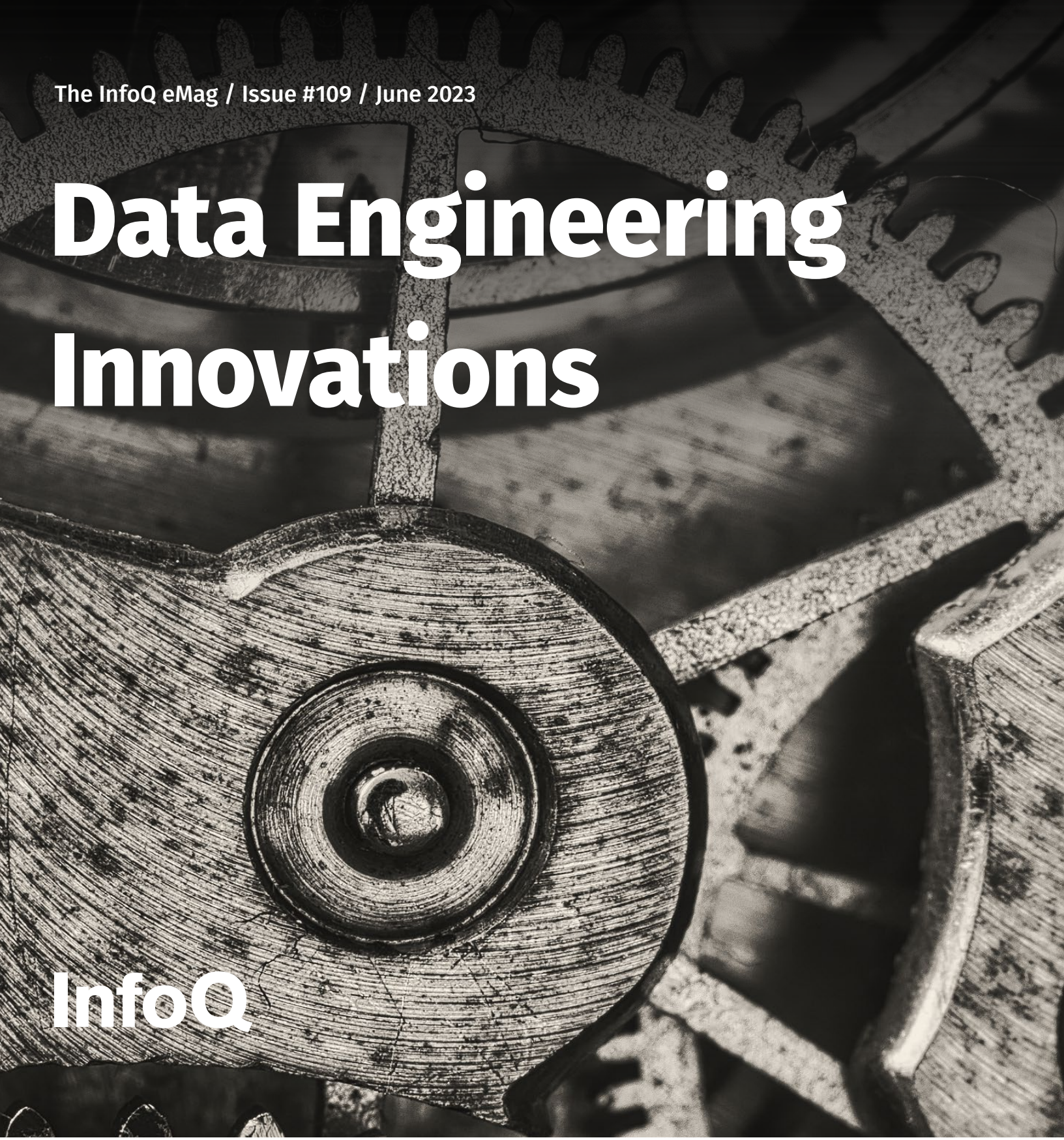
The InfoQ eMag / Issue #109 / June 2023

Data Engineering Innovations



InfoQ

In-Process Analytical Data Management with DuckDB

Create Your Distributed Database on Kubernetes with Existing Monolithic Databases

Understanding and Applying Correspondence Analysis

Data Engineering Innovations

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITOR Susan Conant / DESIGN Dragos Balasoiu, Ana Ciobotaru
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Hannes Mühleisen

is a creator of the DuckDB database management system and Co-founder and CEO of DuckDB Labs, a consulting company providing services around DuckDB. He is also a senior researcher of the Database Architectures group at the Centrum Wiskunde & Informatica (CWI), the Dutch national research lab for Mathematics and Computer Science in Amsterdam. Hannes is also Professor of Data Engineering at Radboud Universiteit Nijmegen. His' main interest is analytical data management systems.

Trista Pan

SphereEx Co-Founder & CTO, Apache Member & Incubator Mentor, Apache ShardingSphere PMC, AWS Data Hero, China Mulan Open Source Community Mentor, Tencent Cloud TVP. She used to be responsible for the design and development of the intelligent database platform of JD Digital Science and Technology. She now focuses on the distributed database & middleware ecosystem, and open-source community.

Rogerio Robetti

is a software engineer with many years of experience, an enthusiast and advocate of modern approaches to software engineering. Has a soft spot for resource usage optimization and automation. Have worked in very heterogeneous environments and different roles from developer to architect and from startup size companies to global organizations, mostly producing enterprise grade web applications but also having experience in R&D, mobile and mainframe applications. Currently lives in Dublin, Republic of Ireland.

Guy Braunstain

is a Tel Aviv-based passionate Full Stack Developer who flourishes in leading development processes from design to implementation through to customer satisfaction. He is continuously searching for innovation and making dev team cycles shorter and more efficient. Prior to working at Jit, Guy served in an elite IDF technological unit, in his current role he is excited to be part of the team making security easier to consume and manage for all dev-teams.

Maarit Widmann

is a data scientist at KNIME. She started with quantitative sociology and holds her bachelor's degree in social sciences. The University of Konstanz made her drop the "social" as a Master of Science. Her ambition is to communicate the concepts behind data science to others in videos and blog posts. Follow Maarit on LinkedIn.

Alfredo Roccato

is an independent consultant and trainer with a focus on data science. He studied statistics at the Catholic University in Milan and has been serving companies with business intelligence and analytics for over 35 years. Follow Alfredo on LinkedIn.

Srini Penchikala

currently works as Senior Software Architect in Austin, Texas. He is also the Lead Editor for AI/ML/Data Engineering community at InfoQ. Srini has over 22 years of experience in software architecture, design and development. He is the author of "Big Data Processing with Apache Spark. He is also the co-author of "Spring Roo in Action" book from Manning Publications. Srini has presented at conferences like Big Data Conference, Enterprise Data World, JavaOne, SEI Architecture Technology Conference (SATURN), IT Architect Conference (ITARC), No Fluff Just Stuff, NoSQL Now and Project World Conference. He also published several articles on software architecture, security and risk management, and NoSQL databases on websites like InfoQ, The ServerSide, OReilly Network (ONJava), DevX Java, java.net and JavaWorld.

A LETTER FROM THE EDITOR

Today's modern data architecture stacks look significantly different from the data architecture models from only a few years ago.

Data streaming and stream processing have become the core components of modern data architecture. Real time data streams are being managed as first-class citizens in data processing analytics solutions. Some companies are even shifting their architecture and technology thinking from "everything's at rest" to "everything's in motion."

Change data capture (CDC) has become a critical design pattern in data engineering use cases. CDC can be used in event-driven microservices based applications, along with data streaming to implement robust solutions.

The emphasis on data streams is also driving innovations in the data governance space such as the stream catalog and stream lineage.

Data mesh architecture, which has been getting a lot of attention recently, is built on four solid principles: domain ownership, data as a product, self-serve data infrastructure platform, and federated governance. Data mesh is expected to have a huge impact on the overall data management programs and initiatives in organizations.

Similar to many compute services in the cloud platforms, data storage services and databases now support serverless models where you only pay for what you use.

On the security and regulatory compliance side, data residency

and data sovereignty are getting a lot of attention to ensure the consumers' data is protected and privacy is maintained throughout the life of the data.

Next-generation data engineering innovations will build on these recent trends to provide even more robust, secure, highly available and resilient data solutions to the development community.

In the InfoQ "Data Engineering Innovations" eMag, you'll find up-to-date case studies and real-world data engineering solutions from technology SME's and leading data practitioners in the industry.

"In-Process Analytical Data Management with DuckDB" by Dr. Hannes Mühleisen highlights the open-source in-process OLAP database designed for analytical data management, how it eliminates the need to copy large amounts of data over sockets, resulting in improved performance. Author also discusses the database support for Morsel-Driven parallelism which allows efficient parallelization across multiple cores while maintaining awareness of multi-core processing.

Trista Pan's article "Create Your Distributed Database on Kubernetes with Existing Monolithic Databases" emphasizes the role Kubernetes plays in supporting cloud native databases and how Apache ShardingSphere can transform any database to a distributed database system, while enhancing it with functions such as sharding, elastic scaling, encryption features, etc. Author demonstrates how to deploy ShardingSphere-Operator, create a sharding table using DistSQL, and test the Scaling and HA of the ShardingSphere-Proxy cluster.

"Design Pattern Proposal for Autoscaling Stateful Systems" by Rogerio Robetti captures the need for proven design patterns for autoscaling stateful systems. Synchronization of data on new nodes is a big challenge when scaling up a stateful system. Robetti discusses various use cases and solution designs that can be used as a foundation for stateful autonomous scalability implementations.

Guy Braunstain's article titled "DynamoDB Data Transformation Safety: from Manual Toil to Automated and Open Source" focuses on data transformation as a continuous challenge in engineering especially in cloud hosted solutions. There is a current lack of tools to perform data transformations programmatically, in an automated way. The open source utility Dynamo Data Transform can be used as a data transformation tool for DynamoDB based systems.

And "Understanding and Applying Correspondence Analysis" authored by Maarit Widmann & Alfredo Roccato describes the simple correspondence analysis (CA) technique to analyze relationships between categorical variables and create profiles based on the projections of the original variables to the new dimensions that it creates. The authors demonstrate how to perform Correspondence Analysis with steps like data collection, data preprocessing, computing CA, and interpreting the results, all using the KNIME open-source analytics platform

We hope that you find value in the articles and resources in this eMag and are able to apply some of these design patterns and techniques in your own data engineering projects and initiatives.

We would love to receive your feedback via editors@infoq.com or on Twitter about this eMag. I hope you have a great time reading it!

In-Process Analytical Data Management with DuckDB

by **Hannes Mühleisen,** Co-founder and CEO @ DuckDB Labs|Co-Creator of DuckDB

Why did I embark on the journey of building a new database? It started with a statement by the well-known statistician and software developer Hadley Wickham:

If your data fits in memory there is no advantage to putting it in a database: it will only be slower and more frustrating.

This sentiment was a blow and a challenge to database researchers like myself. What are the aspects that make databases slow and frustrating? The first culprit is the client-server model.

When conducting data analysis and moving large volumes of data into a database from an application, or extracting it from a database into an analysis environment like R or Python, the process can be painfully slow.

I tried to understand the origins of the client-server architectural pattern, and I authored the paper, "Don't Hold My Data Hostage – A Case For Client Protocol Redesign".

Comparing the database client protocols of various data management systems, I timed how long it took to transmit a fixed dataset between a client program and several database systems.

As a benchmark, I used the Netcat utility to send the same dataset over a network socket.

Compared to Netcat, transferring the same volume of data with MySQL took ten times longer, and with Hive and MongoDB, it took over an hour. The client-server model appears to be fraught with issues.

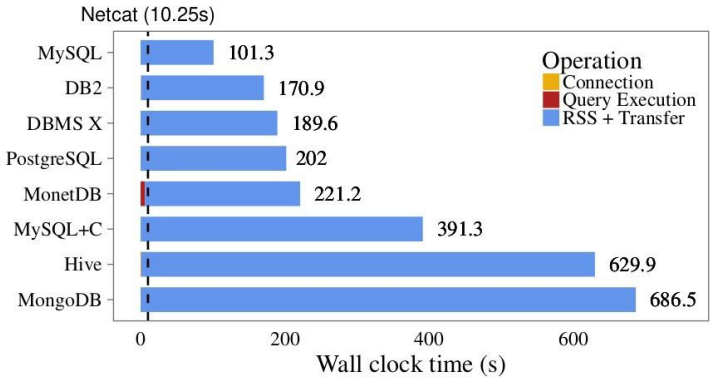


Figure 1: Comparing different clients; the dashed line is the wall clock time for netcat to transfer a CSV of the data

SQLite

My thoughts then turned to SQLite. With billions and billions of copies existing in the wild, SQLite is the most extensively used SQL system in the world. It›s quite literally everywhere: you›re daily engaging with dozens, if not hundreds, of instances unbeknownst to you.

SQLite operates in-process, a different architectural approach integrating the database management system directly into a client application, avoiding the traditional client-server model.

Data can be transferred within the same memory address space, eliminating the need to copy and serialize large amounts of data over sockets.

However, SQLite isn't designed for large-scale data analysis and its primary purpose is to handle transactional workloads.

DuckDB

Several years ago, Mark Raasveldt and I began working on a new database, DuckDB. Written entirely in C++, DuckDB is a database management system that employs a vectorized execution engine. It is an in-process database engine and we often refer to it as the 'SQLite for analytics'. Released under the highly permissive MIT license, the project operates under the stewardship of a foundation, rather than the typical venture capital model.

What does interacting with DuckDB look like?

```
import duckdb
duckdb.sql('LOAD httpfs')
duckdb.sql("SELECT * FROM 'https://
github.com/duckdb/duckdb/blob/
master/data/parquet-testing/
userdata1.parquet'").df()
```

In these three lines, DuckDB is imported as a Python package, an extension is loaded to enable communication with HTTPS resources, and a Parquet file is read from a URL and converted back to a Panda DataFrame (DF).

DuckDB, as demonstrated in this example, inherently supports Parquet files, which we consider the new CSV. The LOAD httpfs call illustrates how DuckDB can be expanded with plugins.

There's a lot of intricate work hidden in the conversion to DF, as it involves transferring a result set, potentially millions of lines. But as we are operating in the same address space, we can bypass serialization or socket transfer, making the process incredibly fast.

We've also developed a command-line client, complete with features like query autocompletion and SQL syntax highlighting. For example, I can initiate a DuckDB shell from my computer and read the same Parquet file:

```
→ parquet-testing git:(master) ✗ duckdb
v0.5.2-dev5502 d43d863378
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
D SELECT * FROM userdata1.parquet limit 10;
```

If you consider the query:

```
SELECT * FROM userdata.parquet;
```

you realize that would not typically work in a traditional SQL system, as userdata.parquet is not a table, it is a file. The table doesn't exist yet, but the Parquet file does. If a table with a specific name is not found, we search for other entities with that name,

such as a Parquet file, directly executing queries on it.

In-Process Analytics

From an architectural standpoint, we have a new category of data management systems: in-process OLAP databases. SQLite is an in-process system, but it is geared toward OLTP (Online Transaction Processing). When you think of a traditional client-server architecture for OLTP, PostgreSQL is instead the most common option.

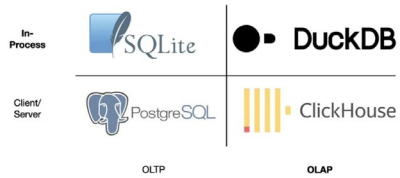


Figure 2: OLTP versus OLAP

On the OLAP side, there have been several client-server systems, with ClickHouse being the most recognized open-source option. However, before the emergence of DuckDB, there was no in-process OLAP option.

Technical Perspective of DuckDB

Let's discuss the technical aspects of DuckDB, walking through the stages of processing the following query in Figure 3.

The example involves selecting a name and sum from the joining of two tables, customer, and sale that share a common column, cid. The goal is to compute the total revenue per customer, summing up all revenue

```
SELECT FIRST(name), SUM(rev+tax)
FROM cust JOIN sale USING(cid) GROUP BY cid
```

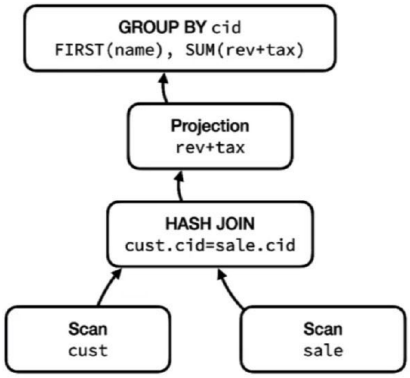


Figure 3: A simple select query on DuckDB

and including tax for each transaction.

When we run this query, the system joins the two tables, aggregating customers based on the value in the cid column. Then, the system computes the revenue + tax projection, followed by a grouped aggregation by cid, where we compute the first name and the final sum.

DuckDB processes this query through standard phases: query planning, query optimization, and physical planning, and the query planning stage is further divided into so-called pipelines.

For example, this query has three pipelines, defined by their ability to be run in a streaming fashion. The streaming ends when we encounter a breaking operator, that is an operator that needs to retrieve the entire input before proceeding.

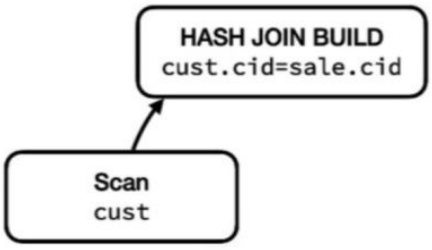


Figure 4: First pipeline

The first pipeline scans the customer table and constructs a hash table. The hash join is split into two phases, building the hash table on one side of the join, and probing, which happens on the other side. The building of the hash table requires seeing all data from the left-hand side of the join, meaning we must run through the entire customer table and feed all of it into the hash join build phase. Once this

pipeline is completed, we move to the second pipeline.

