PostgreSQL vs. MS SQL Server

A comparison of two relational databases from the point of view of a data analyst at pgversusms@gmail.com

0. What's this all about?

I work as a data analyst in a global professional services firm (one you have certainly heard of). I have been doing this for about a decade. I have spent that decade dealing with data, database software, database hardware, database users, database programmers and data analysis methods, so I know a fair bit about these things. I frequently come into contact with people who know very little about these things – although some of them don't realise it.

Over the years I have discussed the issue of PostgreSQL vs. MS SQL Server many, many times. A wellknown principle in IT says: if you're going to do it more than once, automate it. This document is my way of automating that conversation.

Unless otherwise stated I am referring to PostgreSQL 9.3 and MS SQL Server 2014, even though my experience with MS SQL Server is with versions 2008 R2 and 2012 – for the sake of fairness and relevance I want to compare the latest version of PostgreSQL to the latest version of MS SQL Server. Where I have made claims about MS SQL Server I have done my best to check that they apply to version 2014 by consulting Microsoft's own documentation – although, for reasons I will get to, I have also had to rely largely on Google, Stack Overflow and the users of the internet. I know it's not scientifically rigorous to do a comparison like this when I don't have equal experience with both databases, but this is not an academic exercise – it's a realworld comparison. I have done my honest best to get my facts about MS SQL Server right – we all know it is impossible to bluff the whole internet. If I find out that I've got something wrong, I'll fix it.

I am comparing the two databases from the point of view of a data analyst. Maybe MS SQL Server wins over PostgreSQL's as an OLTP backend (although I doubt it), but that's not what I'm writing about here, because I'm not an OLTP developer/DBA/sysadmin.

Finally, as a disclaimer, all the subjective opinions in here are strictly my own based on my own experience.

1. PostgreSQL v MS SQL Server in terms of features for data analytics.

1.1. CSV support

CSV is the de facto standard way of moving structured (i.e. tabular) data around. All RDBMSes can dump data into proprietary formats that nothing else can read, which is fine for backups, replication and the like, but no use at all for migrating data from system X to system Y.

A data analytics platform must be able to look at data from a wide variety of systems and produce outputs that can be read by a wide variety of systems. In practice, this means that it needs to be able to ingest and excrete CSV quickly, reliably, repeatedly and painlessly. Let's not understate this: a data analytics platform which cannot handle CSV robustly is a broken, useless liability.

PostgreSQL's CSV support is top notch. The `COPY TO` and `COPY FROM` commands support the spec outlined in RFC4180 (http://tools.ietf.org/html/rfc4180) (which is the closest thing there is to an official CSV standard) as well as a multitude of common and notsocommon variants and dialects. These commands are fast and robust. When an error occurs, they give helpful error messages. Importantly, they will not silently corrupt, misunderstand or alter data. If PostgreSQL says your import worked, then it worked properly. The slightest whiff of a problem and it abandons the import and throws a helpful error message.

(This may sound fussy or inconvenient, but it is an actual example of a wellestablished design principle (http://en.wikipedia.org/wiki/Failfast). It makes sense: would you rather find out your import went wrong now, or a month from now when your client complains that your results are off?)

MS SQL Server can neither import nor export CSV. Most people don't believe me when I tell them this. Then, at some point, they see for themselves. Usually they observe something like:

MS SQL Server silently truncating a text field

MS SQL Server's text encoding handling going wrong

MS SQL Server throwing an error message because it doesn't understand quoting or escaping (contrary to popular belief, quoting and escaping are not exotic extensions to CSV. They are

fundamental concepts in literally every human readable data serialization specification. Question anyone who doesn't know what these things are.)

MS SQL Server exporting broken, useless CSV

Microsoft's horrendous documentation (http://msdn.microsoft.com/enus/library/ms175937.aspx). How did they manage to overcomplicate something as simple as CSV?

[Update: Bulk Import and Export of Data for SQL Server has been improved in more recent versions, although CSV files are still not support by SQL Server bulk-import operations. See MS documentation: https://docs.microsoft.com/en-us/sql/relational-databases/import-export/bulk-import-and-export-of-data-sql-server]

This is especially baffling because CSV parsers are trivially easy to write (I wrote one in C and plumbed it into PHP a year or two ago, because I wasn't happy with its native CSVhandling functions. The whole thing took perhaps 100 lines of code and three hours – two of which were spent getting to grips with SWIG (http://www.swig.org), which was new to me at the time).

Sad but true: some database programmers I know recently spent a lot of time and effort writing Python code which "sanitises" CSV in order to allow MS SQL Server to import it. They couldn't avoid changing the actual data in this process, though. This is as crazy as spending a fortune on Photoshop and then having to write some custom code to get it to open a JPEG, only to find that the image has been altered slightly.

1.2. Ergonomics

Every data analytics platform worth mentioning is Turing complete, which means, give or take, that any one of them can do anything that any other one can do. There is no such thing as "you can do X in software A but you can't do X in software B". You can do anything in anything – all that varies is how hard it is. Good tools make the things you need to do easy; poor tools make them hard. That's what it always boils down to.

(This is all conceptually true, if not literally true for example, no RDBMS I know of can render 3D graphics. But any one of them can emulate any calculation a GPU can perform.)

PostgreSQL is clearly written by people who care about getting stuff done. MS SQL Server feels like it was written by people who never have to use MS SQL Server to achieve anything. Here are a few examples to back this up:

PostgreSQL supports DROP TABLE IF EXISTS , which is the smart and obvious way of saying "if this table doesn't exist, do nothing, but if it does, get rid of it". Something like this:

```
DROP TABLE IF EXISTS my_table;
```

Here's how you must do it in MS SQL Server:

```
IF OBJE
DROP TABLE dbo.my_table;
```

Yes, it's only one extra line of code, but notice the mysterious second parameter to the OBJECT_ID function. You need to replace that with N'V' to drop a view. It's N'P' for a stored procedur
(why sho
concentration slips for a moment, it's dead easy to do this:

```
IF OBJECT_ID (N'dbo.some_table', N'U') IS NOT NULL DROP TABLE dbo.some_other_table;
```

See what's happened there? This is a reliable source of annoying, timewasting errors.

PostgreSQL supports DROP SCHEMA CASCADE , which drops a schema and all the database objects inside it. This is very, very important for a robust analytics delivery methodology, where teardownandrebuild is the underlying principle of repeatable, auditable, collaborative analytics work.

There is no such facility in MS SQL Server. You have to drop all the objects in the schema manually, and in the right order, because if you try to drop an object on which another object depends, MS SQL Server simply throws an error. This (http://stackoverflow.com/questions/8933976/sqlserver2008deletealltablesunderspecial schema) gives an idea of how cumbersome this process can be.

PostgreSQL supports CREATE TABLE AS . Example:

```
CREATE
* FROM
all_films
imdb_rat
```

This means you can highlight everything but the first line and execute it, which is a useful and common task when developing SQL code. In MS SQL Server, table creation goes like this instead:

```
SELECT
* INTO
good_filr
all_films WHERE
imdb_rating >= 8;
```

So, to execute the plain SELECT statement, you must comment out or remove the INTO bit. Yes, commenting out two lines is easy; that's not the point. The point is that in PostgreSQL you can perform this simple task without modifying the code and in MS SQL Server you can't, and that introduces another potential source of bugs and annoyances.

In PostgreSQL, you can execute as many SQL statements as you like in one batch; as long as you've ended each statement with a semicolon, you can execute whatever combination of statements you like. For executing automated batch processes or repeatable data builds or output tasks, this is critically important functionality.

In MS SQL Server, a CREATE PROCEDURE statement cannot appear halfway through a batch of SQL statements. There's no good reason for this, it's just an arbitrary limitation. It means that extra manual steps are often required to execute a large batch of SQL. Manual steps increase risk and reduce efficiency.

PostgreSQL supports the RETURNING clause, allowing UPDATE , INSERT and DELETE statements to return values from affected rows. This is elegant and useful. MS SQL Server has the OUTPUT clause, which requires a separate table variable definition to function. This is clunky and inconvenient and forces a programmer to create and maintain unnecessary boilerplate code.

PostgreSQL supports $$ string quoting, like so:

```
SELECT $$Hello, World$$ AS greeting;
```

This is extremely useful for generating dynamic SQL because (a) it allows the user to avoid tedious and unreliable manual quoting and escaping when literal strings are nested and (b) since text editors and IDEs tend not to recognize $$ as a string delimiter, syntax highlighting remains functional even in dynamic SQL code.

