Should you go Beyond Relational Databases?

By Martin Kleppmann

Relational databases, such as MySQL, PostgreSQL and various commercial products, have served us well for many years. Lately, however, there has been a lot of discussion on whether the relational model is reaching the end of its life-span, and what may come after it.

Should you care? Which database technology should you be using?

Of course the answer is “it depends”, but that’s not very helpful. Let me ask you a few questions to help you figure out which technology is appropriate to your particular application. Then I can give a few pointers so that you can find out more.

First of all, calm down. Chances are that your current database is perfectly fine for now. But you might want to keep an eye open in case you notice some symptoms which show that you are pushing the relational model to its limits. Some symptoms relate to the structure of your data:

* Do you have tables with lots of columns, only a few of which are actually used by any particular row?
* Do you have “attribute” tables where each row is a triple of (foreign key to row in another table, attribute name, attribute value) and you need ugly joins in your queries to deal with those tables?
* Have you given up on using columns for structured data, instead just serialising it (to JSON, YAML, XML or whatever) and dumping the string into your database?
* Does your schema have a large number of many-to-many join tables or tree-like structures (a foreign key that refers to a different row in the same table)?
* Do you find yourself frequently needing to make schema changes so that you can properly represent incoming data?

Other symptoms relate to the scalability of your system:

* Are you reaching the limit of the write capacity of a single database server? (If read capacity is your problem, you should set up master-slave replication. Also make sure that you have first given your database the fattest hardware you can afford, you have optimised your queries, and your schema cannot easily be split into shards.)
* Is your amount of data greater than a single server can sensibly hold?
* Are your page loads being slowed down unacceptably by background batch processes overwhelming the database?

In my opinion, too much emphasis is often placed on scalability, despite being a very remote problem on most projects. It’s understandable — large-scale computing systems are sexy, and everybody likes to think they are building a service which is going to be massively popular — but more often than not, developers would be better off focussing on their customers’ needs, and solving the scaling problem only if it actually arises.

That said, there is one more reason to consider non-relational databases: they are fashionable. It sounds like a silly idea to base a technical decision on fashion, but remember the human aspects of managing software projects. Great developers generally want to work with cool people in a cool environment

using cool technology. That means if you want to hire great developers, providing all this coolness gives you a better chance of getting the best people to work with you. If you want to get on Hacker News, cool technology is also the way to go. Fashion shouldn't be your primary reason, but all else being equal, you can probably err on the side of coolness. Don't forget the cool people and the cool environment though. And now I'll stop saying cool — it's not very cool.

Document databases and BigTable

The BigTable paper describes how Google developed their own massively scalable database for internal use, as basis for several of their services. The data model is quite different from relational databases: columns don't need to be pre-defined, and rows can be added with any set of columns. Empty columns are not stored at all.

BigTable inspired many developers to write their own implementations of this data model; amongst the most popular are HBase, Hypertable and Cassandra. The lack of a pre-defined schema can make these databases attractive in applications where the attributes of objects are not known in advance, or change frequently.

Document databases have a related data model (although the way they handle concurrency and distributed servers can be quite different): a BigTable row with its arbitrary number of columns/attributes corresponds to a document in a document database, which is typically a tree of objects containing attribute values and lists, often with a mapping to JSON or XML. Open source document databases include Project Voldemort, CouchDB, MongoDB, ThruDB and Jackrabbit.

How is this different from just dumping JSON strings into MySQL? Document databases can actually work with the structure of the documents, for example extracting, indexing, aggregating and filtering based on attribute values within the documents. Alternatively you could of course build the attribute indexing yourself, but I wouldn't recommend that unless it makes working with your legacy code easier.

The big limitation of BigTables and document databases is that most implementations cannot perform joins or transactions spanning several rows or documents. This restriction is deliberate, because it allows the database to do automatic partitioning, which can be important for scaling — see the section on distributed key-value stores below. If the structure of your data is lots of independent documents, this is not a problem — but if your data fits nicely into a relational model and you need joins, please don't try to force it into a document model.

