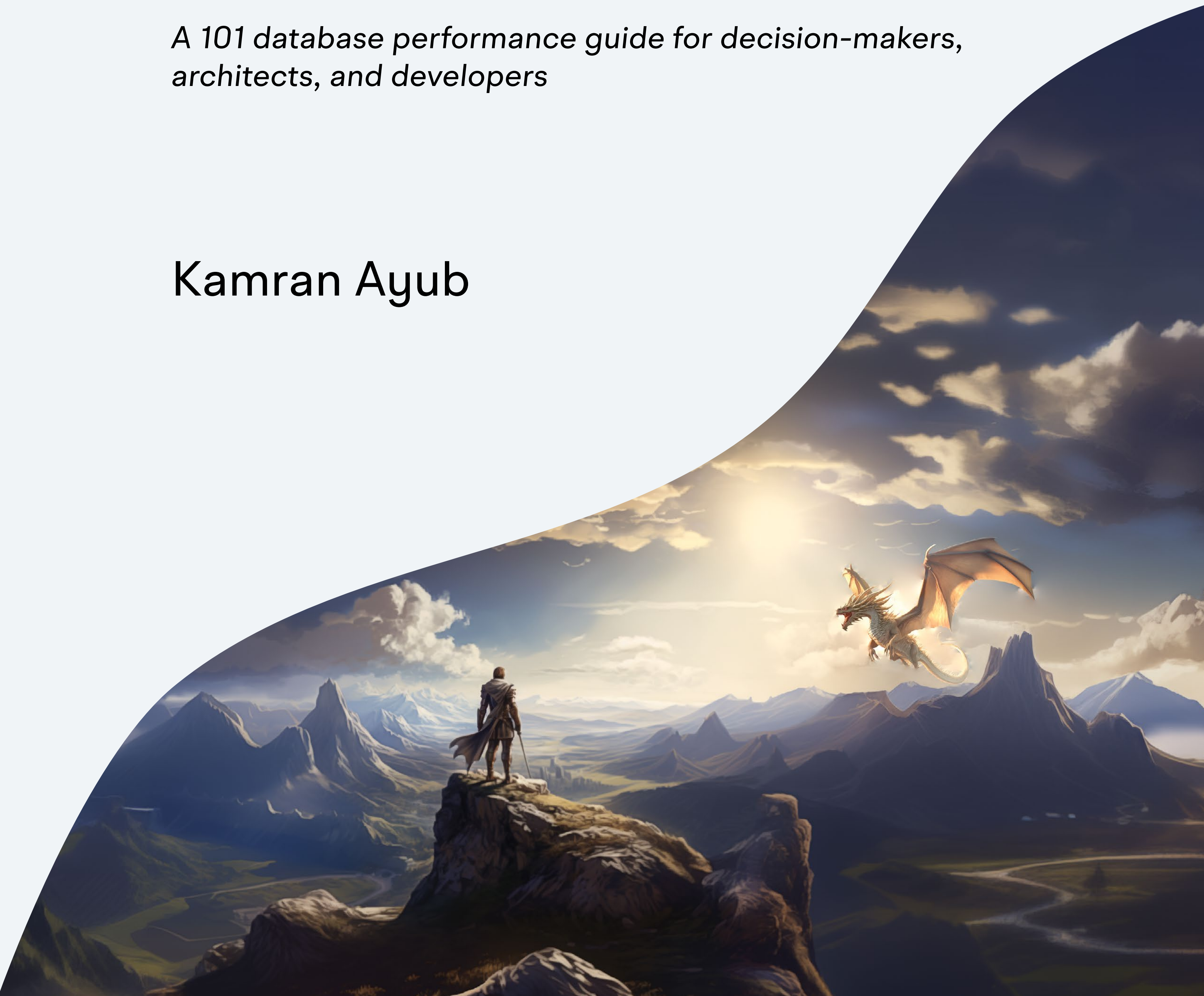
RavenDB

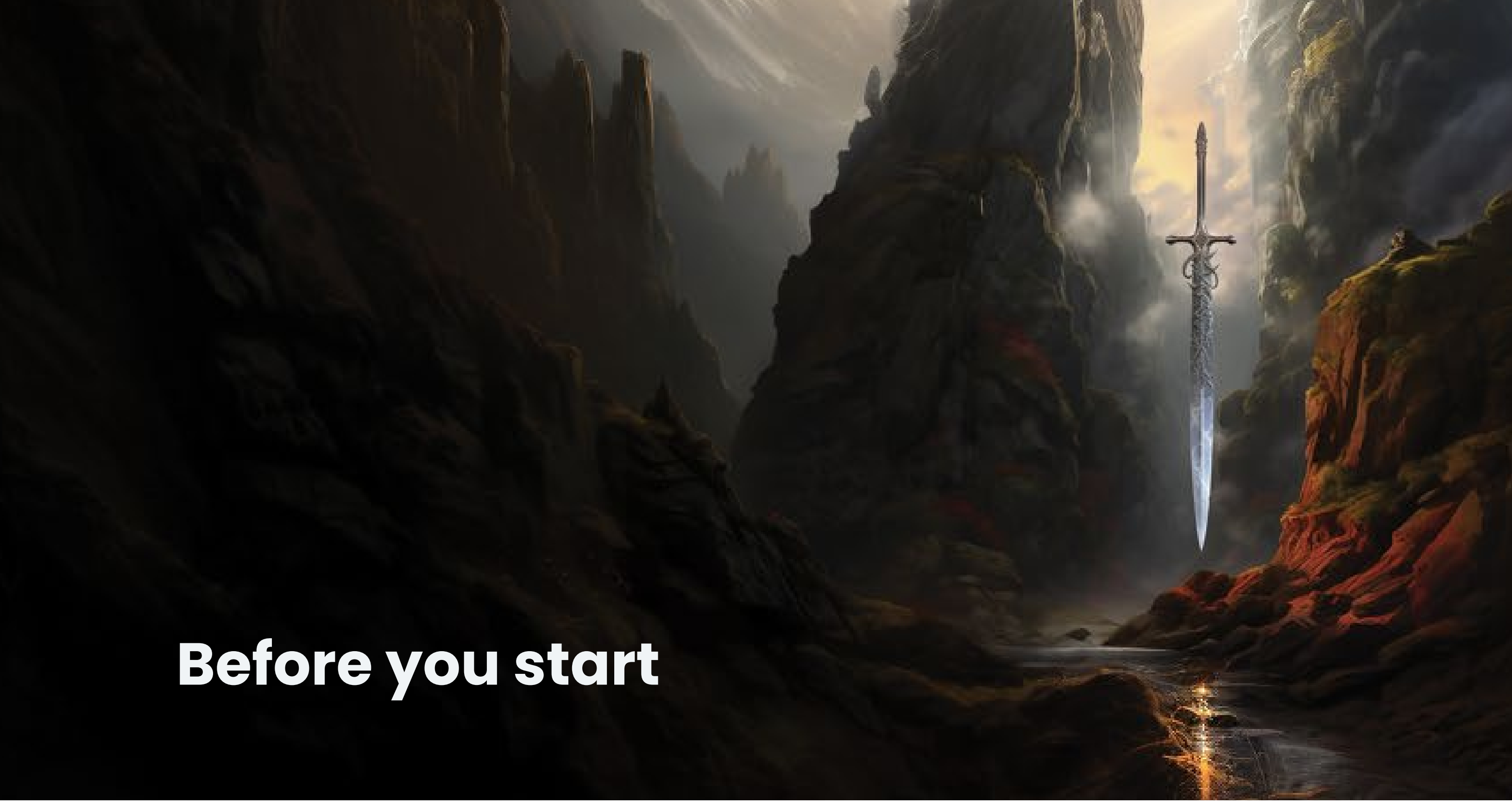
Slaying Database Performance Dragons

A 101 database performance guide for decision-makers, architects, and developers

Kamran Ayub



Contents



Before you start

If you've found yourself at the mercy of database lag, or if you're planning a new project and want to avoid the pitfalls of poor performance, you've picked up the right guide. At RavenDB, we've helped customers ranging from startups to Fortune 500 companies successfully design distributed data architecture and wrangle performance dragons for over a decade.

What to expect

In this 20-minute guide, we'll distill some of the essential knowledge you need to design a responsive data architecture—from understanding the types of dragons (databases) you might encounter to mastering your weapons (query optimization) and shields (indexing). Along the way, we'll introduce you to the magical properties of RavenDB, which provides unique capabilities to keep your databases running efficiently. We've sprinkled in real-world examples and practical advice to make the abstract tangible.

Takeaways

After reading this guide, you'll know how to:

- Identify the strengths and weaknesses of different database technologies
- Design and optimize queries for speed and efficiency
- Understand the crucial role indexing plays (and where it can fall short)
- Be able to make informed decisions when evaluating database vendor performance.

Who is this book for?

This guide is for decision-makers, developers, database administrators, and anyone interested in database performance optimization.

DRAGON WRANGLER TIP:

Look for these Dragon Wrangler Tips – it's where we call out gotchas to pay attention to, or howRavenDB differs from other vendors with our unique take.

Know Thy Dragons: Understanding Database Types and Their Unique Performance Implications

Welcome, brave knight, to the fantastical world of database dragons! Just like different dragons have varying strengths, weaknesses, and magical abilities, different databases have their unique performance implications. Understanding these dragons—or databases, if you prefer—is critical to becoming the dragon wrangler your kingdom (or company) needs.

Types of databases

As Big Data became increasingly prevalent, it exposed the limitations of traditional relational databases, which were initially designed for structured data and single-server deployments. These systems needed help to meet the demands for high throughput and low latency in distributed data-intensive environments. In response, anew generation of databases emerged, commonly known as NoSQL and NewSQL, to better accommodate the challenges of scale, distribution, and varying data structures.

- **Relational Databases (RDBMS):**