PostgreSQL lets you use procedural languages simply by submitting code to the database engine; you write procedural code in Python or Perl or R or JavaScript or any of the other supported languages (see below) right next to your SQL, in the same script. This is convenient, quick, maintainable, easy to review, easy to reuse and so on.

In MS SQL Server, you can either use the lumpy, slow, awkward TSQL procedural language, or you can use a .NET language to make an assembly and load it into the database. This means your code is in two separate places and you must go through a sequence of GUIbased manual steps to alter it. It makes packaging up all your stuff into one place harder and more error prone.

And there are plenty more examples out there. Each of these things, in isolation, may seem like a relatively minor niggle; however, the overall effect is that getting real work done in MS SQL Server is significantly harder and more error prone than in PostgreSQL, and data analysts spend valuable time and energy on workarounds and manual processes instead of focusing on the actual problem.

Update: it was pointed out to me that one very useful feature MS SQL Server has which Postg

```
DECLARE @thing INT = 1;

SELECT @thing + 6;  --returns 7
```

PostgreSQL can't do this. I wish it could, because there are a lot of uses for such a feature.

1.3. You can run PostgreSQL in Linux, BSD etc. (and, of course, Windows)

Anyone who follows developments in IT knows that cross platform is important now. Cross platform support is arguably the killer feature of Java, which is actually a somewhat lumpy, ugly programming language, but nonetheless enormously successful, influential and widespread. Microsoft no longer has the monopoly it once enjoyed on the desktop, thanks to the rise of Linux and Apple. IT infrastructures are increasingly heterogeneous thanks to the flexibility of cloud services and easy access to high performance virtualization technology. Cross platform software is about giving the user control over their infrastructure. (At work I currently manage several PostgreSQL databases, some in Windows and some in Ubuntu Linux. I and my colleagues freely move code and database dumps between them. We use Python and PHP because they also work in both operating systems. It all just works.)

Microsoft's policy is and always has been vendor lock in. They don't opensource their code; they don't provide cross platform versions of their software; they even invented a whole ecosystem, .NET, designed to draw a hard line between Microsoft users and nonMicrosoft users. This is good for them, because it safeguards their revenue. It is bad for you, the user, because it restricts your choices and creates unnecessary work for you.

(Update: a couple of days after I published this, Microsoft made me look like a prat by announcing that it was opensourcing .NET. That's a great step, but let's not crack open the Bollinger just yet.)

Now, this is not a Linux vs. Windows document, although I'm sure I'll end up writing one of those at some point. Suffice it to say that, for real IT work, Linux (and the UNIXlike family: Solaris, BSD etc.) leaves Windows in the dust. UNIXlike operating systems dominate the server market, cloud services, supercomputing (in this field it's a near monopoly) and technical computing, and with good reason – these systems are designed by techies for techies. As a result, they trade user-friendliness for enormous power and flexibility. A proper UNIXlike OS is not just a nice command line – it is an ecosystem of programs, utilities, functionality and support that makes getting real work done efficient and enjoyable. A competent Linux hacker can achieve in a single throwaway line of Bash script a task which would be arduous and time-consuming in Windows.

(Example: the other day I was looking through a friend's film collection and he said he thought the total number of files in the file system was high, considering how many films he had, and he wond
folders. I did a recursive count of filesperfolder for him like this:

```
find . -type f | awk 'BEGIN {FS="/";} {print $2;}' | sort | uniq -c | sort -rn | less
```

The whole thing took about a minute to write and a second to run. It confirmed that some of his folders had a problem and told him which ones they were. How would you do this in Windows?)

For data analytics, an RDBMS doesn't exist in a vacuum; it is part of a tool stack. Therefore, its environment matters. MS SQL Server is restricted to Windows, and Windows is simply a poor analytics environment.

1.4. Procedural language features

This is a biggie.

"Pure" declarative SQL is good at what it was designed for – relational data manipulation and querying. You quickly reach its limits if you try to use it for more involved analytical processes, such as complex interest calculations, time series analysis and general algorithm design. SQL database providers know this, so almost all SQL databases implement some kind of procedural language. This allows a database user to write imperative styled code for more complex tasks.

PostgreSQL's procedural language support is exceptional. It's impossible to do justice to it in a short space, but here's a sample of the goods. Any of these procedural languages can be used for writing stored procedures and functions or simply dumped into a block of code to be executed inline.

PL/PGSQL: this is PostgreSQL's native procedural language. It's like Oracle's PL/SQL, but more modern and feature complete.
PL/V8: the V8 JavaScript engine from Google Chrome is available in PostgreSQL. This engine is stable, feature packed and absurdly fast – often approaching the execution speed of compiled, optimized C. Combine that with PostgreSQL's native support for the JSON data type (see below) and you have ultimate power and flexibility in a single package.

Even better, PL/V8 supports global (i.e. cross function call) state, allowing the user to selectively cache data in RAM for fast random access. Suppose you need to use 100,000 rows of data from table A on each of 1,000,000 rows of data from table B. In traditional SQL, you either need to join these tables (resulting in a 100bn row intermediate table, which will kill any but the most immense server) or do something akin to a scalar subquery (or, worse, cursor based nested loops), resulting in crippling I/O load if the query planner doesn't read your intentions properly. In PL/V8 you simply cache table A in memory and run a function on each of the rows of table B – in effect giving you RAM quality access (negligible latency and random-access penalty; no nonvolatile I/O load) to the 100krow table. I did this on a real piece of work recently – my PostgreSQL/PLV8 code was about 80 times faster than the MS TSQL solution and the code was much smaller and more maintainable. Because it took about 23 seconds instead of half an hour to run, I could run 20 runtestmodify cycles in an hour, resulting in feature complete, properly tested, bug free code.

(All those runtestmodify cycles were only possible because of DROP SCHEMA CASCADE and freedom to execute CREATE FUNCTION statements in the middle of a statement batch, as explained above. See how nicely it all fits together?)

PL/Python: you can use full Python in PostgreSQL. Python2 or Python 3, take your pick, and yes, you get the enormous ecosystem of libraries for which Python is justifiably famous. Fancy running a SVM from scikitlearn or some arbitraryprecision arithmetic provided by gmpy2 in the middle of a SQL query? No problem!

PL/Perl: Perl has been falling out of fashion for some time, but its versatility earned it a reputation as the Swiss army knife of programming languages. In PostgreSQL you have full Perl as a procedural language.

PL/R: R is the de facto standard statistical programming environment in academia and data science, and with good reason it is free, robust, fully featured and backed by an enormous library of high-quality plugins and addons. PostgreSQL lets you use R as a procedural language.

Java, Lua, sh, Tcl, Ruby and PHP are also supported as procedural languages in PostgreSQL.

C: doesn't quite belong in this list because you have to compile it separately, but it's worth a mention. In PostgreSQL it is trivially easy to create functions which execute compiled, optimized C (or C++ or assembler) in the database backend. This is a power user feature which provides unrivalled speed and fine control of memory management and resource usage for tasks where performance is critical. I have used this to implement a complex, stateful payment processing algorithm operating on a million rows of data per second – and that was on a desktop PC.

MS SQL Server's inbuilt procedural language (part of their TSQL extension to SQL) is slow and lacking features. It is also prone to subtle errors and bugs, as Microsoft's own documentation

sometimes acknowledges (http://msdn.microsoft.com/enus/library/ms187308.aspx). I have never met a database user who likes the TSQL procedural language.

What about the fact that you can make assemblies in .NET languages and then use them in MS SQL Server? This doesn't count as procedural language support because you can't submit this code to the database engine directly. Manageability and ergonomics are critically important. Inserting some Python code inline in your database query is easy and convenient; firing up Visual Studio, managing projects and throwing DLL files around (all in GUIbased processes which cannot be properly scripted, version controlled, automated or reviewed) is error prone and is not scalable. In any case, this mechanism is limited to .NET languages.

1.5. Native regular expression support

Regular expressions (regexen or regexes) are as fundamental to analytics work as arithmetic – they are the first choice (and often only choice) for a huge variety of text processing tasks. A data analytics tool without regex support is like a bicycle without a saddle – you can still use it, but it's painful.

