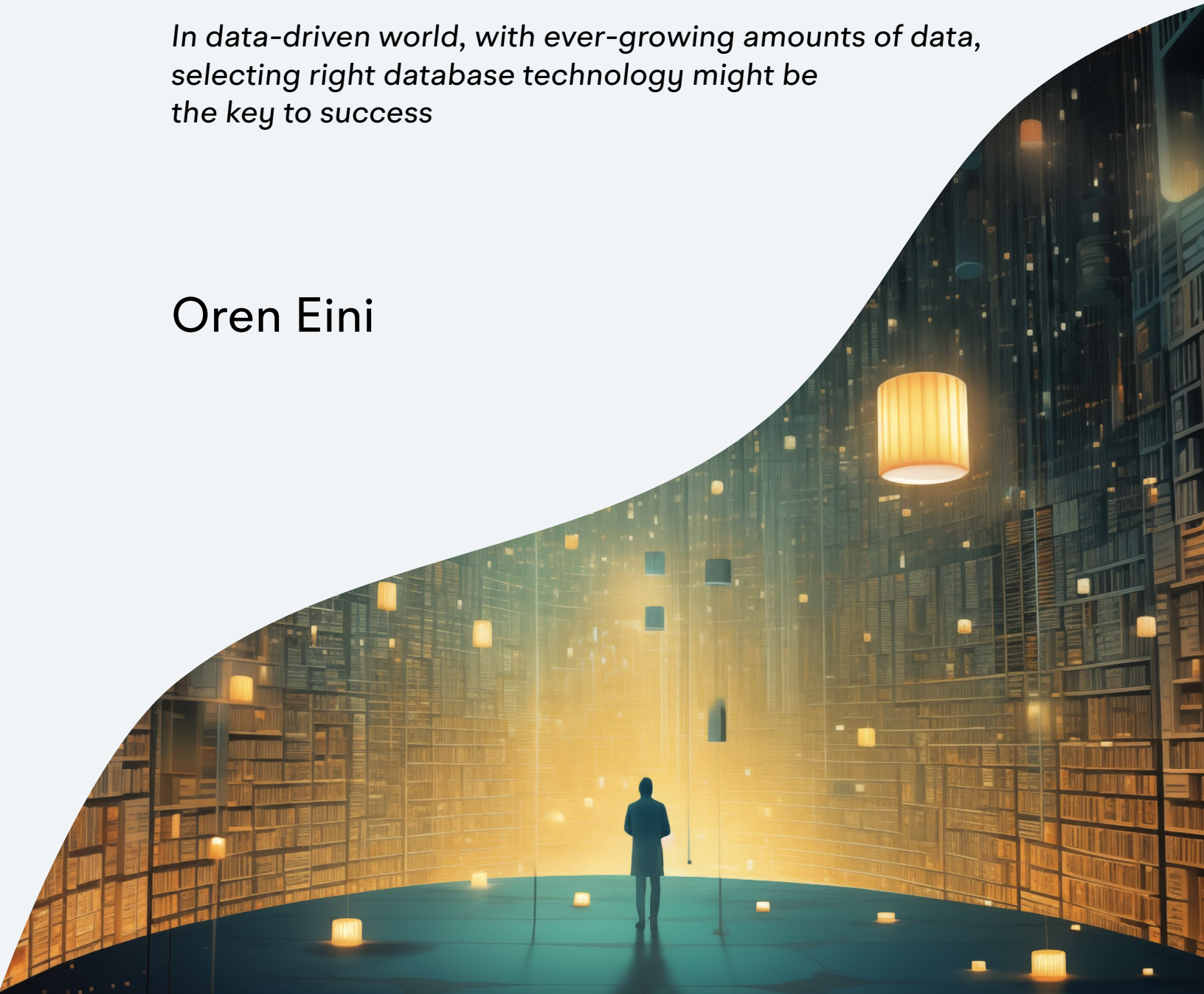
RavenDB

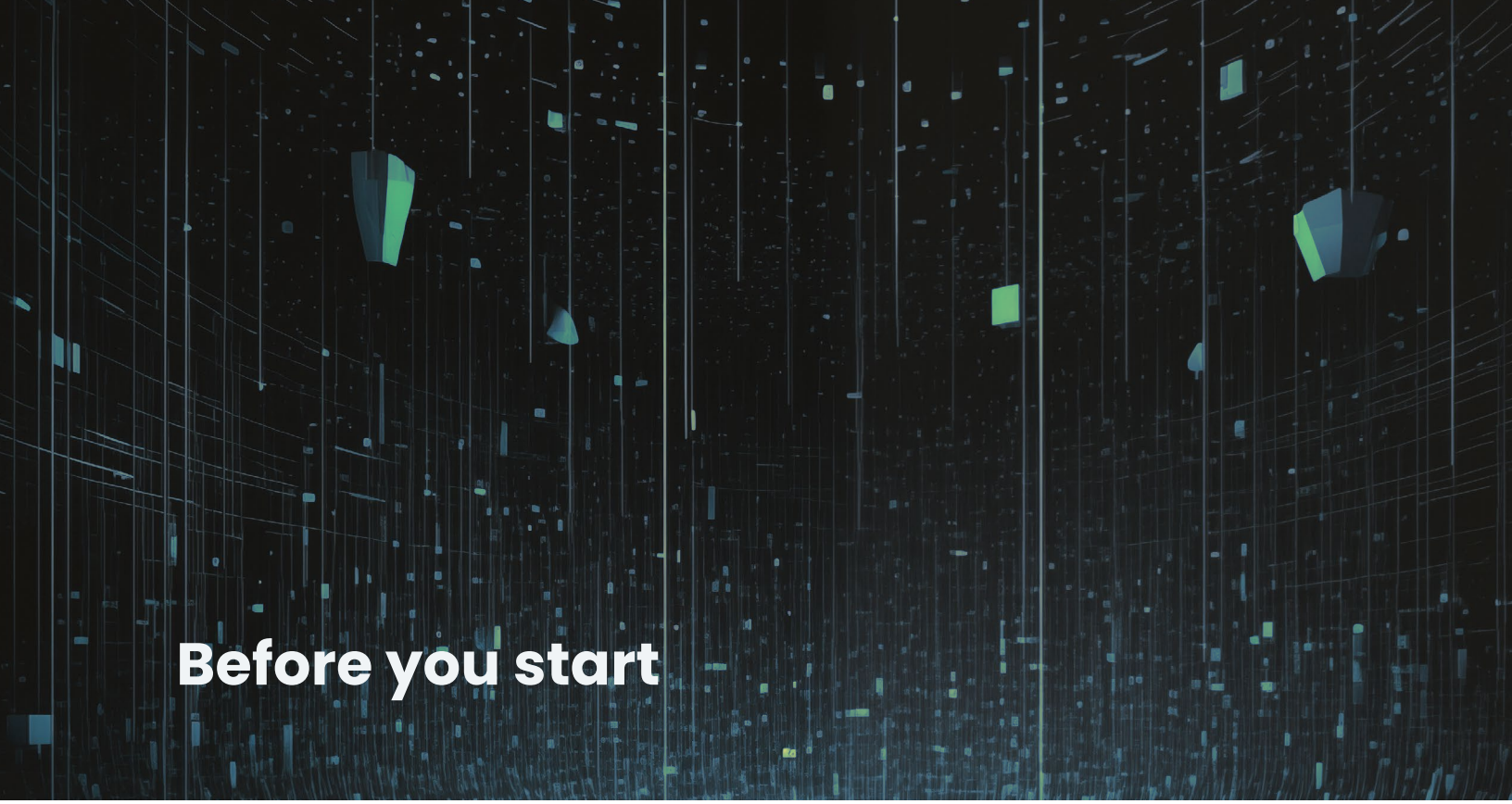
Where Should I Put My Data?

In data-driven world, with ever-growing amounts of data, selecting right database technology might be the key to success

Oren Eini



Contents



Before you start

You may have a world-changing idea rattling in your head, but the very first step in actually getting off the drawing board is going to involve a dizzying array of choices. You must figure out the technology stack, the right business plan, a marketing strategy, and a sales incentives program, to list just a few items.

You will also need to choose the database technology that would serve as the foundation for your new project. While I cannot assist you with everything on the list above, I can provide insights to guide your selection process regarding databases.

We live in a data-driven world, where your data is often far more valuable than anything else in the business. However, making good use of the data can be a big challenge. And the type of databases you employ can greatly affect your velocity and ability to adapt to your needs.

Henry Ford famously said that you can have a car in any color you want, as long as it is black. The same used to be the case for databases as well. It used to be simple. Your database choice was between different *vendors*, all of them providing a relational database using an SQL dialect. The choice was mainly about particular features, licensing considerations, and support agreements.