Think of these like the classic European dragons, massive creatures that hoard treasure. They're the traditional option and have been around for ages. They're reliable but often demand a lot in terms of resources. Common members of this family include MySQL, PostgreSQL, and SQL Server.

- **NewSQL Databases:**

Newly emerging dragons typically designed for niche use cases. Imagine a dragon that can breathe both fire and ice. They support horizontal scaling and schemaless data models like NoSQL databases but distinguish themselves by supporting fixed schemas and SQL queries like their relational counterparts. Common members of this family include NuoDB, VoltDB, Google Spanner, and Clustrix.

- **Platforms (Backend-as-a-Service):**

More like a multi-headed dragon, backend-as-a-service platforms offer a full stack solution to build a backend, including data storage, serverless functions, authentication,hosting, and frontend SDKs. While they simplify and speed up building backend solutions, they require going "all in" on the provider, which may not fit larger organizations. Common members of this family include Firebase, Supabase, AWS Amplify, and Appwrite.

- **NoSQL Databases:**

These are the nimble, more exotic dragons. They can be different types, including document-based, key-value stores, graphs, time series, and more. They're designed for flexibility and horizontal scaling by doing away with the rigidly fixed data schemas of their relational counterparts in favor of schemaless storage. Common members of this family include RavenDB, MongoDB, Redis,InfluxDB, FaunaDB, and Neo4j.

With the recent shift to cloud-native architecture, increasingly cheaper hardware, and an explosion of data needs with ML/AI, there is less and less distinction between these categories. Modern databases are engineered to excel in distributed data architectures, offering more effortless horizontal scalability than traditional vertically-scaling databases.

Additionally, modern databases are tailored to meet the needs of today's developers by offering a more streamlined coding and developer experience. This is often achieved through integrating web technologies like JavaScript, JSON, REST, and GraphQL, making it simpler to model and integrate the application layer.

DRAGON WRANGLER TIP:

NoSQL doesn't necessarily mean "no relationships." There is a variant of "document-relational"database dragons that are schemaless but still offer ways to query relationships.

For example, RavenDB supports loading referenced documents, which effectively allows modeling aSQL JOIN. While this may require more resources at indexing time, it doesn't affect query performancelike in an RDMS – we'll cover why that is soon.

Choosing a data modeling approach

Your data modeling approach is like your battle strategy against these dragons.