PostgreSQL has smashing outofthebox support for regex. Some examples: Get all lines starting with a repeated digit followed by a vowel:

```
SELECT * FROM my_table WHERE my_field ~ E'^([0-9])\\1+[aeiou]';
```

Get the first isolated hex string occurring in a field:

```
SELECT SUBSTRING(my_field FROM E'\\y[A-Fa-f0-9]+\\y') FROM my_table;
```

Break a string on whitespace and return each fragment in a separate row:

```
SELE
-- Re

-- | c
-- ---
-- | The       |
-- | quick     |
-- | brown     |
-- | fox       |
```

Case insensitively find all words in a string with at least 10 letters:

```
```

```
SELECT REGEXP_MATCHES(my_string, E'\\y[a-z]{10,}\\y', 'gi') FROM my_table;
```

MS SQL Server has `LIKE`, `SUBSTRING`, `PATINDEX` and so on, which are not comparable to proper regex support (if you doubt this, try implementing the above examples using them). There are thirdparty regex libraries for MS SQL Server; they're just not as good as PostgreSQL's support, and the need to obtain and install them separately adds admin overhead.

Note also that PostgreSQL's extensive procedural language support also gets you several other regex engines and their various features e.g. Python's regex library provides the added power of positive and negative lookbehind assertions. This is in keeping with the general theme of PostgreSQL giving you all the tools you need to get things done.

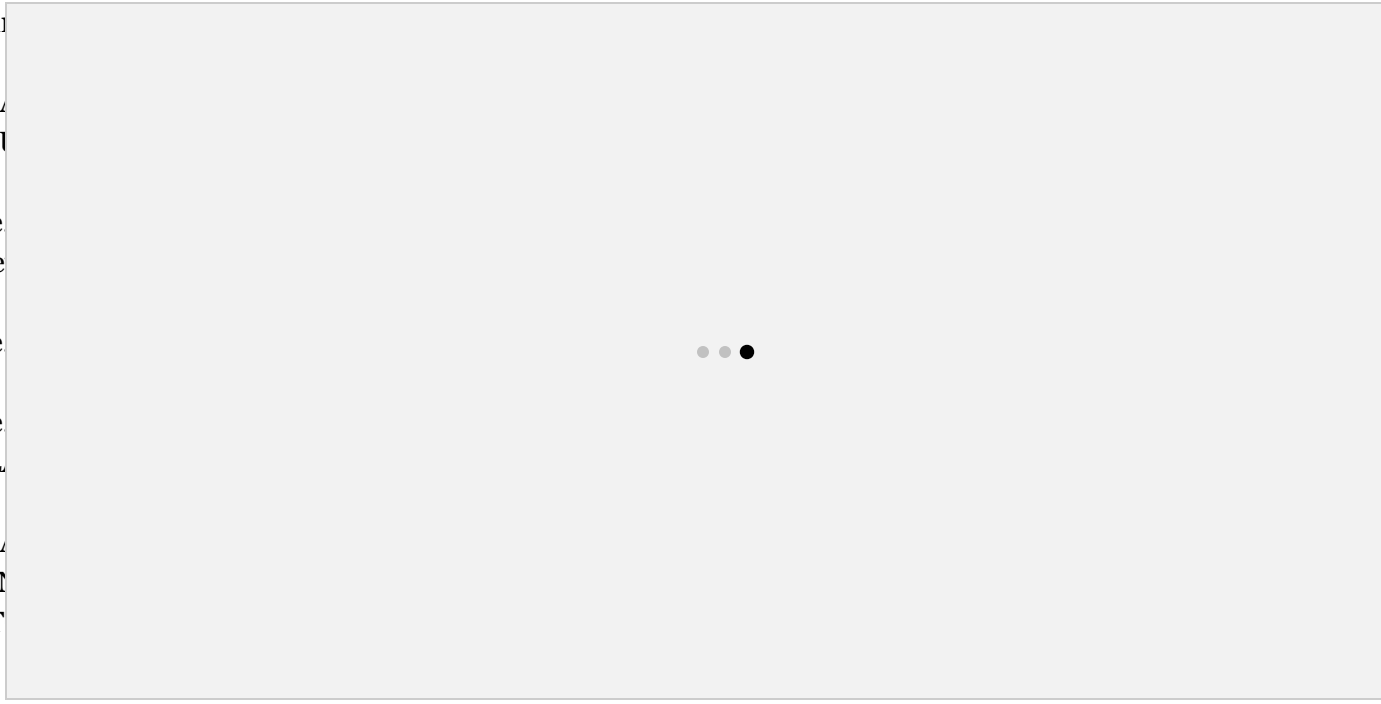
1.6. Custom aggregate functions

This is a feature that, technically, is offered by both PostgreSQL and MS SQL Server. The implementations differ hugely, though.

In PostgreSQL, custom aggregates are convenient and simple to use, resulting in fast problem solvi

```
CRE
RET
$$
state
dt.ge
{
state
}
state
$$ L

CRE
SFU
INIT
);
```



```
--assume accounts table has customer ID, date, interest rate and account movement for each day CREATE TABLE cust_balances AS
SELECT
cust_id,
(interest(movement, rate, dt ORDER BY dt)->>'balance')::FLOAT AS balance FROM accounts GROUP BY
cust_id;
```

Elegant, eh? A custom aggregate is specified in terms of an internal state and a way to modify that state when we push new values into the aggregate function. In this case we start each customer off with zero balance and no interest accrued, and on each day we accrue interest appropriately and account for payments and withdrawals. We compound the interest on the 1st of every month. Notice that the aggregate accepts an `ORDER BY` clause (since, unlike `SUM`, `MAX` and `MIN`, this aggregate is order dependent) and PostgreSQL provides operators for extracting values from JSON objects. So, in 28 lines of code we've created the framework for monthly compounding interest on bank accounts and used it to calculate final balances. If features are to be added to the methodology (e.g. interest rate modifications depending on debit/credit balance, detection of exceptional circumstances), it's all right there in the transition function and is written in an appropriate language for implementing complex logic. (Tragic sidenote: I have seen large organisations spend tens of thousands of pounds over weeks of work trying to achieve the same thing using poorer tools.)

MS SQL Server, on the other hand, makes it absurdly (http://msdn.microsoft.com/enus/library/ms131051.aspx) difficult (http://msdn.microsoft.com/en us/library/ms131056.aspx).

Incidentally, the examples in the second link are for implementing a simple string concatenation aggregate. Note the huge amount of code and gymnastics required to implement this simple function (which PostgreSQL provides out of the box, incidentally. Probably because it's useful). MS SQL Server also does not allow an order to be specified in the aggregate, which renders this function useless for my kind of work – with MS SQL Server, the order of string concatenation is random, so the results of a query using this function are nondeterministic (they might change from run to run) and the code will not pass a quality review.

The lack of ordering support also breaks code such as the interest calculation example above. As far as I can tell, you just can't do this using an MS SQL Server custom aggregate.

(It is actually possible to make MS SQL Server do a deterministic string concatenation aggregation in pure SQL but you have to abuse the RECURSIVE query functionality to do it. Although an interesting academic exercise, this results in slow, unreadable, unmaintainable code and is not a real-world solution).

1.7. Unicode support

Long gone are the days when ASCII was universal, "character" and "byte" were interchangeable terms and "foreign" text was an exotic exception. Proper international language support is no longer optional.

The solution to all this is Unicode. There are a lot of misconceptions about Unicode out there. It's not a character set, it's not a code page, it's not a file format and it's nothing whatsoever to do with encryption. An exploration of how Unicode works is fascinating but beyond the scope of this document – I heartily recommend Googling it and working through a few examples.

The key points about Unicode that are relevant to database functionality are:

Unicode encoded text (for our purposes this means either UTF8 or UTF16) is a variable width encoding. In UTF8 a character can take one, two, three or four bytes to represent. In UTF16 it's either two or four. This means that operations like taking substrings and measuring string lengths need to be Unicode aware to work properly.

Not all sequences of bytes are valid Unicode. Manipulating valid Unicode without knowing it's Unicode is likely to produce something that is not valid Unicode.

UTF8 and UTF16 are not compatible. If you take one file of each type and concatenate them, you (probably) end up with a file which is neither valid UTF8 nor valid UTF16.

For text which mostly fits into ASCII, UTF8 is about twice as space efficient as UTF16.

PostgreSQL supports UTF8. Its CHAR . VARCHAR and TEXT types are, by default, UTF8, meaning they will only accept UTF8 data and all the transformations applied to them, from string concatenation and searching to regular expressions, are UTF8aware. It all just works.