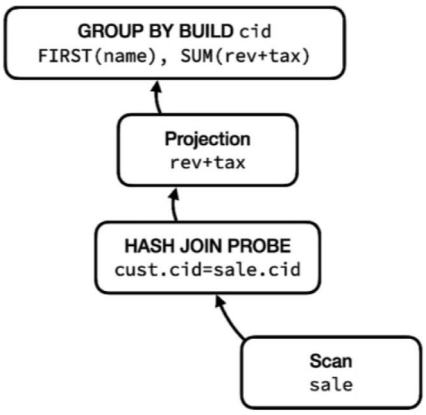


Figure 5: Second pipeline

The second pipeline is larger and contains more streaming operators: it can scan the sales table, and look into the hash table we›ve built before to find join partners from the customer table. It then projects the revenue + tax column and runs the aggregate, a breaking operator. Finally, we run the group by build phase and complete the second pipeline.

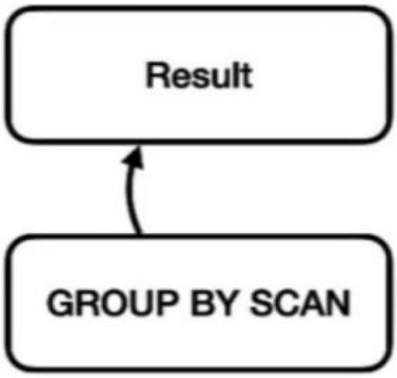


Figure 6: Third pipeline

We can schedule the third and final pipeline hat reads the results of the GROUP BY and outputs the result. This process is fairly standard and many database systems take a similar approach to query planning.

Row-at-a-time

To understand how DuckDB processes a query, let's consider first the traditional Volcano-style iterator model that operates through a sequence of iterators: every operator exposes an iterator and has a set of iterators as its input.

The execution begins by trying to read from the top operator, in this case, the GROUP BY BUILD phase. However, it can't read anything yet as no data has been ingested. This triggers a read request to its child operator, the projection, which reads from its child operator, the HASH JOIN PROBE. This cascades down until it finally reaches the sale table. The sale table generates a tuple, for example, 42, 1233, 422, representing the ID, revenue, and tax columns. This tuple then moves up to the HASH JOIN PROBE, which consults its built hash table. For instance, it knows that ID 42 corresponds to the company ASML and it generates a new row as the join result, which is ASML, 1233, 422.

This new row is then processed by the next operator, the projection, which sums up the last two columns, resulting in a new row: ASML, 1355. This row finally enters the GROUP BY BUILD phase.

This tuple-at-a-time, row-at-a-time approach is common to many database systems such as PostgreSQL, MySQL, Oracle, SQL Server, and SQLite. It's particularly effective for transactional use cases, where single rows are the focus,

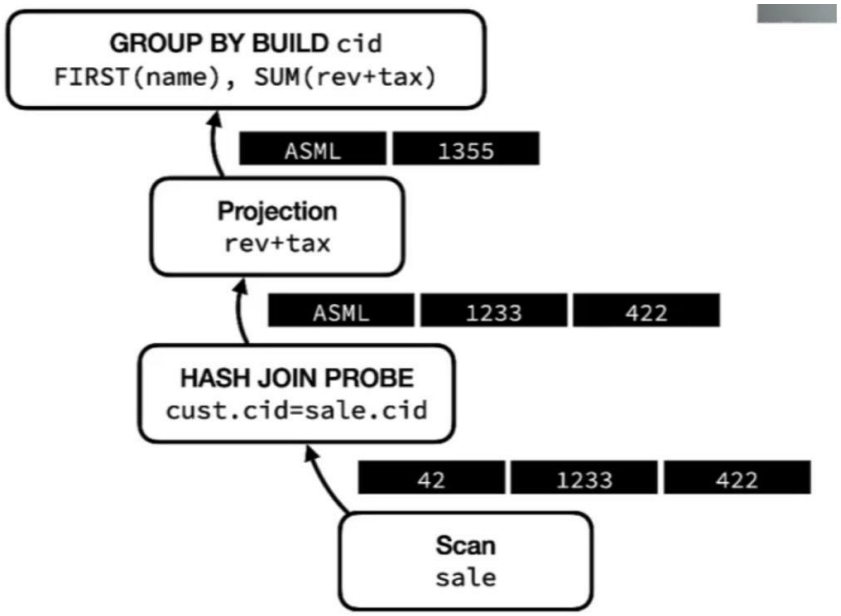


Figure 7: Volcano-style iterator model

but it has a major drawback in analytical processing: it generates significant overhead due to the constant switching between operators and iterators.

One possible improvement suggested by the literature is to just-in-time (JIT) compile the entire pipeline. This option, though viable, isn't the only one.

Vector-at-a-time

Let's consider the operation of a simple streaming operator like the projection.

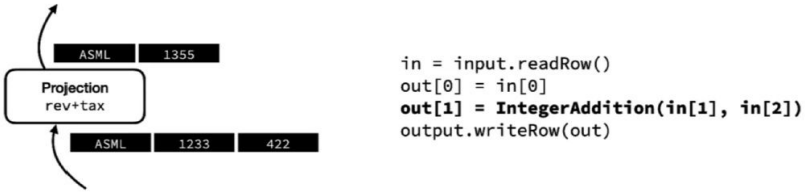


Figure 8: Implementation of a projection

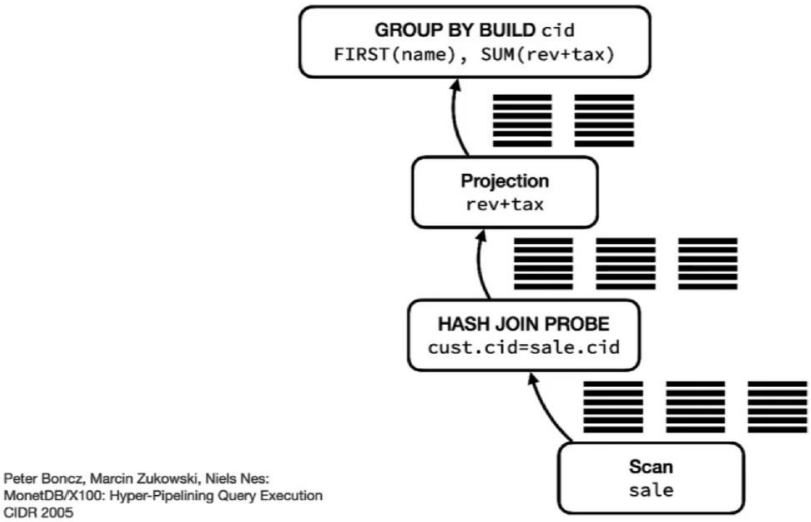


Figure 9: The vector-at-a-time model

We have an incoming row and some pseudocode: input. readRow reads a row of input, the first value remains unchanged, and the second entry in the output becomes the result of adding the second and third values of the input, with the output then written. While this approach is easy to implement, it incurs a significant performance cost due to function calls for every value read.

An improvement over the row-at-a-time model is the vector-at-a-time model, first proposed in "MonetDB/X100: Hyper-Pipelining Query Execution" in 2005.

This model processes not just single values at a time, but short columns of values collectively referred to as vectors. Instead of examining a single value for each row, multiple values are examined for each column at once. This approach reduces the overhead as type switching is performed on a vector of values instead of a single row of values.

The vector-at-a-time model strikes a balance between columnar and row-wise executions. While columnar execution is more efficient, it can lead to memory issues. By limiting the size of columns to something manageable, the vector-at-a-time model avoids JIT compilation. It also promotes cache locality, which is critical for efficiency.

The importance of cache locality is illustrated by the well-known Latency Numbers Everyone Should Know, Figure 10.

The graphic, provided by Google's Peter Norvig and Jeff Dean, highlights the disparity between the L1 cache reference (0.5 nanoseconds) and the main memory reference (100 nanoseconds), a factor of 200.

Given that L1 cache reference has become 200 times faster since 1990 compared to memory reference, which is only twice as fast, there's a significant advantage in having operations fit within the CPU cache.

This is where the beauty of vectorized query processing lies.

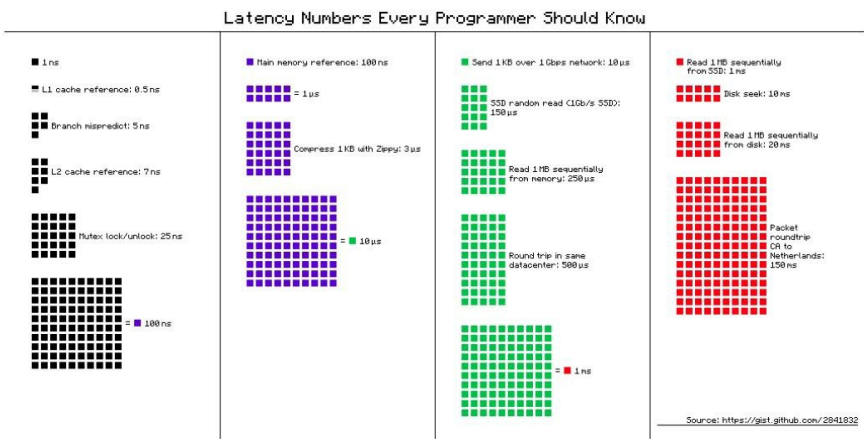
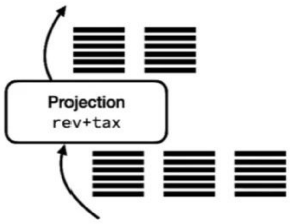


Figure 10: Latency Numbers Everyone Should Know




```
in = input.readChunk()
out[0] = in[0]
res = IntVector()
for i in range(2048):
  res[i] = in[1][i] + in[2][I]
out[1] = res
output.writeChunk(out)
```

Figure 11: Implementation of a projection with vectorized query processing

Let's consider the same projection of revenue + tax example we discussed before. Instead of retrieving a single row, we get as input three vectors of values and output two vectors of values. We read a chunk (a collection of small vectors of columns) instead of a single row. As the first vector remains unchanged, it's reassigned to the output. A new result vector is created, and an addition operation is performed on every individual value in the range from 0 to 2048.

This approach allows the compiler to insert special instructions automatically and avoids function call overhead by interpreting and switching around data types and operators only at the vector level.

This is the core of vectorized processing.

Exchange-Parallelism

Vectorized processing being efficient on a single CPU is not enough, it also needs to perform well on multiple CPUs. How can we support parallelism?

Goetz Graefe, principal scientist at Google, in his paper "Volcano - An Extensible and Parallel Query Evaluation System" described the concept of exchange operator parallelism.

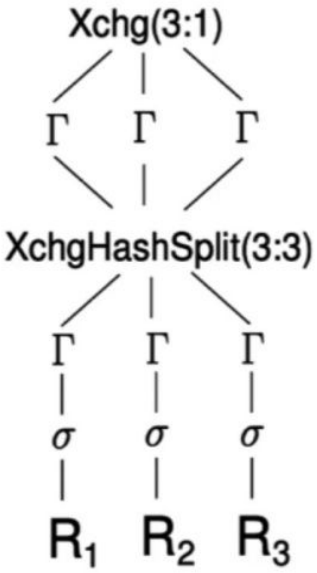


Figure 12: Exchange operator parallelism

In this example, three partitions are read simultaneously. Filters are applied and values are pre-aggregated, then hashed. Based on the values of the hash, the data is split up, further aggregated, re-aggregated, and then the output is combined. By doing this, most parts of the query are effectively parallelized.

For instance, you can observe this approach in Spark's execution of a simple query. After scanning the files, a hash aggregate performs a partial_sum.

Then, a separate operation partitions the data, followed by a re-aggregation that computes the total sum. However, this has been proven to be problematic in many instances.

Morsel-Driven Parallelism

A more modern model for achieving parallelism in SQL engines is Morsel-Driven parallelism. As in the approach above, the input level scans are divided, resulting in partial scans. In our second pipeline, we have two partial scans of the sale table, with the first one scanning the first half of the table and the second one scanning the latter half.

The HASH JOIN PROBE remains the same as it still operates on the same hash table from the two pipelines. The projection operation is independent, and all these results sync into the GROUP BY operator, which is our blocking operator. Notably, you don›t see an exchange operator here.

Unlike the traditional exchange operator-based model, the GROUP BY is aware of the parallelization taking place and is equipped to efficiently manage the contention arising from different threads reading groups that could potentially collide. In Morsel-Driven parallelism, the process begins (*Phase 1*) with each thread pre-aggregating its values. The separate subsets or morsels of input data, are built into separate hash tables.

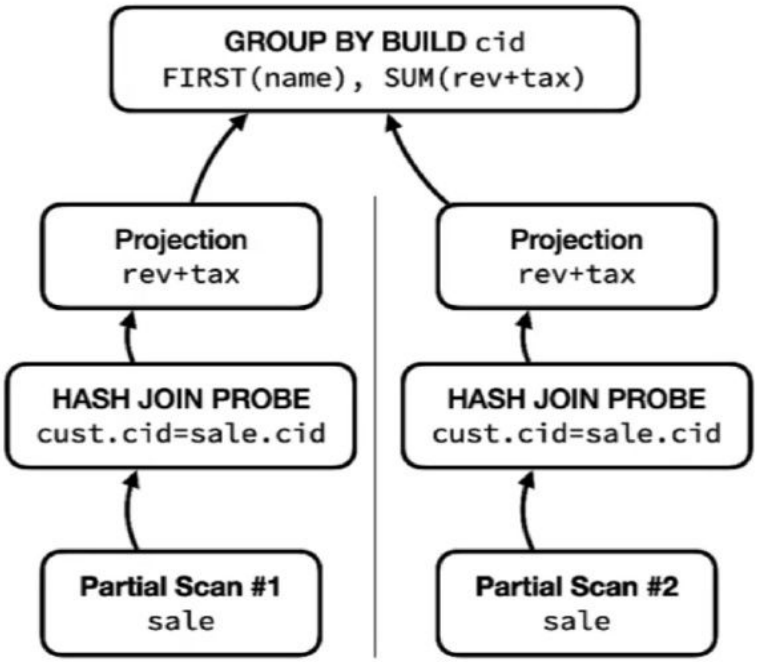


Figure 13: Morsel-Driven parallelism

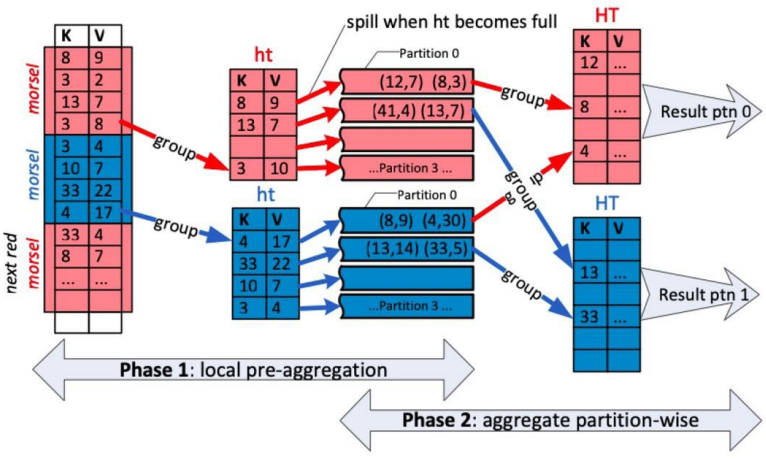


Figure 14: Partitioning hash tables for parallelized merging

The next phase (*Phase 2*) involves partition-wise aggregation: in the local hash tables, data is partitioned based on the radixes of the group keys, ensuring that each hash table cannot contain keys present in any other hash table. When all the data has been read and it's time to finalize our hash table and aggregate, we can select the same partition from each participating thread and schedule more threads to read them all.

Though this process is more complex than a standard aggregate hash table, it allows the Morsel-Driven model to achieve great parallelism. This model efficiently constructs an aggregation over multiple inputs, circumventing the issues associated with the exchange operator.

Simple Benchmark

I conducted a simple benchmark, using our example query with some minor added complexity in the form of an ORDER BY and a LIMIT clause. The query selects

the name and the sum of revenue + tax from the customer and sale tables, which are joined and grouped by the customer ID.

The experiment involved two tables: one with a million customers and another with a hundred million sales entries. This amounted to about 1.4 gigabytes of CSV data, which is not an unusually large dataset.

```
SELECT MIN(name), SUM(rev+tax) total
  FROM cust JOIN sale USING(cid)
GROUP BY cid
ORDER BY total DESC
LIMIT 10;
```

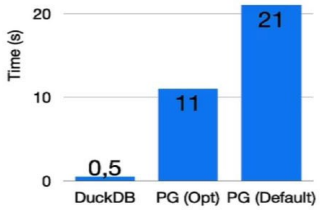


Figure 15: The simple benchmark

DuckDB completed the query on my laptop in just half a second. On the other hand, PostgreSQL, after I had optimized the configuration, took 11 seconds to finish the same task. With default settings, it took 21 seconds.

While DuckDB could process the query around 40 times faster than PostgreSQL, it's important to note that this comparison is not entirely fair, as PostgreSQL is primarily designed for OLTP workloads.

Conclusions

The goal of this article is to explain the design, functionality, and rationale behind DuckDB, a data engine encapsulated in a compact package. DuckDB functions as a library linked directly to the application process, boasting a small footprint and no dependencies and allowing developers to easily integrate a SQL engine for analytics.

I highlighted the power of in-process databases, which lies in their ability to efficiently transfer result sets to clients and write data to the database. An essential component of DuckDB's design is vectorized query processing: this technique allows efficient in-cache operations and eliminates the burden of the function call overhead.

Lastly, I touched upon DuckDB's parallelism model: Morsel-Driven parallelism supports efficient parallelization across any number of cores while maintaining awareness of multi-core processing, contributing to DuckDB's overall performance and efficiency.

SPONSORED ARTICLE

How Disney+ Hotstar Modernized its Data Architecture for Scale

by **Cynthia Dunlop**, Senior Director of Content Strategy @ ScyllaDB

Disney+ Hotstar, India's most popular streaming service, accounts for 40% of the global Disney+ subscriber base.