Then the Internet happened, and the amount of data we keep and the ways in which we work with it simply exploded. Relational databases had a massive issue with adapting to a distributed environment with the new scaling demands.

The need to process ever-growing amounts of data at higher velocity led to the development of new database engines and approaches. Those abandoned the relational model because it often couldn't manage the required scale, and we ended up with many choices for database technology.

In this article, I'm going to try to make sense of the various options and give some tips on how you can select the appropriate database technology for your needs.

Relational Restrictions

Relational databases are modeled using a few core concepts. Tables and rows are at the heart of the relational model. You can *JOIN* data from different tables when you need more complex structures. You can define schema and constraints to ensure that your data makes sense, and you use transactions to ensure overall consistency.

This is a great model, and it has been wildly successful. In fact, the Database Wars of the 1980s have been conclusively decided in favor of relational databases. It's just that the needs of the 2000s exposed several limitations in how relational databases can be used. Those limits became apparent when the size of the data and the number of transactions skyrocketed as every aspect of our lives became digital.

A rigid schema is excellent for consistency, but it also acts as a significant barrier to change when you need to modify your system to additional requirements. Trying to handle dynamic or user-generated data in a relational database is a well-known recipe for pain, slow queries, and a great deal of complexity.

The relational model also contains a hidden assumption: you can access the entire dataset at all times. That assumption was violated when the size of our datasets exceeded the amount of data that could fit into a single machine.

Consider the simple scenario of ensuring that the Email field is unique for all the users in the database. If you split your users among several servers, how can you ensure that you are not creating duplicate entries? Things become a lot harder when you add failures and how to handle them to the mix.

There are solutions for those sorts of questions, but they are quite a bit more complicated than simply marking the field as UNIQUE, as you used to do. In fact, those solutions often required you to have a Ph.D. level of understanding in database internals and distributed system theory. Regardless of your chosen solution, one thing was clear: it was very far from a classic relational database.

Moving away from relational databases meant we had to relinquish some of the capabilities considered standard in the industry, such as schemas or transactions. It also meant we had opportunities to tailor our database engines to suit modern practices better. The age of NoSQL databases began with an absolutely stunning number of options and choices.

The next decade or so was about consolidation and figuring out what features benefit our systems. Transactions, which were discarded early on (it's *complicated* to make them work in a distributed environment), came roaring back. No one actually wanted to live in a non-transactional world, after all. The benefit is that we now have mature and comprehensive database solutions that aren't limited to the relational mindset.

Those database solutions can be utilized effectively to significantly speed up the building of non-trivial applications and systems since they are explicitly geared toward that. In the rest of this article, I will attempt to give you both the background information and specific details on why this is so.

Databases Dossiers

Beyond relational databases, there are quite a few choices. There are document databases where you model the data using JSON documents. There are graph databases in which everything is a node or an edge, and relationships between nodes are paramount. There are column-family databases focused on managing petabytes of information with reasonable speeds. There are key-value databases that simplify their internal model to the maximum extent possible in order to achieve better performance.

As I mentioned, the database world is vast (and deep), and your needs will vary depending on your goals. I assume you are interested in building a business application, the bread and butter of our industry. I'm going to use vehicle rental as an example for this article. It is likely an area you are at least familiar with and has enough complexity to go beyond the basics.

The first choice we must make is to consider what database to use for our vehicle rental project. Let's consider the options. Relational databases used to be the default (and only choice), so we'll keep that option to the side and consider the non-relational databases first, then compare them to the relational option.

A key-value database allows you to store and retrieve data by key. The data may be just binary or have meaning, such as a list of items. Building applications on top of that model is complex since you don't have the ability to query data; you can only fetch by a known key. These databases often implement features such as shopping carts or storing session data. They are rarely used as the primary database of an application.

A document database deals with documents. Not Word or Excel files, but documents in the modeling sense. Your data is a JSON document, which can be arbitrarily complex and extensive. Document databases allow you to query the data and don't place restrictions on its shape and operations. They are well-suited for business applications and OLTP scenarios.

Graph databases are typically utilized when the primary need is some sort of graph operation, whether this is a recommendation engine, social networks, or finding the shortest route for a drive. They are typically used in conjunction with another database and are rarely employed as the system of record.