MS SQL Server 2008 does not support UTF16; it supports UCS2, a deprecated subset of UTF16. What this means is that most of the time, it will look like it's working fine, and occasionally, it will silently corrupt your data. Since it interprets text as a string of wide (i.e. 2byte) characters, it will happily cut a 4byte UTF16 character in half. At best, this results in corrupted data. At worst, something else in your toolchain will break badly and you'll have a disaster on your hands. Apologists for MS are quick to point out that

this is unlikely because it would require the data to contain something outside Unicode's basic multilingual plane. This is completely missing the point. A database's sole purpose is storing, retrieving and manipulating data. A database which can be broken by putting the wrong data in it is as useless as a router that breaks if you download the wrong file.

MS SQL Server versions since 2012 have supported UTF16 properly, if you ensure you select a UTF16compliant collation for your database. It is baffling that this is (a) optional and (b) implemented as late as 2012.

1.8. Data types that work properly

A common misconception is that all databases have the same types – `INT`, `CHAR`, `DATE` and so on. This is not true. PostgreSQL's type system is really useful and intuitive, free of annoyances which introduce bugs or slow work down and, as usual, apparently designed with productivity in mind.

MS SQL Server's type system, by comparison, feels like beta software. It can't touch the feature set of PostgreSQL's type system and it is beset with traps waiting to ensnare the unwary user. Let's take a look:

CHAR, VARCHAR and family

PostgreSQL: the docs actively encourage you to simply use the `TEXT` type. This is a high-performance, UTF8 validated text storage type which stores strings up to 1GB in size. It supports all the text operations PostgreSQL is capable of: simple concatenation and substringing; regex searching, matching and splitting; full text search; casting; character transformation; and so on. If you have text data, stick it in a `TEXT` field and carry on. Moreover, since anything in a `TEXT` field (or, for that matter, `CHAR` or `VARCHAR` fields) must be UTF8, there is no issue with encoding incompatibility. Since UTF8 is the de facto universal text encoding, converting text to it is easy and reliable. Since UTF8 is a superset of ASCII, this conversion is often trivially easy or altogether unnecessary. It all just works.

MS SQL Server: it's a different story (http://msdn.microsoft.com/enus/library/ms187993.aspx). The `TEXT` and `NTEXT` types exist and stretch to 2GB. Bafflingly, though, they don't support casting (http://msdn.microsoft.com/enus/library/ms187928.aspx). Also, don't use them, says MS – they will be removed in a future version of MS SQL Server. You should use `CHAR`, `VARCHAR` and their `N` prefixed versions instead. Unfortunately, `VARCHAR(MAX)` has poor performance characteristics and `VARCHAR(8000)` (the next biggest size, for some reason) tops out at 8,000 bytes. (It's 4,000 characters for `NVARCHAR`.)

Remember how PostgreSQL's insistence on a single text encoding per database makes everything w

> As with earlier versions of SQL Server, data loss during code page translations is not reported. [link (http://msdn.microsoft.com/enus/library/ms176089.aspx)]

In other words, MS SQL Server might corrupt your data, and you won't know about it until something else goes wrong. This is, quite simply, a deal breaker. A data analytics platform which might silently change, corrupt or lose your data is an enormous liability. Consider the absurdity of forking out for a server using expensive ECC RAM as a defense against data corruption caused by cosmic rays, and then running software on it which might corrupt your data anyway.

Date and time types

PostgreSQL: you get `DATE`, `TIME`, `TIMESTAMP` and `TIMESTAMP WITH TIME ZONE`, all of which do exactly what you would expect. They also have fantastic range and precision, supporting microsecond resolution from the 5th millennium BC to almost 300 millennia in the future. They accept input in a wide variety of formats and the last one has full support for time zones.

They can be converted to and from Unix time, which is very important for interoperability with other systems.

They can take the special values `infinity` and `-infinity`. This is not a metaphysicotheologicophilosophical statement, but a hugely useful semantic construction. For example, set a user's password expiry date to `infinity` to denote that they do not have to change their password. The standard way of doing this is to use `NULL` or some date far in the future, but these are clumsy hacks – they both involve putting inaccurate information in the database and writing application logic to compensate. What happens when a developer sees `NULL` or `3499-12-31`? If you're lucky, he knows the secret handshakes and isn't confused by it. If not, he assumes either that the date is unknown or that it really does refer to the 4th millennium, and you have a problem. The cumulative effect of hacks and workarounds like this is unreliable systems, unhappy programmers and increased business risk. Helpful semantics like `infinity` and `-infinity` allow you to say what you mean and write consistent, readable application logic.

They also support the `INTERVAL` type, which is so useful it has its own section right after this one.

Casting and conversion of date and time types is easy and intuitive you can cast any type to TEXT , and the to_char and to_timestamp functions give you ultimate flexibility allowing conversion in

SELECT to_char('2001-02-03'::DATE, 'FMDay DD Mon YYYY'); --this produces the string "Saturday 03 Feb 2001"

and, going in the other direction,

SELECT to_timestamp('Saturday 03 Feb 2001', 'FMDay DD Mon YYYY'); --this produces the timestamp value 2001-02-03 00:00:00+00

As a data analyst, I care very much about a database's date handling ability, because dates and times tend to occur in a multitude of different formats and they are usually critical to the analysis itself.

MS SQL Server: dates can only have positive 4-digit years, so they are restricted to 0001 AD to 9999 AD. They do not support infinity and -infinity . They do not support interval types, so date arithmetic is tedious. You can convert them to and from UNIX time, but it's a hack involving adding seconds to the UNIX epoch, 19700101T00:00:00Z, which you therefore must know and be willing to hardcode into your application.

Date conversion deserves a special mention, because even by MS SQL Server's standards it's awful. The CONVERT function takes the place of PostgreSQL's to_char and to_timestamp but it works l

SELECT CONVERT(datetime, '2001-02-03T12:34:56.789', 126); --this produces the datetime value 2001-02-03 12:34:56:789

That's right – you're simply expected to know that "126" is the code for converting strings in that format to a datetime. MSDN provides a table (http://msdn.microsoft.com/enus/library/ms187928.aspx) of these numbers. I didn't give the same example as for PostgreSQL because I couldn't find a magic number corresponding to the right format for "Saturday 03 Feb 2001". If someone gave you data with such dates in it, I guess you'd have to do some string

manipulation (pity the string manipulation facilities in MS SQL Server are almost nonexistent.)

INTERVAL

PostgreSQL: the INTERVAL type represents a period of time, such as "30 microseconds" or "50 years". It can also be negative, which may seem counterintuitive until you remember that the word "ago" exists. PostgreSQL also knows about "ago", in fact, and will accept strings like '1 day ago' as interval values (this will be internally represented as an interval of 1 days). Interval values let you do intuitive date arithmetic and store time durations as firstclass data values. They work exactly as you expect and can be freely casted and converted to and from anything which makes sense.

MS SQL Server: no support for interval types.

Arrays

PostgreSQL: arrays are supported as a first-class data type, meaning fields in tables, variables in PL/PGSQL, parameters to functions and so on can be arrays. Arrays can contain any data type you like, including other arrays. This is very, very useful. Here are some of the things you can do with arrays:

Store the results of function calls with arbitrarily many return values, such as regex matches
Represent a string as integer word IDs, for use in fast text matching algorithms
Aggregation of multiple data values across groups, for efficient crosstabulation Perform row operations using multiple data values without the expense of a join
Accurately and semantically represent array data from other applications in your tool stack Feed array data to other applications in your tool stack

I can't think of any programming languages which don't support arrays, other than crazy ones like Malbolge (http://en.wikipedia.org/wiki/Malbolge). Arrays are so useful that they are ubiquitous. Any system, especially a data analytics platform, which doesn't support them is crippled.

MS SQL Server: no support for arrays.

JSON

PostgreSQL: full support for JSON, including a large set of utility functions for transforming between JSON types and tables (in both directions), retrieving values from JSON data and constructing JSON data. Parsing and stringification are handled by simple casts, which as a rule in PostgreSQL are intelligent and robust. The PL/V8 procedural language works as seamlessly as you would expect with JSON – in fact, a JSONtype internal state in a custom aggregate (see this example) whose transition function is written in PL/V8 provides a declarative/imperative bestofbothworlds that's powerful and convenient.

JSON (and its variants, such as JSONB) is, of course, the de facto standard data transfer format on the web and in several other data platforms, such as MongoDB and ElasticSearch, and in any system with a RESTful interface. Aspiring AnalyticsasaService providers take note.