- **Normalized:**

This is the old-school, armor-and-sword method. It's effective but can be cumbersome. It's the main approach used in SQL databases.

- **Denormalized:**

Think of this as using a bow and arrow from a distance. It's more flexible and quicker but requires more discipline. You'll see this approach often in NoSQL databases.

Your choice of data modeling can significantly impact database speed. In an e-commerce setting, SQL databases require multi-table joins to fetch a customer's order details, slowing down queries. In contrast,NoSQL databases like RavenDB can store an entire order as a single document, enabling an ultra-fast O(1)lookup.

Don't fear denormalization

A common refrain against schemaless databases and denormalization is that you lose "referential integrity"meaning that data updated in one place does not "stay in sync" with other places it's referenced. This tends to scare people off, and you're here to overcome dragons, not be scared by them, so let's examine the two main motivations behind denormalizing data.

One motivation for denormalization is to store point-in-time data. Take the classic customer order example: a customer purchases a product, and that gets stored as an order detail line item. In this case, referential integrity isn't needed because you want to store the purchase price as a point in time value. It's"denormalized," but that's by design and an important distinction for the business because the intent of storing the price differs (price at purchase vs. current price).

The other motivation is to denormalize reference data for performance reasons – to reduce JOINs, for example,in a complex query. If you wanted to store the books someone reads in a list, in denormalized form, you would store some book details like the name and author. This is cloning reference data to speed up displaying the book list to the end-user.

However, you see the problem immediately. If the author changes their name, such as to match their spouse, you need to go and update every place that the author's name was referenced,making business logic more complicated.

Denormalizing reference data is where extra discipline is required. For any denormalized reference data, you need to ask how often (if at all) the referenced data will update–and what you will do to take action if that happens.

DRAGON WRANGLER TIP:

*When choosing a schemaless database, seek out functionalities that overcome the limitations ofdenormalization. RavenDB helps you avoid this pattern in the first place by letting you **modeldocument relationships**.*

*It also provides features like **Patching**, **Document Revisions**, **DocumentChanges API**, and **Data Subscriptions** to make writing robust data handling logic easier.*

Dragon decision matrix: are you battling the right one?

Before facing your database performance dragon, you need to know its nature – where are its weak spots? Do some weapons work better than others? Database performance is dictated by workload because each has unique advantages and abilities that might increase performance.

For example, choosing PostgreSQL for a high-velocity IoT workload would perform worse than a database designed for time series data ingestion. It would be a recipe to get devoured.

There are **3 main workloads** in the database world:

- **OLAP** (Online Analytical Processing)
- **OLTP** (Online Transaction Processing)
- **Time Series**

Here's a breakdown:

Workload Type	Best For	Strengths	Weaknesses	Use-Case Example
OLAP (Analytics)	Complex queries and data analysis	Efficient in handling large data sets and aggregating information quickly.	Not designed for high-velocity transactional data.	Running sales reports that aggregate data across multiple dimensions (time, geography, etc.)
OLTP (Transactional)	High-velocity transactional data	Fast reads and writes, beneficial for applications requiring real-time data access.	Generally not designed for complex queries that scan large volumes of data.	Stock trading applications where millisecond latency can make a significant difference.
Time Series	Tracking, storing, analyzing time-ordered data	Optimized for inserting and querying data in a time-based order.	Typically not suited for transactional data unrelated to time series.	IoT sensor data tracking for temperature changes over time.

Fig. 1 Database workload comparison

To aid you in deciding which dragon to face, here's a database workload decision matrix that can help you determine which SLAs may be important for you:

	Workload	Read/Write Ratio	Size of 1 Data Item	Important SLAs
IoT / Industry 4.0	Time Series	Write Heavy (strong)	Small (1-5kb)	High Throughput High Scalability
Real-Time Analytics	OLAP	Read Heavy (strong)	Very Large (>50kb)	Low Read Latency
Product / Content Data	OLAP	Read Heavy (slightly)	Large (20-50kb)	High Availability High Consistency

	Workload	Read/Write Ratio	Size of 1 Data Item	Important SLAs
eCommerce	OLTP	Balanced	Medium (5-20kb)	High Availability Low Read Latency High Consistency
Payments	OLTP / Time-Series	Write Heavy (slightly)	Medium (5-20kb)	High Throughput High Consistency High Availability
Gaming / Mobile	OLTP	Balanced	Medium (5-20kb)	Low Read Latency Low Write Latency

Fig. 2 Database workload SLA decision matrix (Source: BenchANT)

There's no one-size-fits-all approach when it comes to deciding on a database that will handle your workload (although we think we come pretty close).

Key Questions to Anticipate Performance Dragons

How do you identify which databases might have hidden performance dragons for your use case? Instead of waiting to hear them roar, read their spec sheets–then dig in further to the documentation and reported issues (for practical problems).

Here are 17 key questions and follow-ups we recommend asking:

1. Is the database ACID (Atomicity, Consistency, Isolation, Durability) compliant? Are there cases where there is no ACID guarantee? Which parts of our workload will ACID be critical, and are those covered?
2. What are the consistency guarantees during a failover? Will we need to introduce retry logic?
3. How does the database support advanced or complex queries, such as aggregations or geospatial searches? How does it ensure they stay performant?
4. How much logic does it push to the application vs. taking on itself? Will it make our applications more or less complex?
5. What is the failover behavior in large clusters? How do node elections take place?
6. How does it maintain business continuity in the event of a catastrophic failure? What will any downtime mean for our business?
7. What is the production hardware sizing, cluster layout, and recommendations? Can we afford that?
8. How early do we have to prepare for sharding? How does sharding affect developer experience? How costly are sharding mistakes?
9. Does it support our key integrations? What will we need to build custom integrations for our needs?
10. What mechanisms are in place to prevent race conditions or corrupted data from being written?

11. How does it integrate with DevOps workflows? Do we need custom code to migrate data?

12. What observability hooks does it provide? Is it easy to monitor?

13. What are the day-to-day operational needs? Do we need to hire a team, or is there a managed provider? Do we need a hybrid hosting architecture?

14. How will we migrate our existing data? Does the vendor offer any migration assistance or tools? How much rework will this cause?

