?

SK: There is something other than open source and PostgreSQL?! No seriously, I like spending time with my family as well as going out for cinema or playing cards with friends and if that gets coupled with a good glass of Austrian white wine I'm even happier...

The key value store everyone ignored

Yes I know you are really happy with your "persistent" Key Value store. But did anybody notice hstore that comes with PostgreSQL ?

I find PostgreSQL to be a really great RDBMS that is overlooked all the time. It even has a great publisher/subscriber system (or LISTEN/NOTIFY in terms of PostgreSQL) that a lot of people may have implemented using Redis, RabbitMQ etc. For people who have not lived anything other than MySQL, I would simply ask them to try out Postgres.

Instead of looking at benchmarks, I will be focusing on a key value store that is ACID compliant for real! Postgres takes advantage of its storage engine and has an extension on top for key value storage. So the plan is to have a table with column(s) that have a datatype of hstore, which in turn has a structure-free storage. Thinking of this model multiple analogies throw themselves in. It can be a Column Family Store just like Cassandra where row key can be PK of the table, and each column of hstore type in the table can be thought of like a super column and each key in the hstore entry can be a column name. Similarly you can imagine it somewhat like Hash structures in Redis (HSET, HDEL) or 2 or 3 level MongoDB store (few modifications required). Despite being similar (when little tricks are applied) to your NoSQL store structures, this gives me an opportunity to demonstrate to you some really trivial examples.

Let's setup our system first. For my experiment I will be using Postgres 9.1 and I will compile it from source. Once inside the source directory you can: `./configure && make install` to install your Postgres. Don't forget to install the extensions in the contrib directory: `cd ./contrib && make install`. Once you have it set up, you can create your own database cluster and start the server (Hints: use initdb and pg_ctl). Then launch psql and make sure you install your hstore extension:

```
CREATE EXTENSION hstore;
SELECT 'foo=>bar'::hstore;
```

If everything goes well you should be able to see a table output. Now we are ready to do some DDL. I created a table named my_store as defined below:

```
CREATE TABLE my_store
(
id character varying(1024) NOT NULL,
doc hstore,
CONSTRAINT my_store_pkey PRIMARY KEY (id)
)
WITH (
OIDS=FALSE
);

CREATE INDEX my_store_doc_idx_gist
ON my_store
USING gist
(doc);
```

As you can see I've created a table with a hstore column type and one GiST index (for operators ?, ?&, ?| etc.). You can check out the documentation to have a look at the different types of operators available. Now that I have a database and tables set up, I use a simple script to populate it with about 115K rows from a Twitter stream. Now keep in mind that it's real life data and I was interested in querying a few basic things from the collected data. For example, how many people are using hash tags, or doing mentions, or were posting links in the tweets? For doing this I wrote a simple python script using tweepy and psycopg2 and ran it for few hours. For each tweet in my store, I added a key value pair of 'has_hashtags⇒:t' if there were any hash tags in the tweet. Similarly I introduced 'has_urls' and 'has_mentions' if they were present in tweet. I will be using these keys along with my GiST index to query my table later on.

“Combining the power of relational and key value style under one store”

So, after populating my data with 115,142 tweets the database grew to a size of 239691780 bytes (just 228MB). Now comes the fun part. I was totally blown away by what I can achieve by combining the power of relational and key value style under one store. So, for example I want to query all the tweets tweeted at unix timestamp of 1323446095 (since I stored the timestamps as a string here is what my query looks like):

```
SELECT doc -> 'text' as tweet, doc -> 'created_at'
as created_at
FROM my_store
WHERE doc @> 'created_at=>00001323446095';
```

I can add a simple count or any other well-known SQL aggregate function without complicating my data store with anything specific like map reduce or the hassle of learning a new language. Note that I padded my timestamp value with zeros since I am only storing strings as values. Also I am utilizing the @> operator, and that's going to use GiST to perform a quick bitmap index scan instead of a sequential scan. That's pretty good for starters. Let's try to fetch out all the tweets that had hash tags in them:

```
SELECT doc -> 'text' as tweet, doc -> 'created_at'
as created_at
FROM my_store
WHERE doc @> 'has_hashtags=>:t';
```

Yes querying the entire database and pulling out the complete data (that you won't probably do because you page the data) gives me 14689 rows in just under 360ms on average. Since we have SQL to hand let's make the query a little more complicated, and use a different operator for the same stuff and also sort the data by ‘created_at’:

```
SELECT doc -> 'text' as tweet, doc -> 'created_at'
as created_at
FROM my_store
WHERE doc @> 'has_hashtags=>:t' AND doc ?
'has_urls'
ORDER BY doc -> 'created_at' DESC;
```

This sounds tasty! But there's more: Postgres has other operators. For example, you can also pull out hash tagged tweets with urls or mentions:

```
SELECT doc -> 'text' as tweet, doc -> 'created_at'
as created_at
FROM my_store
WHERE doc @> 'has_hashtags=>:t' AND doc ?|
ARRAY['has_urls','has_mentions']
```

This is not all you can do with it! hstore comes with all sorts of operators and index systems that you can use to query hash stores. Check them out in the documentation. Now, despite the NoSQL boom I think we have some great examples and reasons of why RDBMS still remains a core part of many market giants (Facebook being something every-one knows). Postgres just gives me one more reason to not ignore RDBMS. So, if you have been moving to some document stores just for the reason that RDBMS doesn't provide them; think again! You can get the same rock solid durability with structure-free systems.

Note

Prior to PostgreSQL 8.2, the containment operators @> and <@ were called @ and ~, respectively. These names are still available, but are deprecated and will eventually be removed. Notice that the old names are reversed from the convention formerly followed by the core geometric data types! Please also note that the ⇒ operator is deprecated and may be removed in a future release. Use the hstore(text, text) function instead.

About the article

The originial article is available at

http://pgmag.org/0113

About the author

Zohaib Sibte Hassan (aka maxpert) is a contributor to DooPHP and creator of micro framework MiMViC. He works at Bumpin Social Media as Senior Software Engineer. He's been busting buzzwords on NoSQL, NodeJS, and various other stuff. Checkout his blog at **http://blog.creapptives.com/**

AWESOME 10 new features in PostgreSQL 9.1

PostgreSQL 9.1

The latest major version of PostgreSQL has so many innovations and improvements ! We really struggled to choose the ten most disruptive features and showcase each of them, with a step-by-step example. Let's see what PostgreSQL 9.1 is gonna change for your data severs !

In September 2011, The PostgreSQL Global Development Group released PostgreSQL 9.1. This latest version of the leading open source database offers innovative technology, unmatched extensibility, and new features.

Responding to Users

Version 9.1 delivers several features which users have been requesting for years, removing roadblocks to deploying new or ported applications on PostgreSQL. These include synchronous replication, per-column collations and unlogged tables.

Advancing the State of the Art

As always the PostgreSQL community innovates with cutting-edge features! Version 9.1 includes 4 items which are new to the database industry, such as: K nearest neighbor (KNN) indexing, serializable snapshot isolation (SSI), writeable common table expressions (wCTE) and security-enhanced (SE) PostgreSQL.

Extending the Database Engine

PostgreSQL's extensibility enables users to add new functionality to a running production database, and use them for tasks no other database system can perform. Version 9.1 adds new extensibility tools, including: SQL/MED support, Extensions and the PostgreSQL extensions network (PGXN).

In PostgreSQL's 25th year of database development, the community continues to advance database technology with every annual release!

"PostgreSQL 9.1 provides some of the most advanced enterprise capabilities of any open source database, and is backed by a vibrant and innovative community with proven customer success. PostgreSQL is well positioned for building and running applications in the cloud," said Charles Fan, Sr. VP R&D, VMware.

Check out the wiki

This article is heavily based on the wiki page called "What's new in PostgreSQL 9.1?".

http://www.pgmag.org/0115

About the author

Marc Cousin has been a PostgreSQL and Oracle DBA since 1999. He is involved in the Postgres community in various ways, including providing help on the French-speaking forum and participating in the PG Day France. He works at Dalibo as a Senior DBA.

SQL/MED

SQL/MED (for Management of External Data) is an extension to the SQL:2003 standard that provides extensions to SQL that define foreign-data wrappers (FDW) and datalink types to allow SQL to access data stored outside of the RDBMS.

PostgreSQL 9.1 already has a bunch of FDW available : Oracle, MySQL, SQLlite, anything providing an ODBC driver, LDAP, couchdb, redis, Amazon S3 storage, CSV files, Twitter, Google Search, HTML pages, RSS feeds, etc. There is even an extension that aims to make FDW development easier when developing them in Python, Multicorn:
http://multicorn.org/

For an almost complete list, check out the PostgreSQL wiki :

http://wiki.postgresql.org/wiki/Foreign_data_wrappers

Here is an example, using the `file_fdw` extension.