MS SQL Server: no support for JSON.
(Update: starting in 2016, json support was added to SQL Server, where it previously was in the MS Azure product line.)

HSTORE

PostgreSQL: HSTORE is a PostgreSQL extension which implements a fast keyvalue store as a data type. Like arrays, this is very useful because virtually every highlevel programming language has such a concept (and virtually every programming language has such a concept because it is very useful). JavaScript has objects, PHP has associative arrays, Python has dicts, C++ has `std::map` and `std::unordered_map` , Go has maps. And so on.

In fact, the notion of a keyvalue store is so important and useful that there exists a whole class of NoSQL databases which use it as their main storage paradigm. They're called, uh, keyvalue stores (http://en.wikipedia.org/wiki/NoSQL#Keyvalue_stores).

There are also some fun unexpected uses of such a data type. A colleague recently asked me if there was a good way to deduplicate a text array. Here's what I came up with:

```
SELECT akeys(hstore(my_array, my_array)) FROM my_table;
```

i.e. put the array into both the keys and values of an HSTORE, forcing a dedupe to take place (since key values are unique) then retrieve the keys from the HSTORE. There's that PostgreSQL versatility again.

MS SQL Server: No support for keyvalue storage.
(Update: There are online reports of support within SQL Azure V12+)

Range types

PostgreSQL: range types represent, well, ranges. Every database programmer has seen fields called `start_date` and `end_date`, and most of them have had to implement logic to detect overlaps. Some have even found, the hard way, that joins to ranges using `BETWEEN` can go horribly wrong, for a number of reasons.

PostgreSQL's approach is to treat time ranges as first-class data types. Not only can you put a range of time (or `INT`s or `NUMERIC`s or whatever) into a single data value, you can use a host of built-in operators to manipulate and query ranges safely and quickly. You can even apply specially developed indices to them to massively accelerate queries that use these operators. In short, PostgreSQL treats ranges with the importance they deserve and gives you the tools to work with them effectively. I'm trying not to make this document a mere list of links to the PostgreSQL docs, but just this once, I suggest you go and see for yourself (http://www.postgresql.org/docs/9.3/static/rangetypes.html).

(Additionally, if the predefined types don't meet your needs, you can define your own ones. You don't have to touch the source code, the database exposes methods to allow you to do this.)

MS SQL Server: no support for range types.

NUMERIC and DECIMAL

PostgreSQL: NUMERIC (and DECIMAL they're synonyms) is arbitrary precision: it supports 131,072 digits before the decimal point and 16,383 digits after the decimal point. If you're running a bank, doing technical computation, landing spaceships on comets or simply doing something where you cannot tolerate rounding errors, you're covered.

MS SQL Server: NUMERIC (and DECIMAL they're synonyms) supports a maximum of 38 decimal places precision in total.

XML

PostgreSQL: XML is supported as a data type and the database offers a variety of functions for working with XML. Xpath querying is supported.

MS SQL Server: MS SQL Server has an XML data type too, and offers plenty of support for working with it. (too bad that XML is going out of style...)

1.9. Scriptability

PostgreSQL can be driven entirely from the command line, and since it works in operating systems with proper command lines (i.e. everything except Windows), this is highly effective and secure. You can SSH to a server and configure PostgreSQL from your mobile phone, if you have to (I have done so more than once). You can automate deployment, performance tuning, security, admin and analytics tasks with scripts. Scripts are very important because unlike GUI processes, they can be copied, version controlled, documented, automated, reviewed, batched and diffed. For serious work, text editors and command lines are king.

MS SQL Server is driven through a GUI. I don't know to what extent it can be automated with Powershell; I do know that if you Google for help and advice on getting things done in MS SQL Server, you get a lot of people saying "rightclick on your database, then click on Tasks...". GUIs do not work well across low bandwidth or high latency connections; text based shells do. As I write I am preparing to do some sysadmin on a server 3,500 miles away, on a VPN via a shaky WiFi hotspot, and thanking my lucky stars it's an Ubuntu/PostgreSQL box.

(Who on Earth wants a GUI on a server anyway?)

1.10. Good external language bindings

PostgreSQL is very, very easy to connect to and use from programming environments, because libpq, its external API, is very well-designed and very well documented. This means that writing utilities which plug into PostgreSQL is very easy and convenient, which makes the database more versatile and a better fit in an analytics stack. On many occasions I have knocked up a quick program in C or C++ which connects to PostgreSQL, pulls some data out and does some heavy calculations on it, e.g. using multithreading or special CPU instructions stuff the database itself is not suitable for. I have also written C programs which use setuid to allow normal users to perform certain administrative tasks in PostgreSQL. It is very handy to be able to do this quickly and neatly.

MS SQL Server's external language bindings vary. Sometimes you have to install extra drivers. Sometimes you have to create classes to store the data you are querying, which means knowing

at compile time what that data looks like. Most importantly, the documentation is a confusing, tangled mess, which makes getting this done unnecessarily time-consuming and painful.

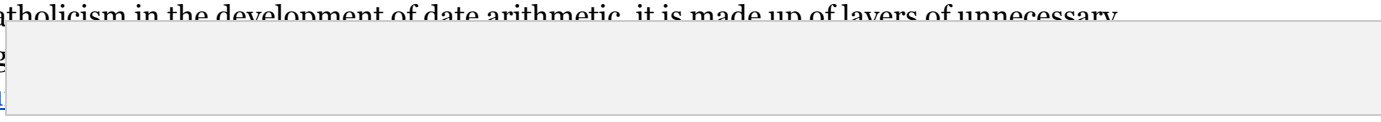
1.11.cumentation

Data analytics is all about being a jack of all trades. We use a very wide variety of programming languages and tools. (Off the top of my head, the programming/scripting languages I currently work with are PHP, JavaScript, Python, R, C, C++, Go, three dialects of SQL, PL/PGSQL and Bash.) It is hopelessly unrealistic to expect to learn everything you will need to know up front. Getting stuff done frequently depends on reading documentation. A well-documented tool is more useful and allows analysts to be more productive and produce higher quality work.

PostgreSQL's documentation (http://www.postgresql.org/docs/9.3/static/index.html) is excellent. Everything is covered comprehensively but the documents are not merely reference manu
progr
all of

> The first century starts at 00010101 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from 1 century to 1 century. If you disagree with this, please write your complaint to: Pope, Cathedral SaintPeter of Roma, Vatican.

MS SQL Server's documentation is all on MSDN, which is a sprawling mess. Because Microsoft is a large corporation and its clients tend to be conservative, the documentation is "business appropriate" – i.e. boring and dry. Not only does it lack amusing references to the historical role of Catholicism in the development of date arithmetic, it is made up of layers of unnecessary categ
docu



MS SQL Server 2012 and try to get from there to something useful. Or try reading this gem (not cherrypicked, I promise):

> A report part definition is an XML fragment of a report definition file. You create report parts by creating a report definition, and then selecting report items in the report to publish separately as report parts.

Has the word "report" started to lose its meaning (http://en.wikipedia.org/wiki/Semantic_satiation) yet?

(And, of course, MS SQL Server is closed source, so you can't look at the source code. Yes, I know source code is not the same as documentation, but it is occasionally surprisingly useful to be able to simply grep the source for a relevant term and cast an eye over the code and the comments of the developers. It's easy to think of our tools as magical black boxes and to forget that even something as huge and complex as an RDBMS engine is, after all, just a list of instructions written by humans in a human readable language.)

1.12. Logging that's useful

MS SQL Server's logs are spread across several places error logs, Windows event log, profiler logs, agent logs and setup log. To access these, you need varying levels of permissions and you have to use various tools, some of which are GUIonly. Maybe things like Splunk (http://www.splunk.com/) can help to automate the gathering and parsing of these logs. I haven't tried, nor do I know anyone else who has. Google searches on the topic produce surprisingly little information, surprisingly little of which is of any use.

PostgreSQL's logs, by default, are all in one place. By changing a couple of settings in a text file, you can get it to log to CSV (and since we're talking about PostgreSQL, it's proper CSV, not broken CSV). You can easily set the logging level anywhere from "don't bother logging anything" to "full profiling and debugging output". The documentation even contains DDL for a table into which the CSVformat logs can be conveniently imported. You can also log to stderr or the system log or to the Windows event log (provided you're running PostgreSQL in Windows, of course).