15. What is the ecosystem around the product? Is there a community of practice around it?

16. What are the production licensing requirements? Is there an open source license?

17. What levels of production support are offered? What are the SLAs?

DRAGON WRANGLER TIP:

*To really know which database best fits your workload, create a proof-of-concept comparing your key workloads in an isolated environment. We even offer **a free 2-day POC program** for startups where our engineers will dedicate their time to helping you evaluate*

Ask your architects, or even the vendors, about what performance tuning options are available.

By now, you should have a foundational understanding of the database dragons you're facing and be better equipped to deal with them.

Dragon Battle Strategies: Optimizing Queries for Maximum Velocity

Now that you know what type (or types) of dragons you're dealing with, it's time to refine your technique. Different databases are built differently and will benefit from different optimization techniques but... dragons are dragons, and most benefit from the same strategies.

Avoid Poisonous JOINs

In the perilous realm of database queries, JOINs are akin to a dragon's slow-acting poison. While they may seem innocent, the more tables you JOIN in a relational database, the more sluggish your query becomes. The database needs to meld rows together based on shared attributes, and this can significantly reduce performance.

As an antidote, ensure indexes are created for joined columns, or use specialized column types like JSON. Consider alternatives like changing the data model, data denormalization, or switching to document or key/value storage models. Some relational databases such as SQL Server and Postgres allow you to write *"materialized views"* which persist query data and can be refreshed on an interval. Document databases such as RavenDB allows you to store all related data in a single location, avoiding the need to JOIN to other locations entirely.

Load Data Ahead of Time

A common problem is performing many separate queries when querying data from a database (called an "N+1 select" problem). This is when you query for a list of items and perform another query for additional information per each item in the list. This is like facing a dragon armed with nothing but a butter knife. Sure, you might eventually get the job done, but it will take far longer and be much more perilous than if you had the right weapon in the first place. This is most commonly encountered when using Object Relational Mappers (ORMs) like Hibernate or Entity Framework. Your database has to make multiple round trips for data, hampering performance.

To avoid this trap, preload data using techniques like eager loading, caching, or batch loading. This is often specific to the ORM being used, such as the *"Include"* method in Entity Framework. On the database side, utilizing materialized views or stored procedures and using field projections to limit the data retrieved can also optimize performance. For distributed internet-scale-scale applications, many companies add a caching persistence layer in-between applications and the database with technologies like Memcached or Redis. This speeds up lookups for commonly used data but adds significant operational overhead.

Your API design is also paramount in this case. Common REST based interfaces are particularly prone to the "N+1 select" issue, since they are hard to compose well. GraphQL, on the other hand, gives you far better performance because it is meant to allow you to compose the data at the server level, instead of continuously going back and forth to the database.

Pay Attention to Query Plans

Entering a dragon's lair without a plan is a recipe for disaster. Similarly, in the database world, inefficient query execution plans can slow down performance considerably. This can happen due to outdated cached plans, improper statistics, or suboptimal query plans.

To counteract this, employ tools like SQL Server Query Store, Execution Plan Analyzer, or SQL Server Profiler. Many databases also provide commands like EXPLAIN and ANALYZE to help you fine-tune your query plans. Those tools usually require a dedicated Wizard to manage on an ongoing basis.

RavenDB is a lizard of a different color, instead of relying on the head wizard for query optimization, the RavenDB Query Optimizer is able to not only produce efficient query plans but to also change the very battlefield for its advantage. Other databases require that you'll have a full suit of armor and understand the entire state of your application to create the appropriate set of indexes. RavenDB is able to read your queries and produce the required indexes (or modify existing ones), leading to optimal query performance and continual improvement without ongoing maintenance.

Similar to the difference between Western and Chinese dragons, indexes in RavenDB compared to other databases are vastly different. An index in SQL Server, for example, is able to serve only a particular set of queries, while another index is required for even slightly different questions. RavenDB is able to serve many different queries from the same index, due to vastly different internal architecture.

Offload Logic to the Database

Trying to find specific data in a non-indexed column is like searching for a magical gem in a dragon's massive hoard—virtually impossible. SQL views, MongoDB's aggregation pipeline, and server-side functions can act like magical maps that quickly point to what you want. This has historically been hard for application developers to accomplish, as traditional relational databases require you to write in specialized languages or keep database logic separate from your codebase.

The landscape is changing now with modern databases like RavenDB allowing developers to write server-side logic in familiar, higher-level languages like C# and JavaScript that can be defined (and deployed) alongside application logic. This makes the developer experience more accessible and reduces operational complexity.

DRAGON WRANGLER TIP:

Modern data architecture usually incorporates some messaging, event sourcing, or queuing infrastructure like Kafka or RabbitMQ. This allows organizations to unify systems together in a distributed environment and make data handling more robust. Modern databases support ETL (Extract, Transform, Load) processes that make it easier to integrate into existing environments. Enterprise RavenDB customers can use it as a native message sink for Kafka and RabbitMQ, as well as push data to SQL, OLAP, Postgres, Power BI, message brokers, or Elasticsearch automatically.

Design a Pragmatic Data Model

With schemaless databases, there are typically no constraints on how data is stored which is both a blessing and a curse. More thought needs to be put into how to model your data domain. Poor data modeling decisions can result in slow queries, inefficient data retrieval, and increased disk usage.

In relational databases, there is a tendency (or an expectation) to be strict with normalization. However, normalization makes queries more complex, and more complex queries take (significantly) more time to execute. In your quest to optimize query performance, you will likely need to introduce denormalization into your data model, especially when choosing a schemaless database. Don't fear it – embrace it. If your data model is not designed for the patterns of data access you require, performance will suffer no matter what database you end up choosing.

With RavenDB, you have the choice to adapt your data model over time with little trouble. You can keep your old data in the previous format or ask RavenDB to convert it. There is no need for complex migrations processes or the usual costs associated with realizing that you need to update your model.

You Probably Don't Need Sharding

Many NoSQL databases allow you to partition data into "shards" to support massive databases. Unfortunately, we see customers all too often reach for sharding before it's really needed – making things overly complicated before it's warranted. For example, MongoDB databases are typically sharded after reaching over *2TB in size*, whereas RavenDB lets you scale to 10TB before you even need to think about sharding.