Disney+ Hotstar offers over 100,000 hours of content on demand, as well as livestreams of the world's most watched sporting events.

The "Continue Watching" feature is critical to the on demand streaming experience for the 300 million-plus monthly active users. That's what lets you pause a video on one device and instantly pick up where you left off on any device, anywhere in the world. It's also what entices you to binge-watch your favorite series: complete one episode of a show and the next one just starts playing automatically.

However, it's not easy to make things so simple. In fact, the underlying data infrastructure powering this feature had grown overly complicated. It was originally built on a combination of Redis and Elasticsearch, connected to an event processor for Apache Kafka streaming data. Having multiple data stores meant maintaining multiple data models, making each change a huge burden. Moreover, data doubling every six months required constantly increasing the cluster size, resulting in yet more admin and soaring costs.

Previous architecture: Here's how the "Continue Watching" functionality was originally architected.

First, the user's client would send a "watch video" event to Kafka. From Kafka, the event would be processed and saved to both Redis and Elasticsearch. If a user opened the home page, the backend was called, and data was retrieved from Redis and Elasticsearch. Their Redis cluster held 500 GB of data, and the Elasticsearch cluster held 20 terabytes.

Their key-value data ranged from 5 to 10 kilobytes per event. Once the data was saved, an API server read from the two databases and sent values back to the client whenever the user next logged in or resumed watching. Redis provided acceptable latencies, but the increase in data size meant that they needed to horizontally scale their cluster. This increased their cost every three to four months. Elasticsearch latencies were on the higher end of 200 milliseconds. Moreover, the average cost of Elasticsearch was quite high considering the returns. They often experienced issues with node maintenance and manual effort was required to resolve the issues.

Modernized architecture: First, the team adopted a new data model that could suit both use cases. Then, they set out to adopt a new database. Apache Cassandra, Apache HBase, Amazon DynamoDB, and ScyllaDB were considered. The team selected ScyllaDB for two key reasons. 1) Consistently low latencies for both reads and writes, which would ensure a snappy user experience for today's demanding customers. 2) ScyllaDB Cloud, a fully managed database as a service (NoSQL DBaaS), offered a much lower cost than the other options they considered.

Try ScyllaDB Cloud with your projects - 30 days free.

Create Your Distributed Database on Kubernetes with Existing Monolithic Databases

by **Trista Pan,** CTO & Co-founder of SphereEx

Background

Most of the recent convenience upgrades that have blessed peoples' lives in the 21st century can be traced back to the widespread adoption of the Internet.

Constant connectivity at our fingertips improved our lives, and created new technical infrastructure requirements to support high-performance Internet services. Developers and DevOps teams have become focused on ensuring the backend infrastructure's availability, consistency, scalability, resilience, and fully automated management.

Examples of issues that tech teams are constantly struggling with include managing and storing large amounts of business data and creating the conditions to ensure that infrastructures deliver optimal service to the applications.

Also, designing technical architecture while thinking ahead to meet future requirements and evolving modern applications to be able to “live” in the cloud.

The cloud is game-changing technology, and if you haven't yet, you should get familiar with it. It has already transformed infrastructure as we know it, from development to delivery, deployment, and maintenance.

Nowadays, modern applications are embracing the concept of anything-as-a-service from various cloud vendors, and developer and operations teams are considering upgrading legacy workloads to future cloud-native applications.

Microservices on Kubernetes

To address the challenges mentioned above, we are witnessing an evolution of the application layer from monolithic services to microservices. By dividing a single monolithic service into smaller units, modern applications can become independent of one another while eliminating unwanted effects of development, deployment, and upgrading.

Moreover, to decouple and simplify communication services, such as APIs and calls, service mesh appeared and took over. Kubernetes provides an abstract platform and mechanism for this evolution, explaining its popularity.

If I had to pinpoint the reason why Kubernetes is so popular, I'd say that it's because, according to the Kubernetes docs:

Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system. (From "Why you need Kubernetes and what it can do" section.)

Kubernetes is an ideal platform for managing the microservice's lifecycle, but what about the database, a stateful service?

Databases

The application layer has adopted microservices as the solution to address the issues previously introduced here. Still, when it comes to the database layer, the situation is a little different.

To answer the pain points we raised, we can look at the database layer. It uses a different method, yet somewhat similar: sharding, a.k.a. distributed architecture.

Currently, this distributed architecture is ubiquitous, whether we're talking about NoSQL databases, such as MongoDB, Cassandra, Hbase, DynamoDB, or NewSQL databases, such as CockroachDB, Google Spanner, Aurora, and so forth. Distributed databases require splitting the monolithic one into smaller units, or shards, for higher performance, improved capability, elastic scalability, etc.

One thing all of these database vendors have in common is that they all must consider historical migration to streamline this evolution process. They all provide data migration from existing Oracle, MySQL, PostgreSQL, and SQLServer databases, just to name a few, to their new database offerings. That's why CockroachDB is compatible with the PostgreSQL protocol, Vitess provides a sharding feature for MySQL, or AWS has Aurora-MySQL and Aurora-PostgreSQL.

Database on Cloud and Kubernetes

The advent of the cloud represents the next challenge for databases. Cloud platforms that are "go-on-demand," "everything-as-a-service," or "out-of-box" are currently changing the tech world.

Consider an application developer. To stay on pace with the current trends, the developer adheres to the cloud-native concept and prefers to deliver the applications on the cloud or Kubernetes. Does this mean it is time for databases to be on the cloud or Kubernetes? The majority of readers would probably answer with a resounding yes - which explains why the market share of the Database-as-a-service (DBaaS) is steadily increasing.

Nevertheless, if you're from the buy side for these services, you may wonder which vendor can promise you indefinite support. The truth is that nobody can give a definitive answer, so multi-cloud comes to mind, and databases on Kubernetes seem to have the potential to deliver on this front.

This is because Kubernetes is essentially an abstraction layer for container orchestration and is highly configurable and extensible, allowing users to

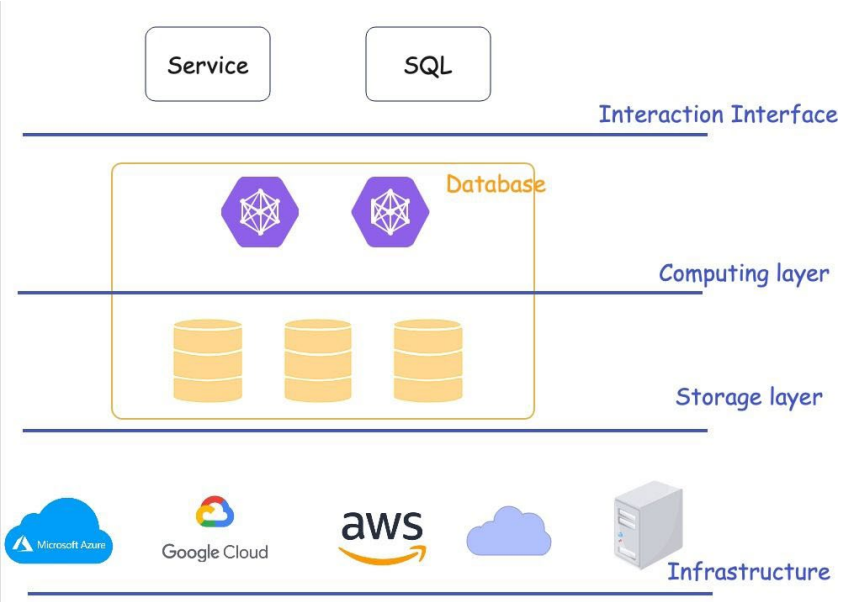
do custom coding for their specific scenarios. Volumes on Kubernetes, for example, are implemented and provided by many cloud vendors. If services are deployed on Kubernetes, applications will be able to interact with Kubernetes rather than different types of specific cloud services or infrastructure. This philosophy has already proven to work well in the case of stateless applications or microservices. As a result of these successful cases, people are thinking about how to put databases on Kubernetes to become cloud neutral.

A drawback to this solution is that it is more difficult to manage than the application layer, as Kubernetes is designed for stateless applications rather than databases and stateful applications. Many attempts to leverage Kubernetes' fundamental mechanisms, such as StatefulSet and Persistent Volume, overlay their custom coding to address the database challenge on Kubernetes. This approach can be seen in operators of MongoDB, CockroachDB, PostgreSQL, and other databases.

Database Compute-Storage Architecture

This approach has become common, but is it the only one? My answer is no, and the following content will introduce you to and demonstrate another method for converting your existing monolithic database into a distributed database system running on Kubernetes in a more cloud-native pattern.

With the help of the following illustration, let's first consider why this is possible.



As you can see from the illustration, the database has two capabilities: computing and storage.

MySQL, PostgreSQL, and other single-node databases combine or deploy two components on a single server or container.

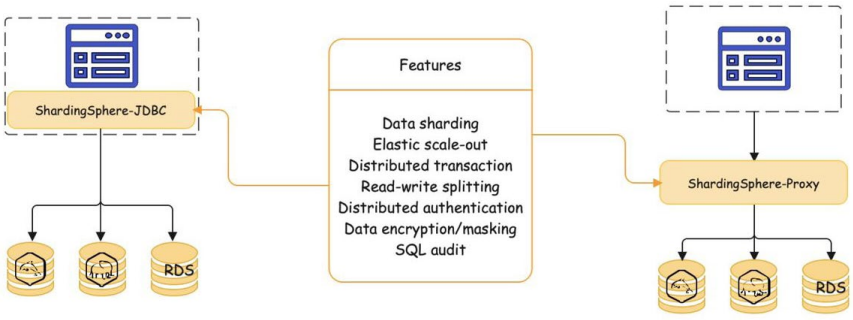
Apache ShardingSphere

Apache ShardingSphere is the ecosystem to transform any database into a distributed database system and enhance it with sharding, elastic scaling, encryption features, and more. It provides two clients, ShardingSphere-Proxy and ShardingSphere-Driver.

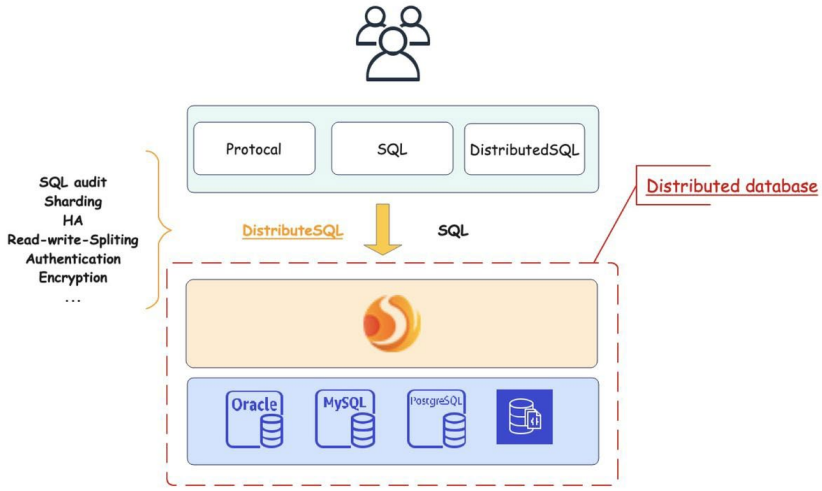
ShardingSphere-Proxy is a transparent database proxy that acts as a MySQL or PostgreSQL database server while supporting sharding databases, traffic governance (e.g., read/write splitting), automatically encrypting data, SQL auditing, and so on. All of its features are designed as plugins, allowing users to leverage DistSQL (Distributed SQL) or a YAML configuration to select and enable only their desired features.

ShardingSphere-JDBC is a lightweight Java framework that brings additional features to Java's JDBC layer. This driver shares most of the same features with ShardingSphere-Proxy.

As I've introduced earlier, if we view monolithic databases as shards (aka storage nodes) and ShardingSphere-Proxy or ShardingSphere-JDBC as the



global server (aka computing node), then ultimately, the result is a distributed database system. It can be graphically represented as follows:



Because ShardingSphere-Proxy acts as a MySQL or PostgreSQL server, there is no need to change the connection method to your legacy databases while ShardingSphere-JDBC implements the JDBC standard interface. This significantly minimizes the learning curve and migration costs.

Furthermore, ShardingSphere provides DistSQL, a SQL-style language for managing your sharding database and dynamically controlling these distributed database system's workloads, such as SQL audit, read/writing splitting, authority, and so on.

For example, you may use `CREATE TABLE t_order ()` SQL to create a new table in MySQL. With ShardingSphere-Proxy, `CREATE SHARDING TABLE RULE t-order ()` will help you create a sharding table in your newly upgraded distributed database system.

ShardingSphere-On-Cloud

So far, we've solved the sharding problem, but how do we make it work on Kubernetes? ShardingSphere-on-cloud provides ShardingSphere-Operator-Chart and ShardingSphere-Chart to help users deploy ShardingSphere-Proxy and ShardingSphere-Operator clusters on Kubernetes.

ShardingSphere-Chart and ShardingSphere-Operator-Chart

Two Charts help users deploy the ShardingSphere-Proxy cluster, including proxies, governance center, and Database connection driver, and ShardingSphere-Operator using helm commands.

ShardingSphere-Operator

ShardingSphere-Operator is a predefined CustomResourceDefinition that describes ShardingSphere-Proxy Deployment on Kubernetes. Currently, this operator provides HPA (Horizontal Pod Autoscaler) based on CPU metric and ensures ShardingSphere-Proxy high availability to maintain the desired replica number. Thanks to community feedback, throughout development iterations, we've found out that autoscaling and availability are our users' foremost concerns. In the future, the open-source community will release even more useful features.

New solution

Users can easily deploy and manage ShardingSphere clusters and create their distributed database system on Kubernetes using these tools, regardless of where their monolithic databases reside.

As previously stated, a database is made up of computing nodes and storage nodes. A distributed database will divide and distribute these nodes. As a result, you can use your existing databases as the new distributed database system's storage nodes. The highlight of this solution is adopting a flexible computing-storage-splitting architecture, utilizing Kubernetes to manage stateless computing nodes, allowing your database to reside anywhere and drastically reducing upgrading costs.

ShardingSphere-Proxy will act as global computing nodes to handle user requests, obtain local resultSet from the sharded storage nodes, and compute the final resultSet for users. This means there is no need to do dangerous manipulation work on your database clusters. You only have to import ShardingSphere into your database infrastructure layer and combine databases and ShardingSphere to make it a distributed database system.

ShardingSphere-Proxy is a stateless application that is best suited to being managed on Kubernetes. As a stateful application, your databases can run on Kubernetes, any cloud, or on-premise.

On the other hand, ShardingSphere-Operator serves as a manual operator working on Kubernetes to offer availability and auto-scaling features for the ShardingSphere-Proxy cluster. Users can scale-in or scale-out ShardingSphere-Proxy (computing nodes) and Databases (storage nodes) as needed. For example, some users simply want more computing power, and ShardingSphere-Operator will automatically scale out ShardingSphere-Proxy in seconds. Others may discover that they require more storage capacity; in this case, they simply need to spin up more empty database instances and execute a DistSQL command. ShardingSphere-Proxy will reshard the data across these old and new databases to improve capacity and performance.

Finally, ShardingSphere can assist users in resolving the issue of smoothly sharding existing database clusters and taking them into Kubernetes in a more native manner. Instead of focusing on how to fundamentally break the current database infrastructure and seeking a new and suitable distributed database that can be managed efficiently on Kubernetes as a stateful application, why don't we consider this issue from the other side. How can we make this distributed database system more stateless and leverage the existing database clusters? Let me show you two examples of real-world scenarios.

Databases on Kubernetes

Consider that you have already deployed databases, such as MySQL and PostgreSQL, to Kubernetes using Helm charts or other methods and that you are now only using ShardingSphere charts to deploy ShardingSphere-Proxy and ShardingSphere-Operator clusters.

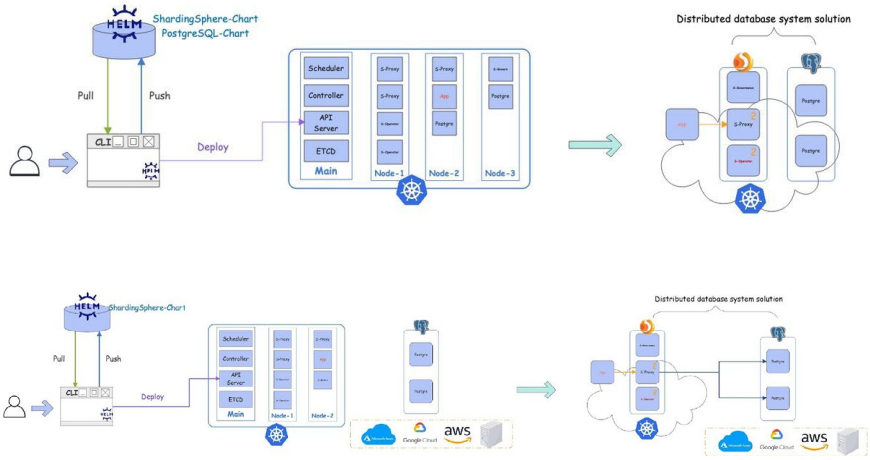
Once the computing nodes have been deployed, we connect to ShardingSphere-Proxy in the original way to use DistSQL to make Proxy aware of databases. Finally, the distributed computing nodes connect the storage nodes to form the final distributed database solution.

Databases on cloud or on-premise

If you have databases on the cloud or on-premises, the deployment architecture will be as shown in the image below. The computing nodes, ShardingSphere-Operator and ShardingSphere-Proxy, are running on Kubernetes, but your databases, the storage nodes, are located outside of Kubernetes.

Pros and Cons

We've seen a high-level introduction to ShardingSphere and some real-world examples of deployment. Let me summarize its pros and cons based on these real-world cases and the previous solution introduction to help you decide whether to adopt it based on your particular case.



