Mega-scale Postgres

How to run 1,000,000 Postgres Databases

Program

What is Heroku & Heroku Postgres?

Organizing principles for mega-scale operations

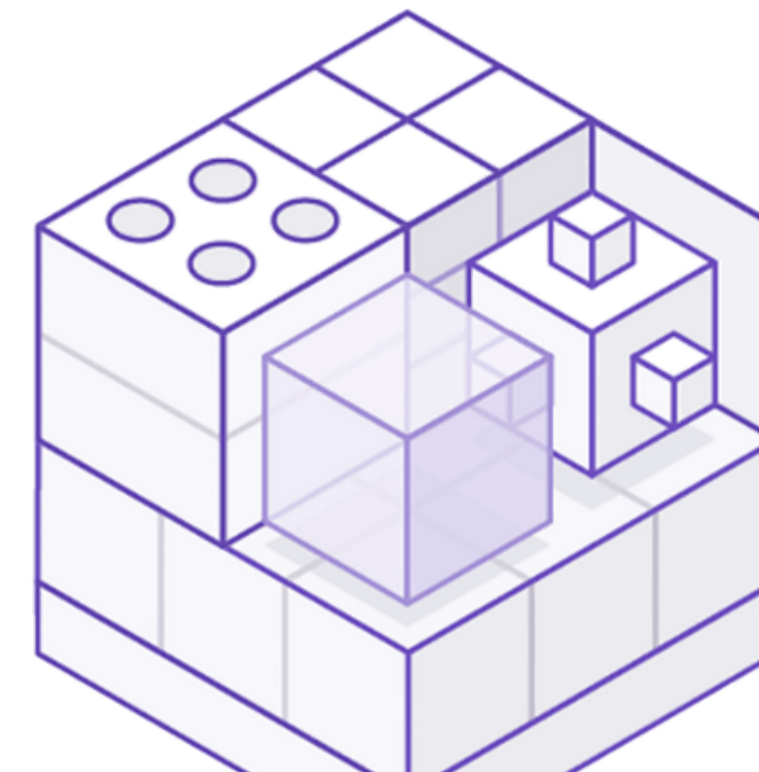
Heroku Postgres

Heroku Postgres

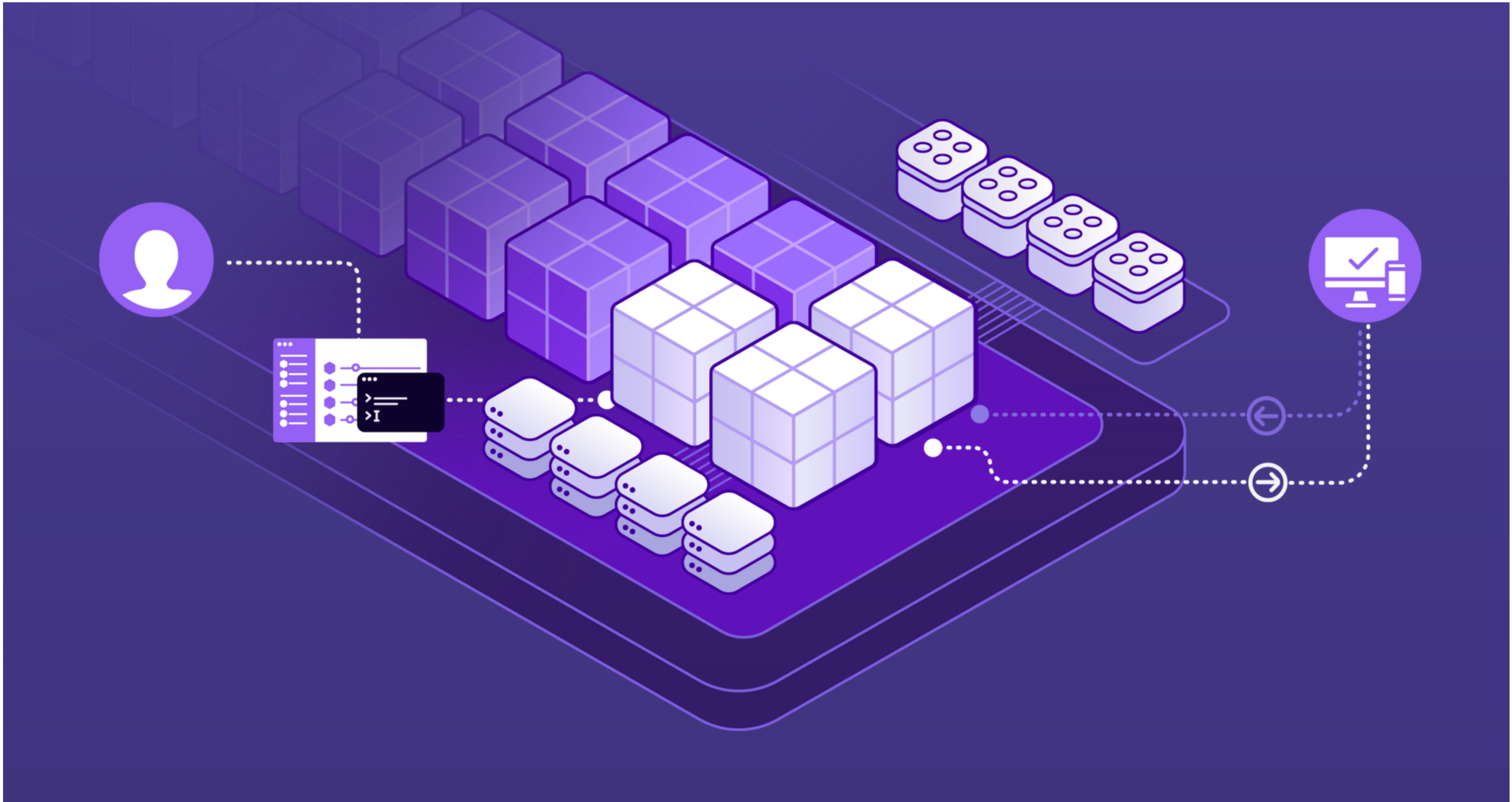
The SQL database service for developers.

