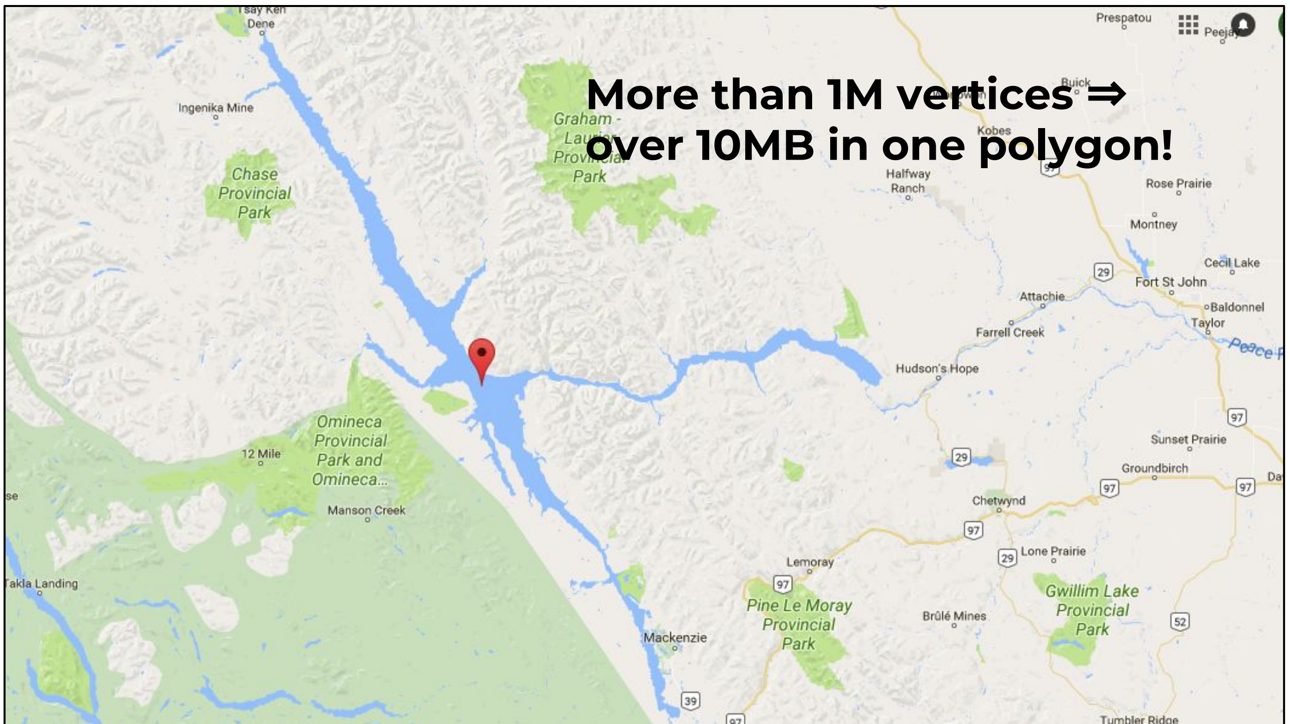
I do drop a lot of references in this talk, to projects and other things, it's a little dense,
so I've compiled all the links into one place,
so take a moment to copy that link if you're interested.

Hello spatial data scientists!
I am an (old) GIS person.

Links @ https://goo.gl/sjL6dB

Greetings! I've been working with spatial data since the mid-1990s,
initially within as a hands-on practitioner and then at increasing removes
as I've gotten into tool building with projects like PostGIS and companies like Carto.

At the time I started, most vector spatial data was created using a rig like this,
for manually capturing data off of aerial photo pairs.
The British Columbia Ministry of Environment
had recently completed a full mapping of the province at 1:20K scale,
which is how we thought about things at the time:
how much data would fit only an E-sized map sheet.
But spatial data was beginning to become standard,
and integrated into business systems,
and it was clear that those systems were just not ready for it.
It broke their core ideas about what a record of data was.



The systems for managing the base map had to deal with what was called “the Williston Lake problem”.
Williston Lake is the 7th largest reservoir in the world.
Mapped at 1:20K scale, and stitched together into a single polygon
it was over 1M vertices: a single 10Mb polygon!

This is not something you’d want to regularly transmit over a 10baseT ethernet link, let alone over the public internet connections of the time, or to a web browser of the period.
How should you store and represent it?
It’s two orders of magnitude bigger than the average lake polygon.

The Geospatial Big Data Challenge

- Float (4-8 bytes) 1D
- Integer (4-8 bytes) 1D
- Date (8-16 bytes) 1D
- String (1-1024 bytes) 1D
- Geometry (16-???????? bytes) ND

Links @ https://goo.gl/sjL6dB

Business systems, and that includes databases,
were built with the expectation that types were simple,
and relatively small.
Numbers and dates and text, not too big, all sortable on a 1 dimensional access,
so indexable using simple btrees.
Spatial data broke that, and broke it pretty badly.

Michael Stonebraker
UC Berkeley

Postgres > Ingres

- complex objects
- extensible
- robust
- active (triggers, user functions)

Links @ https://goo.gl/sjL6dB

Fortunately, academia was way ahead of practice in government information technology,
so there was already work on managing complex objects
a decade before I first started trying to do it practice.

Back in 1986, Michael Stonebraker wrote “The Design of Postgres”
which laid out the goals for his latest research project.

Postgres was to take the relational model and expand on it.
Not just primitive types, also complex objects, and with run-time extension,
so that anyone could code and add a new type.
That’s what we did when we created PostGIS in 2000, we wrote a run-time extension
to Postgres.

Recently PostgreSQL has been “discovered” by the technology cool kids.
The web devs.
But being the “it” database is actually a pretty recent phenomenon.
First it was MySQL’s older stodgier brother, and all the web devs worked with MySQL.
Then it was NoSQL’s older slower brother, and all the web devs worked with MongoDB.

But recently people have realized that a database that has a pedigree,
that respects data integrity,
that has a international development community and
more than one company behind it, is a valuable thing.
It took 20 years to get there, but Postgres is finally getting the respect it deserves.

PostgreSQL Ecosystem
Local Install

- OSX - postgress.app
- Windows - EnterpriseDB
- Linux - APT/YUM packages
- And... Solaris, FreeBSD, HPUX, AIX... and...

And,
A nice thing about a database with a history and a pedigree,
there's a **big ecosystem** around it.
So a lot of things come easy.
It's nicely packaged on all major platforms,
and lots of not-so-major ones.