Pros

- Leverage your existing database capability

Instead of blowing up all your legacy database architecture, it's a smooth and safe way to own a distributed database system.

- Migrate efficiently and steadily

With almost no downtime, ShardingSphere offers a migration process that allows you to move and shard your databases simultaneously.

- Traditional SQL-like approach to harness it

ShardingSphere's DistSQL enables you to use the distributed database system's features, such as sharding, data encryption, traffic governance, and so on, in a database native manner, i.e., SQL Flexible auto-scaling feature for separate computing and storage power

You can scale-in or scale-out ShardingSphere-Proxy and Databases separately and flexibly depending on your needs, thanks to a non-aggressive computing-storage splitting architecture.

- More cloud-native running and governance way

ShardingSphere-Proxy is much easier to manage and natively deploy on Kubernetes because it is essentially a type of stateless global computing server that also acts as a database server.

- Multi-cloud or cross-cloud

As stateful storage nodes, databases can reside on Kubernetes or on any cloud to avoid a single cloud platform lock-in. With ShardingSphere to connect your nodes, you will get a distributed database system.

- More necessary features around databases

ShardingSphere is a database ecosystem that provides data encryption, authentication, read/write splitting, SQL auditing, and other useful features. Users gradually discover their advantages, regardless of sharding.

- More clients for you to choose from, or a hybrid one

ShardingSphere offers two clients based on user requirements: ShardingSphere-Proxy and ShardingSphere-JDBC. Generally, ShardingSphere-JDBC has better performance than ShardingSphere-Proxy, whereas ShardingSphere-Proxy supports all development languages and Database management capabilities. A hybrid architecture with ShardingSphere-JDBC and ShardingSphere-Proxy is also a good way to reconcile their capabilities.

- Open-source support

Apache ShardingSphere is one of the Apache Foundation's Top-Level projects. It has been open-sourced for over 5 years. As a mature community, it is a high-quality project with many user cases, detailed documentation, and strong community support.

Cons

- Distributed transactions

Even in a distributed database system, the transaction is critical. However, because

```
aling
## @param automaticScaling.maxInstance ShardingSphere-Proxy maximum number of scaled-out replicas
## @param automaticScaling.minInstance ShardingSphere-Proxy has a minimum number of boot replicas, and the shrinkage will not be less than this number of replicas
##
automaticScaling:
  enable: true
  scaleUpWindows: 30
  scaleDownWindows: 30
  target: 20
  maxInstance: 4
  minInstance: 1
## @param resources ShardingSphere-Proxy starts the requirement resource, and after opening automaticScaling, the resource of the request multiplied by the percent
of target is used to trigger the scaling action
## e.g:
## resources:
##   limits:
##     cpu: 2
##   requests:
##     cpu: 2
##
-- INSERT --
```

```
  ## @param serverConfig.mode.repository.props.operationTimeoutMilliseconds Milliseconds of operation timeout
  ## @param serverConfig.mode.repository.props.retryIntervalMilliseconds Milliseconds of retry interval
  ## @param serverConfig.mode.repository.props.timeToLiveSeconds Seconds of ephemeral data live
  ## @param serverConfig.mode.repository.type Type of persist repository. Now only support ZooKeeper
  ## @param serverConfig.props.proxy-frontend-database-protocol-type Default startup protocol
  mode:
    repository:
      props:
        maxRetries: 3
        namespace: governance_ds
        operationTimeoutMilliseconds: 5000
        retryIntervalMilliseconds: 500
        server-lists: "{{ printf \"%s-zookeeper.%s:2181\" .Release.Name .Release.Namespace }}"
        timeToLiveSeconds: 600
      type: ZooKeeper
    type: Cluster
  props:
    proxy-frontend-database-protocol-type: PostgreSQL
## @section ZooKeeper chart parameters
```

Pods 4 items Namespace: sharding-test Search Pods...

Name	Namespace	Containers	Restarts	Controlled ...	Node	QoS	Age	Status
shardingsphere-cluster-shardi...	sharding-test		0	ReplicaSet	10.9.74.64	Burstable	6m12s	Running
shardingsphere-cluster-zooke...	sharding-test		0	StatefulSet	10.9.74.64	Burstable	6m13s	Running
shardingsphere-operator-5764...	sharding-test		0	ReplicaSet	10.9.92.106	BestEffort	7m51s	Running
shardingsphere-operator-5764...	sharding-test		0	ReplicaSet	10.9.74.64	BestEffort	7m51s	Running

this tech architecture was not developed from the storage layer, it currently relies on the XA protocol to coordinate the transaction handling of various data sources. It is not, however, a perfect and comprehensive distributed transaction solution.

- SQL-compatibility issue

Some SQL queries work well in a storage node (database) but not in this new distributed system.

This is a difficult issue to achieve 100% support, but thanks to the open-source community, we're getting close.

- Consistent global backup

Although ShardingSphere defines itself as a computing database server, many users prefer to think of it and their databases as a distributed database. As a result, people must think about obtaining a consistent global backup of this distributed database system. ShardingSphere is working on such a feature, but it is not yet supported (release 5.2.1). Users may require manual or RDS backups of these databases.

- Some overhead

Each request will be received by ShardingSphere, calculated, and forwarded to the storage nodes. It is unavoidable that the overhead for each query will increase. This mechanism happens in any distributed database compared to a monolithic one.

Hands-on

This section demonstrates how to use ShardingSphere and PostgreSQL RDS to build a distributed PostgreSQL database that will allow users to shard data across two PostgreSQL instances.

For this demonstration, ShardingSphere-Proxy runs on Kubernetes, and PostgreSQL RDS runs on AWS. The deployment architecture is depicted in the following figure.

This demo will include the following major sections:

- Deploy the ShardingSphere-Proxy cluster and ShardingSphere-Operator.
- Create a distributed database and table using Distributed SQL.
- Test the Scaling and HA of the ShardingSphere-Proxy cluster (computing nodes).

Prepare database RDS

We need to create two PostgreSQL RDS instances on AWS or any other cloud. They will act as storage nodes.

Deploy ShardingSphere-Operator

1. Download the repo, and create a namespace named `sharding-test` on Kubernetes.

```
git clone https://
github.com/apache/
shardingsphere-on-cloud
kubectl create ns
sharding-test
cd charts/shardingsphere-
operator
helm dependency build
cd ../
helm install shardingsphere-
operator shardingsphere-
operator -n sharding-test
cd shardingsphere-
operator-cluster
vim values.yaml
helm dependency build
cd ..
helm install shardingsphere-
cluster shardingsphere-
operator-cluster -n
sharding-test
```

2. Change `automaticScaling: true` and `proxy-frontend-database-protocol-type: PostgreSQL` in values.yaml of `shardingsphere-operator-cluster` and deploy it.
3. Following these operations, you will create a ShardingSphere-Proxy cluster containing 1 Proxy instance, 2 Operator instances, and 1 Proxy governance instance showing as follows.

Create a sharding table by using Distributed SQL

1. Login to ShardingSphere Proxy and add PostgreSQL instances to Proxy.

```
kubectl port-forward
--namespace sharding-
test svc/shardingsphere-
cluster-shardingsphere-
operator-cluster
3307:3307
psql --host 127.0.0.1 -U
root -p 3307 -d postgres
```

```
kubectl port-forward
--namespace sharding-
test svc/shardingsphere-
cluster-shardingsphere-
operator-cluster
3307:3307
psql --host 127.0.0.1 -U
root -p 3307 -d postgres
```

2. Execute DistSQL to create a sharding table `t_user` with MOD (user_id, 4), and show the actual tables of this logic table `t_user`.
3. Insert some test rows and do a query on ShardingSphere-Proxy to get the merged final result.
4. Login to two PostgreSQL instances to get their local results.

This simple test will help you understand that ShardingSphere can help you manage and shard your databases. People don't need to care about the separate data in different shards.

Test the Scaling and HA of the ShardingSphere-Proxy cluster (computing nodes)

If you discover that the TPS (transactions per second) or QPS (queries per second) of this new system are extremely high and users complain that it takes too long to open a webpage, it's time to upgrade your database system's computing power.

Compared to other distributed database systems, ShardingSphere-Proxy is the simplest way to increase computing nodes. ShardingSphere-Operator can ensure ShardingSphere-Proxy availability and autoscale them based on CPU metrics. Furthermore, by modifying its specifications, it is possible to make it scale-in or scale-out, just as follows:

You will receive two ShardingSphere-Proxy instances after upgrading the release. This implies that you have more computing power.

If, as mentioned above, you require more storage capacity, you can take the following steps.

Launch additional PostgreSQL instances in the cloud or on-premises.

Add these new storage nodes to the ShardingSphere-Proxy.

Run distributed SQL to allow ShardingSphere to assist you with resharding.

Wrap-up

The focus of this article is a new sharding database architecture on Kubernetes that leverages your existing monolithic databases, allowing the DevOps team to evolve their database infrastructure to a modern one efficiently and fluently.

The database computing-storage split is a vintage architecture

```
postgres=>
postgres=> ADD RESOURCE ds_0 (
postgres(>     HOST="postgresql-1                    amazonaws.com.cn",
postgres(>     PORT=5432,
postgres(>     DB="postgres",
postgres(>     USER="pos
postgres(>     PASSWORD=
postgres(> ), ds_1 (
postgres(>     HOST="postgresql-2                    mazonaws.com.cn",
postgres(>     PORT=5432,
postgres(>     DB="postgres",
postgres(>     USER="postgres"
postgres(>     PASSWORD=
postgres(> );
SUCCESS
postgres=>
```

```
postgres=>
postgres=> CREATE SHARDING TABLE RULE t_user (
postgres(> RESOURCES(ds_0,ds_1),
postgres(> SHARDING_COLUMN=user_id,TYPE(NAME="hash_mod",PROPERTIES("sharding-count"="4"))
postgres(> );
SUCCESS
postgres=> CREATE TABLE t_user (
postgres(>    user_id int4,
postgres(>    user_name varchar(32),
postgres(>    tel varchar(32),
postgres(>    age varchar(32)
postgres(> );
CREATE TABLE
postgres=> SHOW SHARDING TABLE NODES;
  name  |                                nodes
--------+------------------------------------------------------------------
 t_user | ds_0.t_user_0, ds_1.t_user_1, ds_0.t_user_2, ds_1.t_user_3
(1 row)

postgres=>
```

```
tristaQTristas-MacPro ~ %>PGPASSWORD                psql -hpostgresql-1                amazonaws.com.cn -Up

psql (13.2, server 13.7)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=> select * from t_user_0;
 user_id | user_name |   tel    | age
---------+-----------+----------+-----
       4 | name4     | tel44444 | 46
(1 row)

postgres=> select * from t_user_2;
 user_id | user_name |   tel    | age
---------+-----------+----------+-----
       2 | name2     | tel22222 | 23
(1 row)

postgres=>
```

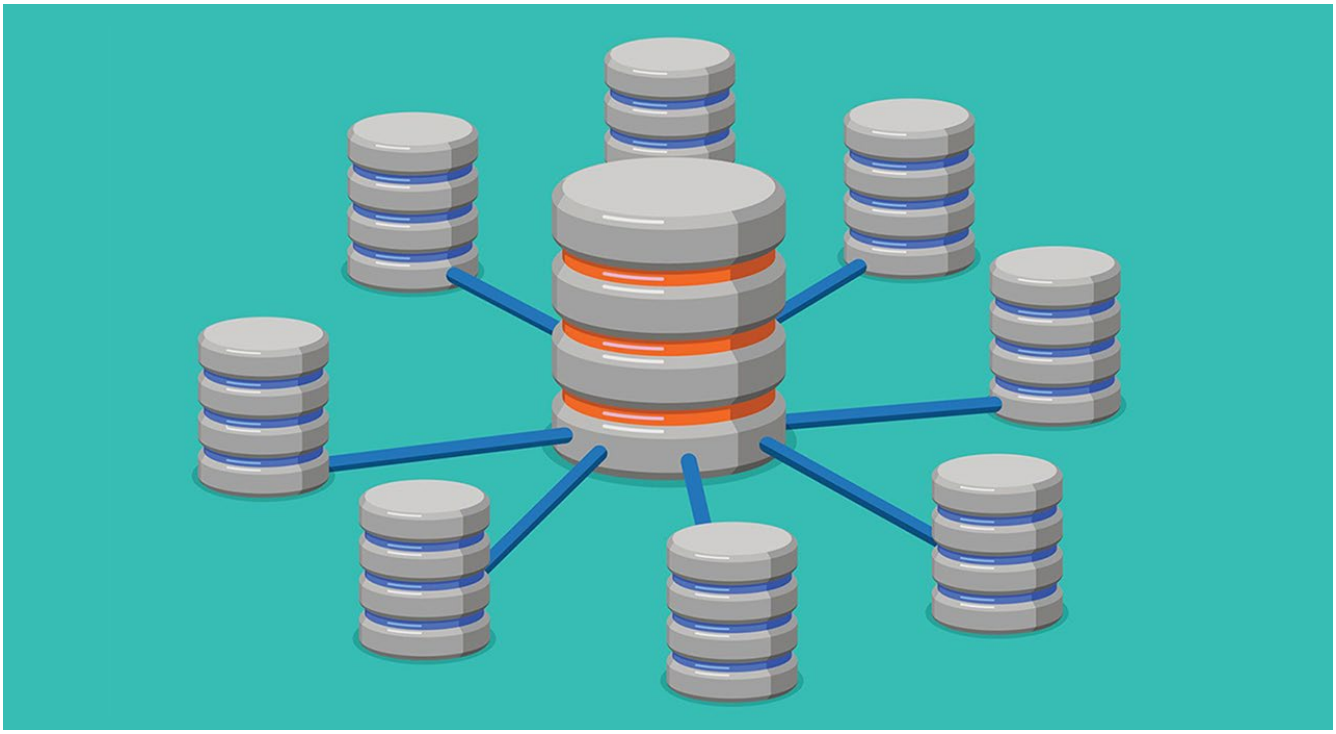
```
postgres=> ^D\q
trista@Tristas-MacPro ~ %>PGPASSWORD=                psql -hpostgresql-2.                .amazonaws.com.cn -U
psql (13.2, server 13.7)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=> select * from t_user_1;
 user_id | user_name |   tel    | age
---------+-----------+----------+-----
       1 | name1     | tel11111 | 22
(1 row)

postgres=> select * from t_user_3;
 user_id | user_name |   tel    | age
---------+-----------+----------+-----
       3 | name3     | tel33333 | 35
(1 row)

postgres=>
```

```
User-supplied values only
auto
1  automaticScaling:
2    enable: true
3    maxInstance: 4
4    minInstance: 2
5    scaleDownWindows: 30
6    scaleUpWindows: 30
7    target: 20
8  common:
9    exampleValue: common-chart
10   global: {}
11 imagePullSecrets: []
12 mySQLDriver:
13   version: 5.1.47
14 proxyVersion: 5.2.0
15 replicaCount: "3"
```

1->2

that is re-interpreted and fully leveraged on Kubernetes today to help users address the governance issue of the stateful database on Kubernetes.

These days, distributed databases, cloud computing, open source, big data, and modern digital transformation are common buzzwords. But they represent useful new concepts, ideas, and solutions that address production concerns and needs. As I always recommend to our end-users, look forward to welcoming new ideas, learning their pros and cons, and then choosing the best one for your specific situation, as there is no such thing as a perfect solution.



Design Pattern Proposal for Autoscaling Stateful Systems

by **Rogerio Robetti,** Software Engineer

Considering the trend in software engineering for segregation and the ever-growing need for scalability, a common challenge arose where autoscaling stateful systems (databases being most common) became complex and, at times, unfeasible. That has led to many companies choosing to over-provision such systems so that, based on expected loads, the systems can cope with the highest expected demands.

This, of course, brings problems as over-provisioning resources is costly. It does not guarantee reliability, as sudden surges of demand or a DOS attack can easily compromise the expected loads. This article aims to dig deeper into the challenges faced when attempting to auto-scale stateful systems and proposes an opinionated design solution on how to address many of those challenges through a mix of existing and novel approaches.

Recapitulating a Little

If we look at how software engineering evolved historically, we see a few significant milestones in terms of building software and the restrictions and expectations of users.

Suppose we took a concise, historical tour of software engineering. We would start with the mainframes and their centralized approach with large servers, pass by the desktop applications, including the Client-Server advent. We would then move into the web applications revolution and the multiple phases within it, from large monoliths to modern microservices.

In all that history, we would see clear trends for segregation. Vertical segregation is where we divide systems by concerns (or context), generally having the

database(s) separated from our applications and in many cases having our UI separated from our business or service layer. And there is horizontal segregation, where the systems can be scaled out by provisioning more nodes to support rising demands, even automatically, with the help of tools like an orchestrator such as Kubernetes.

This segregation hunger led to the inception of architectural approaches like the shared-nothing architecture in which an application is built to not hold state in itself, becoming what we know as a stateless application and making it a lot simpler to scale out. That sounds like an awesome solution, but soon, engineers realized that there is seldom a "truly stateless" application—one that doesn't hold state at all.

What happens instead is that parts of the application (usually the services or microservices) are built stateless. However, they still rely on stateful systems such as databases to hold state on their behalf. This is the central theme of this article. I will discuss this common challenge in software engineering, how to efficiently auto-scale stateful systems in modern applications?

Targeted Use Cases

This article does not target stateful systems that hold state in web servers.

It provides a foundational design upon which software engineers can potentially build their own databases, explicitly addressing the concerns of getting a storage system that works in a single node and turning it into a distributed system with opinionated autoscaling capabilities. For example: Imagine a microservice architecture for online ordering, like the diagram in Figure 1.