The logs themselves are human readable and machine-readable and contain data likely to be of great value to a sysadmin. Who logged in and out, at what times, and from where? Which queries are being run and by whom? How long are they taking? How many queries are submitted in each batch? Because the data is well formatted CSV, it is trivially easy to visualize or analyses it in R or PostgreSQL itself or Python's matplotlib or whatever you like. Overlay this with the wealth of information that Linux utilities like top, iotop and iostat provide and you have easy, reliable access to all the server telemetry you could possibly need.

1.13. Support

How is PostgreSQL going to win this one? Everyone knows that expensive flagship enterprise products by big commercial vendors have incredible support, whereas free software doesn't have any!

Of course, this is nonsense. Commercial products have support from people who support it because they are paid to. They do the minimum amount necessary to satisfy the terms of the SLA. As I type this, some IT professionals I know are waiting for a major hardware vendor to help them with a performance issue in a £40,000 server. They've been discussing it with the vendor for weeks; they've spent time and effort running extensive tests and benchmarks at the vendor's request; and so far, the vendor's reaction has been a mixture of incompetence, fecklessness and apathy. The £40,000 server is sitting there performing very, very slowly, and its users are working 70-hour work weeks to try to stay on schedule.

Over the years I have seen many, many problems with expensive commercial software – everything from bugs to performance issues to incompatibility to insufficient documentation. Sometimes these problems cause a late night or a lost weekend for the user; sometimes they cause missed deadlines and angry clients; sometimes it goes as far as legal and reputational risk.

Every single time, the same thing happens: the problem is fixed by the end users, using a combination of blood, sweat, tears, Google and late nights. I have never seen the vendor swoop to the rescue and make everything OK.

So what is the support for PostgreSQL like? On the two occasions I have asked the PostgreSQL mailing list for help, I have received replies from Tom Lane (http://en.wikipedia.org/wiki/Tom_Lane_(computer_scientist)) within 24 hours. Take a moment to click on the link and read the wiki – the guy is not just a lead developer of PostgreSQL, he's a well-known computer programmer. Needless to say, his advice is as good as advice gets. On one of the occasions, where I asked a question about the best way to implement crossfunction call persistent memory allocation, Lane replied with the features of PostgreSQL I should study and suggested solutions to my problem – and for good measure he threw in a list of

very good reasons why my tentative solution (a C static variable) was rubbish. You can't buy that kind of support, but you can get it from a community of enthusiastic open source developers. Oh, did I mention that the total cost of the database software and the helpful advice and recommendations from the acclaimed programmer was £0.00?

Note that by "support" I mean "help getting it to work properly". Some people (usually people who don't actually use the product) think of support contracts more in terms of legal coverage – they're not really interested in whether help is forthcoming or not, but they like that there's someone to shout at and, more importantly, blame. I discuss this too, here.

(And if you're really determined to pay someone to help you out, you can of course go to any of the organizations which provide professional support for PostgreSQL. Unlike commercial software vendors, whose support functions are secondary to their main business of selling products, these organizations live or die by the quality of the support they provide, so it is very good.)

1.14. Flexible, scriptable database dumps

I've already talked about scriptability, but database dumps are very important, so they get their own bit here. PostgreSQL's dump utility is extremely flexible, command line driven (making it easily automatable and scriptable) and well documented (like the rest of PostgreSQL). This makes database migration, replication and backups – three important and scary tasks – controllable, reliable and configurable. Moreover, backups can be in a spaceeffecient compressed format or in plain SQL, complete with data, making them both human readable and executable. A backup can be of a single table or of a whole database cluster. The user gets to do exactly as he pleases. With a little work and careful selection of options, it is even possible to make a DDLonly plain SQL PostgreSQL backup executable in a different RDBMS.

MS SQL Server's backups are in a proprietary, undocumented, opaque binary format.

1.15. Reliability

Neither PostgreSQL nor MS SQL Server are crash happy, but MS SQL Server does have a bizarre failure mode which I have witnessed more than once: its transaction logs become enormous and prevent the database from working. In theory the logs can be truncated or deleted but the documentation is full of dire warnings against such action.

PostgreSQL simply sits there working and getting things done. I have never seen a PostgreSQL database crash in normal use.

PostgreSQL is relatively bug free compared to MS SQL Server. I once found a bug in PostgreSQL 8.4 – it was performing a string distance calculation algorithm wrongly. This was a problem for me because I needed to use the algorithm in some fuzzy deduplication code I was writing for work. I looked up the algorithm on Wikipedia, gained a rough idea of how it works, found the implementation in the PostgreSQL source code, wrote a fix and emailed it to one of the PostgreSQL developers. In the next release of PostgreSQL, version 9.0, the bug was fixed. Meanwhile, I applied my fix to my own installation of PostgreSQL 8.4, recompiled it and kept working. This will be a familiar story to many of the users of PostgreSQL, and indeed any large piece of open source software. The community benefits from high-quality free software, and individuals with the appropriate skills do what they can to contribute. Everyone wins.

With a closed source product, you can't fix it yourself – you just raise a bug report, cross your fingers and wait. If MS SQL Server were open source, section 1.1 above would not exist, because I (and probably thousands of other frustrated users) would have damn well written a proper CSV parser and plumbed it in years ago.

1.16. Ease of installing and updating

Does this matter? Well, yes. Infrastructure flexibility is more important than ever and that trend will only continue. Gone are the days of the big fat server install which sits untouched for years on end. These days it's all about fast, reliable, flexible provisioning and keeping up with cutting-edge features. Also, as the saying goes, time is money.

I have installed MS SQL Server several times. I have installed PostgreSQL more times than I can remember probably at least 50 times.

Installing MS SQL Server is very slow. It involves immense downloads (who still uses physical install media?) and lengthy, important sounding processes with stately progress bars. It might fail if you don't have the right version of .NET or the right Windows service pack installed. It's the kind of thing your sysadmin needs to find a solid block of time for.

Installing PostgreSQL the canonical way – from a Linux repo – is as easy as typing a single command, like this:

```
sudo apt-get install postgresql
```

How long does it take? I just tested this by spinning up a cheap VM in the cloud and installing PostgreSQL using the above command. It took 16 seconds. That's the total time for the download and the install.

As for updates, any software backed by a Linux repo is trivially easily patched and updated by pulling updates from the repo. Because repos are clever and PostgreSQL is not obscenely bloated, downloads are small and fast and application of updates is efficient.

I don't know how easy MS SQL Server is to update. I do know that a lot of production MS SQL Server boxes in certain organizations are still on version 2008 R2 though...

1.17. The contrib modules

As if the enormous feature set of PostgreSQL is not enough, it comes with a set of extensions called contrib modules. There are libraries of functions, types and utilities for doing certain useful things which don't quite fall into the core feature set of the server. There are libraries for fuzzy string matching, fast integer array handling, external database connectivity, cryptography, UUID generation, tree data types and loads, loads more. A few of the modules don't even do anything except provide templates to allow developers and advanced users to develop their own extensions and custom functionality.

Of course, these extensions are trivially easy to install. For example, to install the fuzzystrmatch exten

```
CREATE EXTENSION fuzzystrmatch;
```

1.18. It's free

PostgreSQL is free as in freedom and free as in beer. Both types of free are extremely important.

The first kind, free as in freedom, means PostgreSQL is opensource and very permissively licensed. In practical terms, this means that you can do whatever you want with it, including distributing software which includes it or is based on it. You can modify it in whatever way you see fit, and then you can distribute the modifications to whomever you like. You can install it as many times as you like, on whatever you like, and then use it for any purpose you like.

The second kind, free as in beer, is important for two main reasons. The first is that if, like me, you work for a large organization, spending that organization's money involves red tape. Red tape means delays and delays sap everyone's energy and enthusiasm and suppress innovation. The second reason is that because PostgreSQL is free, many developers, experimenters, hackers, students, innovators, scientists and so on (the brainybutpoor crowd, essentially) use it, and it develops a wonderful community. This results in great support (as I mentioned above) and contributions from the intellectual elite. It results in a better product, more innovation, more solutions to problems and more time and energy spent on the things that really matter.