Graph databases

Graph databases live at the opposite end of the spectrum. While document databases are good for storing data which is structured in the form of lots of independent documents, graph databases focus on the relationships between items — a better fit for highly interconnected data models.

Standard SQL cannot query transitive relationships, i.e. variable-length chains of joins which continue until some condition is reached. Graph databases, on the other hand, are optimised precisely for this kind of data. Look out for these symptoms indicating that your data would better fit into a graph model:

* you find yourself writing long chains of joins (join table A to B, B to C, C to D) in your queries;
* you are writing loops of queries in your application in order to follow a chain of relationships (particularly when you don't know in advance how long that chain is going to be);
* you have lots of many-to-many joins or tree-like data structures;

* your data is already in a graph form (e.g. information about who is friends with whom in a social network).

There is less choice in graph databases than there is in document databases: Neo4j, AllegroGraph and Sesame (which typically uses MySQL or PostgreSQL as storage back-end) are ones to look at. FreeBase and DirectedEdge have developed graph databases for their internal use.

Graph databases are often associated with the semantic web and RDF datastores, which is one of the applications they are used for. I actually believe that many other applications' data would also be well represented in graphs. However, as before, don't try to force data into a graph if it fits better into tables or documents.

MapReduce

Going on a slight tangent: if background batch processing is your problem and you are not aware of the MapReduce model, you should be. Popularised by another Google paper, MapReduce is a way of writing batch processing jobs without having to worry about infrastructure. Different databases lend themselves more or less well to MapReduce — something to keep in mind when choosing a database to fit your needs.

Hadoop is the big one amongst the open MapReduce implementations, and Skynet and Disco are also worth looking at. CouchDB also includes some MapReduce ideas on a smaller scale.

Distributed key-value stores

A key-value store is a very simple concept, much like a hash table: you can retrieve an item based on its key, you can insert a key/value pair, and you can delete a key/value pair. The value can just be an opaque list of bytes, or might be a structured document (most of the document databases and BigTable implementations above can also be considered to be key-value stores).

Document databases, graph databases and MapReduce introduce new data models and new ways of thinking which can be useful even in a small-scale application; you don't need to be Google or Facebook to benefit from them. Distributed key-value stores, on the other hand, are really just about scalability. They can scale to truly vast amounts of data — much more than a single server could hold.

Distributed databases can transparently partition and replicate your data across many machines in a cluster. You don't need to figure out a sharding scheme to decide on which server you can find a particular piece of data; the database can locate it for you. If one server dies, no problem — others can immediately take over. If you need more resources, just add servers to the cluster, and the database will automatically give them a share of the load and the data.

When choosing a key-value store you need to decide whether it should be opimised for low latency (for lightning-fast data access during your request-response cycle) or for high throughput (which is what you need for batch processing jobs).

Other than the BigTables and document databases above, Scalaris, Dynomite and Ringo provide certain data consistency guarantees while taking care of partitioning and distributing the dataset. MemcacheDB and Tokyo Cabinet (with Tokyo Tyrant for network service and LightCloud to make it distributed) focus on latency.

The caveat about limited transactions and joins applies even more strongly for distributed databases.

Different implementations take different approaches, but in general, if you need to read several items, manipulate them in some way and then write them back, there is no guarantee that you will end up in a consistent state immediately (although many implementations try to become eventually consistent by resolving write conflicts or using distributed transaction protocols; see the algorithm of Amazon's Dynamo for an example). You should therefore only use these databases if your data items are independent, and if availability and performance are more important than ACID properties. For more information, read about Brewer's CAP Theorem, which states that amongst Consistency, Availability and Partition tolerance, you can only choose two, and no database will ever be able to get around that fact.

Richard Jones, co-founder of Last.fm, has written up an excellent overview of distributed key-value stores. Also Tony Bain gives an introduction to the conceptual differences between relational databases and key-value stores, and recently there was a NOSQL event in San Francisco at which a number of different non-relational databases were presented.