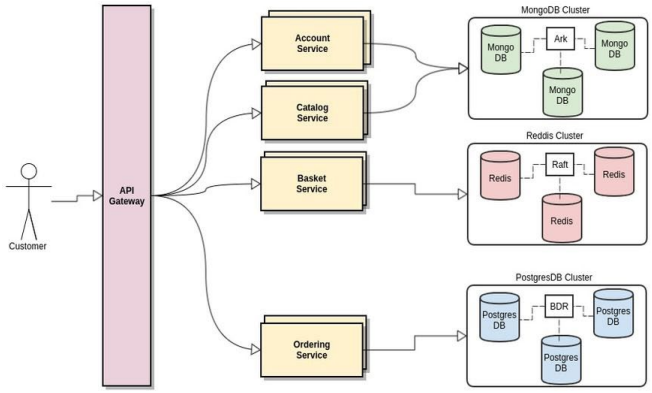


Figure 1: Online shopping example use case

Let's say there are requirements that lead to the conclusion that using RocksDB as a key value storage engine for your project requirements is better than Redis. The trouble is that RocksDB is only a storage engine—it can only be deployed in one node as is. Let's assume your requirements are for a new system that will have a global outreach and require significant extensibility. This article is a good start on how you can go about transforming a single-node storage engine like RocksDB into a distributed and auto-scalable application.

Note that RocksDB is just an example here. It could be any other storage engine or tool like Apache Lucene for text indexation or just in-memory storage without any engine, for that matter. This design pattern is generic and can apply to any storage engine, language, and data type and structure. Also, the designs shared here could be used to give autoscaling capabilities to any of the databases listed (Mongo, Redis, Postgres).

Stateful System Definition

A stateful system is a system where state must be handled. In modern web applications, the job of holding and managing state is typically done by a database, but it can also be a web server, for instance, when user sessions are stored in the web server's memory.

In websites, a typical example of state to be managed is a user's shopping cart (see Figure 1). The cart has to be saved between HTTP requests so that

when the user finishes shopping and proceeds to checkout and payment, the cart is in the correct state with the right products and amounts. The cart information has to be stored somewhere, and that somewhere is a stateful system. In our use case example in Figure 1, the Mongo DB cluster, Redis cluster, and Postgres cluster are the stateful systems.

Autoscaling—The Problems

When we start thinking about autoscaling stateful systems, the main concerns that come to mind are "When should we scale? What should be the trigger? How should this scale take place? How will we move the data? How will the nodes achieve consensus?"

Here are the main problems I will cover in this article:

Consensus

Every distributed system that holds state has to agree on the next valid state of the cluster. This is a well-researched area that we can refer to, we can take as examples Ark, Raft, and BDR, which are the consensus algorithms used by MongoDB, Redis, and Postgres, respectively. These are the databases picked for our example of online ordering presented before in Figure 1.

The need for consensus arises in software applications every time a cluster must agree on the next value of a record stored, being the most notorious use cases of database implementations. In this article, I will propose something new to make Raft consensus smarter when selecting a new leader.

Autoscaling

In stateful systems, even though there is an increase in the offerings of systems that can auto-scale (managed instances of databases in cloud providers, for example), in practice, we see a struggle within companies to be able to implement such scenarios, which appear to be caused by:

- Lack of transparency: When we look at mainstream cloud providers like AWS and Azure it is easy enough to configure autoscaling, but it is not disclosed how exactly the autoscaling will happen, and knowing exactly how it works is vital for critical scenarios with large datasets—questions like "When is the data moved? What is the strategy?" These should be clearly stated on the product offerings.
- Inexistence of public patterns to autoscaling: no publicly available patterns currently exist on how to auto-scale stateful systems.

Data migration lag

A common way of scaling a system is adding new nodes to the cluster. However, when we talk about stateful clusters, time is needed to achieve synchrony with all the data already held by the other nodes—in some cases, these amounts can be massive.

Let's take our online shopping example (Figure 1). If we talk about large organizations that operate in many geographical areas, the number of records can reach billions. At this scale, a clear and efficient approach to synchronizing and moving data is paramount.

Fast vs. slow demand increase

There are two instances where more capacity is required:

1. A steady, medium- to long-term gradual increase in demand. In our ordering example, this would be the numbers of consumers growing consistently over a period of time.
2. A sudden surge in demand that may not be predictable and may risk the service becoming unavailable, which happens when the system is under a DoS attack, for instance.

Closed solutions

There needs to be more publicly available design patterns for the listed problems above. It is not a good idea to simply believe that the cloud provider's approach will work; even if it does, you may find yourself locked to that particular provider, which is not ideal.

The Vision

My vision is a public proposal for a generic, replicable, opinionated approach for autoscaling stateful systems aiming to automatically scale up (vertical) and scale out (horizontal) from a single node up to hundreds or thousands of nodes in a single cluster with minimum configuration and interference of the operator. The solutions presented in this article are theoretical at this stage and require implementation and testing.

Core Principles

Data-type agnostic

The designs are not bound to any specific data types; in other words, you can use the same solutions to handle JSON objects, serialized data, streams, blobs, or other types of data.

The Writer writes and the Reader reads!

The cluster leader responsible for writing new states only does the write operations—it does not perform reads. Read replicas, on the other hand, do all the read operations and never do writes.

Proxy as part of the cluster

You must have a proxy implementation that does not serve reads nor writes as part of the stateful cluster; this enables the cluster to use this proxy as a node that also slowly synchronizes the data of the cluster, eventually becoming ready to serve read or write requests if required.

Trigger autoscaling by average response time

Most autoscaling approaches used by cloud providers use CPU and memory thresholds, but that is not the best way to deliver the best client experience. Even though the resources may be under stress at certain times, it does not necessarily mean that the user is feeling it on the other end; the system may be using 99% of CPU and delivering requests in good time.

Using average response time as the primary trigger changes the decision-making on when to scale, taking the client's perspective of the system's performance.

A priory sharding labeling

Labeling each object/record stored with a shard ID avoids the costs of doing it when the pressure is higher on the system so that whenever you need to start sharding, the labels are already set, and no intervention is required.

Here we go… The designs

In this proposal for autoscaling stateful systems, there are three different actors. Each actor will have specific responsibilities in the cluster. It is worth pointing out that each actor proposed does not necessarily need to be running in its own process or node, and that has much to do with the ability to run the system in a single node, which raises concerns for production purposes. Still, it is paramount for testing, POC, or even some MVP setups.

Without further ado, let's look at each actor and its responsibilities.

Writer (Leader)

The Writer (or leader) is the actor responsible for taking care of the write operations. It writes the new state in its own storage and is responsible for replicating the data to the Read Replica actor(s). There is only one Writer per shard (I will elaborate more on sharding later).

The Writer is the leader of a consensus, and all the write operations are executed through it. No read operations are executed through the Writer.

Read Replica

The Read Replica is the actor that serves all the read requests. It contains a replica of the data from the Writer (leader), except when both Writer and Reader are running in a single node/process, in which case they can share the same storage.

When the consensus protocol elects a new leader, each Read Replica is responsible for opening a multiplex pipe of communication with the leader that remains open until one of the nodes dies or the connection is broken by a network partition. This is important to speed up communication between the

actors/nodes by avoiding the overhead of opening and closing connections.

Load Manager

The Load Manager actor serves as a gateway and load balancer, sending write requests to the leader and read requests to the replicas. It is also a back pressure mechanism that can accept thousands of inbound connections. Still, it maintains the number of parallel threads against the target (Read Replica or Writer) limited to a configurable number, therefore keeping the pressure on these actors controlled. The Apache Tomcat Nio Connector inspires this. And it is vital to defend the cluster against sudden increases in loads or DOS attacks, in which case the pressure will be absorbed by the Load Manager, keeping the read and write actors safe and receiving a steady flow of requests.

The Load Manager is also responsible for routing write requests to the correct shard and sending query requests to each shard when aggregation is required. It also aggregates and sorts the results before returning them to the client, reducing the amount of work required from the Read Replica. The Load Manager addresses these concerns in the same manner as a database proxy would, with the difference that it is part of the cluster and not an external added component in this case.

There can be more than one instance of the Load Manager. Every time the Load Manager instance(s) reach a configurable threshold of CPU and/or memory, a new Load Manager is provisioned, generating a cluster of Load Managers that should have their own independent consensus mechanism.

High-Level Design

The basic interaction between the actors proposed in this design pattern is expressed in the diagram presented in Figure 2:

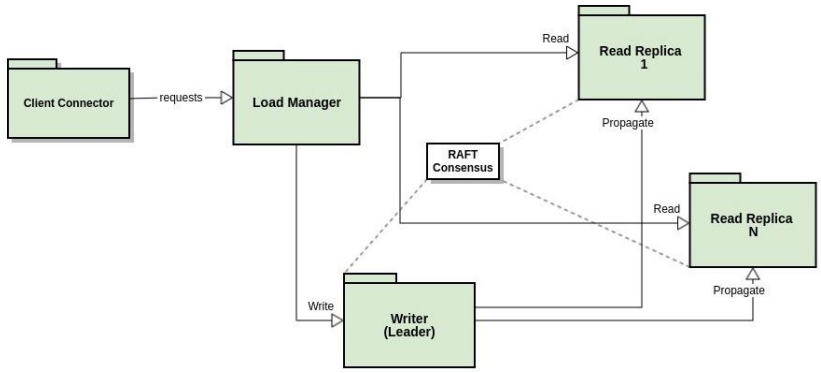


Figure 2: Basic design of the proposed solution

Why Raft?

Raft is a well-known and battle-tested consensus algorithm sometimes comparable to Paxos in performance but a lot simpler, as explained in this Paper review: Raft vs. Paxos.

The fact that Raft only has one leader at a given time is very important for the strategies I will describe here.

Smart Raft

I propose modifying the Raft protocol to increase the overall performance of the cluster by making Raft aware of node differences and selecting the "bigger" node available as a leader. This is especially important when talking about autoscaling write operations, as we only have one leader at a given time. The most obvious way to increase its capabilities is to provision a "bigger" leader and therefore trigger a new election. Raft then needs to be able to identify and elect the new "bigger" node as leader.

An alternative approach is to modify Raft to be able to receive a "switch to" instruction which would cause the cluster to switch the leader to the specified "bigger" node.

The latter approach is preferred, as it would be a smaller change to protocol and would decouple the task of switching the leader from the switch logic.

Bigger in this context is related to CPU, memory, storage technology (SSD), or other resources—it all depends on the purpose of the stateful cluster. If the cluster is intended to serve complex calculations, bigger probably means more CPU, but if it serves requests, it might mean more memory and better storage technology.

Autoscaling Strategy

The phases of scalability that I explain next are named Mach* in an allusion to the speed term used to describe objects as fast or faster than the speed of sound. In this article, each Mach stage effectively implies the cluster's number of nodes.

*Mach terminology is only used in this article—it is not an industry naming convention.

Configurable Scaling Triggers

It is important to understand when is the right time to auto-scale/auto-descale. For instance, it is a bad idea to attempt to scale when the system is under a lot of pressure, and that is why the back pressure offered by the Load Manager actor is so significant.

I will focus on scenarios where the increase in demand happens gradually over time. For that, there are two essential types of configurations that can be used to trigger autoscaling.

In both scenarios, it is the responsibility of the Load Manager to recognize that a trigger should happen and emit a notification to the operator. (Note: the operator may be a human or a software system, preferably the latter.)

The configurable triggers are:

1. By average response time threshold

One of the jobs of the Load Manager is to monitor the average response time of requests. When the average response time of requests reaches a threshold, a trigger for scale is issued.

Example of scaling **UP** configuration: 3 seconds/request on average in the last 60 min.

Example of scaling **DOWN** configuration: < 0.5 seconds/request on average in the last 60 min.

2. By timeout threshold

The timeout threshold is a percentage of requests that may time out within a given period of time before an auto-scale signal is issued.

Example of scaling **UP** configuration: > 1% of requests timed out in the last 5 min.

Example of scaling **DOWN** configuration: Not recommended for timeout threshold as no level of timeouts is advisable to be safe for a downgrade in the cluster.

Refer back to Figure 1, and assume that we have replaced Redis with our auto-scalable RocksDB. The new RocksDB auto-scalable cluster would scale up and down based on these thresholds being breached without any interference of the human operator/admin.

Notes before continuing reading:

- The following examples for each Mach stage focus on increasing read capabilities, which will indirectly increase write capabilities as per the following workload segregation. To scale targeted write capacity, a special section will be dedicated after Mach IV.
- A “Node” in this text means a participant in the cluster and an individual process running—it does not necessarily indicate different hardware.
- The total number of replicas to be provisioned before starting to create shards has to be configurable.
- The cluster can start in any desired setup, being Mach IV the minimum setup recommended for production purposes.
- All three actors' (Writer, Read Replica, and Load Manager) implementations are modular

and always deployed in the nodes. When a node is labeled as Writer, for example, it means that only the Writer module is enabled on it, but it still contains the disabled modules of Read Replica and Load Manager, which is what allows nodes to switch responsibilities if needed.

Mach I

This is the initial state of the system/cluster; all its actors are active in a single node, it is indicated for small use cases or testing pipeline scenarios, and it looks like the diagram in Figure 3.

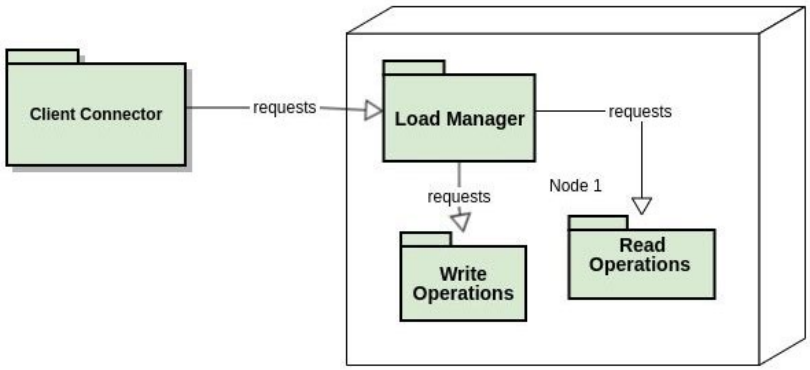


Figure 3: Single node deployment—Mach I.

In Mach I, all components are deployed as a single process in a single node. This single node is responsible for managing all the read and write requests.

Use case: Mostly recommended for testing scenarios—not ideal for production.

Consensus and replication

At Mach I, no consensus or replication is required as the components communicate in memory module to the module.

Mach II

In Mach II, the cluster counts with two nodes deployed—the second node on a scale out is always a Load Manager. That is to ensure back pressure protection on the node responding to the requests and to allow the new node to gradually synchronize the data.

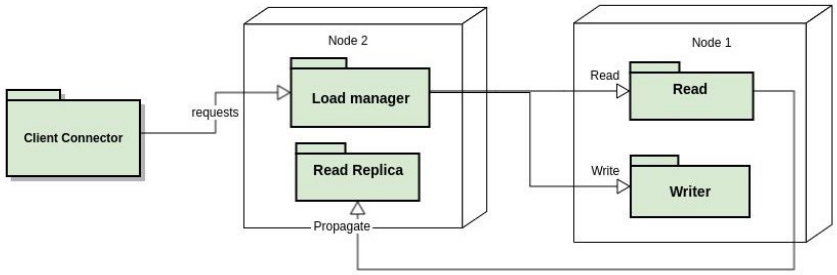


Figure 4: Two-node deployment—Mach II.

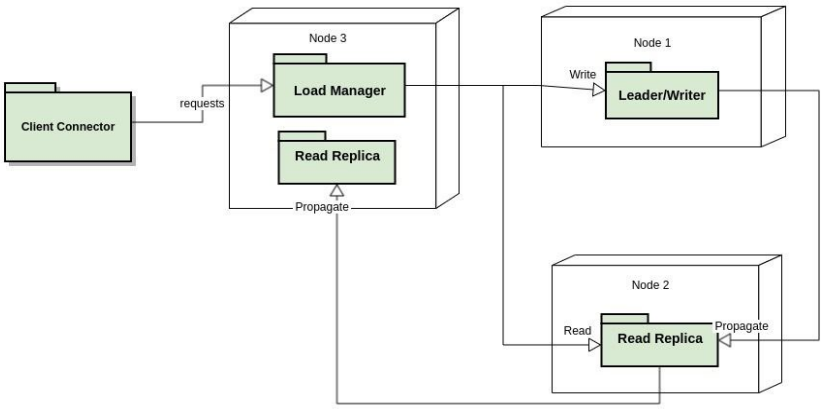


Figure 5: Three-node deployment—Mach III.

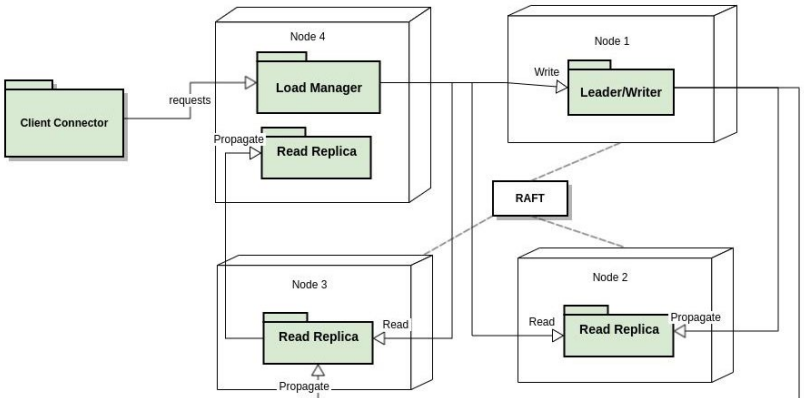


Figure 6: Four-node deployment—Mach IV.

The topology is represented in the diagram in Figure 4.

Consensus and replication

At Mach II, consensus is not required as it is impossible to establish Raft consensus with less than three nodes.

Replication will happen from the Read Replica module deployed on node two to the Read Replica module that also runs in Node 2 alongside the Load Manager. The point to be noted is that the Read Replica in Node 2 (the Load Manager node) does not serve requests; this design decision is to always have a node "nearly" synchronized, which can enter in operation as an extra Read Replica or a leader node extremely fast, as I will explain in Mach III.

Use case: Can be used in scenarios where reliability is not so important and low operational costs are.