PostgreSQL Ecosystem
Cloud Services

- AWS RDS PostgreSQL
- Google Cloud SQL for PostgreSQL
- Azure Database for PostgreSQL

And because it’s an industry standard,
all the cloud vendors have now packaged it up as a managed service.
So standing up a Postgres instance is as simple as stuffing in a credit card and clicking some web forms.
All the Cloud providers include PostGIS support as well,
Which, I think, is a testament to
the importance of spatial data,
and spatial data analysis.

PostgreSQL Ecosystem
3rd Party Connectivity

- Desktop admin tools
- Every language ever
- QGIS desktop
- GDAL utilities and format library
- Mapserver/Mapnik/Geoserver

The ecosystem includes lots of third-party tools,
both for **generic** things
like doing desktop data administration and programming language connectivity,
and for **specific** things like working with spatial data.
You can view and edit your PostgreSQL spatial data with QGIS,
You can import/export it with GDAL,
You can render it to web maps with Mapserver or Mapnik or Geoserver or others.

PostgreSQL Ecosystem Add-ons

- Scripting inside the database!
  - PL/Python
  - PL/R
  - PL/V8 (aka JavaScript)
  - PL/Perl
  - PL/TCL

Stonebraker's design vision included run-time extension,
Low level hooks that allow new features to be added without changing the core code.
One of those unique features is multiple language support.
The ability to script within the database in multiple languages really cannot be undersold.

Here at Carto we've leveraged the Python scripting option to build out analysis routines right inside the database.
Having access to a familiar language is nice,
but the **real power** is in importing **special purpose modules**:
want to do natural language processing? Import that python module.
Have a sci-kit routine that you want to wrap into a trigger? Just import sci-kit.

Now, as data scientists, think about what you can do with R in the database?
Want to update a model whenever the data changes? No problem, that's just a trigger.
Build out an analysis in R desktop and then operationalize it by adding it as a database function.
Now the development team can access it with a SQL call whenever they need it.
So deployment of complex functions starts to look really simple.

PostgreSQL Ecosystem
Add-ons

- Foreign Data Wrappers (FDW)
  https://wiki.postgresql.org/wiki/Foreign_data_wrappers
  - oracle_fdw, mysql_fdw, postgres_fdw
  - hdfs_fdw, hive_fdw
  - ogr_fdw, odbc_fdw
  - … more … more … more …

Similarly the “foreign data wrapper” concept is a very powerful abstraction.
Foreign data wrappers allow
any data source that can be represented as a table,
to be queried directly inside the database.
Obviously other **databases** are easy to represent as a table,
so there are wrappers for Oracle, MySQL, and Postgres itself.
But there’s no reason to stop at that.
There are wrappers for **big data stores**, like HDFS, or Hive, or Impala.
And there’s wrappers for wrappers.
There’s an ODBC wrapper, so any ODBC source can be accessed;
and there’s a wrapper for OGR so any spatial source it supports can be accessed. (I wrote that one.)
The extension add-on system supports very deep integrations like Citus,
which allows horizonal sharding and scaling and I’ll talk about later.
It also is what allowed us to build PostGIS in the first place.

Did you say PostGIS?!? Finally!!!

OK, so the background got a little long-winded,
I’m sorry, but it’s important to know what we’re talking about with Postgres,
it’s a very powerful framework,
and most of the **system level** scaling opportunities
we have in PostGIS are actually
Postgres infrastructure.

- First release in May 2001
- Required PgSQL 7.1
  - TOAST tuples
  - GIST compress
- Was a joke, a toy, a nothing
- 15 years later is industry standard

So, Because Postgres has a robust extension system,
we were able to add a whole new type without patching the core,
and bind it to an index, without having to write the core index code ourselves.
And that was PostGIS.
We released it in 2001, to a blazing fanfare of … nothing much.
But the **open source** GIS folks were very happy with it, and it quickly became the standard database for open source work.
From there the **Postgres** people noticed that GIS users
were actually a large and growing proportion of the Postgres user base.
At now at this point I can confidently state that PostGIS is the industry standard spatial database,
because every new product in the category ends up explaining themselves
in terms of how they are better than PostGIS.

“______ is Scalable”

- Size of data under management
  (there are multi-TB PgSQL databases)
- Amount of concurrency it handles
  (PgSQL happily fills up a 64-way server)
- Size of requests it can fulfill
  (PgSQL frequently runs in a single thread)

Anyways, I am supposed to talk about scalability, which is a **massively** non-specific term.
Are we talking about **size** of data under management?
Postgres has had very big databases in production for a long long time.
Or **number** of users being served?
Postgres has no problems scaling up concurrency a very long ways.
Or the amount of data that has to be **processed** to **answer a query**?
And here we come to a problem,
Because Postgres has long had some limitations in running **large analytical queries**.

“PostgreSQL is Scalable???”

- PostgreSQL is a **general purpose** RDBMS
  - OLTP workloads
  - OLAP workloads

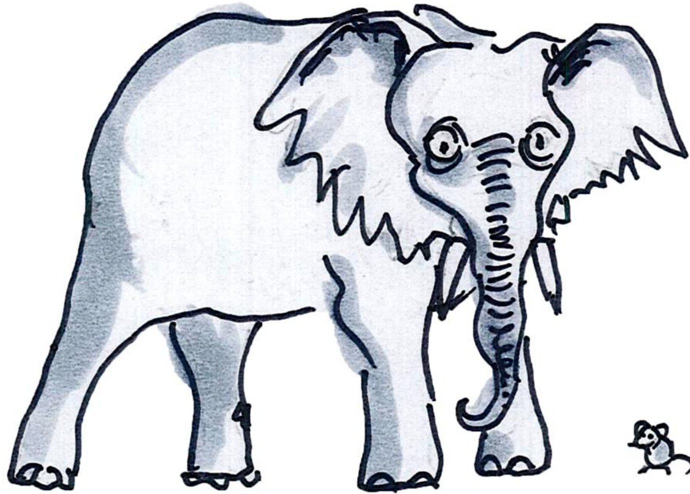
Michael Stonebraker (2012)

- General purpose RDBMS is over. Future is:
  - Column-oriented
  - In-memory
  - Graph
  - Time-series

And those limitations trace back to being a general purpose relational database.
To being a general purpose database where the main use case has been OLTP **transactional** workloads,
where each query is short and operates on relatively few rows, even if the database itself is very very large.
As a general purpose database, OLAP queries, **analytical** queries that scan large amounts of records,
have **also** been supported, but historically as a **second priority** to issues like transactional performance and data integrity.