Sign up for free

Operational Expertise, Built In



Whether it's an index that's not being used, security patches that have to be applied, or guidance relevant to ensuring your database is performing well, we're here to guide you along the way. Our guidance is the result of running the



```
luca@iMac | blog-test | ruby-2.0.0@railsexp | master
>
```

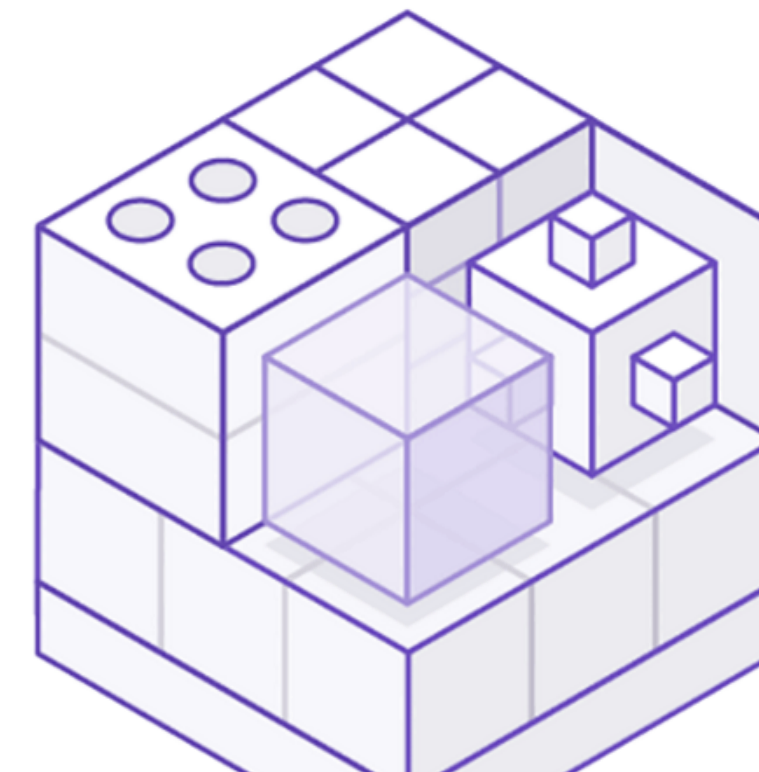
Code deployment is good,
but what about the data?

Heroku Postgres

The SQL database service for developers.

Sign up for free

Operational Expertise, Built In



Whether it's an index that's not being used, security patches that have to be applied, or guidance relevant to ensuring your database is performing well, we're here to guide you along the way. Our guidance is the result of running the

Heroku Postgres

production PostgreSQL on-demand

>1M databases, ~12 staff

provisioning, availability & multi-tenancy

user interface and visibility tools

data sharing functionality (“Dataclips”)

data service integration via FDW

world class customer support

So how do you reach 100,000 databases per engineer?

Organizing Principles for Scalable Operations

rely on programmable infrastructure

email & telephone are not APIs!

treat your servers like livestock, not pets

automate, automate, automate

people who can fix problems
must understand problems

optimize for simplicity
and self-service

What's a user's responsibility?

schema design

query writing

service plan selection

building a business

hiring a team

What is our responsibility?

availability

durability

security

visibility

accessibility

productivity

Make problems solvable!

big tables with sequential scans

poor index utilization

connection limits

lock contention

system resource overuse

idle in transaction

Ask yourself:
What does the user have to know to do the right thing?

Indexes

Can a user tell if a query uses an index?

