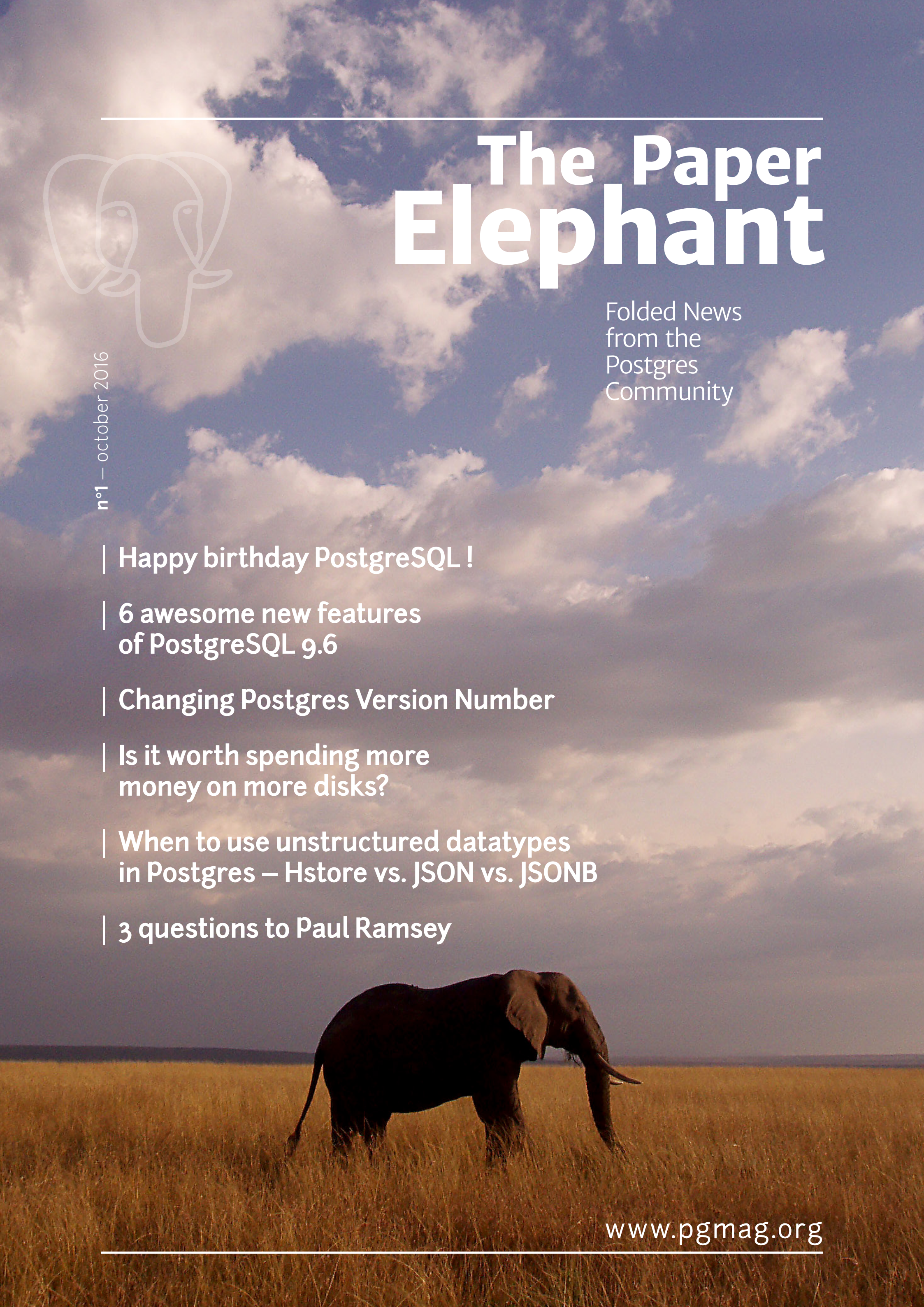


The Paper Elephant

Folded News from the Postgres Community

n°1 — october 2016

www.pgmag.org

Happy birthday PostgreSQL !