Mach III

Mach III indicates that an extra node has been added to the cluster, which now has three nodes in total.

The new node will always enter the cluster as a new Load Manager, the clients will be redirected to the new Load Manager, and the Load Manager provisioned in Mach II takes the role of a Read Replica.

The diagram in Figure 5 represents the Mach III scenario.

Consensus and replication

No consensus is required yet because besides having three nodes, only Node 1 and 2 can actively serve requests.

Use case: Already offers a good performance by separating read and write operations in different nodes. However, if a node fails due to the lack of a second Read Replica node, this would force the Writer to start serving read requests until a new node is provisioned.

Resiliency strategy

Leader crashes: The cluster returns to Mach II topology with Node 2 assuming the write and read operations until a new node is added to the cluster.

Node 2 (Read Replica) goes down: The Leader/Writer starts serving read requests until a new node is added back to the cluster.

Node 3 (Load Manager) goes down: Node 2 will start operating as Load Manager and no longer as Read Replica, and Node 1 will perform the write and read operations.

In all three scenarios, a signal is sent to the operator requesting the provisioning of a new node to replace the fallen one, and the new node always enters the cluster as a Load Manager.

Mach IV

At this stage, there are four nodes in the cluster, and there is a second Read Replica. The deployment will look like the diagram in Figure 6:

Use case: Minimum setup indicated for production workloads, good performance, and good response in case of a failing node.

Consensus and replication

At Mach IV, consensus is introduced, but no election is held initially. Node 1 will remain as leader and centralize the write operations not to waste time switching to a new leader. It is vital that the Raft implementation is extended to support this arrangement. It is also crucial that if the leader goes down, Node 2 or 3 becomes the new leader, reverting to Mach III topology. The Load Manager is responsible for making such a decision.

Leader crashes: Raft protocol can't have an election with only two nodes remaining in the cluster, so the Load Manager will randomly pick a new leader between Nodes 2 and 3 that contain a Read Replica running.

From Mach IV, the resiliency strategy for Replicas and Load Manager nodes is the same as Mach III, and new replica nodes can be added to the cluster to a configurable max number before shards are created.

Mach V, VI, and so on…

New Read Replicas keep being added to the cluster until a configured max number where sharding starts to take place. Note that adding Replicas means always adding a new Load Manager and taking the place of the previous Load Manager. The previous Load Manager then joins the Raft consensus and starts serving read requests.

Use case: As the cluster grows larger, it becomes more reliable since the failure of one node is not as dramatic as in smaller setups.

Resiliency strategy

Leader crashes: Raft elects a new leader among the Read Replicas available, and the new leader communicates to the Load Manager of its election.

Read Replica crashes: A new node is requested to be added to the cluster.

Load Manager crashes: The latest Read Replica added to the cluster assumes the Load Manager responsibilities, no longer serving read requests itself until a new node is provisioned, and as always, it enters the cluster as a Load Manager.

Read Intensive vs. Write Intensive Scenarios

Until Mach IV, the autoscaling neglects the characteristics of the load to the detriment of having what is considered the minimum replication setup for a reliable system. The system will continue to auto-scale, but now differently. It will now consider if the usual load is Read Intensive (80% or more reads), Write Intensive (80% or more write operations), or balanced (all other scenarios). This may not account for exceptional use cases but remember that this is an opinionated pattern and that, if necessary, it can be adapted for special circumstances. The target here is to address most of the use cases—not all.

Scaling Read Intensive scenarios

This requires the simplest strategy where new nodes keep being added in the same manner as Mach I to Mach VII, which represents seven nodes (1 Read Manager, 1 Leader Writer, and 5 Read Replicas) where the cluster will then start using shard labels (more on it soon) to create shards of the existing data and divide the load of incoming requests between the newly created shards.

Before it operates in two shards, a new node is added to the cluster to support the topology introduced in Figure 7, with a minimum of eight nodes.

Use case: Large-scale scenarios with ever-growing demands.

The diagram in Figure 7 represents a setup with two shards, each shard containing one leader and two Read Replicas, plus an extra replica in a Load Manager node.

This can be further scaled into a third or fourth shard when enough nodes are provisioned in each shard. For example, if Shard 2 scales to seven nodes, the next step is to add a new node and divide it into two shards.

Each Load Manager holds a replica of only one shard, and that is the shard the Load

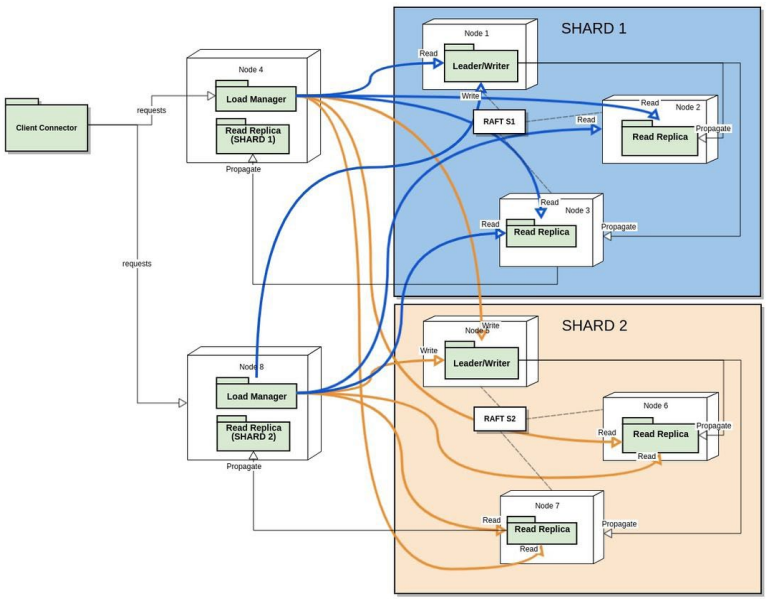


Figure 7: Sharding topology example for two shards.

Manager can assume the read or writer role if needed, but to be able to fulfill requests from clients, it needs to write and read from all shards.

Scaling Write Intensive scenarios

For the scenarios where write operations are the focus or where sensible degradation is observed on write operations, there are two ways to scale the writing capacity of the cluster:

1. Provision of a bigger leader

The first approach is to provision a bigger node (memory, and/or CPU, and/or read/write storage speed) than the current leader. This node will initially be provisioned as the Load Manager (as always) and will remain as Lead Manager until it synchronizes with the current leader exactly like in the Mach transitions explained earlier. Once synchronized, it starts a new election and becomes the new leader of the Raft consensus, switching places with the old leader. Each node in the cluster has specification levels labeled from 1 to 5, where 1 is the lowest level of specification (smaller resources), and 5 is the highest (bigger resources).

2. Sharding

Once scaling vertically has reached its limit (node level 5), sharding begins. A new leader will be provisioned as Read Replica that will first synchronize only the shard it will take over from the current leader and then will become the leader itself only for that shard of the data. The level of the new node added (1 to 5) has to be configurable.

This is a different trigger for sharding than the Read Intensive scenario described before. Read Intensive sharding is triggered based on horizontal scale (number of nodes). For Write Intensive scenarios, sharding is triggered based on vertical scale; in other words, the size of the leader reached the maximum possible (5).

It is also possible to specify that when the new shard is provisioned, both leaders are of a certain level, for instance, level 3. This is important because two level 3s make a "level 6" Writer capable cluster, but this should also be configurable. In this scenario, the previous level 5 Writer node would be replaced by two level 3 nodes, one for each shard.

A priority labeling for sharding strategy

The problem: Figuring out how to separate information into shards efficiently is no easy task, especially if the data is large and complex.

A simple solution: Every time a record/object is stored in the cluster, a bucket ID in a range from 1 to 1000 is assigned to it. This bucket ID is random and guarantees that, at a large scale, each bucket will have a similar number of objects assigned to it, balancing the shards.

For instance, for two shards, the first shard will have objects allocated in the buckets from 1 to 500 inclusive, and the second shard will have objects allocated in the buckets 501 to 1000 inclusive, considering that the total number of buckets was defined to be 1000.

This splitting buckets per shard process will be repeated every time a new shard is required. This means that, for this example, the maximum number of shards is 1000, which is probably unrealistically high for most scenarios.

Conclusion and Future Work

By no means do I believe this article addresses all the nuances of autoscaling a stateful system but instead offers a template and a pattern based on many techniques I have used separately during my career, now put into a single standard that can be used as a foundation for stateful autonomous scalability implementations. These designs may not be the best fit for scenarios where the majority of the read operations have to be executed against many or all shards. In such scenarios, a better shard bucket definition is advisable to attempt to have all the data needed in a single shard or in as few shards as possible.

DynamoDB Data Transformation Safety: from Manual Toil to Automated and Open Source

by **Guy Braunstain,** Full Stack Developer

When designing a product to be a self-serve developer tool, there are often constraints - but likely one of the most common ones is scale. Ensuring our product, Jit - a security-as-code SaaS platform, was built for scale was not something we could embed as an afterthought, it needed to be designed and handled from the very first line of code.

We wanted to focus on developing our application and its user experience, without having challenges with issues and scale be a constant struggle for our engineers. After researching the infrastructure that would enable this for our team - we decided to use AWS with a serverless-based architecture.

AWS Lambda is becoming an ever-popular choice for fast-growing SaaS systems, as it provides a lot of benefits for scale and performance out of the box through its suite of tools, and namely the database that supports these systems, AWS's DynamoDB.

One of its key benefits is that it is already part of the AWS ecosystem, and therefore this abstracts many of the operational tasks of management and maintenance, such as maintaining connections with the database, and it requires minimal setup to get started in AWS environments.

As a fast-growing SaaS operation, we need to evolve quickly based on user and customer feedback and embed this within our product. Many of these changes in application design have a direct impact on data structures and schemas.

With rapid and oftentimes significant changes in the application design and architecture, we found ourselves needing to make data transformations in DynamoDB

very often, and of course, with existing users, it was a priority that this be achieved with zero downtime. (In the context of this article Data Transformation will refer to modifying data from state A to state B).

Challenges with Data Transformation

In the spirit of Brendon Moreno from the UFC:

Maybe not today, maybe not tomorrow, and maybe not next month, but only one thing is true, you will need to make data transformations one day, I promise.

Yet, while data transformation is a known constant in engineering and data engineering, it remains a pain point and challenge to do seamlessly. Currently, in DynamoDB, there is no easy way to do it programmatically in a managed way, surprisingly enough.

While there are many forms of data transformation, from replacing an existing item's primary key to adding/removing attributes, updating existing indexes - and the list goes on (these types are just a few examples), there remains no simple way to perform any of these in a managed and reproducible manner, without just using breakable or one-off scripting.

User Table Data Transform Example

Below, we are going to dive into a real-world example of a data transformation process with production data.

Let's take the example of splitting a "full name" field into its components "first name" and "last name". As you can see in the example below, the data aggregation currently writes names in the table with a "full name" attribute. But let's say we want to transform from a full name, and split this field into first and last name fields.

Before

Id	FullName
123	Guy Br

After

Id	FirstName	LastName
123	Guy	Br

Looks easy, right? Not so, to achieve just this simple change these are the steps that will need to be performed on the business logic side, in order to successfully transform this data.

- Scanning the user records
- Extracting the FullName attribute from each record
- Splitting the FullName attribute into new FirstName and LastName attributes
- Saving the new records
- Cleaning up the FullName attribute

But let's discuss some of the issues you would need to take into account before you even get started, such as - how do you run and manage these transformations in different application environments? Particularly when it's not really considered a security best practice to have access to each environment. In addition, you need to think about service dependencies. For example, what should you do when you have another service dependent on this specific data format?

Your service needs to be backward compatible and still provide the same interface to external services relying on it.

When you have production clients, possibly one of the most critical questions you need to ask yourself before you modify one line of code is how do you ensure that zero downtime will be maintained?

Some of the things you'd need to plan for to avoid any downtime

is around testing and verification. How do you even test your data transformation script? What are some good practices for running a reliable dry run of a data transformation on production data?

There are so many things to consider before transforming data.

Now think that this is usually, for the most part, done manually. What an error-prone, tedious process! It looks like we need a fine-grained process that will prevent mistakes and help us to manage all of these steps.

To avoid this, we understood we'd need to define a process that would help us tackle the challenges above.

The Rewrite Process

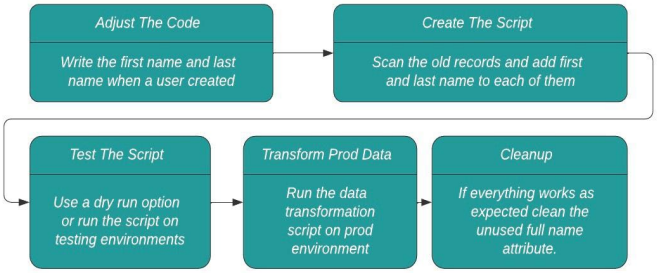


Figure 1: Rewrite Process Flow Chart

First, we started by adjusting the backend code to write the new data format to the database while still keeping the old format, by first writing the FullName, FirstName and LastName to provide us some reassurance of backward compatibility. This would enable us to have the ability to revert to the previous format if something goes terribly wrong.

```
async function createUser(item) {
   // FullName = 'Guy Br'
   // 'Guy Br'.split(' ') === ['Guy',
'Br']
   // Just for the example assume that
the FullName has one space between first
and last name
   const [FirstName, LastName] = item.
FullName.split(' ');
   const newItemFormat = { ...item,
FirstName, LastName };
   return dynamodbClient.put({
       TableName: 'Users',
       Item: newItemFormat,
   }).promise();
};
```

Link to GitHub

Next, we wrote a data transformation script that scans the old records and appends the FirstName and LastName attributes to each of them, see the example below:

```
async function
appendFirstAndLastNameTransformation()
{
  let lastEvalKey;
  let scannedAllItems = false;

  while (!scannedAllItems) {
    const { Items, LastEvaluatedKey } =
await dynamodbClient.scan({ TableName:
'Users' }).promise();
    lastEvalKey = LastEvaluatedKey;

    const updatedItems = Items.
map((item) => {
      const [FirstName, LastName] =
splitFullNameIntoFirstAndLast(item.
FullName);
      const newItemFormat = { ...item,
FirstName, LastName };
      return newItemFormat;
    });

    await Promise.all(updatedItems.
map(async (item) => {
      return dynamodbClient.put({
        TableName: 'Users',
        Item: item,
      }).promise();
    }));

    scannedAllItems = !lastEvalKey;
  };
}
```

Link to GitHub

After writing the actual script (which is the easy part), we now needed to verify that it actually does what it's supposed to. To do so, the next step

was to run this script on a test environment and make sure it works as expected. Only after the scripts usability is confirmed, it could be run on the application environments.

The last phase is the cleanup, this includes taking the plunge and ultimately deleting the FullName column entirely from our database attributes. This is done in order to purge the old data format which is not used anymore, and reduce clutter and any future misuse of the data format.

```
async function cleanup() {
  let lastEvalKey;
  let scannedAllItems = false;

  while (!scannedAllItems) {
    const { Items, LastEvaluatedKey } =
await dynamodbClient.scan({ TableName:
'Users' }).promise();
    lastEvalKey = LastEvaluatedKey;

    const updatedItems = Items.
map((item) => {
      delete item.FullName;
      return item;
    });

    await Promise.all(updatedItems.
map(async (item) => {
      return dynamodbClient.put({
        TableName: 'Users',
        Item: item,
      }).promise();
    }));

    scannedAllItems = !lastEvalKey;
  };
 };
```

Link to GitHub

Lets quickly recap what we have done in the process:

- **Adjusted** the backend code to write in the new data format
- **Created** a data transformation script that updates each record
- **Validated** that script against a testing environment
- **Ran the script** on the application environments
- **Cleaned up** the old data

This well-defined process helped us to build much-needed safety and guardrails into our data transformation process. As we mentioned before, with this process we were able to avoid downtime by keeping the old format of the records until we don't need them anymore. This provided us with a good basis and framework for more complex data transformations.

Transforming Existing Global Secondary Index (GSI) using an External Resource

Now that we have a process––let's be honest, real-world data transformations are hardly so simple. Let's assume, a more likely scenario, that the data is actually ingested from an external resource, such as the GitHub API, and that our more advanced data transformation scenario actually requires us to ingest data from multiple sources.

Let's take a look at the example below for how this could work.

In the following table, the GSI partition key is by GithubUserId.

For the sake of this data transformation example, we want to add a "GithubUsername" column to our existing table.

Before

Id	GithubUserId
123	7654321

After

Id	GithubUserId	GithubUsername
123	7654321	Guy7B

This data transformation looks seemingly as straightforward as the example with the full name, but there is a little twist.

How can we get the Github username if we don't have this information? We have to use an external resource, in this case, it's the Github API.

GitHub has a simple API for extracting this data (you can read the documentation here). We will pass the GithubUserId and get information about the user which contains the Username field that we want.

https://api.github.com/user/:id

The naive flow is similar to the full name example above:

- Adjust our code to write in the new data format.
- Assume that we have the Github username when creating a user.
- Scan the user records (get `GithubUsername` by `GithubUserId` for each record using Github API), and update the record.
- Run that script on the testing environment
- Run it on the application environments

However, in contrast to our previous flow, there is an issue with this naive flow. The flow above is not safe enough. What happens if you have issues while running the data transformation when calling the external resource? Perhaps the external resource will crash / be blocked by your IP or is simply unavailable for any other reason? In this case, you might end up with production errors or a partial transformation, or other issues with your production data.

What can we do on our end to make this process safer?

While you can always resume the script if an error occurs or try to handle errors in the script itself, however, it is important to have the ability to perform a dry run with the prepared data from the external resource before running the script on production. A good way to provide greater safety measures is by preparing the data in advance.

Below is the design of the safer flow:

Adjust our code to write in the new data format (create a user with GithubUsername field)

Create the preparation data for the transformation

Only after we do this, we scan the user records, get GithubUsername for each of them using Github API, append it to a JSON Object `{ [GithubUserId]: GithubUsername }` and then write that JSON to a file.