A few years ago, Stonebraker himself declared that the general purpose database was at a dead-end.
That all the important work in databases from now on
would be in specialized areas like column oriented or in-memory or graph databases.
And then he went on to take part in companies
like Vertica, VoltDB and SciDB that built and sold those specialized databases.

“Uh oh, that sounds bad...”



So, that seems pretty bad.
The father of Postgres wrote off the general purpose database!

Spatial Data Scientists

- Care about scalable OLAP queries
- Which is the hardest PostgreSQL use case to scale!
- Arg!
- However...

And the audience here cares about the use case,
analytical queries,
that is the hardest use case to scale in Postgres!

But no problem!
I’m going to assume for the moment that, notwithstanding the previous,
you do in fact have some data in PostGIS
and you do want to do some classic analytical queries
and you want to make them go faster.
There are tricks for that!
And pitfalls to avoid.
And actually these tricks apply to all spatial databases.

Avoid Magical Thinking on Indexes

- Indexes don't "make things faster"
- Indexes make finding specific objects based on specific conditions faster
- "What objects have bounding boxes that interact with this bounding box?"

So, pitfall number one, is "I added an index, so everything will be fast now".
No.
Indexes make searching a large number of objects for a specific condition faster.
And **spatial** indexes in particular optimize the condition,
"what objects have bboxes that interact with this bbox?"
Now, that seems simple, but bear in mind the Williston Lake problem:
spatial objects have radically variable bounds and sizes.

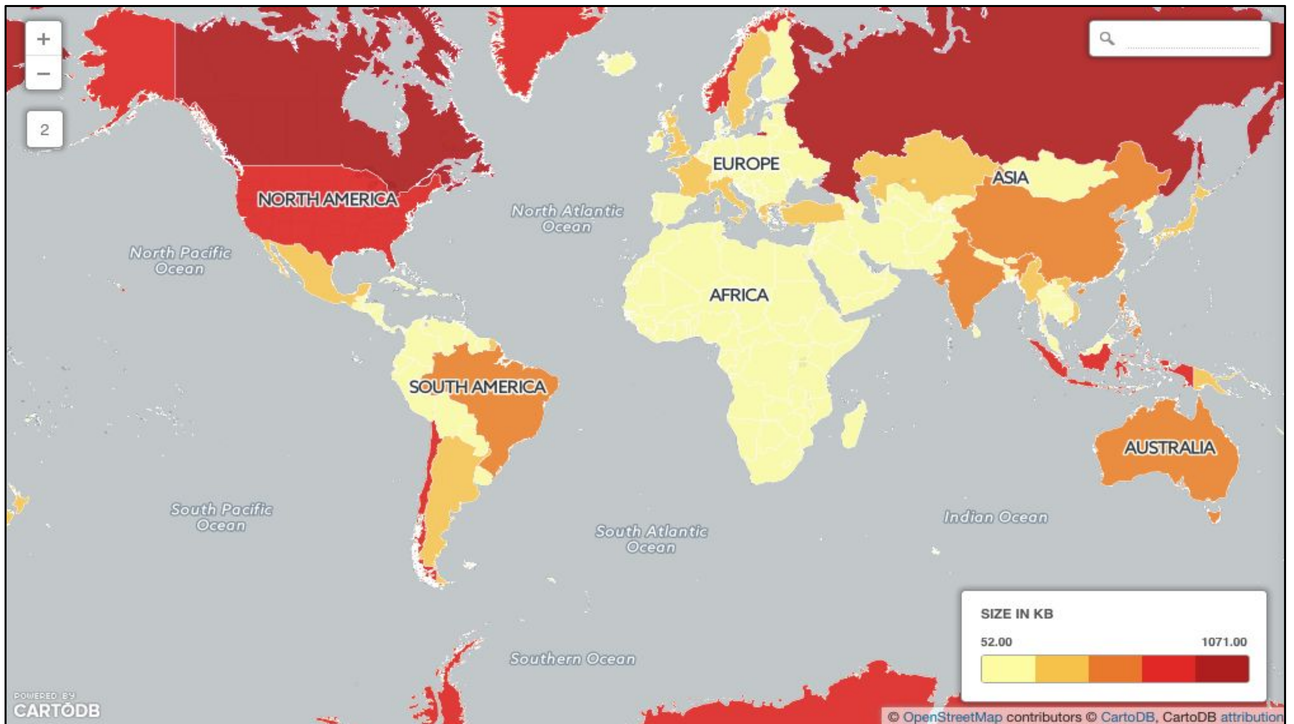
Look at the blue boxes, bounds of lakes.
Now.
Look at the red boxes, example query boxes.
They all are of reasonable small size.
And **yet**... they will all end up retrieving Williston lake,
which is a huge unwieldy polygon that they **don't actually interact with**.