2. The counterarguments

For reasons which have always eluded me, people often like to ignore all the arguments and evidence above and try to dismiss the case for PostgreSQL using misconceptions, myths, red herrings and outright nonsense. Stuff like this:

2.1. But a big-name vendor provides a safety net!

This misconception is a variant of the old adage "noone ever got fired for buying IBM". Hilariously, if you type that into Google, the first hit is the Wikipedia article on fear, uncertainty and doubt (http://en.wikipedia.org/wiki/Fear,_uncertainty_and_doubt) and even more hilariously, the first entry in the "examples" section is "Microsoft". I promise I did not touch the Wikipedia article, I simply found it like that.

In client serving data analytics, you must get it right. If you destroy your reputation by buggering up an important job, your software vendor will not build you a new reputation. If you get sued, then maybe you can recover costs from your vendor but only if they did something wrong. Microsoft isn't doing anything technically wrong with MS SQL Server, they're simply releasing a limited product and being up front about how it is. The documentation admits it's terrible. It works exactly as designed; the problem is that the design is limited. You can't sue Microsoft just because you didn't do your due diligence when you picked a database.

Even if you somehow do successfully blame the vendor, you still have a messed up job and an angry client, who won't want to hear about MS SQL Server's unfortunate treatment of UTF16 text as UCS2, resulting in truncation of a surrogate pair during a substring operation and subsequent failure to identify an incriminating keyword. At best they will continue to demand results (and probably a discount) ; at worst, they will write you off as incompetent – and who

could blame them, when you trusted their job to a RDBMS whose docs unapologetically acknowledge that it might silently corrupt your data?

Since the best way to minimize risk is to get the job done right, the best tool to use is the one which is most likely to let you accomplish that. In this case, that's PostgreSQL.

2.2. But what happens if the author of PostgreSQL dies?!

Same thing that happens if the author of MS SQL Server dies – nothing. Also, needless to say, "the author of PostgreSQL" is as meaningless as "the author of MS SQL Server". There's no such thing.

A senior individual with an IT infrastructure oversight role asked me this question once (about Hadoop, not PostgreSQL). There just seems to be a misconception that all opensource software is written by a loner who lives in his mum's basement. This is obviously not true. Large open source projects like PostgreSQL and Hadoop are
written by teams of highly skilled developers who are often commercially sponsored. At its heart, the development model of PostgreSQL is just like the development model of MS SQL Server: a large team of programmers is paid by an organization to write code. There is no single point of failure.

There is at least one key difference, though: PostgreSQL's source code is openly available and is therefore reviewed, tweaked, contributed to, improved and understood by a huge community of skilled programmers. That's one of the reasons why it's so much better.

Crucially, because opensource software tends to be written by people who care deeply about its quality (often because they have a direct personal stake in ensuring that the software works as well as possible), it is often of the very highest standard (PostgreSQL, Linux, MySQL, XBMC, Hadoop, Android, VLC, Neo4JS, Redis, 7Zip, FreeBSD, golang, PHP, Python, R, Nginx, Apache, node.js, Chrome, Firefox...). On the other hand, commercial software is often designed by committee, written in cube farms and developed without proper guidance or inspiration (Microsoft BOB, RealPlayer, Internet Explorer 6, iOS Maps, Lotus Notes, Windows ME, Windows Vista, QuickTime, SharePoint...)

2.3. But opensource software isn't secure/reliable/trustworthy/enterprise ready/etc!

There's no kind way to say this: anyone who says such a thing is very ignorant, and you should ignore them – or, if you're feeling generous, educate them. Well, I guess I'm feeling generous:

Security: the idea that closed source is more secure is an old misconception, for many (https://www.schneier.com/cryptogram9909.html#OpenSourceandSecurity) good (https://www.schneier.com/cryptogram0205.html#1) reasons (https://www.schneier.com/essays/archives/2004/10/the_nonsecurity_of.html) which I will briefly summarize (but do read the links – they're excellent): secrecy isn't the same as security; an open review process is more likely to find weaknesses than a closed one; properly reviewed open source software is difficult or impossible to build a back door into. If you prefer anecdotal evidence to logical arguments, consider that Microsoft Internet Explorer 6, once a flagship closed source commercial product, is widely regarded as the least secure software ever produced, and that Rijndael, the algorithm behind AES, which governments the world over use to protect top secret information, is an open standard.

In any case, relational databases are not security software. In the IT world, "security" is a bit like "support our troops" (http://en.wikipedia.org/wiki/Support_our_troops#Criticism_and_opponents) in the USA or "think of the children" (http://en.wikipedia.org/wiki/Think_of_the_children#Debate_tactic) in the UK – a trump card which overrules all other considerations, including common sense and evidence. Don't fall for it.

Reliability: Windows was at one point renowned for its instability, although these days things are much better. (Supposedly, Windows 9x would spontaneously crash when its internal uptime counter, counting in milliseconds, exceeded the upper bound of an unsigned 32bit integer, i.e. after 2^{32} milliseconds or about 49.7 days. I have always wanted to try this.) Linux dominates the server space, where reliability is key, and Linux boxes routinely achieve uptimes measured in years. Internet Explorer has always (and still does) failed to comply with web standards, causing websites to break or function improperly; the leaders in the field are the opensource browsers Chrome and Firefox. Lotus Notes is a flaky, crash happy, evil mess; Thunderbird just works. And I have more than once seen MS SQL Server paralyze itself by letting transaction log files blow up, something PostgreSQL does not do.

Trustworthiness: unless you've been living under a rock for the past couple of years, you know who Edward Snowden (http://en.wikipedia.org/wiki/Edward_Snowden) is. Thanks to him, we know exactly what you cannot trust: governments and the large organizations they get their hooks into. Since Snowden went public, it is clear that NSA back doors exist in a vast array of products, both hardware and software, that individuals and organizations depend on to keep their data secure.

The only defense against this is open code review. The only software that can be subjected to open code review is open source software. If you use proprietary closed source software, you have no way of knowing what it is really doing under the hood. And thanks to Mr. Snowden, we now know that there is an excellent chance it is giving your secrets away.

Enterprise readiness:

At the time of writing, 485 of the top 500 supercomputers in the world run on Linux.
As of July 2014, Nginx and Apache, two opensource web servers, power over 70% of the million busiest sites on the net.
The computers on the International Space Station (the most expensive single manmade object in existence) were moved from Windows to Linux in 2013 "in an attempt to improve stability and reliability".
The backend database of Skype (ironically now owned by Microsoft) is PostgreSQL.
GCHQ recently reported that Ubuntu Linux is the most secure commonly available desktop operating system.
The Large Hadron Collider is the world's largest scientific experiment. Its supporting IT infrastructure, the Worldwide LHC Computing Grid, is the world's largest computing grid. It handles 30 PB of data per year and spans 36 countries and over 170 computing centers. It runs primarily on Linux.
Hadoop, the current darling of many large consultancies looking to earn Big Data credentials, is opensource.
Red Hat Enterprise Linux; CEntOS (Community Enterprise OS); SUSE Linux Enterprise Server; Oracle Linux; IBM Enterprise Linux Server etc.

The idea that opensource software is not for the enterprise is pure bullshit. If you work in tech for an organization which disregards open source, enjoy it while it lasts. They won't be around for long.

2.4. But MS SQL Server can use multiple CPU cores for a single query!

This is an advantage for MS SQL Server whenever you're running a query which is CPUbound and not IObound. In reallife data analytics this happens approximately once every three blue moons. On those very rare, very specific occasions when CPU power is truly the bottleneck, you

almost certainly should be using something other than an RDBMS. RDBMSes are not for number crunching.

This advantage goes away when a server has to do many things at once (as is almost always the case). PostgreSQL uses multiprocessing – different connections run in different processes, and hence on different CPU cores. The scheduler of the OS takes care of this.

Also, I suspect this query parallelism is what necessitates the "merge" method which MS SQL Server custom aggregate assemblies are required to implement; bits of aggregation done in different threads have to be combined with each other, MapReducestyle. I further suspect that this mechanism is what prevents MS SQL Server aggregates from accepting ORDER BY clauses. So, congratulations – you can use more than one CPU core, but you can't do a basic string rollup.

2.5. But I have MS SQL Server skills, not PostgreSQL skills!

You'd rather stick with an unreliable system than spend the trivial amount of effort it takes to learn a slightly different dialect of a straightforward querying language?

2.6. But a billion Microsoft users can't all be wrong!

This is a real-life quotation as well, from a senior data analyst I used to work with. Is this implied safety in numbers?