Sharding is powerful but complex. Shards can grow independently, which means over time, they may need to be "rebalanced". Rebalancing is almost always a manual operation and, in the worst case, will require scheduled downtime. In practice, sharding is easy to get wrong initially, and sometimes applications need to be rewritten or updated due to early mistakes.

The worst aspect of sharding is that it front loads all your costs. You have to accept the additional complexity and costs of sharding at the very start of your system, while you reap the benefits only when the system is extremely large. That is a lot of time to pay with additional complexity until you can see the results.

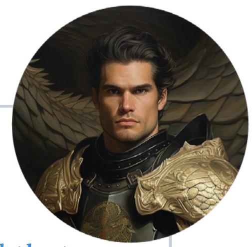
DRAGON WRANGLER TIP:

*One of RavenDB's core design principles is "safe by default" to maintain business continuity. That's why we tried to remove as much complexity from **sharding** as possible. You can start without sharding, introduce it later on, and your application code can remain unchanged, it's transparent by default. While rebalancing is manual, it doesn't incur any downtime. And if you make any mistakes in your sharding approach, RavenDB handles these "bad sharding" scenarios without blowing up, saving you from having to perform expensive migrations.*

Choose Transaction Scope Wisely

Picture a band of knights waiting their turn to strike a dragon—one at a time. This scenario mimics the potential pitfalls of transactions in relational databases, where concurrent transactions can result in locks and blocks, causing performance issues.

To avoid this bottleneck, optimize your transactions and isolation levels. Techniques like row-level locking, shorter transaction durations, and appropriate isolation levels can help you and your knights strike in unison. As you can guess, this is complex and requires very careful coordination.



DRAGON WRANGLER TIP:

*With RavenDB you can avoid all that. The **Voron storage engine** uses a Write-Ahead Journal with a single write thread to avoid locking and blocking issues. Transactions are automatically scoped to the node being written to, and you have the option to perform distributed transactions across a cluster for critical data. RavenDB will also merge concurrent transactions internally (while keeping their ACID properties, of course) to optimize your performance.*

Be Wary of Unbounded Data Sets

Unbounded data, like your Amazon order history, can wreak havoc on both writing to and reading from a database. With NoSQL databases, you may be storing data that contains unbounded lists in a single document or item. Unbounded data grows over time (such as a customer's order history), which leads to large item sizes. Storing large items in a database can impact performance due to slower read and write operations, increased storage costs, and higher network utilization.

When modeling unbounded sets in schemaless databases, consider storing pages of items with range-based keys (e.g. "Customer/A-0035/OrderHistoryItems/1," "Customer/0035-A/OrderHistoryItems/2"). This way, you can take advantage of loading pages of items in the set at once yet keep individual item sizes low. Additionally, use paging or limiting operators (like LIMIT or TOP in SQL) on the application side to prevent memory problems, increased network latency, or slow queries due to large result sets.

Keep Storage Bloat in Check

Like a dragon continually hoarding treasure, most databases grow bloated over time. Fragmentation and document bloat can lead to decreased performance, increased disk usage, and slower write operations.

Some databases require you to configure and perform regular vacuuming to mitigate table bloat and fragmentation. Set up auto vacuuming correctly and consider manual vacuuming based on workload characteristics. Configure alerts for disk usage so you don't get caught by surprise.

Sometimes your data lasts forever, like diamonds, but quite a lot of data is only meaningful for a certain duration. Setting up expiration policies can dramatically reduce the amount of craft that you gather over time.

DRAGON WRANGLER TIP:

Check how much regular maintenance is expected for the databases you evaluate and what tools they provide to make it easier. RavenDB aims for unattended operation. It doesn't require regular maintenance by an administrator, and many janitorial tasks like "vacuuming" aren't needed because the storage of database files is managed internally for you.

RavenDB also allows you to register a document to expire at a certain time automatically, so you won't have to manually clean your warehouse whenever the "disk is nearly full' alert is blaring.

Don't Ignore Configuration and Resource Allocation

Imagine donning armor that's too heavy for battle against a swift dragon. In the database world, this is akin to under-provisioning resources like Azure CosmosDB's Request Units, leading to slow performance and timeouts. Hardware choices have important implications for database stability. For example, NVMe SSD storage will increase disk operations (IOPS) and outperform other types of storage but may require ephemeral disks that your database needs to tolerate losing.

Follow database vendor configuration guides and hardware recommendations to achieve optimal performance. Adjusting working memory size, shared buffers, cache size, max connections, threads/task queue lengths, and connection modes are common knobs to tweak for most databases that can have a profound impact.

DRAGON WRANGLER TIP:

Instead of guessing, relying on trial and error, or waiting until disaster strikes, look for database features that help you optimize resource allocation continuously. RavenDB self-monitors and surfaces any relevant action items like slow disk activity, request performance degradation, or heavy indexing through notifications in the Studio.

It's All in the Technique

Use this chapter as a checklist for your data architects, vendor evaluation, or as a way to get more mileage out of an existing solution. No matter what database you choose, each one has its unique performance dragons to tackle but at least we've covered the most common optimization techniques that will get you 80% of the way there. In the next chapter, we'll explore how indexing acts as a shield against dragon fire.

Shield of Indexing: Keeping Data Lookups Quick and Accurate

If executing a query is your sword swing, an index is your shield. It protects you from the dragon's fiery breath – wasted time and resources. A shield has a direct effect on the types of weapons you can bear – forget about swinging a two-handed sword when you have a tower shield. Similarly, not all databases treat indexes the same way, and you should know what the implications are on the query workloads you expect when comparing vendors.

Why Indexing Works

In essence, indexing is like adding bookmarks to a massive tome of spells (your data). Without bookmarks, you'd have to read the whole book just to find that one spell (data) you need. But with bookmarks, you know exactly where to look. Using different colors, shapes, or sizes to distinguish between groups of bookmarks will make lookups faster at a glance. Indexing in databases works similarly. It creates a data structure (typically a B+Tree) that improves the speed of data retrieval operations at the cost of additional disk and memory space.

It's easy to think that indexes solve most problems with query performance but they fall short in practice because we tend to ask complex questions about our data that most indexes aren't prepared to answer fast enough.