Smaller Objects Index Better, Retrieve Faster

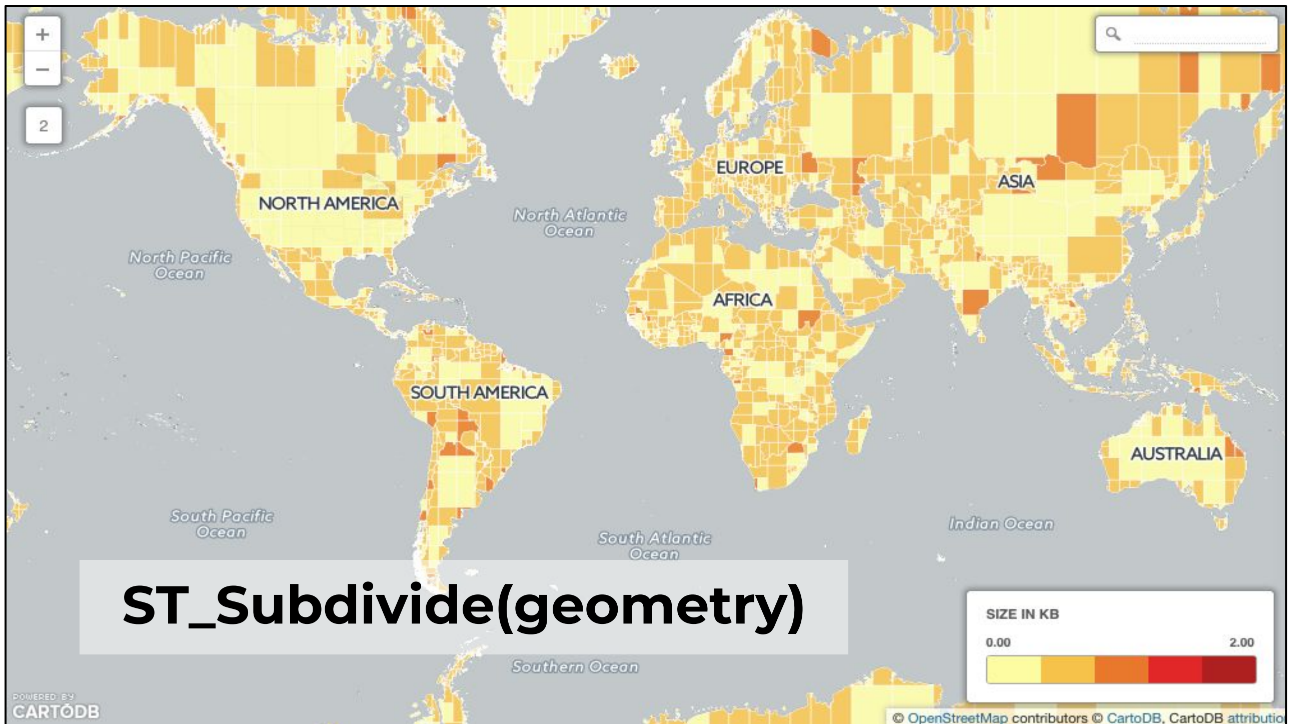
- PostgreSQL page size is 8kb
  - Large tuples are “TOASTed”
- **ST_Subdivide()** will split large geometries into smaller ones covering the same area

You can avoid the Williston Lake problem by pre-processing your data to ensure that the objects are small enough to fit in a single database page, and also small enough to cover a reasonable amount of area.

Slartibartfast is part of the problem.
The places he designed have very detailed edges,
so they require a lot of vertexes and they are much bigger than other places.



So for example, I did a spatial overlay of all the countries of the world
(a few hundred big polygons, colored here by object size in kb)
over all the populated place points in the world (several 10s of thousands of points).
Some countries are really really big, from an object size point of view.
They have big bounds, and they have a lot of vertices.
Canada, Russia, the USA.
So they take a lot of time to access.

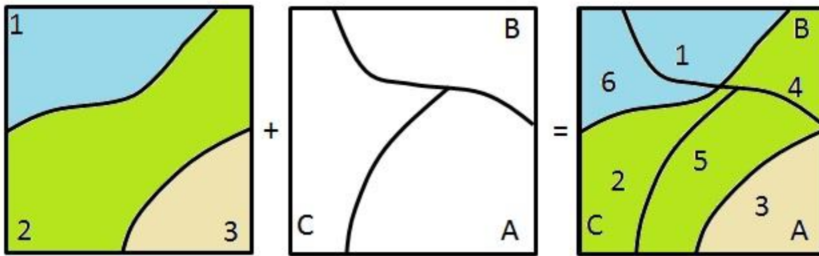


BUT..!
If the data are pre-processed with ST_Subdivide(),
the objects are of more uniform size, they all fit on a database page,
and they have reasonable bounding boxes that don't cover as much area.
Running the same point-on-polygon overlay query on the **subdivided** data
is 30x faster than on the original table, for the same set of input points.
Bringing down the size of objects,
and making their sizes more uniform,
almost always gives you a speed-up.

Avoid Expensive Things by Combining them with Fast Things

- Some spatial functions are faster than others.
  - ST_Intersects, ST_Contains, &&
- Some are slower.
  - ST_Intersection, ST_Buffer, ST_Relate

Another trick is to avoid calling certain functions:
some functions
like intersects and contains and within
are index and cache enhanced.
Others are brute force, and slow.



Polygon Topology

ID	Polygon (A)	Polygon (B)
1	1	B
2	2	C
3	3	A
4	2	B
5	2	A
6	1	C

```
SELECT ST_Intersection(a.geom, b.geom) ...
FROM a JOIN b
ON ST_Intersects(a.geom, b.geom)
```

So a very common spatial analysis use case is the overlay, what Esri calls a “union”.
Take two coverages of the plane and figure the areas that are distinctly part of each coverage.
This is a technique often used for an area-weighted transfer of attributes from one layer to another.
I use it sometimes to move census data from tracts onto electoral voting areas for political analysis.
The simplest way to do that is generating the “intersection” everywhere the objects “intersect”.
But, **intersection** is quite expensive, and has no cache shortcuts.

```sql
SELECT CASE
  WHEN ST_Contains(a.geom, b.geom)
    THEN b.geom
  WHEN ST_Within(a.geom, b.geom)
    THEN a.geom
  ELSE ST_Intersection(a.geom, b.geom)
  END ...
FROM a JOIN b
ON ST_Intersects(a.geom, b.geom)
```

The **fast** way to do it, several times as fast for some input data,
is to only run the **intersection** when absolutely necessary,
and to note that sometimes objects are **fully contained** and can be
added to the result set without any special processing.
This trick can also give a 10x to 20x speed up
for certain input data,
because the PostGIS **contains** and **within** functions have magic speed-ups in them.

Homebrew Parallelism

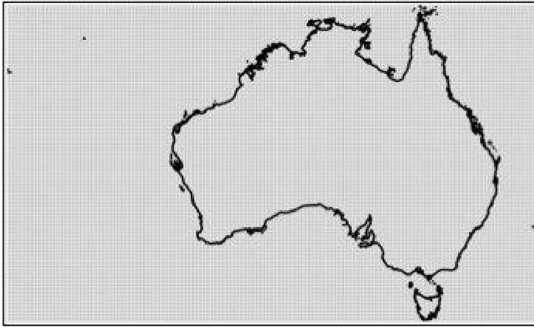
- Spatial overlay is highly parallel problem
- PostgreSQL is happy to process concurrent queries
- Break your processing into independent queries

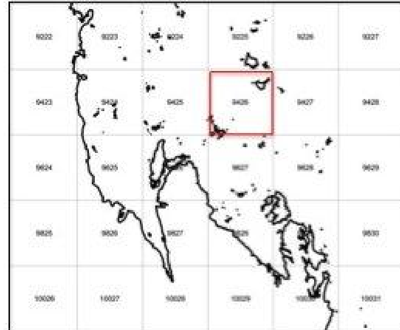
Finally, postgres has traditionally (up until version 9.5)
run queries in a single thread of execution.
This performs **fine** if your load is lots of concurrent short queries.
It performs **poorly** if you are running a **single long query**. (Like you)
However, there is no reason, at a user level,
that you cannot take advantage of postgres' ability
to run multiple short queries at the same time.