Distributed systems are hard… really hard. I suggest that you use them only if you really need the scaling aspects they offer (or just for fun outside of a production environment).
Closing remarks

In this article I have concentrated on open source projects. If you are willing to bind yourself to a particular vendor/hosting provider, Google's Datastore, Amazon SimpleDB, Windows Azure Storage Services or Force.com might be worth considering. They are good technologies, but keep in mind the business risk of potential lock-in.

I can't make judgement about particular projects' suitability for particular purposes. There is some very clever software out there, but also some very new and unstable software. If you want to consider using them, you should do your own research:

* look around their websites for a list of sites using the database in production (and for which aspect of their service they use it);
* check if they have a lively open source community, in case the original developer loses interest and stops maintaining the software;
* try to find some benchmarks (though beware that many benchmarks published on the web are methodologically flawed and/or outdated, so if you are serious about it you should run your own tests, using data which matches your application's characteristics).

As with any fashionable topic, there are many people with strong opinions, both positive and negative; don't let yourself be put off by them. I hope I've given you an overview of the kind of things you can do with different types of databases so that you can choose the right one for your application.
Like this article?

If you enjoyed, this article, feel free to re-tweet it to let others know. Thanks, we appreciate it! :)

Photo Credit: flickr.com/photos/vermininc
33
Enjoy this article?

If you liked this article, feel free to re-tweet it to let others know. Thanks, we appreciate it :)
This post was written by

Martin Kleppmann

Martin Kleppmann is the architect behind Go Test It, an excellent new product for automated multi-browser functional testing of web applications. If you liked this post, you can subscribe to his blog or follow him on Twitter.
Tags for this article

AllegroGraph, BigTable, Cassandra, CouchDB, Databases, DirectedEdge, FreeBase, Hadoop, HBase, Hypertable, Jackrabbit, MapReduce, MongoDB, MySQL, Neo4j, PostgreSQL, Project Voldemort, Sesame, Skynet, ThruDB
Related Articles

1. Doctype: CSS3 gradients and local databases in HTML5
2. Steve Huffman on Lessons Learned at Reddit

We love .net Magazine

We're big fans of .net so they've hooked us up with a rad deal: Save 45 percent on your subscription.
33 Comments

*
Lucas Vogel

June 25, 2009 - 1:10 am

What about the Oracle Berkeley DB product family?
Reply
*
James Pearce

June 25, 2009 - 3:42 am

Yes, of course it's not X vs Y as mutually exclusive choices.

The people that should be really thinking carefully are those who are using X for a problem better solved by Y, or Y for a problem better solved by X.

In web apps, where relationships are often more vague and data more sparse and distributed, these new store approaches are fantastic.

On the other hand, in certain enterprise applications, an RDBMS may continue to be the ideal platform, come what may. (Would it be ideal for a bank to store your account activity in a CouchDB document?)

Also there are still unique advantages that RDBMS have – for balance, you could mention how these are lacking in the newer alternatives.

(One example: little native support for aggregation – apparently your application must keep running totals of everything as it goes – and put that in a document too. In Google's App Engine, it's even

impossible to *count* more than 1,000 rows in a table, would you believe. Imagine running a bank on that.)

So, thanks: your article is fairly balanced, and just on the right side of the naive "new vs old" arguments that have popped up on this topic.

Reply

*

Stefan

June 25, 2009 - 6:23 am

Very good overview, thank you for that.

I've seen a presentation on column-based databases by Hasso Plattner a few weeks ago. He spoke about enormous performance increases based on huge datasets with real-world enterprise data. That was quite impressive..

Reply

*

Chris Anderson

June 25, 2009 - 7:27 am

CouchDB is actually ideal for storing bank account activity (probably the best of the bunch discussed here), as running totals are incredibly cheap to calculate using incremental map reduce. CouchDB can easily give you a consistent snapshot of an account and its total balance using a very simple reduce function. It's one of the examples we use in our training courses because it shows just how good a fit CouchDB is for that sort of application.