We'll map a CSV file to a table.

```
=# CREATE EXTENSION file_fdw WITH SCHEMA extensions;
\dx+ file_fdw
         Objects in extension "file_fdw"
                Object Description
-----------------------------------------------------
foreign-data wrapper file_fdw
function extensions.file_fdw_handler()
function extensions.file_fdw_validator(text[],oid)
```

This next step is optional. It's just to show the 'CREATE FOREIGN DATA WRAPPER' syntax:

```
=# CREATE FOREIGN DATA WRAPPER file_data_wrapper
-# HANDLER extensions.file_fdw_handler;
CREATE FOREIGN DATA WRAPPER
```

The extension already creates a foreign data wrapper called `file_fdw`. We'll use it from now on.

Why SQL/MED is cool

For more details, check out Robert Haas' blog post on SQL/MED at:
http://pgmag.org/0116
The original article is available at:
http://pgmag.org/0110

SQL/MED makes PostgreSQL a powerful tool for enterprise data integration

We need to create a 'server'. As we're only retrieving data from a file, it seems to be overkill, but SQL/MED is also capable of coping with remote databases.

```
=# CREATE SERVER file
-# FOREIGN DATA WRAPPER file_fdw ;
CREATE SERVER
```

Now, let's link a '/tmp/data.csv' file to a `stats` table:

```
=# CREATE FOREIGN TABLE stats (
-#    field1 numeric,
-#    field2 numeric
-#    )
-#    server file options (
-#      filename '/tmp/data.csv',
-#      format 'csv',
-#      delimiter ';'
-#     );
CREATE FOREIGN TABLE
=# SELECT * from stats ;
 field1 | field2
--------+--------
    0.1 |    0.2
    0.2 |    0.4
    0.3 |    0.9
    0.4 |    1.61.6
```

For now, foreign tables are SELECT-only but more improvements will be coming in the next major versions.

Per-column collation

In multilingual databases, users can now set the collation for strings on a single column. This permits true multilingual databases, where each text column is a different language, and indexes and sorts correctly for that language.

Let's say you are using a Postgres 9.0 database, with an UTF8 encoding and a `de_DE.utf8` collation (alphabetical sort) order, because most of your users speak German. If you had to store french data too, and had to sort, some french users could have been disappointed:

```
=# SELECT * from (values ('élève'),('élevé'),('élever'),('Élève'))
-# as tmp order by column1;
  column1
---------
  élevé
  élève
  Élève
  élever
```

It's not that bad, but it's not the french collation order: accentuated (diactric) characters are considered equal on first pass to the unaccentuated characters. Then, on a second pass, they are considered to be after the unaccentuated ones. Except that on that second pass, the letters are considered from the end to the beginning of the word. That's a bit strange, but that's the french collation rules...

With **Postgres 9.1**, two new features are available:

You can specify collation at query time:

```
=# SELECT * FROM (VALUES ('élève'),('élevé'),('élever'),('Élève'))
-# AS tmp ORDER BY column1 COLLATE "fr_FR.utf8";
  column1
---------
  élève
  Élève
  élevé
  élever
```

The collation order is not unique in a database anymore.

You can specify collation at table definition time:

```
=# CREATE TABLE french_messages (message TEXT
COLLATE "fr_FR.utf8");
=# INSERT INTO french_messages VALUES
('élève'),('élevé'),('élever'),('Élève');
=# SELECT * FROM french_messages ORDER BY message;
  message
---------
  élève
  Élève
  élevé
  élever
```

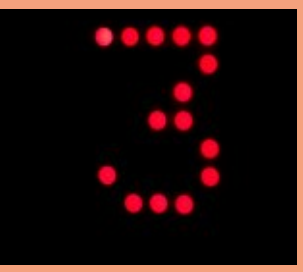
And of course you can create an index on the message column, that can be used for fast french sorting. For instance, using a table with more data (400k rows) and without collation defined:

```
=# CREATE TABLE french_messages2 (message TEXT);
=# INSERT INTO french_messages2
-# SELECT * FROM french_messages, generate_series(1,100000);
=# CREATE INDEX idx_fr_ctype
-# ON french_messages2 (message COLLATE "fr_FR.utf8");

=# EXPLAIN SELECT * FROM french_messages2
-# ORDER BY message;
                                QUERY PLAN
---------------------------------------------------------------------------
 Sort  (cost=62134.28..63134.28 rows=400000 width=32)
   Sort Key: message
   ->  Seq Scan on french_messages2  (cost=0.00..5770.00 rows=400000 width=32)

=# EXPLAIN SELECT * FROM french_messages2
-# ORDER BY message COLLATE "fr_FR.utf8";
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using idx_fr_ctype on french_messages2(cost=0.00..17139.15 rows=400000 width=8)
```

Unlogged tables



When performance is more important than durability, unlogged tables provide a way to improve performance while keeping the data managed within PostgreSQL. Removing logging reduces I/O overhead, yielding performance improvements up to 10 times faster when compared to logged tables. Scenarios expected to leverage unlogged tables include web session data, real time logging, ETL and temporary/intermediate tables for functions.

Introduced in **PostgreSQL 9.1**, unlogged tables are designed for ephemeral data. They are much faster on writes, but won't survive a crash (it will be truncated at database restart in case of a crash). Currently, **GiST** indexes are not supported on unlogged tables, and cannot be created on them.

They don't have the WAL maintenance overhead, so they are much faster to write to. Here is a (non-realistic) example:

First let's create an unlogged table:

```
# CREATE UNLOGGED TABLE testu (a int);
CREATE TABLE
```

Now for comparison purposes, we'll create a second table — identical, but logged:

```
# CREATE TABLE test (a int);
CREATE TABLE
```

Let's put an index on each table:

```
# CREATE INDEX idx_test on test (a);
CREATE INDEX
# CREATE INDEX idx_testu on testu (a);
CREATE INDEX
```

Now we'll see how faster we can go:

```
=# \timing
Timing is on.
=# INSERT INTO test SELECT
generate_series(1,1000000);
INSERT 0 1000000
Time: 17601,201 ms
=# INSERT INTO testu SELECT
generate_series(1,1000000);
INSERT 0 1000000
Time: 3439,982 ms
```

With this example, the unlogged table is 5 times faster than the regular one. Even when using the `COPY` command, the write performance is much faster with unlogged tables.

Performances are greatly improved for ephemeral data

But remember! If your PostgreSQL crashes you will lose all the content, even if you force a checkpoint:

```
$ cat test.sql
INSERT INTO testu VALUES (1);
CHECKPOINT
INSERT INTO testu VALUES (2);

$ psql -f test.sql ; killall -9 postgres postmaster
INSERT 0 1
CHECKPOINT
INSERT 0 1
postmaster: no process found
```

The server was forced to shutdown immediatly. And after restart:

```
# select * from testu;
    a
------------

(0 rows)
```

Many “NoSQL” engines advertise a speed advantage over “traditional” RDBMS engines, by employing a similar unlogged approach. PostgreSQL 9.1 provides an option, allowing a database designer to choose between performance and crash safety for tables in the same database, without having to abandon the rich feature set provided by PostgreSQL.

In other words

unlogged tables are very efficient for caching data, or for anything that can be rebuilt in case of a crash.

Synchronous replication

Synchronous Replication enables high-availability with consistency across multiple nodes. Synchronous replication supports "2-safe replication" which ensures transactions have been confirmed by a standby server in addition to the master, greatly limiting the possibility of data loss. Only PostgreSQL has transaction level synchronous replication, allowing users to choose per transaction between response time and data safety.

Let's say you have set up a "classic" PostgreSQL Hot Standby cluster with streaming replication. To get synchronous, just change the following option in the master's `postgresql.conf`:

```
synchronous_standby_names = 'newcluster'
```

This is the `application_name` from our `primary_conninfo` from the slave. Just do a

```
pg_ctl reload
```

and this new parameter will be set. Now any commit on the master will only be reported as committed on the master when the slave has written it on its own journal, and acknowledged it to the master.

One of the really great features of synchronous replication is that it is controllable per session. The parameter `synchronous_commit` can be setted to local in a session, if it does not require this synchronous guarantee, or turned off completely if you want asynchronous commit even on master. If you don't need it in your transaction, just type the command below and you won't pay the penalty.

```
SET synchronous_commit TO local;
```

Of course, Synchronous replication is not the only improvement of this new version. There are many other new replication features for PostgreSQL 9.1! Here's a quick list:

- In 9.0, the user used for replication had to be a superuser. It's no longer the case, there is a new 'replication' privilege.
- `pg_basebackup` is a new tool to create a clone of a database, or a backup, using only the streaming replication features.
- The slaves can now ask the master not to vacuum records they still need.
- Two new systems views called pg_stat_replication and pg_stat_database_conflicts
- Replication can now be easily paused on a slave.
- Restore points can now be created.