Postgres was created in the eighties as a replacement of Ingres at the university of Berkeley, but for years it remained a research object without a significant user base. The real starting point of the project was in July 1996, when a small group of people created the first non-university development server for PostgreSQL, thus putting the cornerstone of the global collective effort we now call "postgres community".

Marc Fournier, Bruce Momjian and Vadim B. Mikheev among other pioneers saw the promise of the system, and devoted themselves to its continued development. Contributing enormous amounts of time, skill, labor, and technical expertise, this global development group radically transformed Postgres. During the early years, they brought consistency, improved stability, set up mailing lists for bug reports, wrote documentation, etc.

Two decades later, PostgreSQL is now an iconic open source project, along with Linux (also celebrating its 25th anniversary this year). A database software that it as the same time robust, standard compliant and constantly evolving. A complex alchemy of innovation and stability...

So how did PostgreSQL manage to become the main alternative to Oracle, a company employing 37,000 engineers around the world. A true David vs. Gotliath story ! That's because the power of PostgreSQL is the user base itself. Open source is all about erasing the frontier between producers and consumers and the PostgreSQL community is a great example of that philosophy. Everyday and everywhere, thousands of users are testing beta versions, translating documentation, organizing meetups, answering questions on forums or simply promoting PostgreSQL at their workplace... A swarm of passionate happy users working both individually and collectively to build the best database ever.

It's the vision of the pioneers who almost instantly had the intuition that this code had to be shared,maintained and developped by an open, transparent and decentralized community. So happy birthday to PostgreSQL and kudos to the founders of this great human adventure !

Looking back at 1996, the key to success was not in the code.

The Paper Elephant is an independent media edited by and for the PostgreSQL Community.
This issue was created with the financial help of PostgreSQL Europe - www.postgresql.eu

General Editor: Damien Clochard

Layout Editor: Damien Cazeils — www.damiencazeils.com

Many Thanks to: Olivier Courtin, Josh Berkus, Hans-Jürgen Schönig, Craig Kersteins, Paul Ramsey

License: Creative Commons Attribution-ShareAlike 3.0

Photos : p1 - Allison Edwards / flickr / CC BY-NC-ND, p3 - kestrana / flikr / CC BY-NC

Disclaimer: The views and opinions in this media should in no way be assumed to be endorsed by the PostgreSQL Global Development Group. This magazine is provided with absolutely no warranty whatsoever; neither the contributors nor PostgreSQL Magazine accept any responsibility or liability for loss or damage resulting from readers choosing to apply this content to theirs or others computers and equipment.

6 awesome new features of PostgreSQL 9.6

Version 9.6 is another huge release for the PostgreSQL community ! The release notes are way too long, so we've decided to pick up 6 big improvements that may change the way you will use PostgreSQL

1 Parallelism is probably the main attraction of this version : a long-awaited feature that will open the door to many new usecases. In a nutshell, in the previous versions, Postgres could only use one single core per query, even if other processors were available. The limitation is now over and many different forms of queries can use parallelism : sequential scans, joins and aggregates can now be run in parallel on multiple core, if you want to.

2 Better Lock Monitoring : `the pg_stat_activity` view now provides much better wait information. When a process is waiting for a lock, you'll see the type of lock and details of the wait event that's putting your query on hold. Also with the `pg_blocking_pids()` function you'll know what's blocking a given server process. Such monitoring will let DBA know how how long a backend for particular event and therefore identify bottlenecks.

3 Multiple Synced Standbys : Previously synchronous replication was possible for at most one node. PopstgreSQL 9.6 now supports multiple synchronous standby servers. It enables users to consider one or more nodes as synchronous and increase the level of transaction durability by ensuring that transaction commits wait for replies from all of those nodes.

4 Preventing bloat : Until now a long-running report or cursor displaying query results could block cleanup of dead rows, therefore bloating all volatile tables in the database, causing performance problems and excessive use of storage space. A new parameter called `old_snapshot_threshold` allows the cluster to cleanup a dead rows when the transaction which modified it has completed and all snapshots which can still see it have reached a certain age...

5 PostgreSQL FDW improvements : With more than 80 Foreign Data Wrappers (FDW) you can connect PostgreSQL to almost any remote data store. Version 9.6 brings improvements to the postgres_fdw connector, such as an option to control the fetch_size and the ability to `push down` operations (joins and sorts) to the remote PostgreSQL instance. If you want to aggregate data from multiple PostgreSQL servers, this is a huge win.