Column-family databases are laser-focused on managing stupendous amounts of data. They model their data in tables and columns, but columns can contain lists of values and queries always follow the primary key. The entire system is built around the constraint of working in a distributed environment. That is both its most significant advantage and its most annoying stumbling block. Unless you expect to deal with datasets exceeding the 100s of TB mark, I recommend against choosing this option.

For almost all business applications (OLTP - Online Transaction Processing), I would recommend a document database as the most suitable choice for your database. Of the choices we have explored so far, it is by far the most suitable. However, there is still one contender that we haven't addressed. How would a document database compare to a relational database for business application needs?

Document / Relational Dilemma

A while ago, the term du-jour was data-driven systems. I'm probably dating myself because that was over 25 years ago. That was the era when everything started to become digital, and the quick availability of data transformed our world. Consider the typical application from that era. You might still encounter those; airline reservation systems are still running on the same command line terminals, for example.

The level of complexity and sophistication that we demand today from our systems, as compared to what was required from applications of the past, is vast. However, the databases we used for those systems are the same relational databases we have today. Conceptually, not much has changed for relational databases since that era. But what types of systems do we build? They are drastically different.

Consider the data model aspect in the vehicle rental scenario. In a relational database, you would have tables for Vehicle, Customer, and Rental. Applying this to rental scenarios is fairly simple and quite obvious.

Until you realize that this is a highly simplified model of the actual business needs. Focusing just on the rental aspect of the business, you'll expect to be able to select the specific car type and its features, as well as any number of add-ons (such as a baby car seat or a GPS). In the rental agreement, in addition to the period you'll rent the car, there are many additional types of information that you need to track, from the allowed drivers on the vehicle to the type of insurance selected, the fuel strategy, etc. Payments may be handled using credit card or cash, or multiple credit cards. When you start looking at the *actual* model you have to deal with, the number of entities and tables explodes. That has a significant impact on your system's overall architecture and behavior.

The interesting thing about this is that 30 years ago, it wouldn't have been an issue. There was no *concept* of trying to manage everything in one system. The features you gained from digitizing your system were limited compared to what we expect today. Primarily because of the capabilities of the systems available at the time.

How would a document database address such issues? What would make it simpler to build complex modern systems using a document database versus a relational one?

The answer starts with the different models. In a relational database, complex entities are stitched together using multiple tables, joins, and complicated queries. In contrast, a document database allows you to express a complex entity directly as a single document.

Consider the rental agreement you signed when you accepted the car. It has many clauses and options. Almost each one of them would require creating an additional table to be joined in a relational system. In a document database, we can represent that rental agreement, with all of its complexity, as a single document.

I will provide specific examples to make the distinction between relational and document databases easier to understand.

Tackling Tangled Data

Consider the dashboard for an agent at the vehicle rental. They have a dashboard for the vehicles in the

lot, showing which are available. The problem is that in order to actually show usable information, you need to find the vehicles that are located in the lot. That is easy enough, but you now need to check the maintenance status and fuel level for each of those (to know which are available for rental).

By far, the most common way to actually handle that is to issue a separate query for each available vehicle to check its status. If you have 50 vehicles available, you'll issue one query to get the available vehicles and then 50 separate queries for each of them.

That is insanely inefficient, but it is common due to how the API is exposed to the application. You have one query to load available vehicles and another method to load their status. Another common reason for this is the use of OR/M tools such as Hibernate or Entity Framework.

You can use more complex queries to JOIN the data together, of course, but that only takes you so far. Joining data from multiple tables risks a Cartesian product and, even without it, can greatly increase the amount of data that the relational database needs to process.

A document database, on the other hand, enables more efficient modeling of such data. A document in such a database allows you to store complex information in a single location. This capability enables you to retrieve all the related data as a single query without having to deal with multiple queries or complex joins.

Furthermore, document databases aren't limited to the relational model. Handling relations in a document database doesn't require you to deal with a Cartesian product. You can request the database to provide both the query results and the related data in a single remote call without hassle.

This sounds like a small thing, but it has a massive impact on both development time and runtime costs. The reduced number of queries not only improves the efficiency but also greatly reduces the latency of the system - the time spent processing your requests. That is a key metric in both usability and sales conversions.

When using RavenDB, you have the freedom to model your data as you see fit, including working with complex objects and object graphs. You aren't limited only to single documents but can also work with relations between documents.