Reply

*

irf

June 25, 2009 - 10:46 am

Are Object Databases no longer cool? ;)

Anyone wanting to be impressed by a database have a look at http://www.statbank.dk – I think it's the worlds largest statistical DB.

Reply

*

Alessio Saltarin

June 25, 2009 - 12:18 pm

Thank you for your article. I agree that RDBMS are at EOL phase (end-of-life). RDBMS engines were born when computer and I/O were slow. Now, this is no more the case. Well serialized, well "hibernated" data, written in a hierarchical fashion (JSON or XML) just do the job in most cases. Also, we won't need SQL anymore: common languages, such as Python, Ruby, Java or C# are enough, and deal with data (collections, dictionaries) in an easier and more effective way.

Reply

*
james

June 25, 2009 - 12:45 pm

> Standard SQL cannot query transitive relationships, i.e. variable-length chains of joins
> which continue until some condition is reached. Graph databases, on the other hand, are
> optimised precisely for this kind of data. Look out for these symptoms indicating that your
> data would better fit into a graph model:

The SQL Standard defines syntax for recursive queries (using WITH RECURSIVE), so yes, you can query transitive relationships. Oracle, DB2, MS SQL are among the commercial vendor systems that support them, and the free RDBMSes Firebird and the almost released PostgreSQL 8.4 will have support too. Please take what you read at neo4j with a couple packets of salt (recursive queries aren't slow due to 'increasing number of joins', either).

As for having to add columns to your database to match incoming data, that really indicates poor upfront design. But still, in most modern RDBMSes, adding to (not altering) table schemas is not very expensive, and you can always create another table with a foreign key to the master table (and update your views, because you aren't querying tables directly from your application, right?) Empty columns take from zero to a byte in most RDBMSes, so not really an issue for storage. If you feel you really need a 'schemaless' "design", some RDBMSes have a 'hash' data type which can be indexed on (key, value) (e.g., the 'hstore' extension for PostgreSQL).

Generally the 'need' for non-relational databases has been spurred by the poor quality of RDMSes such as MySQL, not by limitations of SQL and Relational Databases, or the Relational Model.

Reply

*
Shaun Bruner

June 25, 2009 - 1:13 pm

I too wonder why Berkeley DB never makes these lists? Is it because BDB doesn't have built-in server capability? Every time someone mentions key/value databases, I immediately think of the old dbm-style databases, of which BDB is probably the best known member.

Reply

*
Kris Zyp

June 25, 2009 - 1:24 pm

Sorry to gripe about missing one, but Persevere seems like a pretty obvious omission, since it is one of the more popular non-relational JSON-based databases.

Reply

*
buck nuggets

June 25, 2009 - 3:41 pm

Very nice – this is the first report I've seen that tries to make sense of the options. Right now we've

got a lot of people trying to apply lessons learned in one area to another without realizing what the pros & cons really are. Here's two more considerations.

1. data quality – which might not be a huge deal for a large social networking site – where your data may vary slightly based on which server you hit. But it's a killer for finance, reservations, etc. The RDMBS achieves this with both ACID compliance, model consistency and declarative constraints. The last two are as important as the first in ensuring that all data in the database, regardless of which application wrote it, regardless of when it was written (four years ago or today) all meet the current rules within the database.

2. reporting – there are very, very few applications that don't need reporting. One benefit of the RDBMS today is that it can make reporting very easy to deliver. One of the primary reason that object oriented databases failed ten years ago is that they couldn't support reporting. Most applications that used them had to deal with horrible performance, expensive cost to build reports, or copied data out regularly into an RDBMS and had to pay twice the admin, hardware & licensing costs. Most of the new alternatives to the RDBMS suffer from the same limitations.