6 Remote Apply : PostgreSQL 9.6 adds a new replication method called `remote_apply` where the master waits for the transaction to be applied on the remote side, not just written to disk. This is slower than the classic replication mode but not that much and it guarantees that all "commited data" are available for slave read. If you want to distribute read queries on multiple standby nodes, this new mode is for you !

Of course spotting only 6 items is a hard choice.

Of course, there are many others enhancements in this release such as : Phrase Full Text Search, the pg_visibility extension, better vacuums on frozen pages, index-only scans for partial indexes, command progress reporting, and as always... numerous performance improvements !

Changing Postgres Version Number

The PostgreSQL Project has decided to switch to a new version numbering policy.

First, let's explain how we do version numbers now. Our current version number composition is:

9 . 5 . 3
Major1 . Major2 . Minor

That is, the second number is the "major version" number, reflecting our annual release. The third number is the update release number, reflecting cumulative patch releases. Therefore "9.5.3" is the third update to version 9.5.

The problem with that first number is that we have no clear criteria when to advance it. Historically, we've advanced it because of major milestones in feature development: crash-proofing for 7.0, Windows port for 8.0, and in-core replication for 9.0. However, as PostgreSQL's feature set matures, it has become less and less clear on what milestones would be considered "first digit" releases. The result is arguments about version numbering on the mailing lists every year which waste time and irritate contributors.

As a result, the PostgreSQL Project has decided to change the version numbering to the following two-digit format:

10 . 2
Major . Minor

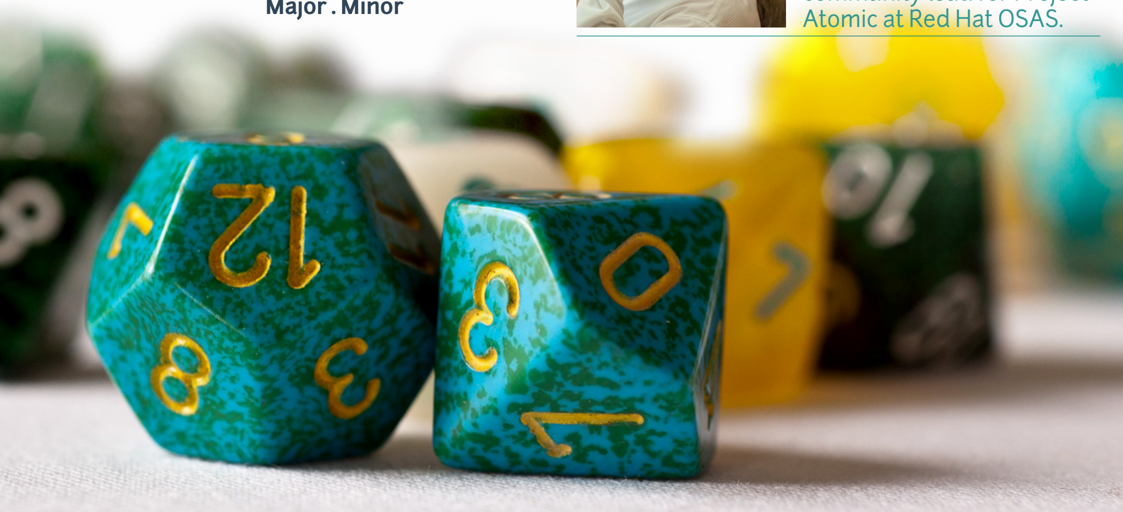
Thus "10.2" would be the second update release for major version 10. The version we release in 2017 will be "10" (instead of 10.0), and the version we release in 2018 will be "11".

The "sortable" version number available from the server, libpq, and elsewhere would remain the same six digits, zero-filled in the middle. So 10.2 will be 100002.

The idea is that this will both put an end to the annual arguments, as well as ending the need to explain to users that 9.5 to 9.6 is really a major version upgrade requiring downtime. Obviously, there is potential for breakage of a lot of tools, scripts, automation, packaging and more in this. So we're announcing this now, almost a year before 10 beta is due to come out. PostgreSQL 9.6 is the last major release that uses the three-digit numbering sheme.