In particular, you may *include* related documents in a query, which means that in addition to the resulting matching documents RavenDB will fetch the related documents as part of processing the query in a single remote call.

Alternatively, you can *load* data from related documents during the query, projecting just the parts of the documents and their relations that you care about.

Seamless Scaling & Handling High Availability

One of the biggest challenges in designing your database solution at scale is addressing both the need to manage a large number of requests and ensure availability during failures.

Relational databases often require external assistance to handle this requirement. It's common to have a primary node that accepts writes and secondary nodes that serve as read replicas. The problem is

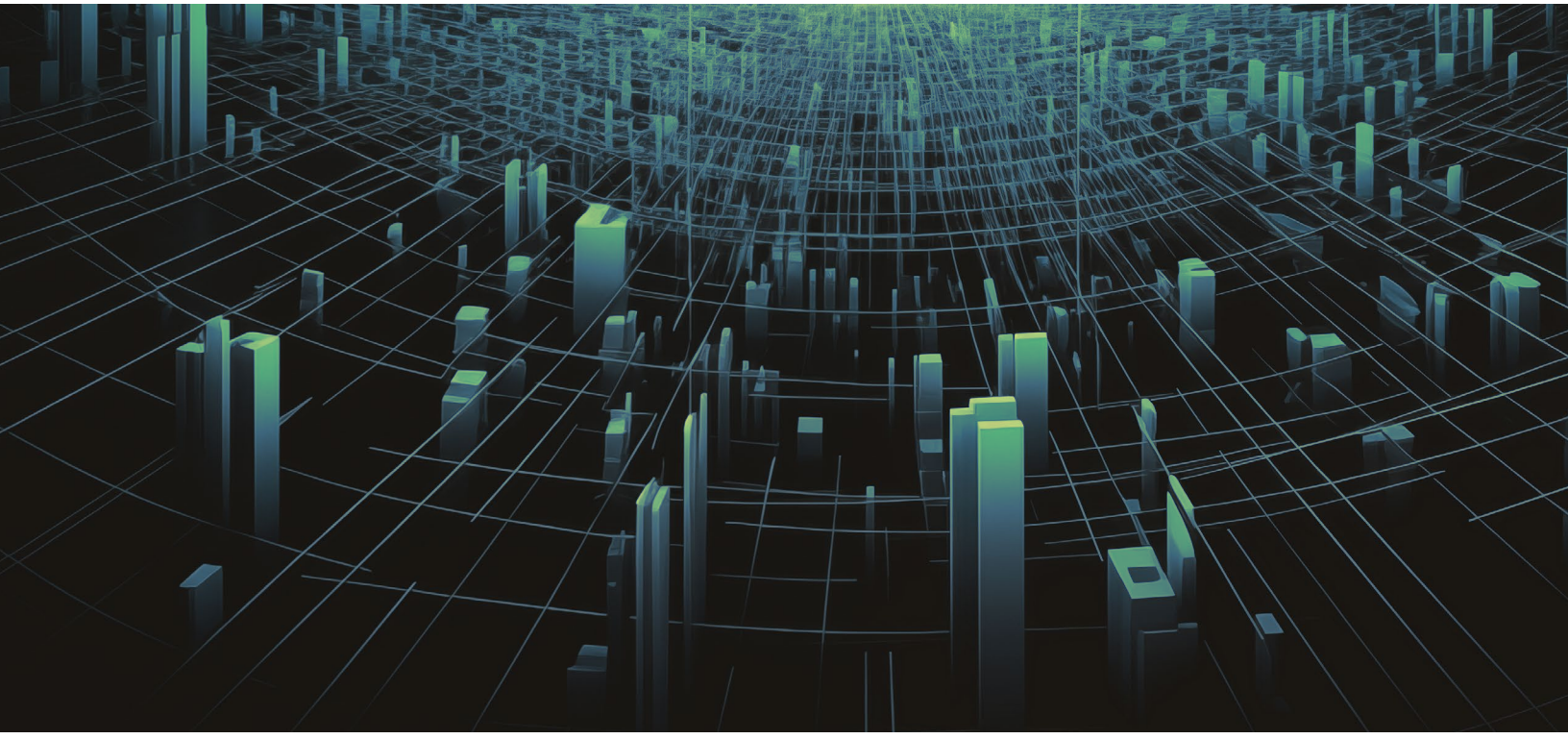
that this setup is *complex* and *fragile*. For example, the *2018 GitHub outage* is directly linked to a transient failure of a router (lasting less than a minute) that brought down GitHub for over a day.