Reply

*

Jags Ramnarayan

June 25, 2009 - 4:43 pm

In-memory data grid or fabric might be something you should consider mentioning in your report. They offer distributed key-value stores without losing support for querying or transactions. The products typically get deployed as a middle tier caching solution that can "read-through", "write-through" or "write-behind" to one or more backend databases. They even offer reliable publish-subsribe features so apps can now receive data change notifications enabling scalable "real time push" all the way to browser clients. Finally, one might find how these products can horizontally partition the data (by colocating related data) and support moving behavior to where the data resides (map-reduce).

Hopefully, this white paper – http://community.gemstone.com/display/gemfire/EDF... provides some insight into the important concepts.

Reply

*

Peter

June 25, 2009 - 5:44 pm

I was just going to post about Persevere as well. I like it because of the way you can extend document types – the object model makes sense to the way I handle document based data.

Reply

*

james

June 25, 2009 - 6:45 pm

Martin: too busy to approve my comment huh?

Chris: Thanks for the chuckle of the day.

Reply

*

Gergely Orosz

June 25, 2009 - 8:39 pm

Nice overview. I am missing some file-based closer to object oriented than relational databases e.g. db4o (http://www.db4o.com/) or Karvonite (http://code.msdn.microsoft.com/karvonite) in the comparison.

Having mentioned open source document-like databases and Jackrabbit I would also add the Sense/Net Content Repository (http://wiki.sensenet.hu/index.php?title=SenseNet_... to the list which is similar to Jackrabbit and the JSR 170 in the Java world.

Reply

*

John Turnbull

June 25, 2009 - 10:27 pm

So many new choices. Here's another: EMC's xDB – a native xml DB with a development solution built around it.

Reply

o

Jerry Silver

August 3, 2009 - 11:41 pm

You can find more information on EMC’s native XML database, xDB, at http://developer.emc.com/xmltech. Free download too.

Reply

*

Julian Browne

June 26, 2009 - 5:09 am

A very good and unbiased round up.

The problem with scalability these days it that the coolness factor is making people want to play at being Amazon or Google so there's a tendency to jump right into solving imaginary scale issues when, as you say, it's a "very remote problem on most projects" and it's also very hard so the end result is often unnecessary complexity, which perversely doesn't then scale.

It ain't like we haven't got enough on our respective plates providing high-quality, extensible, useful, reliable, software in the first place.

Reply

*

Drakanor

June 26, 2009 - 3:27 pm

I'm still waiting for a database natively supporting the associative data model. To me it's the best

solution for 90% of all database based application use cases. Right now, we simulate it with PostgreSQL by putting in an additional "relational-to-associative" layer which of course may raise performance issues in some cases.

Reply

*

Jonathan Gray

June 26, 2009 - 4:56 pm

There's another reason small companies are trying to build on scalable technologies early.

It's not only that they want to be prepared for an increase in customers, but their datasets are very large from day one (and growing the dataset 100X could be a matter of flipping a switch).

Many of these companies are mining data, web crawling, etc… It's the original and canonical use case for BigTable, though it's now used for a wide variety of applications.

Reply

*

Anti-RDBMS and distributed key-value stores

July 1, 2009 - 2:44 pm

[...] 1st, 2009 Update: Think Vitamin has a good article on reasons for switching to a non-relational database and also compares the tradeoffs between the different data storage options. SHARETHIS.addEntry({ [...]

Reply

*

Michael_Marth

July 2, 2009 - 10:00 am

Very good article. I like that you stress the need to fit the actual data to the model.

One small correction: Jackrabbit can deal with transactions

(see http://jackrabbit.apache.org/frequently-asked-que... In the upcoming JSR-283 specification (for which Jackrabbit is the reference implementation and which it partially implements already) query joins are also supported.

Reply

*

Rob Tweed

July 3, 2009 - 7:17 am

Also check out M/DB which is an open source clone of SimpleDB, and another from the same stable: M/DB:X which is an HTTP-interfaced lightweight Native XML Database. Indeed M/DB:X is soon to become even more interesting with a new JSON/XML mapping capability built in: input a JSON string and it will be stored as an equivalent XML DOM that can be modified and manipulated via DOM API methods and/or searched using XPath….and then returned as JSON.