This is what such a flow would look like:

```
async function
prepareGithubUsernamesData() {
  let lastEvalKey;
  let scannedAllItems = false;

  while (!scannedAllItems) {
    const { Items, LastEvaluatedKey } =
await dynamodbClient.scan({ TableName:
'Users' }).promise();
    lastEvalKey = LastEvaluatedKey;

    const currentIdNameMappings = await
Promise.all(Items.map(async (item) => {
      const githubUserId = item.
GithubUserId;
      const response = await
fetch(`https://api.github.com/
user/${githubUserId}`, { method: 'GET'
});
      const githubUserResponseBody =
await response.json();
      const GithubUsername =
githubUserResponseBody.login;

      return { [item.GithubUserId]:
GithubUsername };
    }));

    currentIdNameMappings.
forEach((mapping) => {
```

```
      // append the current mapping to
the preparationData object
      preparationData = {
...preparationData, ...mapping };
    });

    scannedAllItems = !lastEvalKey;
  };

  await fs.writeFile('preparation-
data.json', JSON.
stringify(preparationData));
};
```

Link to GitHub

Next we scan the user records (get GithubUsername by GithubUserId for each record using **Preparation Data**), and move ahead to updating the record.

```
async function appendGithubUsername() {
  let lastEvalKey;
  let scannedAllItems = false;

  while (!scannedAllItems) {
    const { Items, LastEvaluatedKey } =
await dynamodbClient.scan({ TableName:
'Users' }).promise();
    lastEvalKey = LastEvaluatedKey;

    const updatedItems = Items.
map((item) => {
      const GithubUsername =
preparationData[item.GithubUserId];
      const updatedItem =
currentGithubLoginItem ? { ...item,
GithubUsername } : item;
      return updatedItem;
    });

    await Promise.all(updatedItems.
map(async (item) => {
      return dynamodbClient.put({
        TableName: 'Users',
        Item: item,
      }).promise();
    }));

    scannedAllItems = !lastEvalKey;
  };
};
```

Link to GitHub

And finally, like the previous process, we wrap up by running the script on the testing environment, and then the application environments.

Dynamo Data Transform

Once we built a robust process that we could trust for data transformation, we understood that to do away with human toil and ultimately error, the best bet would be to automate it.

We realized that even if this works for us today at our smaller scale, manual processes will not grow with us. This isn't a practical long-term solution and would eventually break as our organization scales. That is why we decided to build a tool that would help us automate and simplify this process so that data transformation would no longer be a scary and painful process in the growth and evolution of our product.

Applying automation with open source tooling

Every data transformation is just a piece of code that helps us to perform a specific change in our database, but these scripts, eventually, must be found in your codebase.

This enables us to do a few important operations:

- Track the changes in the database and know the history at every moment. Which helps to investigate bugs and issues.
- No need to reinvent the wheel - reusing existing data transformation scripts already written your organization streamlines processes.

By enabling automation for data transformation processes, you essentially make it possible for every developer to be a data transformer. While you likely should not give production access to every developer in your organization, applying changes is the last mile. When only a handful of people have access to production, this leaves them with validating the scripts and running them on production, and not having to do all of the heavy lifting of writing the scripts too. We understand

it consumes more time than needed for those operations and it is not safe.

When the scripts in your codebase and their execution are automated via CI/CD pipelines

other developers can review them, and basically, anyone can perform data transformations on all environments, alleviating bottlenecks.

Now that we understand the importance of having the scripts managed in our codebase, we want to create the best experience for every data-transforming developer.

Making every developer a data transformer

Every developer prefers to focus on their business logic - with very few context disruptions and changes. This tool can assist in keeping them focused on their business logic, and not have to start from scratch every time they need to perform data transformations to support their current tasks.

For example - dynamo-data-transform provides the benefits of:

- Export utility functions that are useful for most of the data transformations
- Managing the versioning of the data transformation scripts
- Supporting dry runs to easily test the data transformation scripts
- Rollback in the event the transformation goes wrong - it's not possible to easily revert to the previous state
- Usage via CLI––for dev friendliness and to remain within developer workflows. You can run the scripts with simple commands like `dynamodt up`, `dynamodt down` for rollback, `dynamodt history` to show which commands were executed.

Dynamo Data Transform:

Quick Installation for serverless:

The package can be used as a standalone npm package see here.

To get started with DynamoDT, first run:

```
npm install dynamo-data-transform
--save-dev
```

To install the package through NPM (you can also install it via…)

Next, add the tool to your serverless.yml Run:

```
npx sls plugin install -n dynamo-data-
transform
```

You also have the option of adding it manually to your serverless.yml:

```
plugins:
  - dynamo-data-transform
```

You can also run the command:

```
sls dynamodt --help
```

To see all of the capabilities that DynamoDT supports.

Let's get started with running an example with DynamoDT. We'll start by selecting an example from the code samples in the repo, for the sake of this example, we're going to use the example `v3_insert_users.js`, however, you are welcome to test it out using the examples you'll find here.

We'll initialize the data transformation folder with the relevant tables by running the command:

```
npx sls dynamodt init --stage local
```

For serverless (it generates the folders using the resources section in the serverless.yml):

```
resources:
```

```
 Resources:
   UsersExampleTable:
     Type: AWS::DynamoDB::Table
     Properties:
       TableName: UsersExample
```

The section above should be in serverless.yml

The data-transformations folder generated with a template script that can be found here.

We will start by replacing the code in the template file v1_script-name.js with:

```
const { utils } = require('dynamo-data-
transform');

const TABLE_NAME = 'UsersExample';

/**
 * The tool supply following
parameters:
 * @param {DynamoDBDocumentClient} ddb
- dynamo db document client https://
docs.aws.amazon.com/AWSJavaScriptSDK/
v3/latest/clients/client-dynamodb
 * @param {boolean} isDryRun - true if
this is a dry run
 */
const transformUp = async ({ ddb,
isDryRun }) => {
  const addFirstAndLastName = (item) =>
{
    // Just for the example:
    // Assume the FullName has one
space between first and last name
    const [firstName, ...lastName] =
item.name.split(' ');
    return {
      ...item,
      firstName,
      lastName: lastName.join(' '),
    };
  };
  return utils.transformItems(ddb,
TABLE_NAME, addFirstAndLastName,
isDryRun);
};

module.exports = {
  transformUp,
  transformationNumber: 1,
};
```

Link to GitHub

For most of the regular data transformations, you can use the util functions from the dynamo-data-transform package. This means you don't need to manage the versions of the data transformation scripts, the package will do this work for you. Once you've customized the data you'll want to transform, you can test the script using the dry run option by running:

```
npx sls dynamodt up --stage local --dry
```

The dry run option prints the records in your console so you can immediately see the results of the script, and ensure there is no data breakage or any other issues.

Once you're happy with the test results, you can remove the --dry flag and run it again, this time it will run the script on your production data, so make sure to validate the results and outcome.

Once you have created your data transformation files, the next logical thing you'd likely want to do is add this to your CI/CD. To do so add the command to your workflow/ci file for production environments.

The command will run immediately after the `sls deploy` command, which is useful for serverless applications.

Finally, all of this is saved, as noted above so if you want to see the history of the data transformations, you can run:

```
npx sls dynamodt history --table UserExample --stage local
```

```
History for table UsersExample
┌─────────┬──────────────────────┬─────────┬──────────────────────┐
│ (index) │         Date         │ Command │ Transformation Number │
├─────────┼──────────────────────┼─────────┼──────────────────────┤
│    0    │ '10/25/2022, 6:28:27 PM' │  'up'   │          1           │
│    1    │ '10/25/2022, 6:28:47 PM' │  'up'   │          2           │
│    2    │ '10/25/2022, 6:28:48 PM' │  'up'   │          3           │
│    3    │ '10/25/2022, 6:30:06 PM' │ 'down'  │          3           │
│    4    │ '10/25/2022, 6:36:56 PM' │ 'down'  │          2           │
│    5    │ '10/25/2022, 6:37:01 PM' │ 'down'  │          1           │
│    6    │ '10/25/2022, 6:37:14 PM' │  'up'   │          1           │
│    7    │ '10/25/2022, 6:37:14 PM' │  'up'   │          2           │
│    8    │ '10/25/2022, 6:37:15 PM' │  'up'   │          3           │
│    9    │ '10/25/2022, 6:38:06 PM' │ 'down'  │          3           │
└─────────┴──────────────────────┴─────────┴──────────────────────┘
"history" command run successfully.
```

The tool also provides an interactive CLI for those who prefer to do it this way.

And all of the commands above are supported via CLI as well.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
guybr@Guys-MBP serverless-localstack % dynamodt -i