EATING THE ELEPHANT IN SMALL BITES – AN INTRODUCTION TO VECTOR TILING AND MAP REDUCE IN POSTGIS

BY MARK WYNTER ON FEBRUARY 23, 2015 IN BLOG

"Stop trying to eat the elephant with one bite"





24,321 tiles

32km x 32km each

There's a very nice blog post about one approach to this,
in Australia, where the user broke up the problem into 24000 independent tiles,
and then ran each tile-based calculation as an independent parallel process.
There's a little-known unix tool, GNU parallel,
that allows you to run as many scripts in parallel as you like,
so it's easy to launch multiple processes
in parallel from the commandline.
He was able to take his computation from hours to minutes
with some simple scripting and data segmentation.

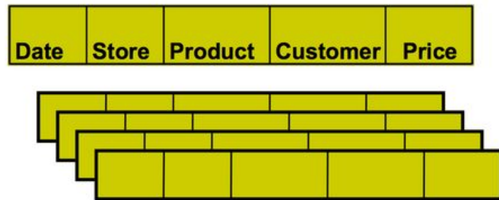
http://dimensionaledge.com/intro-vector-tiling-map-reduce-postgis/

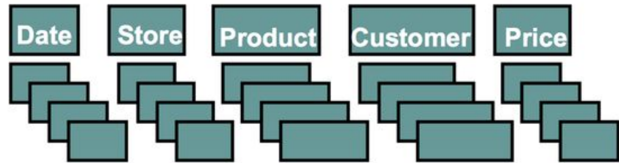
But of course, we all want the **magic** magic,
the secret configuration parameter, or the obscure add-on
And because Postgres is extensible,
there are some add-ons you can put into Postgres
and potentially get big performance benefits for analytical workloads.

Column Orientation

- Common OLAP database pattern
- Columns compress better than rows
- Allows wide tables without performance penalty of width
  - OLAP star schema
- **cstore_fdw** extension

Firstly, it’s possible to store tables in column-oriented form.
For large data sets, where you cannot fit the whole set into memory,
using column orientation
(a) allows more data to float into memory, thanks to better compression on columns and
(b) gives faster scans over compressed data and
(c) avoids having to read all columns in a row when only a few are required to fulfill a query.





- Copy tables into **cstore_fdw** table
- Data are pivoted and compressed
- Big wins are for data too large to fit into memory

This is all possible using the “cstore” foreign data wrapper,
which manages the column oriented tables,
and provides the same standard SQL interface to the data.
A lot of other big data platforms use variations on this trick,
things like the “parquet” format and “ORC” format are blocked versions of column orientation,
and the underlying format for “cstore” is also a variant of the ORC format.
If all your data can floats in memory you won’t see huge gains
from moving from row to column oriented,
but if you have enough data,
a column oriented approach can yield big wins.

GPUs

- Yes, GPUs
- **pg-strom** extension
- Uses CUDA to apply GPUs to filters, joins and aggregates
- 10x improvement for some cases
- Promise “drop in” performance boost

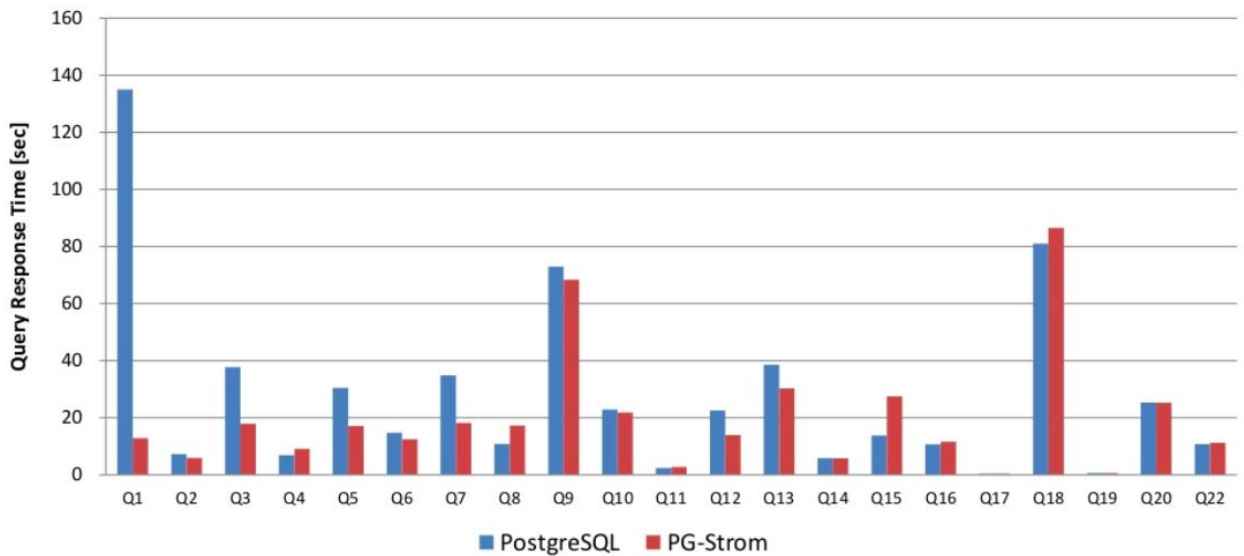
There is also an option for using GPUs to accelerate calculations in Postgres.
The “pg-strom” extension uses GPUs and
even more impressively promises “drop in” support.
You don’t have to change your workflow,
just build and enable the extension and magic will occur.
Obviously, you then require a server with lots of CUDA-compatible GPUs,
But hey, GPUs!