Reply

*
Glenn Wiens

July 13, 2009 - 11:09 pm

Coming to the end of the comment section, I was disappointed to see no response to a question raised more than once (and one that rolls around in my mind as well): What about Berkeley DB? It has a long-established history and is being used by some of the busier websites to assist in making their RDBMS systems fast enough to keep up with web hits.

Would BDB be considered a NoSQL candidate? Or is the direction of NoSQL just new shinier things?
Reply
*
Kai's daily tech #44 … | Kai Richard König

July 18, 2009 - 8:08 pm

[...] Carsonified » Should you go Beyond Relational Databases? [...]
Reply
*
Notes from OSCON 2009 in San Jose | ecmarchitect.com

July 27, 2009 - 10:04 pm

[...] was the third "noSQL"-themed talk I saw. He made a good point that when we design apps, we should be saying, "I [...]
Reply
*
Jim McCoy

July 30, 2009 - 5:10 am

Glenn: While Berkeley DB is good tech, I think that a lot of us got burned by it in the past (three catastrophic loss of entire db situations and two others that required several days to recover from in my own experience of using BDB for more than a decade) and if given the choice today I would always pick Tokyo Cabinet over BDB. The fact that it is now in Oracle's hands is not really a point in its favor either.
Reply
*
Alex Popescu

August 1, 2009 - 9:45 pm

1. Very nice post! A while back I've started a comparison of "alternative" data storage solution. It's far from being complete, I think it might offer some quite details to newcomers (http://themindstorms.blogspot.com/2009/05/quick-reference-to-alternative-data.html).

2. I've been using BDB as a storage implementation of Jackrabbit for 4 years. It was couple of days

ago (in fact it happened exactly on Jul. 30th, the date of the last comment mentioning failures of BDB) that this database got corrupted.

3. There are a couple of things about all these solution that concern me and I'm referring here to the fact that most of them have chosen to implement custom protocols and connection APIs. While I may find some reasons for this (f.e. most of them have been initially developed to solve an internal problem and open sourced afterwards), it is quite strange to see that they lock you in. Basically, before starting to use any of these solution you'll have to spend a lot more time to figure out if the solution you are picking will fit all your needs, as later migration will be extremely difficult.

I hope there will be a time when people involved will sit down and figure out some common protocols and APIs. And I hope this will happen sooner than later, as the more we postpone this step the more difficult will be to ask those already using to change their implementations.

Reply

*

Más allá de las bases de datos relacionales en El blog de Javier Aranda

August 29, 2009 - 11:08 pm

[...] Traducción libre del texto original Should you go Beyond Relational Databases? by Martin Kleppmann [...]

Reply

*

The Python Paradox is now the Scala Paradox – Martin Kleppmann at Yes/No/Cancel

September 18, 2009 - 10:32 pm

[...] an article about non-relational databases which Ryan Carson asked me to write a few months ago, I suggested that fashion can and should play [...]

Reply

*

Eamonn O'Brien-Strain :: links for 2009-09-20

September 20, 2009 - 11:04 pm

[...] Carsonified » Should you go Beyond Relational Databases? Nicely organized overview of non-SQL databases. (tags: programming database scaling) [...]

Reply

*

Neo Technology Secures Seed Funding To Lead NoSQL Movement

October 28, 2009 - 10:41 am

[...] both the future of databases and Neo Technology, but until then I highly recommend this excellent article by Martin Kleppmann. It gives you a great overview of different database models and an excellent [...]

Reply

*

InfoGrid Blog - Carsonified: Why Graph Databases

October 28, 2009 - 5:20 pm

[...] Kleppman summarizes the case for Graph Databases at carsonified.com. This is exactly why InfoGrid is built around a [...]

Reply

*

NoSQL – Non-relational databases « Lars Barkman

May 14, 2010 - 9:22 am

[...] “if you want vast, on-demand scalability, you need a non-relational database”.” Should you go Beyond Relational Databases? – “Relational databases, such as MySQL, PostgreSQL and various commercial products, [...]