About the Author :

Josh Berkus (@fuzzychef) has been a member of the PostgreSQL Core Team since 2003. He's also the community lead for Project Atomic at Red Hat OSAS.



Is it worth spending more money on more disks?

“Our database is slow.
What if we just buy more disks?
Is it going to fix things?”

Every PostgreSQL database consultant in the world has heard this kind of question already more than once. While more disks are surely a nice thing to have, it is not always economical to buy more hardware to fix problems…

pg_stat_statements: Digging into details

To answer the question whether additional disks make sense or not, it is important to extract statistics from the system. The best tool to do that is in my judgement pg_stat_statements, which is currently part of the PostgreSQL contrib module.

It will give you a deep insight into what is going on inside the server and it will also give a clue of what happens on the I/O side. In short: It will measure “disk waits”. Therefore it is always a good idea to enable this module by default. The overhead is minimal, so it is definitely worth to add this thing to the server.

The pg_stat_statements extension will install a new view describing how often a query was called, the total runtime of a certain type of query, the caching behaviour and so on. This view will contain 4 fields, which will be vital to our investigation: query, total_time, blk_read_time and blk_write_time.

The blk_* fields will tell us how much time a certain query has spent on reading and writing. We can then compare this to the total_time value to see, if I/O time is relevant or not. In case you got enough memory, data will reside in RAM anyway and so the disk might only be needed to store changes. There is one important aspect, which is often missed: blk_* is by default empty as PostgreSQL does not sum up I/O time by default due to potentially high overhead.

pg_test_timing: Measuring overhead

To sum up I/O times, set track_io_timing to true in postgresql.conf. In this case pg_stat_statements will start to show the data you need. However, before you do that, consider running pg_test_timing to measure how much overhead there is:

```
iMac:~ hs$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 37.97 nsec
```

On my iMac the average overhead for a call is 37.97 nano seconds. On a good Intel server you can maybe reach 14-15 nsec. If you happen to run bad virtualization solutions this number can easily explode to 1400 or even 1900 nsec.

Buying more and better disks really only makes sense if you are able to detect a disk bottleneck using pg_stat_statements. However, before you do that: Try to figure out, if those queries causing the problems can actually be improved. More hardware is really just the last option.

About the Author :

Hans-Jürgen Schönig (@postgresql_007) has 15 years of experience with PostgreSQL. He is consultant and CEO of Cybertec Schönig & Schönig GmbH which has served countless customers around the globe.

When to use unstructured datatypes in Postgres–Hstore vs. JSON vs. JSONB

Since Postgres started supporting NoSQL (via hstore, json and jsonb), the question of when to use Postgres in relational mode vs NoSQL mode has come up a lot. Do you entirely abandon traditional table structures, and go with documents all the way? Or do you intermingle both?

“The answer unsurprisingly is: it depends...”

HSTORE

If you exclude XML, this was the first truly unstructured datatype to arrive in Postgres. Hstore arrived way back in Postgres 8.3, before upsert, before streaming replication, and before window functions. Hstore is essentially a key/value store directly in Postgres. With hstore you're a little more limited in terms of the datatypes you have: you essentially just get strings. You also don't get any nesting; in short it's a flat key/value datatype. Its obvious benefit is flexibility, but where it really shines is being able to leverage various index types. In particular, a GIN or GiST index will index every key and value within the hstore. This way when you filter on something it'll use the index if it makes sense to the planner within Postgres.

As hstore isn't a full document equivalent, it's a stretch to consider using it as such. If you have relational data as well as some data that may not always exist on a column: it can be a great fit. In the most basic case attributes of a product catalog can be a great candidate. In certain categories such as books you'd have things like whether it's fiction or not; but in others such as clothes you might have things like size, and color. Having columns for every possible attribute for a product can at times very much be overkill.

JSON

When Postgres 9.2 arrived it was well received as the JSON release. Finally, Postgres can now complete against Mongo. (Although the JSON functionality in Postgres 9.2 was probably a little oversold.)