DYNAMO DATA TRANSFORM

Select a command:
› up - Run the pending data transformation scripts
  down - Rollback the last data transformation
  history - Show the history of data transformations
  prepare - Prepare data for the data transformation script
  init - Initialize the data-transformations folder
```

With Dynamo Data Transform, you get the added benefits of being able to version and order your data transformation operations and manage them in a single place. You also have the history of your data transformation operations if you would like to roll back an operation. And last but not least, you can reuse and review your previous data transformations.

We have open-sourced the Dynamo Data Transform tool that we built for internal use to perform data transformations on DynamoDB and serverless-based environments and manage these formerly manual processes in a safe way.

The tool can be used as a Serverless Plugin and as a standalone NPM package.

- NPM
- GitHub

Feel free to provide feedback and contribute to the project if you find it useful.

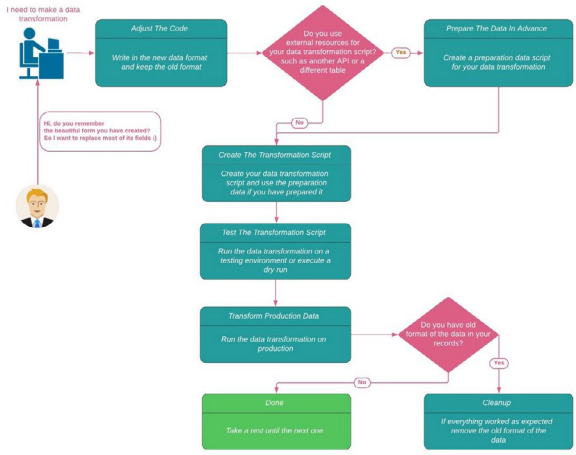


Figure 2: Data Transformation Flow Chart

Understanding and Applying Correspondence Analysis

by **Maarit Widmann,** Data Scoentist @KNIME, **Alfredo Roccato**, Data Science Independent Consultant

In this article, the authors explain how correspondence analysis functions with an example of real social survey data. Also provided is an implementation of the example in KNIME Analytics Platform, an open source software, so that you can try out the analysis hands-on.

Introduction

Customer segments, personality profiles, social classes, and age generations are examples of effective references to larger groups of people sharing similar characteristics.

The characteristics that shape these groups are often manifold and thus require multivariate analysis.

One way to access the variables is via questionnaires. Because the variables are mostly qualitative, the questionnaires produce categorical data with predefined categories, for example, on a Likert-type scale.

The starting point to analyze the relationships between categorical variables is a contingency table which compares the categories pairwise.

As the next step, correspondence analysis (CA) performs a multivariate analysis on multiple contingency tables.

It projects them into a numeric feature space, which captures most of the variability in the data by fewer dimensions.

What Is Simple Correspondence Analysis?

Simple correspondence analysis is a technique to analyze relationships between categorical variables and create

Row ID	name	edition	idno	cntry	netusoft	polintr	rlgdnm	gndr	yrbrn	rshpsts	eisced	pdwrk	edctn	uempla
Row0	ESS9e03_1	3.1	27	AT	5	3	1	1	1975	66	3	1	0	0
Row1	ESS9e03_1	3.1	137	AT	5	2	1	1	1951	1	3	0	0	0
Row2	ESS9e03_1	3.1	194	AT	4	4	1	2	1978	1	2	1	0	0
Row3	ESS9e03_1	3.1	208	AT	5	3	66	1	1955	1	3	1	0	0
Row4	ESS9e03_1	3.1	220	AT	1	2	1	2	1947	66	2	0	0	0
Row5	ESS9e03_1	3.1	254	AT	2	2	1	1	1954	1	3	0	0	0
Row6	ESS9e03_1	3.1	290	AT	1	4	1	1	1962	1	3	1	0	0
Row7	ESS9e03_1	3.1	301	AT	1	3	1	2	1944	66	3	0	0	0
Row8	ESS9e03_1	3.1	305	AT	5	3	1	1	1981	1	3	1	0	0
Row9	ESS9e03_1	3.1	400	AT	4	4	1	2	1996	66	3	0	0	0
Row10	ESS9e03_1	3.1	413	AT	1	2	66	1	1970	66	3	1	0	0
Row11	ESS9e03_1	3.1	438	AT	5	2	66	2	1959	1	4	0	0	0

Figure 1. Raw survey data as a starting point of CA

profiles based on the projections of the original variables to the new dimensions that it creates. This is useful, for example, when analyzing and visualizing survey data.

CA processes a two-way contingency table that displays the frequency distribution between two variables. It represents the frequency distribution on numeric, orthogonal dimensions. Based on the proximity along the first few of these dimensions, we can visually explore the individuals' and categories' associations.

We can investigate, for example, if there is a relationship between interest in politics and demographics data such as age. Also, we can interpret a dimension generated by CA as a new, synthetic dimension, such as "status," that captures several categories which together contribute to "high" or "low" status.

How To Perform Correspondence Analysis

Step 1: Data collection

We start the data collection by accessing survey data, with records for N individuals who have answered K questions.

As an example, we use the European Social Survey data from the year 2018 measuring the attitudes, beliefs and behavior patterns in European nations. The data contains metadata and answers from 49,519 individuals recorded in 572 columns. We consider only a subset of the variables and perform CA to analyze the relationships between interest in politics, country, income, family relationship, gender, education, age, and internet usage.

These variables are transformed into a two-way contingency table (see the next step) based on the definition of row variables, column variables and supplemental variables as described below:

- **Row variables** refer to variables that represent the row IDs. In our example, the interest in politics is the row variable. It contains the following four nominal classes: not at all, hardly, quite, and very interested. The data for 98 participants who didn't provide the information about their interest in politics (not applicable, refusal, no answer, don't know) were discarded from the analysis.
- **Column variables** refer to variables that represent the column headers. The column variables are income, family relationship, gender, education, age, and internet usage.
- **Supplementary variables** can be used to interpret the resulting profiles, but they are not used in computing CA. In our example, "country" is the supplementary variable.

Note that if there were numeric variables, these had to be discretized before performing CA.

The survey data can be stored in varying formats, for example, in a csv file. Here, each row corresponds to an individual filling out the survey. Each column represents a survey question or metadata, such as the ID of the participant. (see Figure 1)

Row ID	GE:Female	GE:Male	FI:VeryHigh	FI:High	FI:MidHigh	FI:Middle	FI:MidLow	FI:Low	FI:VeryLow
PI:Hardly	10201	7636	826	2416	1449	3027	1680	3287	1407
PI:Not at all	6424	3721	311	906	590	1410	864	2117	1182
PI:Quite	7708	8308	1298	2768	1497	2907	1410	2657	1051
PI:Very	2116	3307	685	971	530	954	448	818	339

Figure 2. A two-way contingency table showing the preprocessed survey data before computing

Notice that the column and row variables may need to be binned or encoded to help give a better understanding of the CA results. For example, the survey data reports 10 income deciles, which we encoded to seven income classes: very low, low, mid low, middle, mid high, high, and very high.

Step 2: Data preprocessing

In data preprocessing, we create a two-way contingency table that shows the frequency distribution of the row and column variables.

Figure 2. shows a part of the contingency table for the survey data in our example.

In the first row, it shows how the 17,837 survey participants hardly interested in politics are distributed into two categories, male and female, as well as into the seven categories describing family income. The more column variables there are, and the more categories in each column variable, the wider the table.

The transformation of the raw data into a contingency table is required to perform CA via the algorithms available, for example, in R software. In the next step, we explain how CA functions under the hood, although it is not necessary for executing such algorithms.

Step 3: Computing CA

Projecting the data into new numeric dimensions in CA works the same way as in principal component analysis (PCA), by sequentially constructing orthogonal dimensions of the data. This can be performed by singular value decomposition.

However, while in PCA, the decomposition is based on maximizing the variance; in CA it is based on maximizing the inertia.

For each row variable *i*, inertia is calculated with the following formula:

$$Inertia\ (i/G_J) = f_i . d^2_x{}^2(i,G_J)$$

Where f_i is the weight, i.e., the marginal sum of row variable *i*, and $d^2_x{}^2(i,G_J)$ is the chi-squared distance from the mean profile defined by the marginal probabilities of column variables *J*. The total inertia is calculated by summing up these inertias for all row variables *I*. In the extreme case, if the row reflects the mean profile, the inertia of that row variable is zero.

For column variables, the inertia is the sum of inertias of their categories *j*:

$$Inertia\ (j/G_I) = f_{.j}\, d^2_x{}^2(j,G_I),$$

Where f_j is the weight, the marginal sum of column variable *j*, and $d^2_x{}^2(j,G_I)$ is the chi-squared distance from the mean profile defined by the marginal probabilities of *I* row variables.

The sum of inertias of all column variables *j* produces the same total inertia as the sum of inertias of all individuals *i*.

Step 4: Interpreting the results

In this step, we explain how to interpret the results of CA visually in a scree plot and biplot and numerically via the output statistics.

Scree plot

To compare the percentages of total inertia that the new dimensions explain, we can take a look at a scree plot as shown in Figure 3

In our example, the first dimension explains 89.4% of the inertia, while the second dimension explains 10.19% of it. Together, the first two

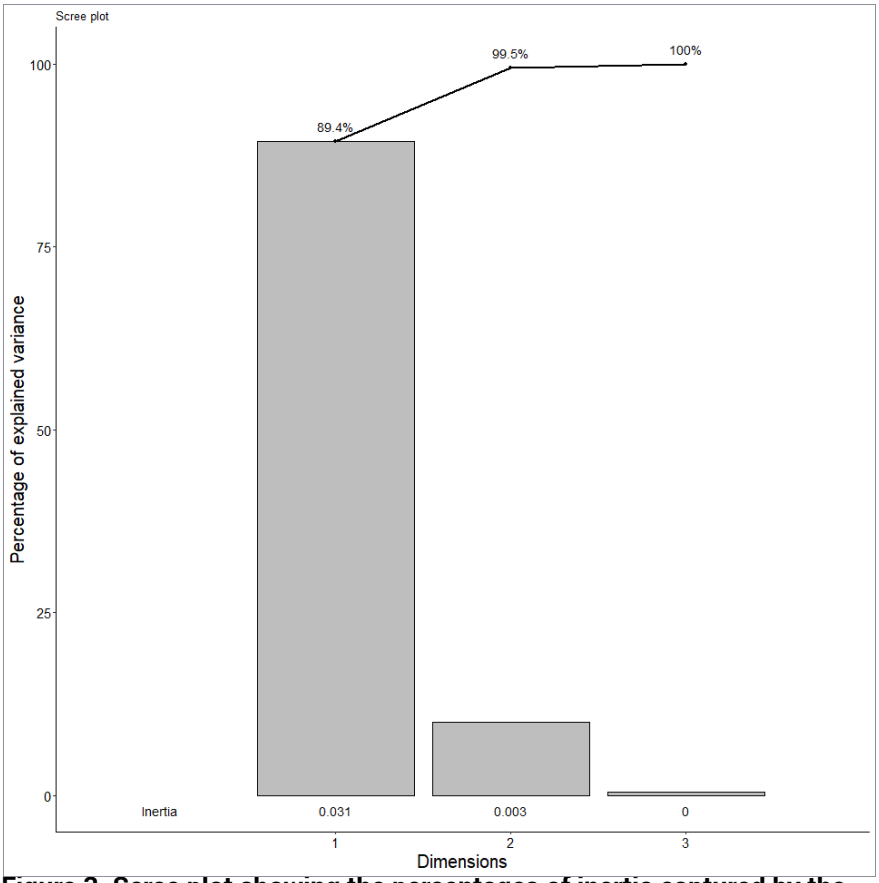


Figure 3. Scree plot showing the percentages of inertia captured by the new dimensions generated by CA

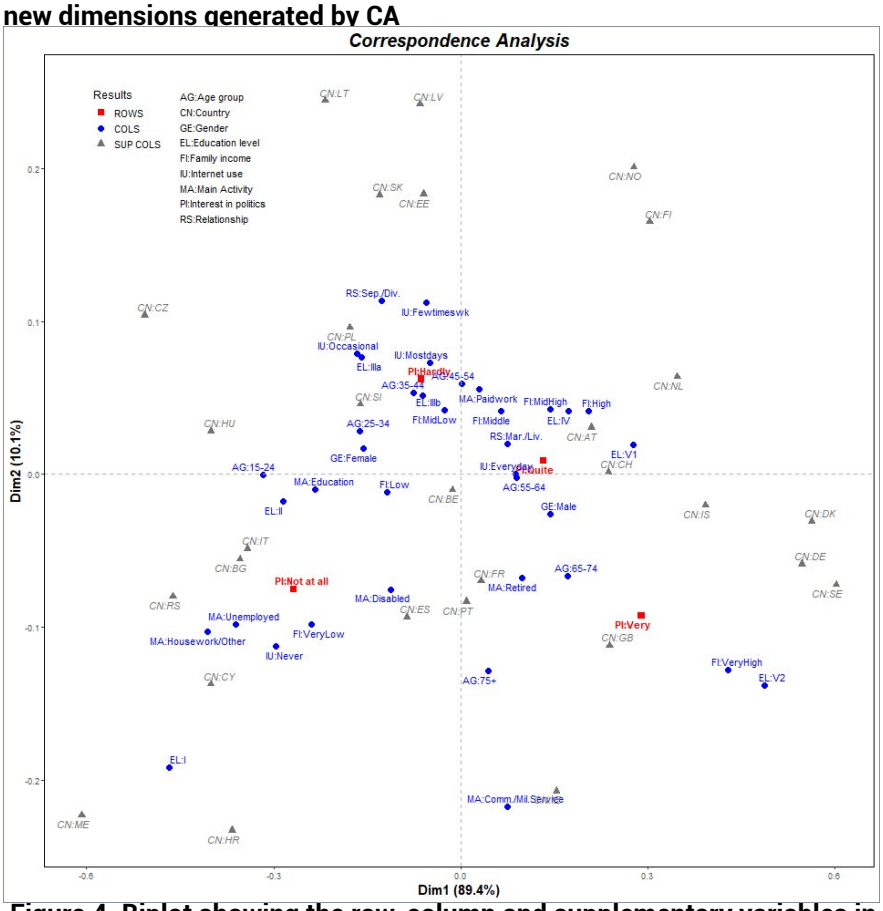


Figure 4. Biplot showing the row, column and supplementary variables in two-dimensional space

dimensions explain 99.5% of the total inertia.

Biplot

Next, we project the row and column variables into the first two dimensions and explore them visually in a biplot. (see Figure 3)

The biplot shows the first two dimensions on the x- and y-axis. It is possible to show the row, column and supplementary variables along the same axes using transition formulas between the coordinates of row, column and supplementary variables.

Proximity in the feature space indicates positive association. For example, the group of individuals who are very interested in politics (PI: Very) is close to the category of very high income (FI:VeryHigh), and these variables are therefore strongly associated. Also, the categories of very high income and MA level education (EL: V2) are strongly associated. This implies that people with very high income have MA level education more often than an average person from any income class.

Also, the closer the angle between two groups/categories is to 90°, the less they are associated. For example, the categories "IU: Everyday" and "Age: 45-54" lie on the x- and y-axis, respectively. Therefore, this association is very weak.

netusoft	agegrp	Frequency	Expected	Deviation
IU:Everyday	?	124	135.487	-11.487
IU:Everyday	AG:15-24	4,698	3,173.128	1,524.872
IU:Everyday	AG:25-34	5,517	3,881.499	1,635.501
IU:Everyday	AG:35-44	5,986	4,605.336	1,380.664
IU:Everyday	AG:45-54	5,921	5,247.509	673.491
IU:Everyday	AG:55-64	4,666	5,412.074	-746.074
IU:Everyday	AG:65-74	2,816	4,854.039	-2,038.039
IU:Everyday	AG:75+	847	3,265.928	-2,418.928
IU:Fewtime...	?	15	12.137	2.863
IU:Fewtime...	AG:15-24	72	284.258	-212.258
IU:Fewtime...	AG:25-34	153	347.716	-194.716
IU:Fewtime...	AG:35-44	324	412.56	-88.56
IU:Fewtime...	AG:45-54	561	470.088	90.912
IU:Fewtime...	AG:55-64	689	484.83	204.17
IU:Fewtime...	AG:65-74	642	434.839	207.161
IU:Fewtime...	AG:75+	283	292.572	-9.572
IU:Mostdays	?	32	18.598	13.402
IU:Mostdays	AG:15-24	253	435.572	-182.572

Figure 5. A sample of a contingency table between analyzed variables

The contingency table in Figure 5 below confirms this: There is little deviation between the observed and expected value.

The categories of the supplementary variable, i.e., the countries, help to interpret the dimensions. It seems that Switzerland (CN:CH) and the Nordic countries such as Denmark and Sweden (CN:DK and CN:SE) are strongly associated with the first dimension. Instead, the Baltic countries such as Estonia and Latvia (CN:EE and CN:LV) are strongly associated with the second dimension.

Results	Label	Quality	Mass	Inertia	Dim1	Dim2	Contr1	Contr2	SqCos1	SqCos2
COLS	GE:Female	1	0.084	0.002	-0.156	0.017	0.066	0.007	0.989	0.011
COLS	GE:Male	0.999	0.073	0.002	0.144	-0.026	0.049	0.014	0.967	0.032
COLS	FI:High	0.985	0.022	0.001	0.204	0.041	0.03	0.011	0.946	0.039
COLS	FI:Low	1	0.028	0	-0.118	-0.012	0.013	0.001	0.99	0.01
COLS	FI:MidHigh	1	0.013	0	0.144	0.043	0.009	0.007	0.919	0.081
COLS	FI:MidLow	0.954	0.014	0	-0.026	0.042	0	0.007	0.268	0.686
COLS	FI:Middle	0.989	0.026	0	0.064	0.041	0.004	0.013	0.699	0.29
COLS	FI:VeryHigh	0.994	0.01	0.002	0.429	-0.128	0.059	0.047	0.913	0.082
COLS	FI:VeryLow	1	0.013	0.001	-0.239	-0.098	0.023	0.035	0.856	0.144
COLS	EL:I	0.995	0.012	0.003	-0.467	-0.192	0.085	0.127	0.852	0.144
COLS	EL:II	1	0.026	0.002	-0.285	-0.018	0.07	0.002	0.996	0.004
COLS	EL:IIIa	1	0.036	0.001	-0.159	0.076	0.029	0.059	0.813	0.187
COLS	EL:IIIb	0.918	0.026	0	-0.062	0.051	0.003	0.019	0.54	0.378
COLS	EL:IV	0.998	0.019	0.001	0.172	0.041	0.019	0.009	0.944	0.054
COLS	EL:V1	0.987	0.018	0.001	0.277	0.019	0.043	0.002	0.982	0.005
COLS	EL:V2	0.997	0.02	0.005	0.487	-0.138	0.152	0.109	0.923	0.074
COLS	RS:Mar./Liv.	0.977	0.09	0.001	0.075	0.02	0.016	0.01	0.912	0.065
COLS	RS:Sep./Div.	0.993	0.001	0	-0.126	0.114	0	0.002	0.548	0.445
COLS	MA:Disabled	0.954	0.004	0	-0.112	-0.075	0.002	0.007	0.657	0.297
COLS	MA:Education	0.982	0.01	0.001	-0.233	-0.01	0.018	0	0.98	0.002
COLS	MA:Paidwork	1	0.082	0	0.03	0.056	0.002	0.073	0.224	0.776
COLS	MA:Retired	0.999	0.043	0.001	0.098	-0.068	0.013	0.057	0.675	0.323
COLS	MA:Unemplo...	1	0.008	0.001	-0.361	-0.098	0.035	0.023	0.931	0.069
COLS	MA:Housew...	0.999	0.007	0.001	-0.406	-0.103	0.038	0.022	0.939	0.06
COLS	IU:Everyday	0.999	0.097	0.001	0.088	0	0.024	0	0.999	0
COLS	IU:Fewtimes...	0.999	0.009	0	-0.055	0.112	0.001	0.032	0.195	0.804
COLS	IU:Mostdays	0.979	0.013	0	-0.049	0.073	0.001	0.02	0.308	0.671
COLS	IU:Never	0.999	0.028	0.003	-0.296	-0.113	0.08	0.104	0.873	0.127
COLS	IU:Occasional	0.945	0.009	0	-0.167	0.079	0.008	0.017	0.772	0.173
COLS	MA:Comm./...	0.81	0	0	0.075	-0.217	0	0.001	0.086	0.724
COLS	AG:15-24	0.993	0.016	0.002	-0.317	-0	0.053	0	0.993	0
COLS	AG:25-34	0.995	0.02	0.001	-0.161	0.028	0.017	0.005	0.966	0.029
COLS	AG:35-44	1	0.024	0	-0.076	0.053	0.004	0.019	0.668	0.332
COLS	AG:45-54	0.94	0.027	0	0.002	0.059	0	0.027	0.001	0.939
COLS	AG:55-64	0.98	0.028	0	0.089	-0.002	0.007	0	0.98	0.001
COLS	AG:65-74	0.998	0.025	0.001	0.172	-0.067	0.024	0.032	0.868	0.13
COLS	AG:75+	0.999	0.017	0	0.044	-0.129	0.001	0.08	0.105	0.894
ROWS	PI:Hardly	0.988	0.36	0.003	-0.063	0.062	0.046	0.399	0.5	0.488
ROWS	PI:Not at all	1	0.2	0.015	-0.268	-0.075	0.463	0.321	0.927	0.072
ROWS	PI:Quite	0.986	0.329	0.006	0.133	0.009	0.188	0.007	0.982	0.004
ROWS	PI:Very	0.995	0.112	0.01	0.29	-0.092	0.303	0.273	0.903	0.091
SUP COLS	CN:AT	?	?	?	0.209	0.031	?	?	?	?

Figure 6 . The output statistics of CA

Statistics

Finally, we can inspect the groups, categories and new dimensions by looking at the output statistics. The table in Figure 6 shows a sample of the output statistics of CA for our example:

The table contains the variables as row IDs and the statistics as column headers. For supplementary columns, there are no statistics, except dimensions 1 and 2, because they don't contribute to the dimensions. Dimension 1 seems to relate positively to high levels of interest in politics, family income, education, age group, etc. Therefore, Dimension 1, which is highly important, could be interpreted as a "status" dimension (high vs. low).

In the next table we explain what the output statistics of CA quantify and state an example question that we can answer by them.

Next, we show how to perform CA and produce the results introduced above in KNIME Analytics Platform.

Practical Implementation of CA

In this section, we will introduce how to perform the example

Statistics	Definition	Question
Mass	The relative frequency of each variable. The percentages of row and column variables, respectively, sum up to 1.	Which group of individuals/ category is the most/least frequent?
Inertia	The inertia of a single row/column variable	How diverse is each group of individuals/category?
Dim1-2	The coordinates of the variable in the reduced feature space as defined by dimensions 1 and 2	How to visualize the variables in a two-dimensional feature space?
Contr1-2	The percentage of the variable's inertia of the total inertia of dimension 1 and 2, respectively	Which variable explains the most variation of dimensions 1 and 2?
SqCos1-2	The percentage of the variable's inertia projected onto dimension 1 and 2, respectively	How well do dimension 1 and 2 alone explain the variation in this variable?
Quality	The representation quality of the variable by dimensions 1 and 2 together. Equal to the sum of SqCos 1–2. Value 1 corresponds to a perfect representation.	How well do dimensions 1 and 2 together explain the variation in this variable?

Table 1. Definitions and interpretations of the output statistics produced by CA

application of this article, analyzing social survey data via CA, in KNIME Analytics Platform. The KNIME workflow below shows the steps.

You can download the Exploring categorical data via Correspondence Analysis workflow from the KNIME Hub and open it in KNIME Analytics Platform. KNIME Analytics Platform is open source and can be downloaded from the KNIME website.

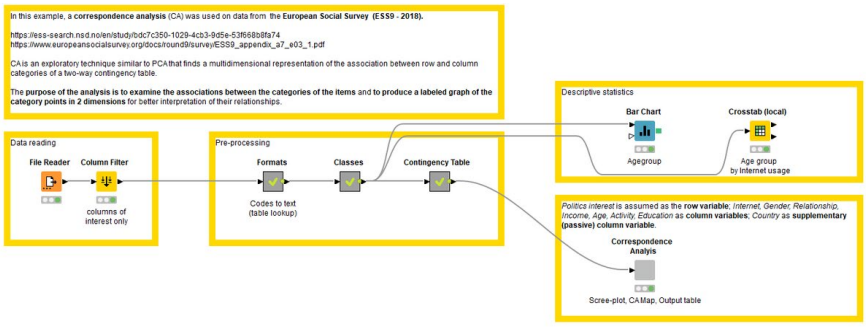


Figure 7. Example of KNIME workflow performing CA on European Social Survey data. You can download the workflow from then KNIME Hub.

The workflow progresses in four steps: data reading, preprocessing, descriptive analysis, and CA.

First, it accesses the data as a CSV file, which stores the data as shown in Figure 1. Second, it accesses the dictionaries that contain the descriptions of the codes to replace them in the data. After that, it bins some of the variables into fewer categories. Then, it creates the contingency table as shown in Figure 2.

Lastly, it computes the CA and produces a view that displays the scree plot, biplot, and statistics table (Figures 3-4 and Table 1). It performs CA using functions of the R software, in particular, the ca () function of the ca package. For the views, it uses the function ggplot () of the ggplot2 package. The KNIME Interactive R Statistics Integration allows us to write the script within the visual workflow.

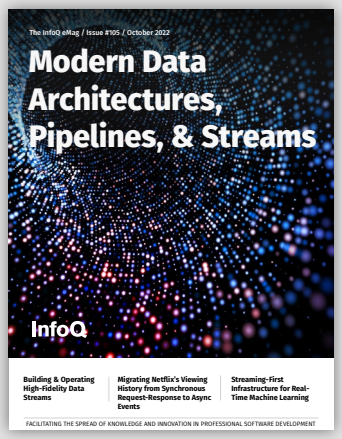
In addition, it displays a bar chart and contingency table to explore the frequency distributions in the data in parallel to performing CA (Figure 5).

Summary

In this article, we introduced correspondence analysis,

which analyzes associations in categorical data, and showed how it helps to analyze categorical data beyond a contingency table by projecting the categories of the variables onto new numeric dimensions. You can find these associations based on the proximity of the variables in a reduced feature space that could not otherwise be discovered through a pairwise analysis.

Read recent issues



Modern Data Architectures, Pipelines, & Streams

In this eMag, you'll find up-to-date case studies and real-world data architectures from technology SME's and leading data practitioners in the industry.

The InfoQ Trends Reports 2022

The InfoQ trends reports provide a snapshot of emerging software technology and ideas. We create the reports and accompanying graphs to aid software engineers and architects in evaluating what trends may help them design and build better software. Our editorial teams also use them to help focus our content on innovator and early adopter trends.

The Platform Engineering Guide: Principles and Best Practices

Platform Engineering has quickly become one of the hottest topics in DevOps. The explosion of new technologies has made developing software more interesting but has substantially increased the number of things that development teams need to understand and own.

InfoQ

InfoQ @InfoQ InfoQ InfoQ