Big Oh-No: How Indexing Affects Query Performance

To understand indexes and query performance, we have to dig a little deeper. Let's talk about Big-O notation. It's a way to describe how well an algorithm performs. In layman's terms, if your data grows significantly, the database has to do more work to lookup data, so queries begin taking longer and longer.

Here's a visualization to illustrate how significant of a problem this is (Figure 3):

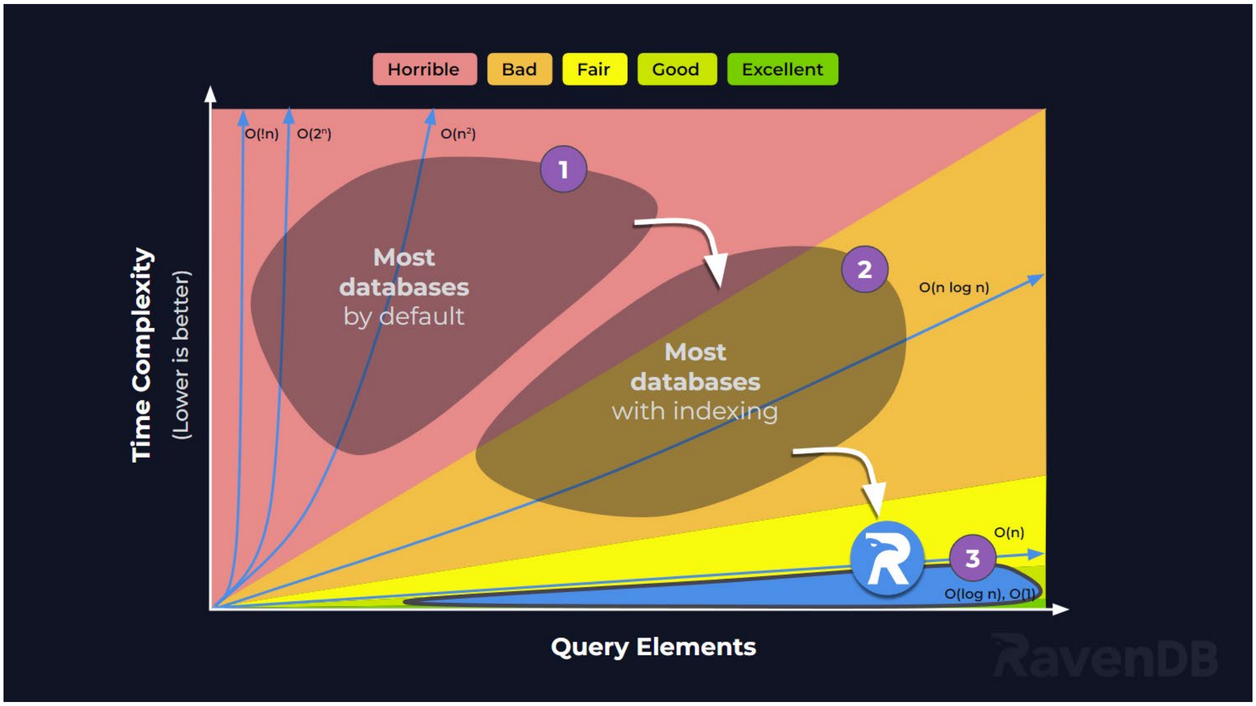


Fig. 3 Database query time complexity

"Query elements" refers to the amount of "parts" of a query, such as a filter expression (WHERE), aggregation (COUNT BY), or JOIN. So while simple queries across any database perform similarly (such as a lookup by key), more complex queries have more elements that can significantly increase the time it takes to compute a result if the database isn't optimized for them.

- Point 1: Out of the box, some databases could be as slow as O(n^2), O(2^n), or even O(!n), where each added piece of data drastically slows down the query (think "table scanning" or "N+1 issues").
- Point 2: Even after tuning with indexes, most traditional databases struggle to get below O(n log n) query complexity, meaning the query time still grows as a factor of your data size logarithmically.

Both these points translate to something very simple: most databases have a performance ceiling as you scale no matter what you do. The more complex your queries are and the more data you have to query across, the worse the problem becomes. This is why nearly any database will serve your needs at first. But as you scale, at a certain point the database becomes a bottleneck and starts to cause significant problems.

Exposing the Indexing Dragon in the Room

The implications of this can be widely seen in downstream application architecture decisions: logic is offloaded elsewhere to serverless functions or the application layer, and in many circumstances, organizations turn to event-sourcing or messaging architectures like Kafka or RabbitMQ. While these solutions do add a layer of consistency in processing data streams, they also introduce the need for custom integrations or non-native solutions. The end result is still a juggling act of components that should ideally work in harmony but often don't, increasing your total cost of ownership with additional maintenance, operating costs, and development overhead.

Compiling Queries Ahead of Time

The reality is that with most databases, you're penalized for trying to optimize data access. Luckily, RavenDB isn't like most databases. To address this issue, we need to fundamentally redefine indexing. In computer science, ahead-of-time (AOT) compilation compiles a higher-level language to a lower-level language at build time to save work at run time. With RavenDB, indexes are written in higher-level languages like C# and JavaScript, and then mapped or aggregated results are compiled to B+ data structures ahead of time in the background (what we call "indexing time"). This allows you to optimize your queries more than you ever could at application runtime. RavenDB indexes combine field-level indexing, aggregation pipelines, and stream processing into a custom indexing engine called Corax. Corax enables RavenDB to remove most runtime query complexity by allowing you to run complex data logic ahead of when you query it.

This approach leads to a dramatic reduction in runtime query complexity (Figure 3, Point 3). RavenDB queries never exceed O(N) complexity, translating to a performance boost of 800-3000% or beyond with internet-scale workloads. In essence, you're pre-computing what you'll most likely query for, eliminating the lag that could otherwise derail your applications.

This isn't "for free" of course, you pay the "complexity tax" at indexing time (Figure 4), but this is easier to scale with hardware and data modeling optimizations. While indexes are "eventually consistent" they always return data immediately as it's available, constantly re-indexing document changes in near real-time. This approach ensures that the data is always up-to-date and minimizes the risk of missing or outdated information. Moreover, indexes can be defined in your codebase or automatically generated based on your application queries, like a shield that magically appears when the time is right. This keeps everything in sync with your domain model simplifying deployment and operational overhead.