A word of warning

In synchronous mode, transactions are considered committed when they are applied to the slave's journal, not when they are visible on the slave. It means there will still be a delay between the moment a transaction is committed on the master, and the moment it is visible on the slave. This still is synchronous replication because no data will be lost if the master crashes.

Extensions

While PostgreSQL has always been extensible, now users can easily create, load, upgrade, and manage any of dozens of database extensions using the EXTENSION database object.

This item and the following one are another occasion to present several features in one go. We'll need to install `pg_trgm`. In a nutshell, `pg_trgm` provides functions and operators for determining the similarity of ASCII alphanumeric text based on trigram matching, as well as index operator classes that support fast searching for similar strings. With PostgreSQL 9.1, it is now an extension.

Let's first install `pg_trgm`. Up to 9.0, we had to run a script manually; the command looked like this:

```
\i /usr/local/pgsql/share/contrib/pg_trgm.sql
```

This was a real maintenance problem: the created functions defaulted to the public schema, were dumped “as is” in pg_dump files, often didn't restore correctly as they depended on external binary objects, or could change definitions between releases.

With 9.1, you can use the `CREATE EXTENSION` command:

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema ]
             [ VERSION version ]
             [ FROM old_version ]
```

The most important options are extension_name, of course, and schema: extensions can be stored in a schema.

Extend Extensions

As a developper, you have very few new things to do in order to package your code as an extension. You must provide a "control file" which must be named the same as the extension with a suffix of .control, and must be placed in the installation's SHAREDIR/extension directory. There must also be at least one SQL script file, which follows the naming pattern extension--version.sql. You can also provide a Makefile that relies on PGXS.

Check out the documentation for more details!

http://pgmag.org/0120

Easily create, load, and manage new database features

So let's install `pg_trgm` for the next example:

```
=# CREATE schema extensions;
CREATE SCHEMA
=# CREATE EXTENSION pg_trgm
-# WITH SCHEMA extensions;
CREATE EXTENSION
```

Now, `pg_trgm` is installed in an 'extensions' schema. It will be included in database dumps correctly, with the `CREATE EXTENSION` syntax. So if anything changes in the extension, this extension will be restored with the new definition.

You can get the list of extensions under psql:

```
\dx
              List of installed extensions
  \dx
    List of installed extensions
 Name    | Version | Schema     | Description
---------+---------+------------+-------------
pg_trgm | 1.0     | extensions | .....
plpgsql | 1.0     | pg_catalog | .....
(2 rows))
```

K nearest neighbor (KNN) Indexing

KNN indexes provide an innovative method to avoid expensive table scans. They enhance PostgreSQL's query capabilities by using mathematical "distance" for indexing and searches. These indexes can be used to improve common text searches, text-similarity searches, geospatial location comparisons and other queries. Text search indexes can now be configured to provide indexing support for LIKE '%string%' queries without changing any SQL. PostgreSQL is among the first database systems to have KNN.

GiST indexes can now be used to return sorted rows, if a 'distance' has a meaning and can be defined for the data type. For now, this work has been done for the point datatype, the `pg_trgm` extension, and many `btree_gist` datatypes. This feature is available for all datatypes to use, so there will probably be more in the near future.

Here is an example with `pg_trgm`. `pg_trgm` uses trigrams to compare strings. Here are the trigrams for the 'hello' string:

```
SELECT show_trgm('hello');
            show_trgm
----------------------------------
 {"  h","  he",ell,hel,llo,"lo "}
```

Trigrams are used to evaluate similarity (between 0 and 1) between strings. So there is a notion of distance, with distance defined as '1-similarity'.

For our example, we need the `pg_trgm` extension and an indexed table. The table contains 5 million text records, for 750MB.

```
CREATE TABLE test_trgm (text_data text);
CREATE INDEX test_trgm_idx ON test_trgm USING gist
(text_data extensions.gist_trgm_ops);
```

Until 9.0, if we wanted the two closest text_data to hello from the table, here was the query:

```
SELECT text_data, similarity(text_data, 'hello')
FROM test_trgm
WHERE text_data % 'hello'
ORDER BY similarity(text_data, 'hello')
LIMIT 2;
```

On the test database, it takes around 2 seconds to complete.

You can now create index on "distance" for faster location and text-search queries

With 9.1 and `KNN`, one can write:

```
SELECT text_data, text_data <-> 'hello'
FROM test_trgm
ORDER BY text_data <-> 'hello'
LIMIT 2;
```

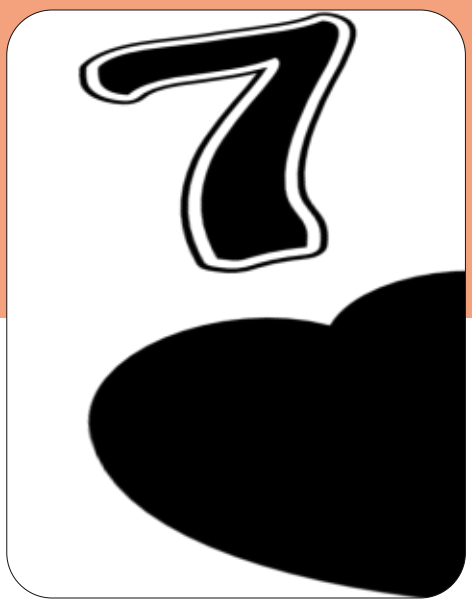
The `<->` operator is the distance operator available in the `btree_gist` extension. It runs in 20ms, using the index to directly retrieve the 2 best records.

While we're talking about `pg_trgm` and KNN, another new feature is that the `LIKE` and `ILIKE` operators can now automatically make use of a trgm index. Still using the same table:

```
SELECT text_data
FROM test_trgm
WHERE text_data LIKE '%hello%';
```

This query will use the `test_trgm_idx` index instead of scanning the whole table.

PGXN



The PostgreSQL Extension Network (PGXN) is a central distribution system for open source PostgreSQL extension libraries. It is inspired by the Comprehensive Perl Archive Network (CPAN).

The new site PGXN.org offers a repository for contributing and downloading extensions

We've seen that **PostgreSQL 9.1** allows developers to create extensions. With **PGXN**, extension authors can submit their work together with metadata describing them: the packages and their documentation are indexed and distributed across several servers. The system can be accessed through a web interface or command line clients thanks to a simple API.

A comprehensive PGXN client is being developed. It can be installed with:

```
$ easy_install pgxnclient
Searching for pgxnclient
...
Best match: pgxnclient 0.2.1
Processing pgxnclient-0.2.1-py2.6.egg
...
Installed pgxnclient-0.2.1-py2.6.egg
```

Among other commands, it allows you to search for extensions on the website:

```
$ pgxn search pair
pair 0.1.3
  ... Usage There are two ways to construct key/value *pairs*: Via the
  *pair*() function: % SELECT *pair*('foo', 'bar'); *pair* ------------
  (foo,bar) Or by using the ~> operator: % SELECT 'foo' ~> 'bar';
  *pair*...

semver 0.2.2
    *pair* | 0.1.0 | Key/value *pair* data type Note that "0.35.0b1" is less
    than "0.35.0", as required by the specification. Use ORDER BY to get
    more of a feel for semantic version ordering rules: SELECT...
```

To build and install them on the system:

```
$ pgxn install pair
INFO: best version: pair 0.1.3
INFO: saving /tmp/tmpezwyEO/pair-0.1.3.zip
INFO: unpacking: /tmp/tmpezwyEO/pair-0.1.3.zip
INFO: building extension
...
INFO: installing extension
[sudo] password for piro:
/bin/mkdir -p
'/usr/local/pg91b1/share/postgresql/extension'
...
```

And to load them as database extensions:

```
$ pgxn load -d mydb pair
INFO: best version: pair 0.1.3
CREATE EXTENSION
```

What can be easier than this?

SE Postgres

Security enhanced (SE) Postgres is designed for military-grade data stores. It allows mandatory access control with full-integration support for Security enhanced Linux (SE Linux). SE Linux is a NSA project that modifies the Linux kernel with security-related functions. It has been endorsed by and included with major Linux distributions including, but not limited to, Red Hat, CentOS, Debian and Ubuntu. PostgreSQL is the only database system which has tight integration with SE Linux.

Databases are significant facilities for managing information assets. Databases enable information to be searched for, retrieved, and stored in more elegant ways than with file systems.

Most existing RDBMSs apply their own access controls, for example, `GRANT` and `REVOKE`, without collaborating with the operating system. This can result in inconsistent access controls compared to the ones on filesystem and so on.

With SE Postgres, you can deploy military-grade security and mandatory access control

Some modern operating systems have enhanced access control features, such as SE Linux. The design of most of these features is based upon the reference monitor model (which came from 1980's researches), that allows all system access to be managed by a centralized security policy. The reference monitor model assumes that object managers (for example, operating systems) can capture all system access, and then make decisions about whether that access is allowed or denied.

SE PostgreSQL is a built-in enhancement of PostgreSQL, providing fine-grained mandatory access control (MAC) for database objects. SE PostgreSQL makes access control decisions based on SELinux security policy, the same way user access to file system objects is managed by the operating system. It provides the following significant features:

1. Mandatory access controls : PostgreSQL uses the concept of a database superuser that can bypass all access controls of native PostgreSQL. On the contrary, SE PostgreSQL enforces its access control on any client, without exception, even if the client is a database superuser. Clients can only access database objects when access is allowed by both native PostgreSQL and SE PostgreSQL.
2. Fine-grained access controls : SE PostgreSQL allows access control to be configured at the column and row levels (only a few proprietary RDBMSs support column and row level access-control options).
3. Consistency in access controls : SE Linux requires all processes and objects to have a security context that represents its privileges and attributes.
   SE PostgreSQL assigns a security context on each database object (which are tuples in system/general tables), and makes access control decisions based on the privileges and attributes as if SE Linux applies them in the kernel.

An operating systems is not the object manager in all cases. A Relational Database Management System (RDBMS) is an object manager for database objects, similar to operating systems being object managers for file system objects. Previously, RDBMSs made access control decisions independently from a centralized security policy, and as such, meticulous care is needed to keep its consistency between OS and RDBMS.

Learn more

SE Postgres is far too complex and powerful to be showcased on a single page. For more details, please check out the documentation at http://pgmag.org/0123

Serializable snapshot isolation

This new feature allows users to enforce arbitrarily complex user-defined business rules within the database without blocking, by automatically detecting live race conditions in your SQL transactions at runtime. This feature currently exists only in PostgreSQL.

With Serializable Snapshot Isolation (**SSI**), if you can show that your transaction will do the right thing if there are no concurrent transactions, then it will do the right thing in any mix of serializable transactions or else be rolled back with a serialization failure.

Now let's try a “Simple Write Skew” example where two concurrent transactions each determine what they are writing based on reading a data set which overlaps what the other is writing. You can get a state which could not occur if either had run before the other. This is known as write skew, and is the simplest form of serialization anomaly against which SSI protects you.

SSI keeps concurrent transactions consistent without blocking, using “true serializability”

In this case there are rows with a color column containing 'black' or 'white'. Two users concurrently try to make all rows contain matching color values, but their attempts go in opposite directions. One is trying to update all white rows to black and the other is trying to update all black rows to white. If these updates are run serially, all colors will match. If they are run concurrently in `REPEATABLE READ` mode, the values will be switched, which is not consistent with any serial order of runs. If they are run concurrently in `SERIALIZABLE` mode, SSI will notice the write skew and roll back one of the transactions.

The example can be set up with these statements:

```
session 1                                               session 2
begin;
update dots set color = 'black' where color = 'white';

                                                        begin;
                                                        update dots set color = 'white' where color = 'black';
                                                        At this point one transaction or the other is doomed to fail.
                                                        commit;
                                                        First commit wins.
                                                         select * from dots order by id;
                                                         id | color
                                                         ----+-------
                                                           1 | white
                                                           2 | white
                                                           3 | white
                                                         (3 rows)
                                                         This one ran as if by itself.
 commit;
 ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
 DETAIL:  Cancelled on identification
       as a pivot, during commit attempt.
 HINT:  The transaction might succeed if retried.
 A serialization failure. We roll back and try again.
 rollback;
 begin;
 update dots set color = 'black' where color = 'white';
 commit;
 No concurrent transaction to interfere.
 select * from dots order by id;
  id | color
 ----+-------
   1 | black
   2 | black
   3 | black
 (3 rows)

 This transaction ran by itself, after the other.
```

More examples

Check out the SSI wiki page for additional real life examples such as “Intersecting Data”, “Overdraft Protection” or “Deposit Report”.

http://wiki.postgresql.org/wiki/SSI

Writeable common table expressions

This features (also known as wCTE) supports the relational integrity of your data by allowing you to update multiple, cascading, related records in a single statement. By using the results of one query to execute another query, you can update recursively, hierarchically, across foreign keys, or even more creatively. PostgreSQL provides the most complete and flexible implementation of this SQL feature available.

Common table expressions were introduced in PostgreSQL 8.4 (see the `WITH` syntax). Now, data modification queries can be put in the `WITH` part of the query, and the returned data used later.

Let's say we want to archive all records matching `%hello%` from our `test_trgm` table:

```
CREATE TABLE old_text_data (text_data text);

WITH deleted AS (
  DELETE FROM test_trgm
  WHERE text_data like '%hello%'
  RETURNING text_data
)
INSERT INTO old_text_data SELECT * FROM deleted;
```

All in one query.

wCTE execute complex multi-stage data updates in a single query

As a more ambitious example, the following query updates a pgbench database, deleting a bunch of erroneous transactions and updating all related teller, branch, and account totals in a single statement:

```
WITH deleted_xtns AS (
  DELETE FROM pgbench_history
  WHERE bid = 4 and tid = 9
  RETURNING *
),
deleted_per_account AS (
   SELECT aid, sum(delta) as baldiff
   FROM deleted_xtns
   GROUP BY 1
),
accounts_rebalanced as (
   UPDATE pgbench_accounts
   SET abalance = abalance - baldiff
   FROM deleted_per_account
   WHERE deleted_per_account.aid =
pgbench_accounts.aid
   RETURNING deleted_per_account.aid,
pgbench_accounts.bid,
     baldiff
 ),
 branch_adjustment as (
     SELECT bid, SUM(baldiff) as branchdiff
     FROM accounts_rebalanced
     GROUP BY bid
 )
 UPDATE pgbench_branches
 SET bbalance = bbalance - branchdiff
 FROM branch_adjustment
 WHERE branch_adjustment.bid = pgbench_branches.bid
 RETURNING
branch_adjustment.bid,branchdiff,bbalance;
```

Money in Open Source, funding PostgreSQL features

Open source is moving fast and flourishing. In most cases, free software projects start with little or no funding at all. Money is not necessary to implement a good idea in an open source environment. However as a project grows and reaches commercial success, it becomes possible to raise funds directly to add new major features.

Once upon a time, a newbie would come to our community and say, “I want feature X.” The response was generally either, “Why?” or “Where’s your patch?” Over the last couple of years (I would put it at around the release of 8.3), the project has matured and the responses to feature requests are now “How?” instead of “Why?” In other words, “How can we get this done?” versus “Why in the world would we want that?” This is a positive step and has led to many world class leading features within PostgreSQL over the last few major releases.

As with most popular, mature and large open source projects, our project has become primarily driven by commercial interests, and the majority of key contributors to the project have financial incentive.

This works well within our ecosystem because we have quite a few different companies that contribute, whether it be 2ndQuadrant, CMD, Credativ, EnterpriseDB, or PgExperts. However, these companies are driven by a need to pay employees, preferably in a manner that allows them to eat more than high-sodium 25 cent noodles. This means that while they may all contribute, they also have different profit centers and their priorities can sway based on customer demands.

Paying for feature development is also difficult. Last year, CMD was sponsored by multiple companies to develop the FKLocks (Foreign Key Locks) patch. I can say with certainty that we will lose money on the development of that patch. We underestimated the amount of work it would take to get the job done. The financial loss is our responsibility but when you are trying to balance the investment of reviewing many thousands of lines of code to interpret the best way to modify that code for a new feature and do so in a fashion that does not mean you have written the feature before you get it funded, it makes for a very careful process. That is not to say the development of the patch wasn’t worth it; but it does point to a particular difficulty in getting a company such as CMD to continue feature development. If we lose money on the majority of patches we develop, we have to divert resources from developing features to more profitable ventures such as training or professional services. As I recall, 2ndQuadrant ran into a similar problem when they developed Hot Standby.

“The key is to have a multitude of avenues for people and companies to fund the continued development of PostgreSQL.”

There is no single solution to this problem because each entity will have to find what works for them. The key is to have a multitude of avenues for people and companies to fund the continued development of PostgreSQL. At PostgreSQL Conference West this year we were able to fund two small features, one from CMD and one from 2ndQuadrant. Although the CMD feature (URI connection strings) was funded by a single entity (Heroku), 2ndQuadrant’s feature was funded by multiple sponsors from the conference itself. This model works well for well defined, smallish features (<15k) as we can use the conferences as a vehicle to communicate the need. I know that at Postgres Open there were some fund-raising efforts as well.

One option would be to have a subscription pool. If X number of people give Y amount, the community can employ Z number of engineers full time. One complication with this is that in order for it to work, it would have to run through one of the PostgreSQL companies. I know if every CMD customer were willing to commit to 1000.00 per year, we could easily employ (with benefits) 5 engineers full time for only PostgreSQL contribution development. Of course that brings about other problems, such as who decides what features are worked on and what if subscriptions drop off? When companies give money, they want to have a say in how it is spent and just because you write a patch, it doesn’t mean it is going to get committed. When employees are working they want to make sure they have stability.

The idea of using the non-profits (PgUS, PgEU, SPI) has come up on occasion, but it has a wealth of legal and administrative issues that could burden the corporations and actually hinder development. That is certainly not what we want to have happen. It would be simple enough to offer grants for development, but grants are not tax-free and it would put the tax and administrative burden on the developer, something that most developers (let’s be honest) want nothing to do with and aren’t good at anyway. The non-profits also have to raise the money for development and all of them are operated by people with day jobs (and many have families). Fund-raising takes a lot of time that most people do not have.

What does the community think? How could we go about continuing to drive feature development and meet the required financial responsibility to developers? Is this something that even matters any more? Has our community grown past that? One thing that is certain, the amount of money being invested in PostgreSQL is only going to increase. This is easily evidenced by announcements from VMware, or just by reviewing the number of contributions that are commercially sponsored.

About the author

Joshua Drake (@Linuxpoet) is the founder of Command Prompt, the oldest dedicated PostgreSQL support provider in North America. Since 1997, he's been developing, supporting, deploying and advocating PostgreSQL.

Cascading streaming replication

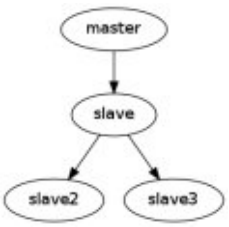
PostgreSQL's built-in replication system is simple and solid. But so far, it has lacked a key feature: cascading replication. With PostgreSQL 9.2, you'll be able to design sophisticated replication clusters.

On 19th of July, Simon Riggs committed this patch:

```
Cascading replication feature for streaming log-based
replication. Standby servers can now have WALSender
processes, which can work with either WALReceiver or
archive_commands to pass data. Fully updated docs,
including new conceptual terms of sending server,
upstream and downstream servers. WALSenders is
terminated when promoted to master.

Fujii Masao, review, rework and doc rewrite by Simon
Riggs
```

Streaming replication is relatively new, added in 9.0. Since the beginning it shared a limitation with normal WAL-files based replication, in that there is only one source of data: that is the master. While it makes sense, it is also pretty cool to be able to make a slave a source of replication for some other systems; for example, not to keep the master occupied with such tasks.



Now, with the patch, we can set up a replication schema like this:

So, let's test it.

To make it work I will need a master database and 3 slaves, made off the master. Seems simple enough.

```
=$ mkdir master
=$ initdb -D master
...
=$ vim master/postgresql.conf
```

In postgresql.conf, I change:

```
port = 4001
wal_level = hot_standby
checkpoint_segments = 20
archive_mode = on
archive_command = '/bin/true'
max_wal_senders = 3
wal_keep_segments = 100
logging_collector = on
log_checkpoints = on
log_connections = on
log_line_prefix = '%m %r %u %d %p: '
```

I also set `pg_hba.conf` to something that matches my test environment:

```
# TYPE  DATABASE     USER    ADDRESS       METHOD
local   replication  all                   trust
local   all          all                   trust
host    all          all     127.0.0.1/32  trust
```

With the master prepared that way, I can start it:

```
=$ pg_ctl -D master start
server starting


=$ psql -p 4001 -d postgres -c "select version()"

version
-------------------------------------------------
 PostgreSQL 9.2devel on x86_64-unknown-linux-gnu,
compiled by gcc-4.5.real (Ubuntu/Linaro 4.5.2-
8ubuntu4) 4.5.2, 64-bit
(1 row)
```

All looks ok.

One note — you might not understand why I used `/bin/true` as archive_command. The reason is very simple — archive_command has to be set to something, otherwise archive_mode cannot be enabled, and this will cause problems with backups; but on the other hand I will not need to use the WAL archive, since I have a pretty large value for `wal_keep_segments`.

Now, we'll set up the slaves. Starting with the first one of course:

```
=$ psql -p 4001 -d postgres -c "select
pg_start_backup('whatever')"
 pg_start_backup
 -----------------
 0/2000020
 (1 row)
=$ rsync -a master/ slave/
=$ psql -p 4001 -d postgres -c "select
pg_stop_backup()"
NOTICE:  pg_stop_backup complete, all required WAL
segments have been archived
pg_stop_backup
----------------
 0/20000D8
(1 row)
```

Of course we need some tidying of 'slave':

```
=$ rm -f slave/pg_xlog/????????????????????????
slave/pg_xlog/archive_status/* slave/pg_log/*
slave/postmaster.pid

=$ vim slave/postgresql.conf

  port = 4002
  hot_standby = on
```

And I also create `recovery.conf` in `slave/`, with this content:

```
restore_command = '/bin/false'
standby_mode = 'on'
primary_conninfo = 'port=4001 user=depesz'
trigger_file = '/tmp/slave.finish.recovery'
```

With this in place I can start the 'slave':

```
=$ pg_ctl -D slave start
server starting
=$ head -n 1 slave/postmaster.pid | xargs -IPG ps uwf -p PG --ppid PG
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
depesz   13082  1.5  0.0  66484  7492 pts/3    S    12:51   0:00 /home/pgdba/work/bin/postgres -D slave
depesz   13083  0.0  0.0  26136   716 ?        Ss   12:51   0:00  \_ postgres: logger process
depesz   13084  0.0  0.0  66556  1428 ?        Ss   12:51   0:00  \_ postgres: startup process   recovering 000000010000000000000006
depesz   13087  2.7  0.0  81504  3064 ?        Ss   12:51   0:00  \_ postgres: wal receiver process   streaming 0/6000078
depesz   13091  0.0  0.0  66484  1012 ?        Ss   12:51   0:00  \_ postgres: writer process
depesz   13092  0.0  0.0  26132   896 ?        Ss   12:51   0:00  \_ postgres: stats collector process

=$ head -n 1 master/postmaster.pid | xargs -IPG ps uwf -p PG --ppid PG
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
depesz   12981  0.2  0.0  66456  7520 pts/3    S    12:50   0:00 /home/pgdba/work/bin/postgres -D master
depesz   12982  0.0  0.0  26140   724 ?        Ss   12:50   0:00  \_ postgres: logger process
depesz   12984  0.0  0.0  66456  1016 ?        Ss   12:50   0:00  \_ postgres: writer process
depesz   12985  0.0  0.0  66456  1012 ?        Ss   12:50   0:00  \_ postgres: wal writer process
depesz   12986  0.0  0.0  67296  2096 ?        Ss   12:50   0:00  \_ postgres: autovacuum launcher process
depesz   12987  0.0  0.0  26136   732 ?        Ss   12:50   0:00  \_ postgres: archiver process
depesz   12988  0.0  0.0  26136  1040 ?        Ss   12:50   0:00  \_ postgres: stats collector process
depesz   13088  0.3  0.0  67428  2480 ?        Ss   12:51   0:00  \_ postgres: wal sender process depesz [local] streaming 0/6000078
```

One note — I used "`user=depesz`" in `primary_conninfo`, because I run the tests on the depesz system account, so initdb made a superuser named depesz, not postgres.

We now have replication between 'master' and 'slave' set up, and so we can test it:

```
=$ psql -p 4001 -d postgres -c "create table i (x
int4)"; psql -p 4002 -d postgres -c '\d i'

CREATE TABLE Table "public.i"
Column |  Type   | Modifiers
--------+---------+-----------
 x      | integer |
```

All looks OK. Now, we can add 'slave2' and 'slave3'. Since I'm lazy, I will just stop 'slave', copy 'slave' to 'slave2' and 'slave3' and then modify them:

```
=$ pg_ctl -D slave stop
waiting for server to shut down.... done
server stopped
=$ rsync -a slave/ slave2/
=$ rsync -a slave/ slave3/
=$ pg_ctl -D slave start
server starting
```

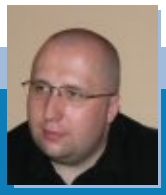
'slave2' and 'slave3' will be basically the same as 'slave', but with different ports, and connecting to 4002 (slave) instead of 4001 (master) for their WAL. So, let's do the changes:

About the article

The originial article is available at

http://pgmag.org/0134

About the author



Hubert "Depesz" Lubaczewski is a Database Architect at OmniTI. His blog (http://depesz.com/) is dedicated to the new features of the forthcoming version of PostgreSQL.

How to test this at home ?

If you are not a code contributor, testing PostgreSQL Alphas and Betas is one of the best things you can do for the project. By participating in organized testing, you help get the release out faster, with more features and less bugs.

If you are able to contribute to PostgreSQL by doing alpha testing, please read the Alpha/Beta Testing Page in the wiki:

http://wiki.postgresql.org/wiki/HowToBetaTest

```
=$ perl -pi -e 's/port = 4002/port = 4003/' slave2/postgresql.co
=$ perl -pi -e 's/port = 4002/port = 4004/' slave3/postgresql.conf
=$ perl -pi -e 's/port=4001/port=4002/' slave{2,3}/recovery.conf
=$ perl -pi -e 's/slave.finish.recovery/slave2.finish.recovery/' slave2/recovery.conf
=$ perl -pi -e 's/slave.finish.recovery/slave3.finish.recovery/' slave3/recovery.conf
```

Now, let's start them and see the processes:

```
=$ for a in slave2 slave3; do pg_ctl -D $a/ start; done
server starting
server starting

=$ head -n 1 -q */*.pid | xargs -IPG echo "-p PG --ppid PG" | xargs ps uwf
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
depesz    14031  0.0  0.0  66488  7496 pts/3    S    13:03   0:00 /home/pgdba/work/bin/postgres -D slave3
depesz    14032  0.0  0.0  26140   720 ?        Ss   13:03   0:00  \_ postgres: logger process
depesz    14033  0.0  0.0  66556  1400 ?        Ss   13:03   0:00  \_ postgres: startup process   recovering
000000010000000000000006
depesz    14063  0.0  0.0  79456  2148 ?        Ss   13:03   0:00  \_ postgres: wal receiver process   streaming
0/6012ED0
depesz    14069  0.0  0.0  66488  1532 ?        Ss   13:03   0:00  \_ postgres: writer process
depesz    14070  0.0  0.0  26136   900 ?        Ss   13:03   0:00  \_ postgres: stats collector process
depesz    14026  0.0  0.0  66492  7496 pts/3    S    13:03   0:00 /home/pgdba/work/bin/postgres -D slave2
depesz    14042  0.0  0.0  26144   720 ?        Ss   13:03   0:00  \_ postgres: logger process
depesz    14043  0.0  0.0  66560  1400 ?        Ss   13:03   0:00  \_ postgres: startup process   recovering
000000010000000000000006
depesz    14067  0.0  0.0  79460  2148 ?        Ss   13:03   0:00  \_ postgres: wal receiver process   streaming
0/6012ED0
depesz    14071  0.0  0.0  66492  1532 ?        Ss   13:03   0:00  \_ postgres: writer process
depesz    14072  0.0  0.0  26140   900 ?        Ss   13:03   0:00  \_ postgres: stats collector process
depesz    14021  0.0  0.0  66488  7528 pts/3    S    13:03   0:00 /home/pgdba/work/bin/postgres -D slave
depesz    14037  0.0  0.0  26140   724 ?        Ss   13:03   0:00  \_ postgres: logger process
depesz    14038  0.0  0.0  66560  1572 ?        Ss   13:03   0:00  \_ postgres: startup process   recovering
000000010000000000000006
depesz    14048  0.0  0.0  66488  1536 ?        Ss   13:03   0:00  \_ postgres: writer process
depesz    14050  0.0  0.0  26136   904 ?        Ss   13:03   0:00  \_ postgres: stats collector process
depesz    14052  0.0  0.0  79460  2136 ?        Ss   13:03   0:00  \_ postgres: wal receiver process   streaming
0/6012ED0
depesz    14064  0.0  0.0  67332  2476 ?        Ss   13:03   0:00  \_ postgres: wal sender process depesz [local]
streaming 0/6012ED0
depesz    14068  0.0  0.0  67452  2476 ?        Ss   13:03   0:00  \_ postgres: wal sender process depesz [local]
streaming 0/6012ED0
depesz    12981  0.0  0.0  66456  7524 pts/3    S    12:50   0:00 /home/pgdba/work/bin/postgres -D master
depesz    12982  0.0  0.0  26140   724 ?        Ss   12:50   0:00  \_ postgres: logger process
depesz    12984  0.0  0.0  66456  1780 ?        Ss   12:50   0:00  \_ postgres: writer process
depesz    12985  0.0  0.0  66456  1012 ?        Ss   12:50   0:00  \_ postgres: wal writer process
depesz    12986  0.0  0.0  67296  2156 ?        Ss   12:50   0:00  \_ postgres: autovacuum launcher process
depesz    12987  0.0  0.0  26136   732 ?        Ss   12:50   0:00  \_ postgres: archiver process
depesz    12988  0.0  0.0  26136  1040 ?        Ss   12:50   0:00  \_ postgres: stats collector process
depesz    14053  0.0  0.0  67444  2520 ?        Ss   13:03   0:00  \_ postgres: wal sender process depesz [local]
streaming 0/6012ED0
```

Please note that 'master' Postgres has only one sender process (pid 14053), 'slave' Postgres has one receiver (14052) and two senders (14064 and 14068), and 'slave2' and 'slave3' have only one receiver each (14067 and 14063).

Now we should test if it all works well, so:

```
=$ psql -d postgres -p 4001 -c \
          'insert into i(x) values (123)'
for port in 4002 4003 4004
do
  echo "port=$port"
  psql -p $port -d postgres -c \
       "select * from i"
done
INSERT 0 1
port=4002
 x
---
(0 rows)

port=4003
 x
---
(0 rows)

port=4004
 x
---
(0 rows)
```

The tables are empty. They should have some data, but it might be simply because of replication lag. So let's retry the check, without the insert this time:

```
=$ for port in 4002 4003 4004
do
  echo "port=$port"
  psql -p $port -d postgres -c "select * from i"
done
port=4002
  x
-----
 123
(1 row)

port=4003

  x
-----
 123
(1 row)

port=4004

  x
-----
 123
(1 row)
```

And all works fine now. Great! The only missing feature is the ability to make slave sourcing from a slave still work when a slave gets promoted to standalone, but unfortunately, it's not implemented yet:

```
=$ touch /tmp/slave.finish.recovery; sleep 5; head -n 1 -q */*.pid | xargs -IPG echo "-p PG --ppid PG" | xargs ps uwf
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
depesz    14896  0.1  0.0  66488  7524 pts/3    S    13:18   0:00 /home/pgdba/work/bin/postgres -D slave3
depesz    14897  0.0  0.0  26140   720 ?        Ss   13:18   0:00  \_ postgres: logger process
depesz    14898  0.0  0.0  66556  1696 ?        Ss   13:18   0:00  \_ postgres: startup process   waiting for
000000010000000000000006
depesz    14901  0.0  0.0  66488  1276 ?        Ss   13:18   0:00  \_ postgres: writer process
depesz    14902  0.0  0.0  26136   900 ?        Ss   13:18   0:00  \_ postgres: stats collector process
depesz    14883  0.1  0.0  66492  7528 pts/3    S    13:18   0:00 /home/pgdba/work/bin/postgres -D slave2
depesz    14885  0.0  0.0  26144   724 ?        Ss   13:18   0:00  \_ postgres: logger process
depesz    14886  0.0  0.0  66560  1700 ?        Ss   13:18   0:00  \_ postgres: startup process   waiting for
000000010000000000000006
depesz    14890  0.0  0.0  66492  1280 ?        Ss   13:18   0:00  \_ postgres: writer process
depesz    14891  0.0  0.0  26140   904 ?        Ss   13:18   0:00  \_ postgres: stats collector process
depesz    14021  0.0  0.0  66488  7528 pts/3    S    13:03   0:00 /home/pgdba/work/bin/postgres -D slave
depesz    14037  0.0  0.0  26140   724 ?        Ss   13:03   0:00  \_ postgres: logger process
depesz    14048  0.0  0.0  66488  1780 ?        Ss   13:03   0:00  \_ postgres: writer process
depesz    14050  0.0  0.0  26136  1032 ?        Ss   13:03   0:00  \_ postgres: stats collector process
depesz    15018  0.0  0.0  66488  1016 ?        Ss   13:20   0:00  \_ postgres: wal writer process
depesz    15019  0.0  0.0  67320  2100 ?        Ss   13:20   0:00  \_ postgres: autovacuum launcher process
depesz    15020  0.0  0.0  26136   912 ?        Ss   13:20   0:00  \_ postgres: archiver process   last was
00000002.history
depesz    12981  0.0  0.0  66456  7524 pts/3    S    12:50   0:00 /home/pgdba/work/bin/postgres -D master
depesz    12982  0.0  0.0  26140   724 ?        Ss   12:50   0:00  \_ postgres: logger process
depesz    12984  0.0  0.0  66456  1780 ?        Ss   12:50   0:00  \_ postgres: writer process
depesz    12985  0.0  0.0  66456  1012 ?        Ss   12:50   0:00  \_ postgres: wal writer process
depesz    12986  0.0  0.0  67296  2164 ?        Ss   12:50   0:00  \_ postgres: autovacuum launcher process
depesz    12987  0.0  0.0  26136   732 ?        Ss   12:50   0:00  \_ postgres: archiver process
depesz    12988  0.0  0.0  26136  1040 ?        Ss   12:50   0:00  \_ postgres: stats collector process
```

As you can see the sender in 'slave' got killed, and thus 'slave2' and 'slave3' are still slaves, but without a source of WAL. Logs of 'slave2' and 'slave3' PostgreSQL show a clear reason why it doesn't work:

```
2011-07-26 13:26:41.483 CEST    16318: FATAL:
timeline 2 of the primary does not match recovery
target timeline 1
```

Clearly 'slave' is using timeline 2, while 'slave2' and 'slave3' are still on timeline 1. Theoretically it should be simple to fix, since slave has pg_xlog/00000002.history file, but the functionality to switch timelines in recovery is simply not there yet.

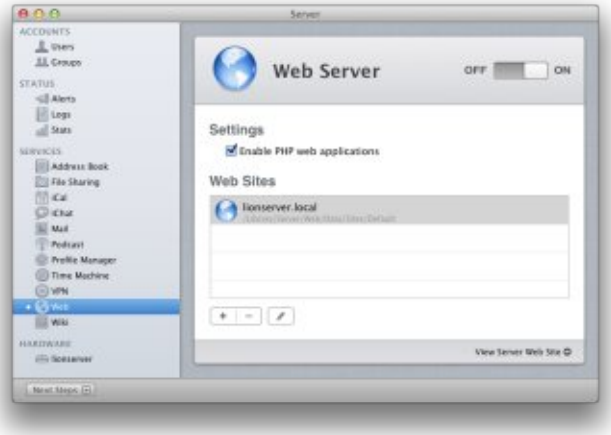
Anyway, the ability to have slaves that are receiving WAL from other slaves is pretty cool, and definitely a welcome addition.

Elephants and Lions

With the recent release of Lion and Lion Server, one noticeable absentee was MySQL, which we later discovered was replaced with PostgreSQL. In this article we will take a look at how to connect to Lion Server's built-in PostgreSQL services. We'll be focusing on the 2 most popular managment tools: an application called pgAdmin and a web service known as PgPhpAdmin.

How to use pgAdmin...

First let's take a look at pgAdmin! We need to open Server.app and enable the web service, as in most cases people want to use PostgreSQL to run a database for their web install. In this example, I want to do just that! I've also gone ahead and enabled PHP web applications. (Please note that this step is not important to the running of PostgreSQL, but will be required for your web applications that need PostgreSQL).



Next we need to download pgAdmin. To download the latest Mac release, open up the DMG and drop it in your Applications folder (you can also link it in your dock to make it more accessible). Now lets check a few things before firing up PostgreSQL. Open up Terminal and type:

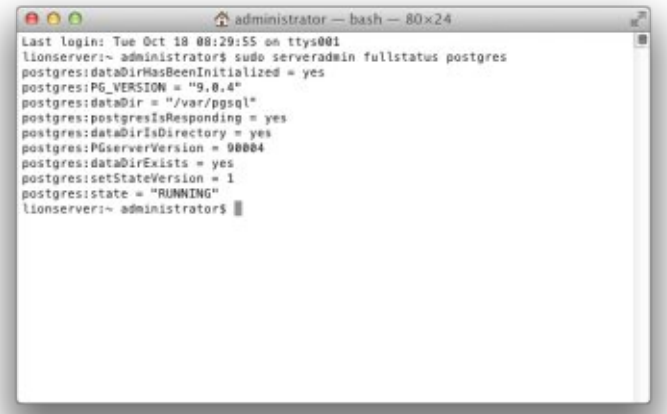
```
sudo serveradmin list
```

This will display all of the available services of Lion Server. Check that 'postgres' is in there.

Next type in:

```
sudo serveradmin fullstatus postgres
```

This will tell you the current status of the 'postgres' service. If you have already been using the Server.app, then it's possible the state will already be 'running'.



If not, then you can start it using the following command:

```
sudo serveradmin start postgres
```

This should result in the postgres state being 'RUNNING'.

If you were now to open pgAdmin and try to connect it would fail. This is because 'postgres' is not listening to connections on localhost. To fix this, we need to edit a file:

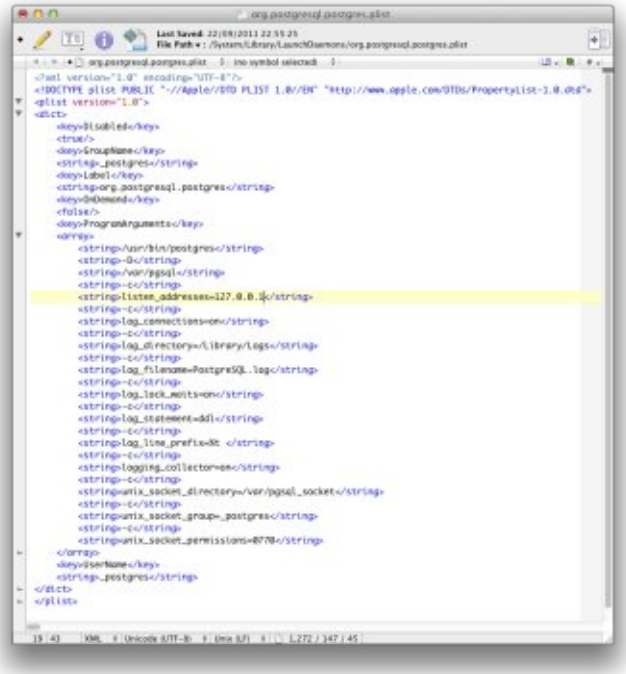
```
/System/Library/LaunchDaemons/org.postgresql.postgres.plist
```

Search for the line:

```
<string>listen_addresses=</string>
```

Add in our localhost address:

```
<string>listen_addresses=127.0.0.1</string>
```



Now lets stop and start 'postgres'. In Terminal:

```
sudo serveradmin stop postgres
sudo serveradmin start postgres
```

Now lets open up pgAdmin and connect to PostgreSQL. Click the connect icon in the top left corner and enter the settings for your setup.

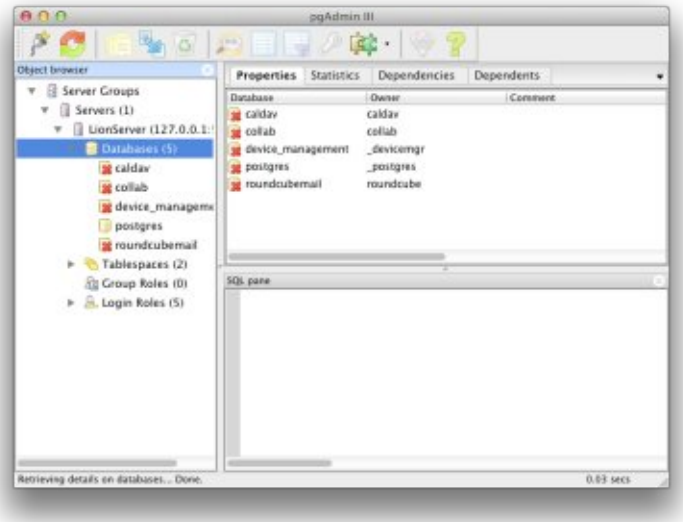
Enter a descriptive Name, Host (which we've just opened as 127.0.0.1), Username and Password (by default these are '_postgres' and your administrator password).

About the article

The originial article is available at

http://pgmag.org/0132

Voila! You are connected and can create new databases.



Known Issue

If you experience issues with permissions or accessing the pg_hba.conf file, you may need to change the permissions on the pgsql folder in /private/var/ – Get Info on the pgsql folder and add your current administrative user to have read/write access.

Download

pgAdmin is available at:

http://www.pgadmin.org/

PhpPgAdmin can be found at:

http://phppgadmin.sourceforge.net/

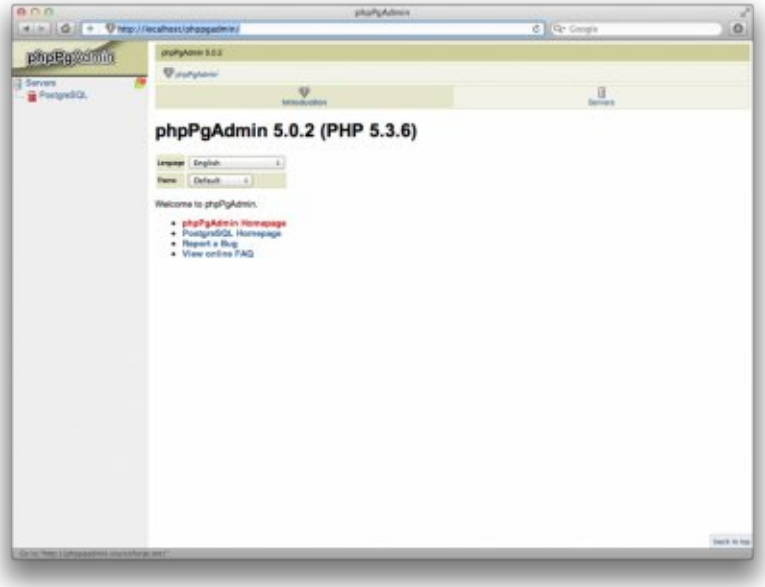
... or phpPgAdmin

Now let's try **phpPgAdmin**, just head over to the phpPgAdmin website and download the latest release. Unzip the contents and rename the folder to something like 'phpPgAdmin' to make it easier to browse to. You need to place this folder in your web documents folder in the following location:

```
/Library/Server/Web/Data/Sites/Default/
```

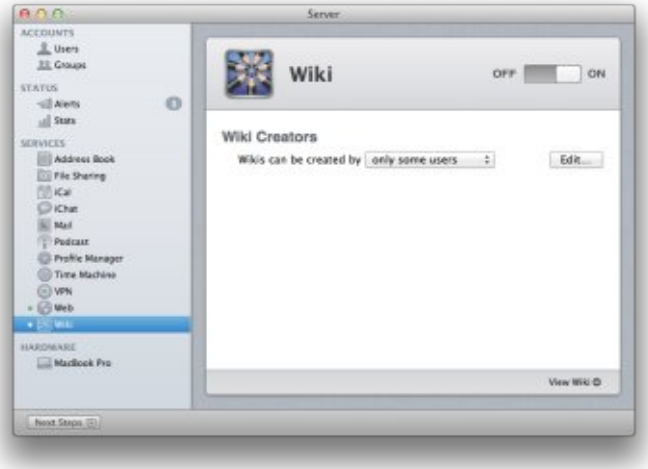
Open up Safari and browse to the 'phpPgAdmin' folder on your localhost web server:

```
http://localhost/phppgadmin/
```

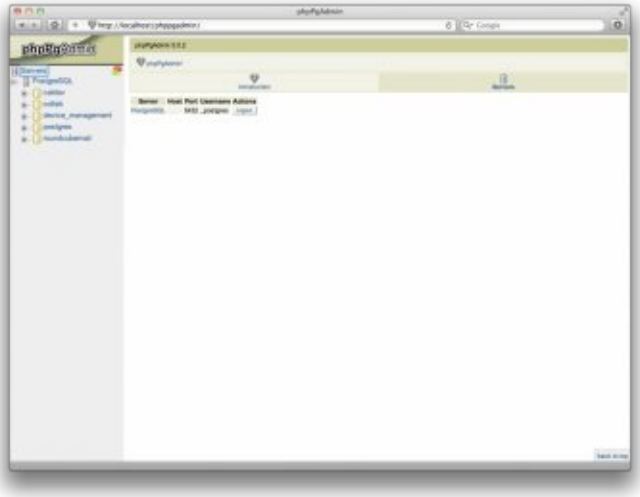


At this point if the PostgreSQL server has a red X next to it, this means it's not running.

Like above the PostgreSQL service must be running. An alternative way to start PostgreSQL is to open Server App and launch the 'Wiki' service. You may want to change the 'Wiki Creators' to be Admin's only or otherwise everybody can create Wikis:



Now if you head back to the phpPgAdmin page in Safari and select the database you should now be able to login with the user '_postgres' and your administrator password.

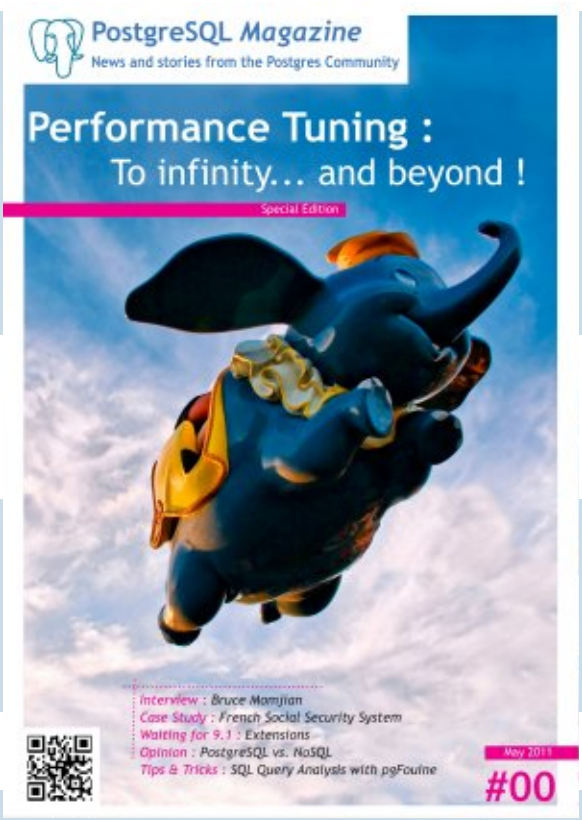


About the author

Simon Merrick (@Mactasia) has worked in education based IT for the past 7 years, alongside offering Apple Consultancy services. He often blogs and writes "how to" articles for his website Mactasia (mactasia.co.uk)

How to Contribute

This magazine is a community-driven initiative. You can help us in various ways!!



Translators

You like the magazine, but you'd like to read it in your own language? Don't worry, this is our next challenge! So far we have 4 translating teams coming off the ground: Spanish, Portuguese, French and Chinese. For now, there's no translation available but hopefully we'll have something to present by the end of the year. If you want to help one of these teams, check out our website and contact the team leader. If you want to create a new team for another language, please send us a message at contact@pgmag.org.

http://pgmag.org/Contribute#translating_an_issue

Writers

Do you have something to say about PostgreSQL? As article? An tutorial? Some insights? Just send us your text at articles@pgmag.org. Our next issue will talk about many things, including PostGIS 2.0 and PostgreSQL 9.2... The deadline is July 24th, 2012.

Copy Editors

We're looking for editors to help us improve the formatting, style, and accuracy of the articles we receive. Typically, this involves correcting spelling, avoiding jargon, cutting article when they're long and ensuring that to the text adheres style guidelines. This job is probably the most interesting part of the magazine as you need to be both an experienced PostgreSQL DBA and a skilled wordsmith!

Distribute PG Mag

If you organize an event or meeting dedicated to PostgreSQL, please contact us. We will try to send you free copies of the magazine. You can also get the PDF version here: http://pgmag.org/01/download

Photo Credits

Front: © claudiaveja (fotolia) / *p4:* © *ITPUG* / p6 : © wicked (Flickr)[1] / *p8:* © *MagnusHagander* / p11: © ThomasVondra (Flickr) / *p12:* © r_leontiev (flickr) / p16: © andrechinn (Flickr)[2] / *p17:* © *ilovememphis (Flickr)*[3] / p18: © DanBrady (Flickr)[2] / p19: © *boklm (Flickr)*[2] / p20: © / p21: © *CarbonNYC (Flickr)*[2] / p22: / p23: © *SmSm (Flickr)*[2] / p24: / p 25: © *QUOI Media Group (Flickr)*[4] / p26: © DMM Photography Art (fotolia) / p27: © *remik44992 (fotolia)* / p30: © droolcup (Flickr)[5] / p32-34 : © *SimonMerrick* / P34 : ©Silvercupcake1

[1] : Creative Commons BY-NC-SA [2] : Creative Commons BY
[3] : Creative Commons BY-ND [4] : Creative Commons BY-SA
[5] : Creative Commons BY-NC

www.pgmag.org