The underlying cause was improper handling of failure scenarios when using a relational database in a distributed environment. The key issue is that relational databases were never designed from the outset for distributed environments and deploying them successfully is a complicated chore that is easy to get wrong.

Document databases are universally meant to be used in a distributed environment. As such, scenarios such as scaling, load balancing and automatic failover are part of the core feature set of the database.

RavenDB follows the model of active-active nodes. In a cluster of servers, any node in the cluster may accept a write, although usually a single node will be designated as the primary one. That ensures that as long as at least one node is operating, your system can continue chugging along. During normal operations, you can require reads to go from the primary node, from a secondary node, or from the nearest node to the client.

For failure modes, RavenDB follows the adage that everything should just work. A node going down shouldn't take down your system. Nor should you require any special configuration or setup to guarantee high availability. Merely setting up a RavenDB cluster ensures automatic failover, with clients and servers cooperating seamlessly to maintain the functionality of your system.



The Triumph of Transactions

Earlier in this article, I mentioned that one of the things that early non-relational databases gave up on was transactions. I cannot truly express how horrified I was when I heard that this was the case.

Transactions and ACID principles are the foundation for maintaining consistency in databases. RavenDB, as a non-relational database, sets itself apart by providing full transaction support from its very inception.

Recent years have shown how important that is, as the vast majority of non-relational databases are now supporting transactions. However, there is a distinct difference in the level of support between different products. For instance, while MongoDB and Couchbase support transactions, enabling them requires a case-by-case basis and comes at a high cost.

RavenDB, on the other hand, is always transactional and is performant enough to compete with other databases at their best without compromising your data.

How does RavenDB compare to relational databases, with regards to transactions? It turns out that there is a very important difference in the manner in which relational databases and RavenDB implement transactions. A relational database using SQL needs to process each operation in the transaction individually, usually confirming each command's success or failure to the client before accepting the next command in the transaction.

RavenDB, on the other hand, uses a different protocol. The entire set of commands is sent to the data-

base in a single unit, which gives RavenDB a major advantage in performance, latency, and optimization opportunities. Instead of going back and forth between client and server, a single remote call is sufficient to process the whole transaction.

Having the entire transaction to process in one go also has another impact on RavenDB's design. It means that the database engine doesn't need to manage locks at the same level as a relational database would. Lock management may consume *over 30% of the total time* a relational database spends processing requests. RavenDB can eliminate all of that cost entirely, without compromising on safety.

Dealing with Dynamic Data

Relational databases operate on top of a fixed schema, which is great if your model is well-known and doesn't change. In practice, you often have to evolve your model over time and frequently have to deal with user-defined and dynamic data. Schema evolution in a relational database is a complex and delicate process, enough to have *whole books written about it.*

Document databases are schemaless, meaning that you don't need to take special actions to evolve your model. Working with user-defined and dynamic data is a breeze (including querying on dynamic data, traditionally a complex and slow process).

Document databases require more overhead than relational databases on a per-record basis (since they cannot assume a schema and need to keep metadata per record instead of per table). That can usually be mitigated using compression usage.

The fact that each document in the database is standalone means that you can also evolve your documents over time. You don't need to have a big bang moment where you convert all your data to a new format. This is particularly important if you subscribe to agile methods and wish to continuously make small changes over time rather than be forced into what is effectively a paradigm shift whenever you need to update the structure of your entities.

The mere fact that adding or removing a field is not a painful process means that your development team doesn't have to fight to make changes to your application, leading to rapid development and improvement.

In RavenDB, the database can derive a compression dictionary from the actual data you are storing. This means that the database can compress individual documents very well (effectively removing the per document metadata costs) while retaining the ability to work with schemaless format.

Compression is usually seen as costly because of the additional computation required, but it turns out to be a net positive in the real world. The reduction in I/O costs achieved from the 50% - 70% storage reduction that documents compression brings to the table more than compensates for the compression costs.

Documents Diversification

Documents are a really good fit for business applications. This is mostly because you can usually draw a 1:1 mapping between the actual forms and documents that run the business and your entities in the database model. This alignment of the way the

business thinks about its processes and the way the database stores them has a lot of value.