DRAGON WRANGLER TIP:

"Eventual consistency" practically means that sometimes queries to the database return stale data. 99% of the time, this isn't a problem – not everything shown to the user needs to be up-to-date at a millisecond granularity. For the 1% of times when it is critical, RavenDB gives you the flexibility to wait for non-stale results so you can make informed decisions for the best end-user experience. We also distinguish between document operations and queries – operations like reading and writing a document is an O(1) transaction and is ACID-compliant so it's guaranteed to be up-to-date. While queries give you the freedom to return results immediately or wait until everything is tallied up.

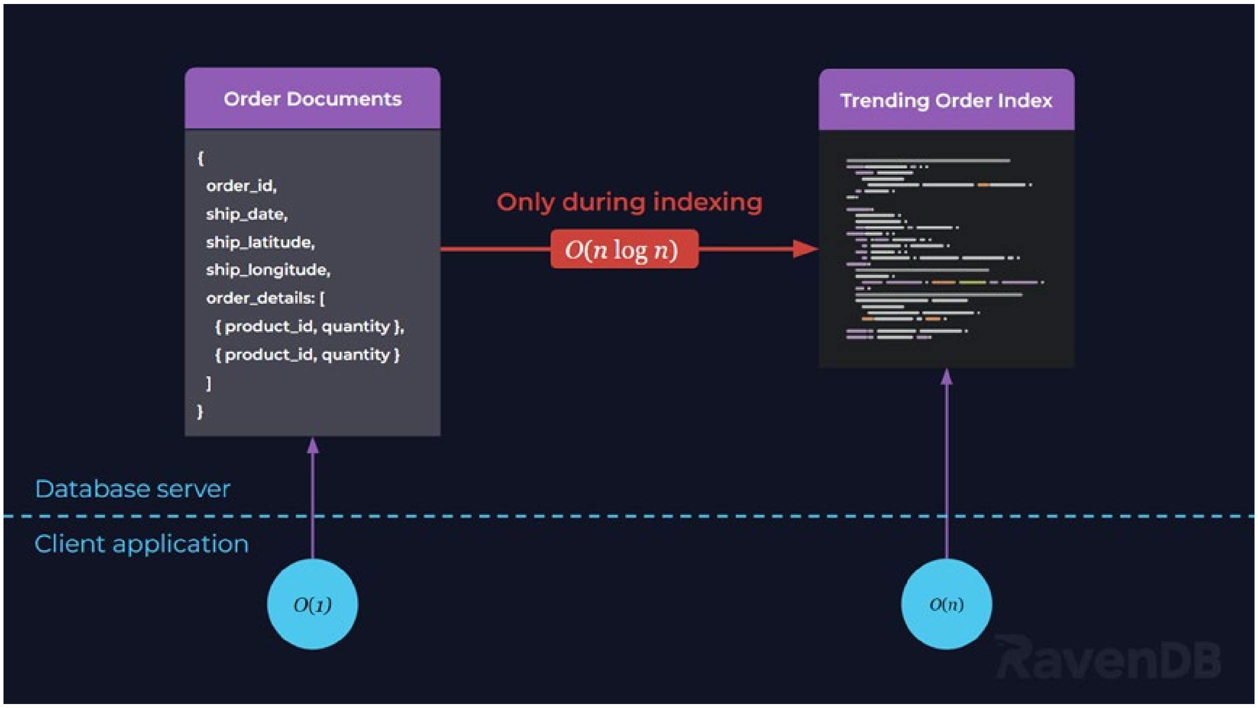


Fig. 4 How RavenDB indexes offload query complexity

Indexing, in whatever form it takes, is your shield in the battle for performance. It can make your queries not just faster but also more efficient and accurate. While traditional databases offer some level of protection, RavenDB takes it to a whole new level with ahead-of-time indexing.

Why Not Avoid Battling Dragons In the First Place?

Business is all about picking the right battles–your database shouldn't be one. For business-intensive applications, complex queries are a given. Instead of forcing you to battle poor query performance at runtime, RavenDB helps you avoid that in the first place.

RavenDB is an Open Source NoSQL document-relational database that is fully transactional without sacrificing performance. It supports aggregate queries with native support for time series and counters.

Prefer To Get A Closer Look On Your Own?

- Multi-document & distributed ACID transactions
- Ahead-of-time compiled query indexes
- Sharding and replication
- Free community license
- Host on RavenDB Cloud or self-host on Linux/ Windows
- Native support for Time Series and Counters
- Full-text search
- ETL to SQL, OLAP, Kafka, RabbitMQ, Elasticsearch, & PowerBI
- SDKs for .NET, Java, Node.js, Go, Python, and Elixir

BOOK A DEMO

RavenDB is open source and can be used freely with a community license.

Victory Feast: Real-World Case Studies of RavenDB's Performance Triumphs

How does ahead-of-time indexing translate to real-world results? Here are some customer victory feasts you can sink your teeth into.

Rakuten Kobo: Saw 80-90% lower total operating cost vs. Couchbase

In a large-scale deployment comparison conducted by Rakuten Kobo, one of the world's largest digital booksellers, RavenDB emerged as a far more cost-effective solution than Couchbase. Both databases were tested for performance, scalability, and cost efficiency, with RavenDB showing clear advantages in hardware scalability.

The annual costs for a high-performance RavenDB cluster were estimated to be $30,000, whereas the comparable Couchbase cluster amounted to a staggering $150,000 per year. Notably, the Couchbase cluster needed a minimum of 5 nodes with 192GB RAM each and suffered repeated failures when downscaled, showcasing its limitations in hardware scalability.

