Query Service—The Challenge of Developing a Flexible Managed Database Service

3.1 Introduction

IIJ began a Tech Challenge scheme in fiscal 2019 to provide opportunities for engineers to breathe life into new ideas for services and technologies they ponder on a daily basis. My project was selected for the inaugural Tech Challenge, and over the course of a year through September 2020, I worked on my own to design the service specifications and develop a prototype of the service.

The theme of my project was the development of a query service. While deploying applications in Kubernetes containers is common practice these days, we still do not have an optimal solution for using Kubernetes containers when it comes to database persistence, availability, and performance. My aim was to address these issues by developing a query service (a managed database service) in the form of an external service that runs alongside Kubernetes and provides the same level of flexibility as when using containers, as well as data persistence and availability.

3.2 Key Features Developed

So I had a big development theme set for the Tech Challenge—that of developing a query service—and while I could see what issues I needed to solve, once I began working on the project, I found I had to rethink specifically what features I would be developing. I spoke to administrators who develop and operate IIJ's services to learn about the issues surrounding databases. Those conversations made a lot of sense to me, and I identified some overlap with the issues in my existing development role. Several potential features that would be highly useful for developers/operators came out of this, and I set to work laying out the requirements and designing and developing the query service.

Databases are a crucial component when developing and operating services, but building and operating a database is a heavy burden for the small teams that focus on developing and running service applications. Developers and operators do not want a database server. They want database functionality that allows them to connect via a client to a data store and use CRUD operations. Moreover, when it comes to non-functional requirements, they want these to be met, but they do not want to deal with them. So I saw the need for a service that completely hides (from the user) the deployment of a database before it is used and the design of instances in a way that takes into consideration high database availability, backups, security configuration, and performance. Yet these really are features you would expect from a database service, and I didn't feel the query service would be offering anything novel in that respect.

Yet there was something that really made sense to me in what the administrators who develop and run IIJ's services were saying. When a service application's execution speed drops off, they wanted some way of just powering through it. For developers, the standard responses when database performance fails to improve include tuning SQL queries and adding indexes. But when an application suddenly slows down, a feature that sort of "magically" increases database power without having to stop the application would be wonderful. This is something that I certainly would have benefited from when I was developing services, so I decided to implement this sort of magic feature into my query service. This is something that not even Kubernetes database operators offer, so the technical challenge it posed made it feel all the more worthwhile and motivating for me.

One other issue came up quite often in my conversations, which can be summarized as follows: "Data outlives systems, so we need long-term access to the database, but we have to migrate the data every time the system is replaced. The database itself also needs to be updated to new versions, but no one in our team is very familiar with the process, so we continue to use the old version as is." I spent many years in our system integration team providing technical support on a lot of database migrations and version updates due to customer system replacements, and I know firsthand how exhausting such projects can be.

I knew the query service should naturally allow users to use the same database, and that I should aim for the hardware and software on which the database runs to be constantly upgraded, but I also understood that it was crucial for this not to put a burden on the users. This is the sort of thing Kubernetes' rolling upgrades aim to achieve, and I think this would be a similar feature, but I developed a completely original implementation for the query service.

Although I had a year, the reality is that I was actually developing it entirely on my own. There was a chance it would end up half baked if I tried building all sorts of stuff into it. But I knew it had to have the bare minimum for modern-day databases in terms of the non-functional requirements of high availability and backups. In terms of what makes the query service distinct, I wanted it to solve the issues that I and the engineers I knew faced and, in line with the initial concept, to provide the same or greater level of functionality as databases in a Kubernetes container. So I moved forward with the following as my development priorities.

(1) Develop features that let users control the performance of the database without having to stop the database
(2) Develop an interface to allow (1) above to be used freely and easily from within the application
(3) Develop flexible billing functionality that makes (1) above easy to use
(4) Functionality that lets the system control (1) above on behalf of the user and always provide optimal performance
(5) A service update feature to facilitate ongoing use of the query service

Figure 1 is a conceptual overview of the features developed. At its core, the service uses Oracle Database (we are looking at supporting other databases too).