The JSON datatype in Postgres is under the covers still largely just a text field. With the JSON datatype what you do get is validation on it as it comes in though. Postgres does enforce that it's actually JSON. One small potential benefit of it over JSONB (which we'll get to next) is that it preserves the indentation of the data coming in. So if you are extremely particular about the formatting of your JSON, or have some need for it in a particular structure, JSON can be useful.

Furthermore, over time Postgres has picked up a number of niceties in the form of functions that can help. So, the question is: should you use JSON? At the end of the day, Postgres' JSON type simply provides JSON validation on a text field. If you're storing some form of log data you rarely need to query, JSON can work fine. Because it's so simple, it will have a lot higher write throughput. For anything more complex, I'd recommend using JSONB, which is covered below.

JSONB

Finally in Postgres 9.4 we got real and proper JSON in the form of JSONB. The B stands for better. JSONB is a binary representation of JSON, this means it's compressed and more efficient for storage than just text. It also has a similar plumbing of hstore underneath.

JSONB is largely what you'd expect from a JSON datatype. It allows nested structures, use of basic datatypes, and has a number of built in functions for working with it. Though the best part similar to hstore is the indexing. Creating a GIN index on a JSONB column will create an index on every key and value within that JSON document. That with the ability to nest within the document means JSONB is the superior to hstore in most cases.

That still leaves a bit of question of when to use only JSONB though. If you want a document database, instead of one of the other options out there you could go directly to Postgres. With a package like MassiveJS this can become quite seamless as well But even then, there are some clear examples where going more document heavy does make most sense.

In conclusion

In most cases JSONB is likely what you want when looking for a NoSQL, schema-less, datatype. Hstore and JSON can have their place as well but it's less common. More broadly, JSONB isn't always a fit in every data model. Where you can normalize there are benefits, but if you do have a schema that has a large number of optional columns (such as with event data) or the schema differs based on tenant id then JSONB can be a great fit. Use JSON if you're just processing logs, don't often need to query, and use as more of an audit trail. Hstore can work fine for text based key-value looks, but in general JSONB can still work great here.

About the Author :

Craig Kersteins (@craigkerstiens) works at Citus Data in San Francisco. He curates “Postgres Weekly”, a weekly email newsletter with Postgres content

3 Questions to Paul Ramsey

Paul Ramsey (@pwramsey) **is a PostGIS core contributor and a general PostgreSQL enthusiast. He's lives in Canada and works at CARTO.**

Q: For someone who has not heard about PostGIS, and even about GIS, how could you describe briefly what PostGIS does ?

A: PostGIS adds user defined types to PostgreSQL: geometry, geography, and raster. Geometry and geography are used to represent features in the world: point features, linear feature, area features. Raster is used to represent fields of values, for example a weather grid of temperatures or wind speeds.

The types allow you to store location information alongside other information in tables. So an address table could include a geometry column with the addresses represented as points. And then queries can act on those columns, so finding all the addresses near by to a location is a simple spatial query.

Q: Some people use or refer a motto in life. What's the one that makes sense for you ?

A: It's pretty hard to beat the golden rule "do unto others as you would have them do unto you" as a general philosophy of living. But for my professional life I've also been drawn to a phrase that Tim O'Reilly uses to guide his business decisions: "create more value than you capture". He means that you should build things bigger than yourself and not begrudge other people getting value out of them, but neither should you shy away from getting some value. But the balance is clear: leave more for others than you take yourself. I think this works especially well for open source, since building value for others in a free piece of software has clearly also built value for myself, as a professional whose expertise is in that software.

Q: What is your biggest hope/wish for PostGIS project ?

A: In some ways, my big wish has already come true: PostGIS is spoken of in industry events as a reasonable option alongside products like Oracle and SQL Server and ArcSDE. Most third party GIS software supports PostGIS as a spatial database option. PostGIS has already arrived.

The only way things could get better is if, on the technical side, PostgreSQL makes the leap from a single-threaded transactional server model, to a multi-threaded cloud server model. It is already taking steps in that direction, with new work on parallel processing, and ongoing improvements in replication. But the single most requested PostGIS feature I'm getting in my current work is horizontal scaling, and that is something that only the core PostgreSQL team can provide.

The Paper Elephant is an open and community-driven project.
Joins us at http://www.pgmag.org/contribute