Documents aren't the only thing you'll deal with in most applications, and document databases aren't limited to *just* JSON data. Binary data, such as images, Office documents, and audio and video files are all data that you need to commonly handle.

Storing them as documents would be suboptimal. Fortunately, you aren't forced to do that; you can store binary data and files directly in a document database. In MongoDB, that is called GridFS, while RavenDB allows you to use Attachments, which are binary data that is attached to documents and stored alongside them. That gives you an efficient way to deal with binary data.

Note that attachments are stored as-is, without compression. Most common file types already include compression, so trying to compress already compressed files is a waste. If you are attempting to store uncompressed binary data, compressing it first is recommended.

Beyond handling binary attachments, RavenDB supports several other data types that provide specialized storage solutions tailored to specific needs.

Chief among them is support for storing time series data natively. Let's assume that the vehicle rental agency has a GPS tracker on all their vehicles. It is important for the agency to track where each vehicle was at all times. Every 5 seconds, it will ping the server with the vehicle ID, the current time, and the GPS coordinates. That can grow very large over the lifetime of a vehicle, and isn't very well suited for storage in document format.

RavenDB supports storing such time series data directly and efficiently, allowing you to manage billions of data points, aggregate and query over them.

Revisionist History

Your application needs a system of record where all data, operations, and actions are recorded. That is the role of the database, of course. Going beyond the obvious portion of storing and retrieving data, there is also a host of desirable ancillary functions.

One of them is the need to track changes in your data over time. That can be because you would like to see your documents as they used to be in the past or because of strict regulations. Many fields, such as healthcare or finance, have audit requirements that you must meet.

RavenDB allows you to define a versioning scheme in which the database keeps track of old revisions of your documents as they change. That gives you an immutable audit trail of your changes. Even the database administrator cannot modify those revisions, they are truly immutable.

The revisions feature is useful for audit trails but also simply to see what has changed in a document over time and can be quite a boon for complex business scenarios.

Quicker Queries & Indexing Insights

So far, we have dealt primarily with how you model your data, perform transactions, and store and retrieve data. That is only a part, and usually not the interesting part, of using a database.

Once your data resides in the database, you are usually interested in querying it, often in all sorts of interesting ways. Here, for a very long time, relational databases held an absolute primacy.

The original buzzword for non-relational databases was: NoSQL databases. The idea was to drop not just the relational model but also the querying language that came with it. That proved to be a serious issue down the road.

Today, aside from MongoDB, I cannot think of a single database that *doesn't* have a SQL-like querying language. The relational model is limiting, but the ability to express your queries in a structured manner, regardless of how the data is actually stored, is a huge benefit to users.

One of the major innovations of the relational model was the break between how the data is stored and how it is queried. Luckily, the querying language and the underlying model are not tied together, and it is entirely possible to use a SQL-like query language on a non-relational database.

In the case of RavenDB, for example, it supports querying using RQL (Raven Query Language). This language gives you both the flexibility and readability of SQL and, at the same time, is tailored to work well on JSON documents. Native support for documents makes querying your data a breeze.

Another aspect of querying is the manner in which the database is able to find the relevant data. Usually, if you just issue a query to the database, the query engine needs to scan through all the records in the system to find the matches. As you can imagine, that is not a good solution when you have a serious amount of data.

To solve that issue, databases utilize indexes. Those allow you to find the relevant data quickly. The issue is that you now need to not only manage your data model, but also your indexes. Anyone who has experienced database performance problems has a story of how adding a simple index dropped a query latency from minutes or hours to milliseconds.

Relational databases require that you define indexes, and then the database will attempt to use the relevant indexes to speed up your queries. That usually works, but sometimes the database engine decides to use a different index (or not use one at all), leading to what is effectively a black art of database optimizations. And all of that requires that you have defined the appropriate set of indexes in the first place, which isn't an obvious or easy task.

RavenDB, on the other hand, does not make assumptions about indexes that have been deployed. In fact, one of the core tenets that RavenDB subscribes to is that it should do the Right Thing. When you issue a query with RavenDB, it isn't going to scan through the entire dataset. It will always use an index to give you the results *fast*.

What if there is no matching index? The query optimizer will go ahead and create one for you!

The cost is roughly the same between creating an index and scanning the entire dataset, but the next time you issue this query or a similar one, the index

will already exist, and you'll be able to get the results immediately.