Starting in the next section, I describe the query service's core features, with (1) and (2) above covered in "Online Resource Reallocation Feature", (3) in "Per-second Billing Feature", (4) in "Autoscaling Feature", and (5) in "Service Update Feature".

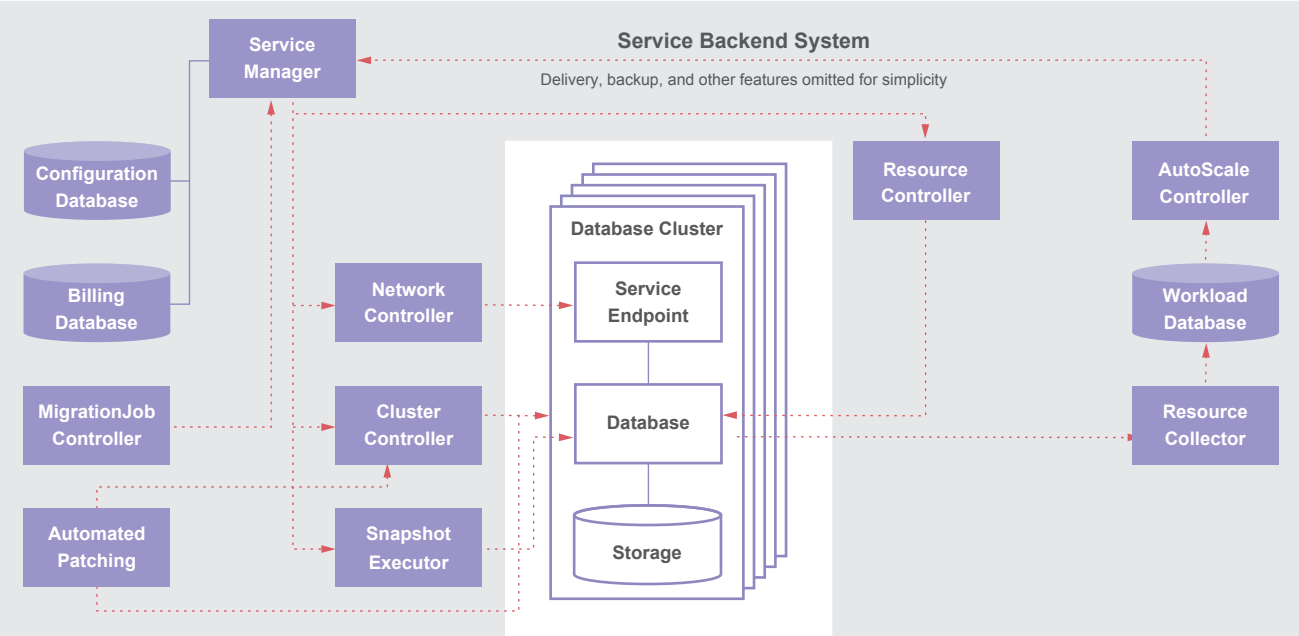

Figure 1: The Orchestration System Developed

3.2.1 Online Resource Reallocation Feature

To address the first development priority—functionality allowing the user to freely control database performance without stopping the database—I developed an online resource feature.

Setting databases up to easily allow for a distributed configuration to be adopted as loads increase, like with Web and application servers, is difficult. Replication, sharding, and configurations that sacrifice integrity are often used to facilitate scalability. The effects of such strategies are limited to read operations, however, and they come with restrictions on transaction processing and so forth. Particularly when performance issues arise with a single, large, database-specific process, scalability through load balancing can be ineffective.

A database's multi-workload capabilities are another aspect of its performance, and characteristically the amount of resources needed by a database can vary greatly depending on how it is used. Database processing varies greatly mainly depending on what combination of the following is in play: (1) SQL syntax, (2) amount of data to be processed, and (3) number of concurrent processes.

What combination of these factors is in play changes substantially throughout the day (Table 1). For example, many users will use a database concurrently during the day, but each operation is small. At night, however, daily jobs are executed, and while only a few of these run concurrently, it is not uncommon for any single operation to be large. Adding to this are seasonal fluctuations and spontaneous events, making workload even more complicated.