Can a user tell if they have unused indexes?

Does a user know how much I/O an index uses?

Does the user know if an expensive query would benefit from an index?

What kind of index would be best?

Replication

What does a user need to know about replication?

Which of these things can you make irrelevant?

How do you prevent long-running queries from damaging the system?

Are the default replication settings appropriate?

How would a user ever discover sync vs. async replication?

Example Solutions

Creating a Replica

```
$ heroku addons:add heroku-postgresql --follow=PRIMARY_DATABASE
Creating database... done.
```

Expensive Queries

pg:diagnose

2. will@skadi: ~ (zsh)

```
~ ➤ heroku pg:diagnose --app will
RED: Long Queries
Pid    Duration                 Query
-----  -----------------------  ---------------------------------
30597  9 days 21:17:40.170191  select 'lol', pg_sleep(8675309);

RED: Hit Rate
Name            Ratio
--------------  ------------------
table hit rate  0.9029378531073446

YELLOW: Indexes
Reason                 Index                           Index Scan Pct  Scans Per Write  Index Size  Table Size
---------------------  ------------------------------  --------------  ---------------  ----------  ----------
Never Used Indexes     public.logs::logs_created_at    0.00            0.00             6150 MB     35 GB

GREEN: Bloat
GREEN: Connection Count
GREEN: Idle in Transaction
GREEN: Blocking Queries
GREEN: Load
```

Organizing the Team

Not dev-ops, dev **is** ops.

Everyone answers support, participates in on-call.

Prioritize ops & support work.

Convert repeat problems into automation, or documentation.

Collaborative autonomy.

Short-term planning interlocks with long-term planning.

Team & Technical Culture

Heroku Data Team

1 Designer

1 Web Developer

6 Backend Engineers (Shared Infrastructure)

1 PostgreSQL Developer

3 Other Data Store Specialists (Redis, etc)

2 Product Managers

1 Engineering Manager

TOTAL 10 ENGINEERS, 0 OPS

Tools

GitHub for source code

AWS for servers

Heroku to run our web services

Trello for planning

Google Groups (email) for discussion

HipChat for conversation

Weekly planning meetings

Bi-annual strategic sessions

Stages of Team Growth

Inception
(n > 10^2)

team size: 2.5

simple product: create, destroy, connect, backup, restore

early “developer experience” work

no super-user for customers

pg_terminate_backend() required superuser

Early Growth
($n > 10^3$)

team size: 4

develop robust AWS understanding (EBS volumes pain)

learning the pain of scale, focusing on automation

early web front-end (previously only CLI)

use of hstore to encourage Ruby community adoption

extension support for Postgres (thanks Dim!)

“We have recovered your disks. Some **might** be corrupt.”

not a nice thing to hear from your service provider

PostgreSQL Conquers Ruby (n > 10^4)

team size: 6-8

json begins challenge to MongoDB

dataclips: gist/pastebin meets SQL

customer dashboard

PITR

PostGIS support

PostgreSQL Takes Over
(n > 10^5)

team size: 8-10

larger databases - >>1Tb

(reasonable postgres OLTP ceiling O(2-5Tb) per node)

additional robust HA mechanisms to reduce MTTR

customer experience leads to pg:diagnose

jsonb decisively defeats MongoDB performance

Heroku Postgres Today (n > 10^6)

team size: 10-15

expensive queries visualization

FDW connections to other Heroku Data Services

VPC “private spaces” support

new infrastructure to support greater scale

3 other data services: Redis, and two unannounced

Observations

Why is Postgres Succeeding?

Compelling features attract new projects: jsonb, PL/V8, PostGIS

Extensions enable experimentation but require further investment

Effective advocacy in some language communities - Ruby, Python, Go

MySQL is collapsing / forks not competitive

MongoDB is popular but immature

Long-term Challenges for Postgres

scalability

no native solution for large OLTP datasets

node.js

85%+ choose non-Postgres
fastest growing language community in the world

failure tolerance

no architecturally robust solution for HA
(think: zookeeper, cassandra, etc.)

back-end services

Firebase, GraphQL, Parse, etc.

Tactical Issues with Postgres

heavy-weight connections

scaled-out services create hundreds and hundreds of connections

very large table pain

difficulty with partitioning, for example

bloat, VACUUM, freezing

transaction wraparound, idle in transaction, standby feedback

performance visibility

pg_lock confusion, EXPLAIN format, no explain for DDL

Thoughts on Cloud in Russia

Cloud is powered by
on-demand resources.

Cloud will not really start until Russia has a strong IaaS.

CORE cloud infrastructure

Virtual Machines (“EC2”)

Load Balancing (“ELB”)

SAN / Block Store (“EBS”)

File store (“S3”)

From those,
you can build the rest.*

*: At least you can make applications and a DBaaS.

Большое спасибо

fin

Peter van Hardenberg / @pvh