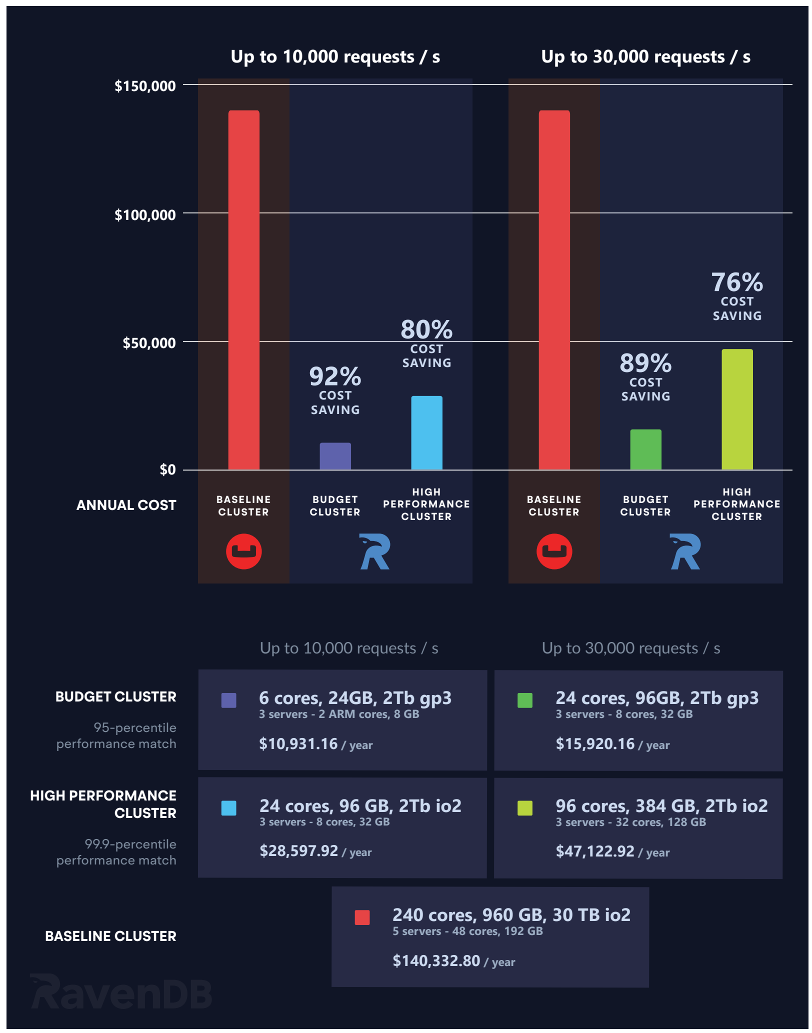


Fig. 5 Latency distribution in milliseconds for the highlights for user query (lower is better)

This price disparity between RavenDB and Couchbase offers potential cost savings of 80-90% with RavenDB.

Even more critically, Couchbase required a separate system (like Elasticsearch) for handling queries under real load conditions, potentially adding even more costs and complexity. In contrast, RavenDB successfully filled both roles of data storage and query processing at a much-reduced price tag, demonstrating its value not just in performance but also in cost-efficiency. This makes RavenDB a compelling choice for companies looking to maximize their ROI in database solutions.

READ THE WHITE PAPER

IoT Bridge: Increased Request Throughput By 4,000X

Before switching to RavenDB, IoT Bridge faced serious performance limitations with their initial choice of MongoDB as their NoSQL database. Despite optimizing their application code, they struggled to meet customer demands for a guaranteed ingestion rate. Their **maximum request throughput was capped at 26-30 requests per second**, a bottleneck that raised concerns about scalability as they were planning for much higher levels of performance.

After transitioning to RavenDB, IoT Bridge experienced a transformative improvement in data processing speeds, **reaching over 120,000 requests per second**—a leap that far exceeded their initial expectations. RavenDB's fully transactional nature, automatic indexes, and ACID compliance added further value without sacrificing performance. They now operate on an entry-level server at a minimal cost, with room to scale as their customer and device numbers grow. This dramatic increase in performance has not only resolved their previous limitations but also better positioned them for future expansion.

READ THE USE CASE

PartsSource: Sped Up Real-Time Price Calculations by 3,000X

PartsSource, a leading medical replacement parts supplier serving U.S. hospitals, grappled with a performance bottleneck that severely affected their online shopping experience. Offering **over 1.3 million products and navigating through more than 10,000 pricing rules**, their system was painfully slow, forcing customers to wait up to 3 seconds to see a single product's price. Their former relational database had to run real-time queries across 12 separate tables, putting both performance and customer satisfaction at risk.

By utilizing ahead-of-time indexes, RavenDB took on the heavy lifting of pre-computing these intricate pricing calculations. The payoff was astronomical: instead of displaying 1 product in 3 seconds, **they can now display a page of 30 products in 30 milliseconds, marking a 3,000X improvement in speed**. This upgrade didn't just enhance the shopping experience—it redefined what was possible.

READ THE USE CASE

DPG Media: Increased complex query speed by 800% and reduced operational costs by 1/3

DPG Media, a large media conglomerate, faced significant challenges with operational complexity and cost-efficiency in their data management. Their system was reliant on a combination of multiple big-name NoSQL databases like DynamoDB, Redis, and Elasticsearch in conjunction with their main PostgreSQL database. This resulted in complex, table-spanning queries that compromised on performance and increased operational costs. Despite tests on alternative solutions like MongoDB and CouchDB, integration remained a pain point, keeping their costs high and making system maintenance laborious.

DPG Media made a strategic decision to integrate RavenDB into their architecture, yielding transformative results. The transition simplified their data management by consolidating multiple databases into a single RavenDB platform. **This led to an 800% performance increase for complex queries and reduced operational costs by about 1/3**. RavenDB's query language was similar to SQL, making it easier for their existing team to adapt without altering much of their codebase. With RavenDB, the company achieved up to 400K operations per second in tests and provided over 1M transactions per second on commodity hardware, optimizing cost and performance. RavenDB's versatility and feature-rich nature—from client-side caching to default ACID transactions—also simplified many aspects of their operational processes, effectively reducing their overall system complexity.

READ THE USE CASE

Kamran Ayub is a developer, educator, speaker, and the founder of Keep Track of My Games. He teaches developers modern web development, cloud, and NoSQL as a Pluralsight author. He also helps maintain the Excalibur.js game engine. Previously, he helped build and scale massive websites for Fortune 500 companies.

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/