Furthermore, the RavenDB query optimizer is smart enough to create the appropriate set of indexes for you, simply based on the queries that you issue. And as your query patterns change over time, it will adjust itself to the optimal configuration for your needs.

Instead of spending a lot of time and effort trying to get the right indexing strategy, you can simply ignore the entire topic since RavenDB will take care of that for you.

RavenDB even supports creating indexes for you in advanced scenarios, such as geospatial queries, full-text searches, or aggregations.

You can always define your own indexes explicitly, but what is important is that you don't need to shell these features out to another platform (for example, replicating data to ElasticSearch for full-text search capabilities) as they can be handled natively and directly in RavenDB.

Beyond Babysitter Duty

Relational databases were created in an era where computers cost more than a house, were operated by highly paid experts, and were temperamental and delicate machines. That had a huge impact on the way they are structured, deployed, and expected to be operated.

A relational database is usually meant to be used by an expert, with a dedicated staff on call to handle any issues that may arise. I talked about this previously when discussing indexing. Relational databases assume that a database administrator with deep insight into the applications and systems that utilize the database is at the helm. If you don't have the dedicated staff, or they don't have complete insight into everything that is going on (an almost impossible task), you aren't likely to get the most out of your database.

RavenDB, on the other hand, was born in a very different era. It runs without a lot of guidance, often with none at all, and is able to provide equivalent or better performance and capabilities than a relational database with staff taking care of it on a full-time basis.

There are dozens of features inside of RavenDB that aim to allow you to operate it in unattended mode. It will adjust itself automatically to changing conditions, such as new versions being deployed, shifts in user patterns, etc.

Running RavenDB in a cluster is another scenario where you can truly appreciate the generational gap from relational databases. Setting up a cluster, ensuring high availability, load distribution, and optimizing your application performance are complicated and time-consuming for relational databases.

For RavenDB, that is just the way things are. There is no complexity to deal with since the database assumes that it is the one that needs to encapsulate all of it, leaving you free to add business value to your systems.

Database Delight: A Sweet Summary

RavenDB and relational databases both allow you to store and retrieve data and on the initial look, they behave very similarly. Both systems have transac-

tions, can be dynamically queried using a dedicated querying language, and do well for business applications and Online Transaction Processing (OLTP).

RavenDB has a more flexible modeling approach, which is more suitable for modern applications, both in terms of allowing you to change in an agile fashion over time and dealing with dynamic and user-generated data. Relational databases turn modifying your schema into a complicated chore and dynamic data into a big challenge.

Features such as automatic indexing and self-tuning mean that you can set up RavenDB very quickly and, in general, forget that it even exists. It will hum along and do its work, letting you do yours. Relational databases, however, require frequent hand-holding and ongoing care and maintenance by qualified staff.

When working in a distributed environment, the choice of your database plays a crucial role. Using a relational database can introduce complexities that may impact the overall efficiency of your solution.

In contrast, RavenDb was purposefully crafted for distributed scenarios, which is evident in its operational simplicity and overall ease of use.

In conclusion, RavenDB stands out as an excellent choice in the dynamic landscape of modern applications. Offering seamless scalability, robust transactional features, and freedom from the complexities that often burden traditional relational databases, RavenDB is a powerful ally in the realm of distributed environments. Its design empowers you to navigate the challenges of a data-driven world with agility and confidence.

For those in search of a reliable, adaptable, and user-friendly foundation for their applications, RavenDB proves to be a steadfast and future-ready choice.

I hope that this article has been helpful to you. We would welcome the opportunity to discuss this further with you on our *GitHub Community*. If you have any questions, feel free to contact us at *support@ravendb.net*.

About RavenDB

RavenDB is a pioneer in NoSQL database technology with over 2 million downloads and thousands of customers from startups to Fortune 100 Large Enterprises.

Mentioned in both Gartner and Forrester research, over 1,000 businesses use RavenDB for IoT, Big Data, Microservices Architecture, fast performance, a distributed data network, and everything you need to support a modern application stack for today's user.

For more information please visit

ravendb.net

Contact us at

info@ravendb.net

Documentation
https://ravendb.net/learn/docs-guide

Use Cases
https://ravendb.net/news/use-cases

Free Online Training
https://ravendb.net/learn/bootcamp

Webinars
https://ravendb.net/learn/webinars

RavenDB Download
https://ravendb.net/download

RavenDB Cloud Database as a Service
https://cloud.ravendb.net/


RavenDB