Table 1: Characteristics of Database Operations

	Day	Night
SQL syntax	Simple	Complicated
Data volume per operation	Low	High
Concurrent operations	Many	Few
Desired performance characteristic	Response	Throughput
Most important resource	CPU/memory	I/O

In traditional on-premises computing environments, databases have always had the most resources allocated within a system given that they combine different characteristics using inelastic resources.

In the last decade, it has become commonplace to use cloud services for computing resources. The days of people being skeptical about running databases in the cloud are long gone, and databases now run on cloud services seemingly as a matter of course (this is reminiscent of today's efforts to run databases on Kubernetes).

The multitude of computing resources that cloud services provide has greatly expanded users' resource options, but databases in the cloud do have their issues. If you build and run a database on IaaS, for instance, just because the location of virtual machines shifts into the cloud does not mean the fundamental problems you face when working with an on-premises system are solved. And a lot of the database services that cloud vendors provide on a PaaS basis need to be restarted when CPU resources, in particular, are reallocated. Stopping the database means stopping the entire service, so resources reallocations cannot be executed on a whim.

Scaling out a database with Kubernetes and Kubernetes operators is also effective and has gained a lot of attention of late (with replication, increasing the system's overall processing capacity is effective in the case of high-volume load balancing). But as discussed above, scaling up is probably more effective than scaling out when a single operation is slow (NewSQL is also an option, but I personally don't think it's quite mainstream yet). "Well then don't write inefficient

SQL," I hear you cry, but I think others working in this area will understand that this actually happens a lot in reality. I do not mean to criticize the scale-out strategy. In fact, I like scaling out, and while I do not mention it in this report, my query service also supports scaling out.

But I digress. Turning back to the query service, in an effort to increase processing performance without stopping the database, I developed an online resource reallocation feature that abstracts the computing resources. It provides the following functionality.

(1) CPU and I/O data throughput affects database performance, and this feature makes available as much of this as is needed when it is needed, without having to stop the database.
(2) It provides an interface in SQL so that resources can be easily reallocated from within the application that is the source of the resource request.

Table 2 shows (1) expressed as a service specification.

The CPU core and I/O data throughput resources can be increased or decreased separately, from the basic level up to the maximum spec. The change in resources happens within a few seconds, as I will discuss, and this is immediately reflected in the billing information.

One characteristic of (2) is that the online resources feature can be used via SQL as well as an API. The online resources feature is easy to understand even when controlled manually, and I developed it so that it would be easy to reallocate resources from the application that is the source of the resource request, so it is easy to embed into programs. This makes it possible to use resources in a way not previously possible, since you can, for example, change the number of CPUs before and after executing a large processing operation, and it is easy for developers to implement this (Figure 2).

Table 2: Query Service Specifications

	Fixed	Variable	
Resource	Basic	Options	
		Maximum spec	Increment
Database	1	—	—
CPU score	1	6	1
Data throughput (MB/s)	100	2000	100
No. of concurrent connections	50	300	Linked to CPU score
Data area (GB)	50	8000	Automatically increased

```
        if maxpom <= 2000 and maxgon >= 100 then
            dbms_output.put_line('Current max power ==>'||rc1.ag_power);
            vSQL := 'exec cpumod(6)';
            execute immediate vSQL;
            insert into testtab as select * from testman;
            commit;
            vSQL := 'exec cpumod(2)';
            execute immediate vSQL;
            select  testcol, to_char(modified_datetime,'YYYY-MM-DD HH24:MI:SS') as monday into testaa
            dbms_output.put_line('New maxmbps count ==>'||testaa);
            dbms_output.put_line('Resource was modified at '||moddatechar);
        else
            dbms_output.put_line('ERR-xx1 : Invalid argument [ '||aaaaa||' ]');
            dbms_output.put_line('ERR-xx2 : Valid argument is between 100 and 2000 ');
        end if;
    end loop;
end;
/
```

Change no. of CPUs to 6

Parallel processing to increase performance

Change no. of CPUs to 2

Figure 2: CPU Parallel Processing

Next, I explain how the online resources feature works. If we're going to provide a paid service, then allowing users to directly change the number of CPU cores and I/O throughput would quickly put us out of commission. Hence, although the service provides an interface via SQL procedures, the system changes are not made by executing a simple program that wraps the SQL command for making the change. Instead, the resource reallocation request is sent to the backend system's service manager via an external program (Figure 3).