2.7. But if it were really that good then it wouldn't be free!

People actually say this too. I feel sorry for these people, because they are unable to conceive of anyone doing anything for any reason other than monetary gain. Presumably they are also unaware of the existence of charities or volunteers or unpaid bloggers or any of the other things people do purely out of a desire to contribute or to create something or simply to take on a challenge.

This argument also depends on an assumption that open source development has no benefit for the developer, which is nonsense. The reason large enterprises opensource their code and then pay their teams to continue working on it is because doing so benefits them. If you open up your code and others use it, then you have just gained a completely free source of bug fixes, feature contributions, code review, product testing and publicity. If your product is good enough, it is used by enough people that it starts having an influence on standards, which means broader industry acceptance. You then have a favored position in the market as a provider of support and deployment services for the software. Open sourcing your code is often the most sensible course of action even if you are completely self-interested. As a case in point, here I am spending my free time writing a web page about how fabulous PostgreSQL is and then paying my own money to host it. Perhaps Teradata or Oracle are just as amazing, but they're not getting their own pages because I can't afford them, so I don't use them.

2.8. But you're biased!

No, I have a preference. The whole point of this document is to demonstrate, using evidence, that this preference is justified. If you read this and assume that just because I massively prefer PostgreSQL I must be biased, that means you are biased, because you have refused to seriously consider the possibility that it really is better.

If you think there's actual evidence that I really am biased, let me know.

2.9. But "PostgreSQL" is a stupid name!

This one is arguably true; it's pretty awkward. It is commonly mispronounced, very commonly misspelt and almost always incorrectly capitalized. It's a good job that "stupidness of name" is not something serious human beings take into account when they're choosing industrial software products.

That being said, "MS SQL Server" is literally the most boring possible name for a SQL Server provided by MS. It has anywhere from six to eight syllables, depending on whether or not you abbr

syllables for a product name. Microsoft has a thing for very long names though – possibly its greatest achievement ever is

> Microsoft® WinFX™ Software Development Kit for Microsoft® PreRelease Windows Operating System CodeNamed "Longhorn", Beta 1 Web Setup

I count 38 syllables. Wow.

2.10. But SSMS is better than pgAdmin!

(Update: both of these products have changed tremendously wince time of writing. pgAdmin had 4 releases during 2017 alone.)
It's slicker, sure. It's prettier. It has code completion, although I always turn that off because it constantly screws things up, and for every time it helps me out with a field or table name, there's at least one occasion when it does something mental, like auto"correcting" a common SQL keyword like "table" to a Microsoft monstrosity like "TABULATION_NONTRIVIAL_DISCOMBOBULATED_MACHIAVELLIAN_GANGLYON_ID" or something.

For executing SQL and looking at the results in a GUI, pgAdmin is fine. It's just not spectacular.

SSMS is obviously Windowsonly. PGAdmin is crossplatform. This is actually quite convenient. You can run PGAdmin in Windows, where you have all your familiar stuff – Office, Outlook etc. – whilst keeping the back end RDBMS in Linux. This gets you the best of both worlds (even an open source advocate like me admits that if you're a heavy MS Office user, there is no serious alternative). Several guys I work with do this.

One point in SSMS's favour is that if you run several rowreturning statements in a batch, it will give you all the results, pgAdmin returns only the last result set. This can be a drag when doing data analytics, where you often want to simultaneously query several data sets and compare the results.

There's another thing though: psql. This is PostgreSQL's commandline SQL interface. It's really, really good. It has loads of useful catalogquerying features. It displays tabular data intelligently. It has tab completion which, unlike SSMS's code completion, is actually useful, because it is context sensitive. So, for example, if you type DROP SCHEMA t and hit tab, it will suggest schema names starting with "t" (or, if there is only one, autofill it for you). It lets you jump around in the file system and use ultrapowerful text editors like vim inline. It automatically

keeps a list of executed commands. It provides convenient, useful data import and export functionality, including the "COPY TO PROGRAM" feature which makes smashing use of pipes and commandline utilities to provide another level of flexibility and control of data. It makes intelligent use of screen space. It is fast and convenient. You can use it over an SSH connection, even a slow one.

It's only serious disadvantage is that it is unsuitable for people who want to be data analysts, but are scared of command lines and typing on a keyboard.

2.11. t MS SQL Server can import straight from Excel!

Yes. Excel can output to CSV (in a rare moment of sanity, Microsoft made Excel's CSV export code work properly) and PostgreSQL can import CSV. Admittedly, it's an extra step. Is the ability to import straight from Excel a particularly important feature in an analytics platform anyway?

2.12. But PostgreSQL is slower than MS SQL Server!

A more accurate rephrasing would be "MS SQL Server is slightly more forgiving if you don't know what you're doing".

For certain operations, PostgreSQL is definitely slower than MS SQL Server – the easiest example is probably `COUNT(*)` , which is (I think) always instant in MS SQL Server and in PostgreSQL requires a full table scan (this is due to the different concurrency models they use). PostgreSQL is slow outofthe box because its default configuration uses only a tiny amount of system resources – but any system being used for serious work has been tuned properly, so raw outofthebox performance is not a worthwhile thing to argue about.

I once saw PostgreSQL criticized as slow because it was taking a long time to do some big, complex regex operations on a large table. But everyone knows that regex operations can be very computationally expensive, and in any case, what was PostgreSQL being compared to? Certainly not the MS SQL Server boxes, which couldn't do regexes.

PostgreSQL's extensive support for very clever indexes, such as range type indexes and trigram indexes, makes it orders of magnitude faster than MS SQL Server for a certain class of operations. But only if you know how to use those features properly.

The immense flexibility you get from the great procedural language support and the clever data types allows PostgreSQLbased solutions to outperform MS SQL Server based solutions by orders of magnitude. See my earlier example.

In any case, the argument about speed is never only about computer time; it is about developer time too. That's why highlevel languages like PHP and Python are very popular, despite the fact that C kicks the shit out of them when it comes to execution speed. They are slower to run but much faster to use for development. Would you prefer to spend an hour writing maintainable, elegant SQL followed by an hour of runtime, or spend three days writing buggy, desperate workarounds followed by 45 minutes of runtime?

2.13. But you never mentioned suchandsuch feature of MS SQL Server!

As I said in the banner and the intro, I am comparing these databases from the point of view of a data analyst, because I'm a data analyst and I use them for data analysis. I know about SSRS, SSAS, inmemory column stores and so on, but I haven't mentioned them because I don't use them (or equivalent features). Yes, this means this is not a comprehensive comparison of the two databases, and I never said it would be. It also means that if you care mostly about OLTP or data warehousing, you might not find this document very helpful.

2.14. But Microsoft has opensourced .NET!

Yeah, mere hours after I wrote all about how they're a vendor lockin monster and are antiopen source. D'oh.

However, let's look at this in context. Remember the [almighty ruckus (http://en.wikipedia.org/wiki/Standardization_of_Office_Open_XML)](http://en.wikipedia.org/wiki/Standardization_of_Office_Open_XML) when the Office Open XML standard was being created? Microsoft played every dirty trick in the book to ensure that MS Office wouldn't lose its dominance. Successfully, too – the closest alternative, LibreOffice, is still not a viable option, largely because of incompatibility with document formats. The OOXML standard that was finally pushed through is immense, bloated, ambiguous, inconsistent and riddled with errors. That debacle also started with an apparent gesture toward open standards on Microsoft's part.

If that seems harsh or paranoid, let's remember that this is an organization that has been in legal trouble with both the USA and the EU for monopolistic and anticompetitive behavior and abuse

of market power, in the latter case being fined almost half a billion Euros. Then there's the involvement in SCO's potentially Linuxkilling lawsuit against IBM. When Steve Ballmer was CEO he described Linux as "a cancer" (although Ballmer also said "There's no chance that the iPhone is going to get any significant market share. No chance", so maybe he just likes to talk nonsense). Microsoft has a long-established policy of preferring conquest to cooperation.

So, if they play nice for the next few years and their magnanimous gesture ushers in a new era of interoperability, productivity and harmony, I (and millions of developers who want to get on with creating great things instead of bickering over platforms and standards) will be over the moon. For now, thinking that MS has suddenly become all warm and fuzzy would just be naive.

2.15. I don't like your tone / you sound like a fanboy / is this a rant?

This page is unprofessional by definition – I'm not being paid to write it. That also means I don't have to hide the way I feel about things. I hope you appreciate the technical content even if you don't like the way I write. My sincere hope is that you can benefit from my experiences, the good, the bad and all things in-between.

pgversusms@gmail.com