SOURCE: CUDA C Programming Guide

- Massive parallel cores
- Much higher DRAM bandwidth
- Better price / performance ratio
- Advantage
  - Simple arithmetic operations
  - Agility in multi-threading
- Disadvantage
  - complex control logic
  - no operating system

	GPU	CPU
Model	Nvidia GTX TITAN X	Intel Xeon E5-2690 v3
Architecture	Maxwell	Haswell
Launch	Mar-2015	Sep-2014
# of transistors	8.0billion	3.84billion
# of cores	**3072 (simple)**	12 (functional)
Core clock	1.0GHz	2.6GHz, up to 3.5GHz
Peak Flops (single precision)	6.6TFLOPS	998.4GFLOPS (with AVX2)
DRAM size	12GB, GDDR5 (384bits bus)	768GB/socket, DDR4
Memory band	**336.5GB/s**	68GB/s
Power consumption	250W	135W
Price	$999	$2,094

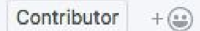
The advantages of GPUs for pg-strom are the same as for mapd:
GPUs just provide access to a **lot more cores**,
so as long as you can feed them data **fast enough**,
there are potential huge wins available in calculation speed.
This is a comparison from the pg-strom developers,
Looking at two units that had about the same price point in 2015,
and with the Nvidia, you get 3072 simple GPU cores,
compared to 12 very flexible CPU cores with the Intel.



For specific use cases, you can see 10x performance boosts,
but even for a suite of generic decision support queries,
with no changes in what queries are run,
there are improvements of a solid fraction or a few times faster.

PostGIS compatibility #305

kaigai commented 10 days ago



We positions PostGIS support one of the major feature on v3.0, however, please don't expect all the feature of PostGIS. We begin to support a part of them at the initial support.

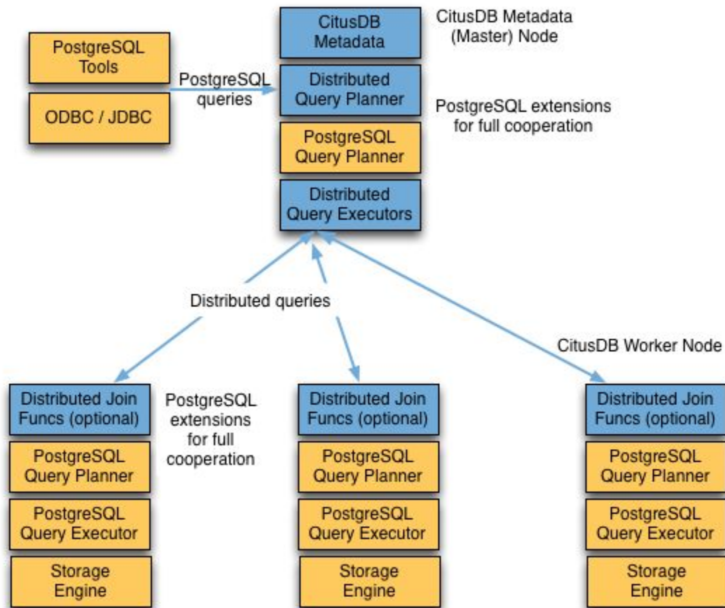
And for spatial data folks,
there's some good and interesting news from pgstrom,
which is that they've targeted their 3.0 release to include some basic spatial functions.

Horizontal Scaling

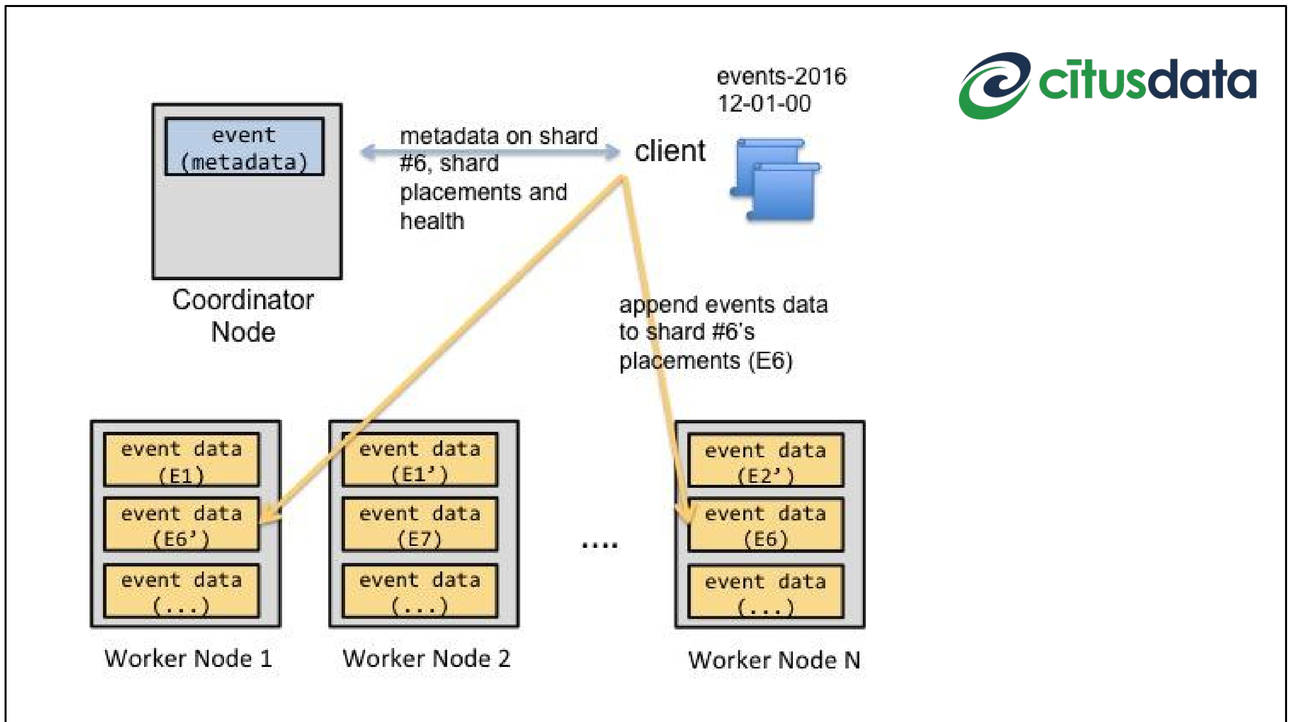
- **citus** extension
- Sharding and replication solution
- Parallel query on worker nodes
- Scale linearly with number of worker nodes, resilient to node failures
- Open source, extension only, no patching

Finally, it's actually possible to extend Postgres
to support horizontal data partitioning,
which allows scaling up of writes, so big streaming data sources can be supported,
and also scaling up reads, so queries that scan large quantities of data can be parallelized across multiple worker nodes.

The citus extension
used to be called "citusdb" and
used to be a proprietary patch against Postgres core.
It's now open source, and it's been re-factored into an extension,
so it can be added at run-time and
doesn't require any changes to the core code.



Like most horizonal database solutions,
a citus installation consists of a master node and multiple worker nodes.
But, Most of the code is plain vanilla Postgres.
The master contains the standard planner, and some special logic
that hooks in at run time for planning queries over the worker nodes.
The workers have a little special code for things like distributed join queries,
but mostly they are just vanilla Postgres databases,
they just run the queries as sent by the master and return the results.



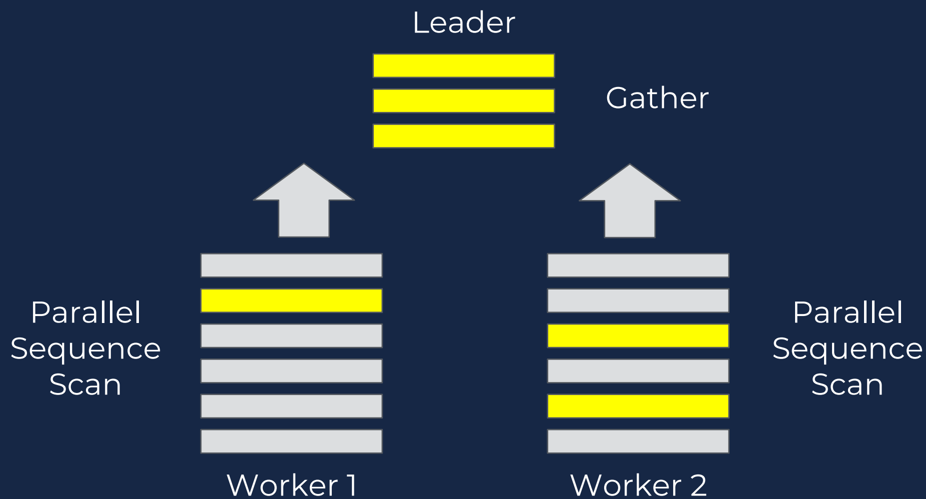
Citus solves the horizontal partitioning problem in a clever way.
A number of other older sharing solutions
for Postgres (postgres-xc, pl/proxy)
distributed data to worker nodes, but did so without redundancy.
That made running in production harder,
because ensuring reliability required setting up replicas for every data node.
It also made adding nodes harder,
because rebalancing data between nodes was a manual process.
Citus realized if they made **far more logical partitions** of tables than physical nodes, they could
(a) put copies of partitions on multiple physical nodes for redundancy, so losing one node never caused data loss, and
(b) rebalance to new nodes by moving logical partitions automatically between nodes, which is far easier to automate than moving rows between nodes.

So that's the **add-ons** available now, but
there's even more good scalability built right into the **standard Postgres server**,
particularly as you work with the most recent releases of Postgres,
like version 10.

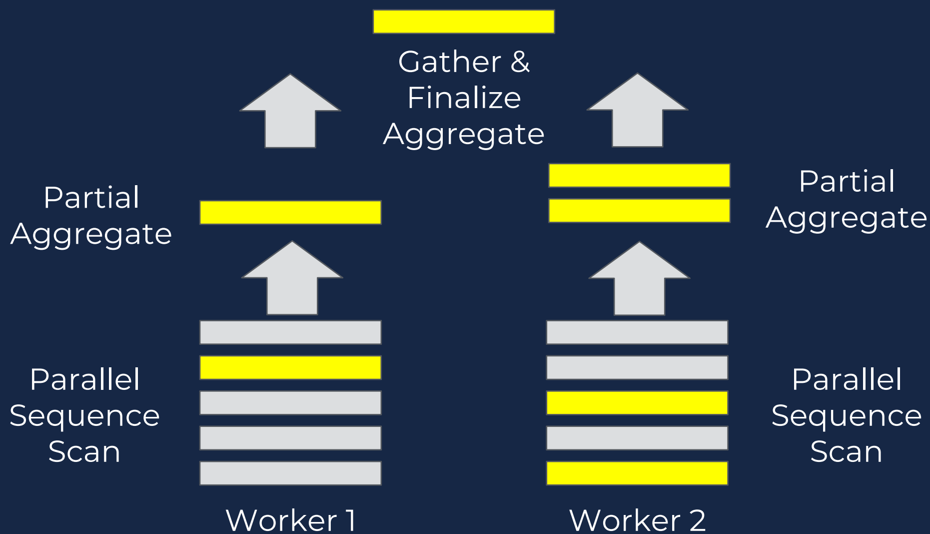
Parallel Query

- Since PgSQL 9.5
- Beyond the single thread of execution
- Database operates row by row
- Calculations are usually independent
- Planner enhancements
    - Gather node

Most database operations can be conceived as parallel problems,
the rows are independent, the execution is row by row,
it's easy to visualize how work could run in parallel.
Like most parallel problems, it's easy in theory, hard in practice.
But, over the last several years, Postgres **has** broken free of the single thread of execution.
The key enhancements came in version 9.5,
with "worker" processes and a "gather" node in the planner
to support breaking jobs into parallel chunks.
So now, a growing number of queries will
automatically parallelize to take advantage of multiple cores on the server.



So in 9.5 we got the first primitive parallel features.
The first kind of parallelism added was a parallel scan.
It allowed multiple workers to apply filter conditions to an input tuple-set,
and a gather node to collate the results of the filter into a final tuple-set.
This provided some minor speed-ups,
but since scans tended to be I/O bound anyways, wasn't a massive win.



However, in 9.6,
when the parallel scan was joined to parallel **aggregation**,
things got very interesting.
Now aggregates could run parallel scans that generated **partial aggregate** results,
that the gather node could collate and finalize.
What's a "partial aggregate"? It's the minimum amount of information you need
for independent summaries to be themselves summarized.
As an example of a partial aggregate,
imagine what you need for two people to calculate an average simultaneously:
if each person independently calculates a sum and a count for the records they are working on,
the final answer can be easily synthesized from their partial results.
With parallel aggregation, we're seeing improved performance of aggregate queries
that are linear with the number of cores available to the calculation.

Parallel Query Today (v10)

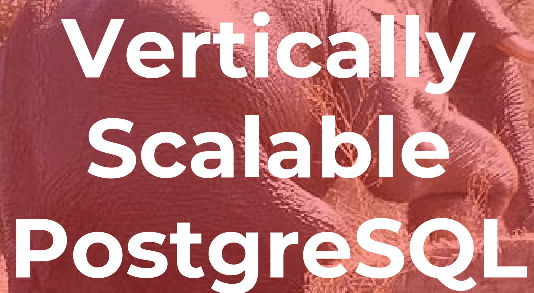
- Parallel Sequence Scan (9.5)
- Parallel Aggregate (9.6)
  - Requires "PartialAggregate"
- Parallel Join
  - NestedLoop (9.6)
  - MergeJoin (10)
- Parallel Index Scan (10)
  - btree only

We started with just parallel scan in 9.5, and the worker infrastructure.
In 9.6, we got parallel aggregate
and the first parallel join.
In 10 we got improved parallel join,
much better planning of parallel problems,
and the first parallel index scans.

Parallel Query Tomorrow (v11)

- Parallel Sort
- Parallel Append
    - Unions and Partitions
- Parallel Hash Join
- Parallel Index Build

And because parallel processing is seen as key for the future of PostgreSQL performance,
there are big investments in that work by corporate members of the Postgres development community.
EnterpriseDB, Fujitsu, and NTT all have multiple developers working on parallel query features.
Version 11 will see support for parallel sorting,
parallel append, so multiple portions of a plan can be parallelized,
parallel hash joins, and
parallel index creation.



So, PostgreSQL has been highly scalable for transactional workloads for a long time.
All this **new** parallel work is leading to an increasingly
vertically scalable PostgreSQL for the kinds of **analytical** workloads that this audience runs.
What's "vertical" scalability? It's scaling through better computers.
Want things to go faster? Run a bigger box, with more CPUs, more cores,
more compute power.
That's vertical scalability.

Parallel ST_AsMVT()

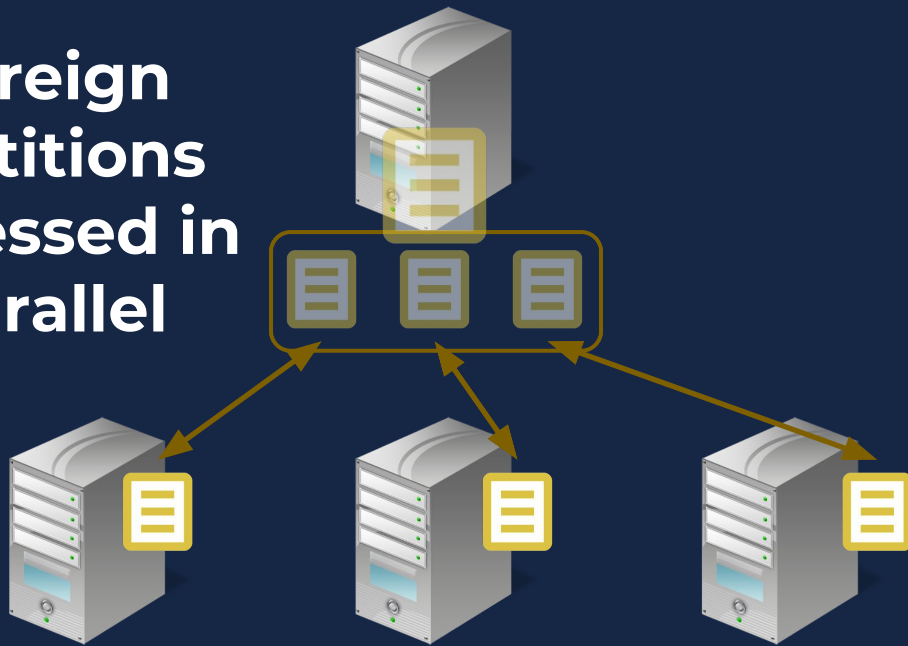
```
CREATE AGGREGATE ST_AsMVT(anyelement)
(
    parallel = safe,
    stype = internal,
    sfunc = pgis_asmvt_transfn,
    combinefunc = pgis_asmvt_combinefn,
    finalfunc = pgis_asmvt_finalfn,
    serialfunc = pgis_asmvt_serialfn,
    deserialfunc = pgis_asmvt_deserialfn
);
```

Here at Carto,
We’re taking advantage of some of the new parallelism already,
both the stuff we get for free on our existing queries,
and by updating our code to be parallel ready.
So for vector tile creation,
by making our ST_AsMVT() aggregate function parallel
we allow the underlying scans to run in parallel,
and the creation of the final tiles to be done on multiple cores.

All this work on parallel planning,
and on foreign data wrappers, which are also a relatively new feature,
is not happening at random.
All these Postgres companies are beavering away at this things because
there's a larger plan at work here...

Partitioning
+
Parallelism
+
Foreign Data Wrappers

Because when you put together features like
data partitioning, where one large logical table
is modelled as multiple smaller physical tables,
And parallel planning, which recognizes
where queries can be broken up and run separately,
And foreign data wrappers, that allow you
to locate physical data on a server remove from a logical table,



You’re building all the parts necessary to create a horizontally partitioned system.
In a horizontal system, you model a large table as a partition,
with the logical table sitting on a master node
and the physical partitions are on separate worker nodes.
The physical partitions are accessed on the master via foreign data wrappers.
The parallel planner is used to divide queries between the appropriate foreign tables,
and the aggregate the results in a gather node as they finish.
All the pieces fit together, and over the next couple releases,

We will see a full,
horizontally scalable PostgreSQL emerge,
that has all the vertical scaling strengths we have now,
along with the ability to partition workloads over multiple physical nodes,
to scale up queries to as many worker nodes as we wish.

"PostgreSQL is Scalable???"

- PostgreSQL is a **general purpose** RDBMS
  - OLTP workloads
  - OLAP workloads

Michael Stonebraker (2012)

- General purpose RDBMS is over. Future is:
  - Column-oriented
  - In-memory
  - Graph
  - Time-series

Even when we reach the nirvana of horizontally scalable Postgres,
the system will still remain a general purpose relational database.
It'll have the overhead of transactional integrity.
It'll have assumptions about row orientation even it it has pluggable column orientation.
So… Michael Stonebraker isn't wrong,
there will **always** be use cases better served by **specialized** databases,
he's just way too pessimistic about
how much utility most people can squeeze out of a general purpose relational database,
given the advances in underlying hardware performance
and the flexibility of his original Postgres design.

~~Is PostgreSQL Scalable?~~

Is PostgreSQL Scalable Enough?

So when looking at PostgreSQL, the right question isn't "is Postgres scalable",
It's "is postgres scalable enough for my problem",
and for a large and ever growing number of problems,
the answer is going to be yes.

Scaling PostgreSQL and PostGIS

We are smart, we can make it go.

Links @ https://goo.gl/sjL6dB

PAUL RAMSEY <pramsey@carto.com>

CARTO

thanks