So the procedure used to execute a request via SQL or the API is a simple program that takes the user request, determines on the user database whether the values are valid, and passes the request to an external program.

The service manager, having received the request via the external program, changes the information about the user database in the backend system's configuration database and changes the user database's billing criteria in the billing system. It then runs the database resource manager via the resource controller to change the user database's CPU core and I/O throughput configuration. Once this is done, it returns a message saying that the configuration is complete to the connected user session via the procedure on the user database (Table 3).

I don't have enough space to explain the program's implementation in detail, but these processes complete within a few seconds, so the user is able to adjust resources almost in

Table 3: Query Service Billing

	Daytime	Billing increment	Charges
Basic	IIJ Query Service, basic fee	1	Fixed monthly
Options	Additional CPU cores	1 core	
	Additional data throughput	100MB/s	Per-second billing
	Data capacity of 51GB or more	1GB	

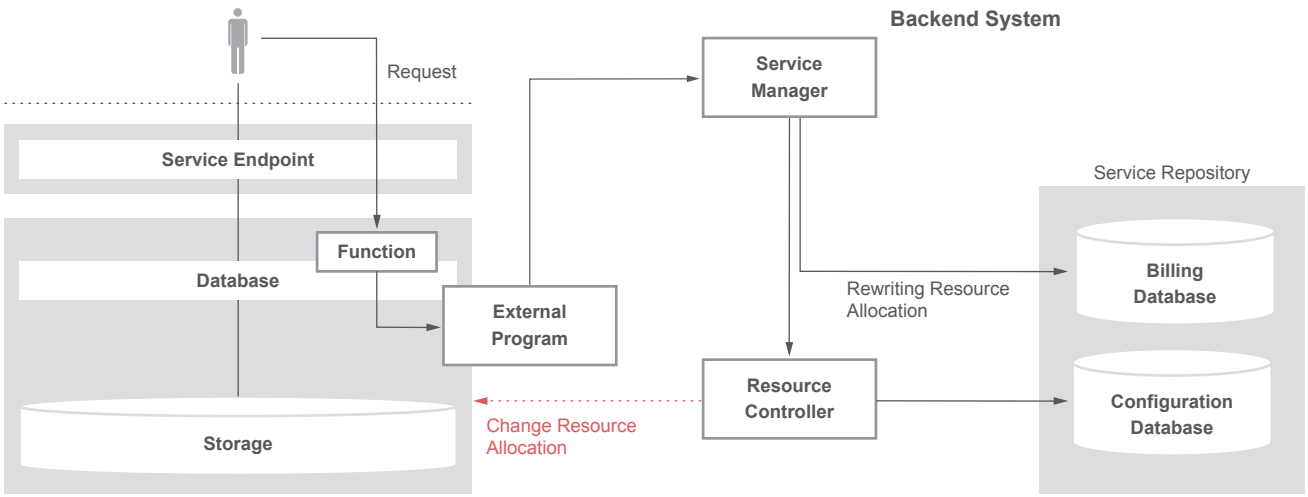

Figure 3: Resource Reallocation Backend

real time. Although possibly an extreme example, embedding the online resource reallocation feature into a program opens the door to an entirely new way of using database services as it means an application can dynamically measure the amount of data to process and dynamically change the resource allocation for every operation based on how much data there is (Figure 4).

3.2.2 Per-second Billing Feature

The online resources feature makes it possible to expand and shrink the resource allocation at any time, but it wouldn't be convenient if the resources billing were overly rigid. Aiming for a flexible billing structure for the query service, I also developed per-second resources billing for when the basic specs (CPU cores, data throughput, etc.) are exceeded.

Figure 5 illustrates how the query service pricing is implemented. CPU core counts, data throughput, and data area that exceed the basic spec are each billed for separately. This means users are billed only for the resources that they use when they need them.

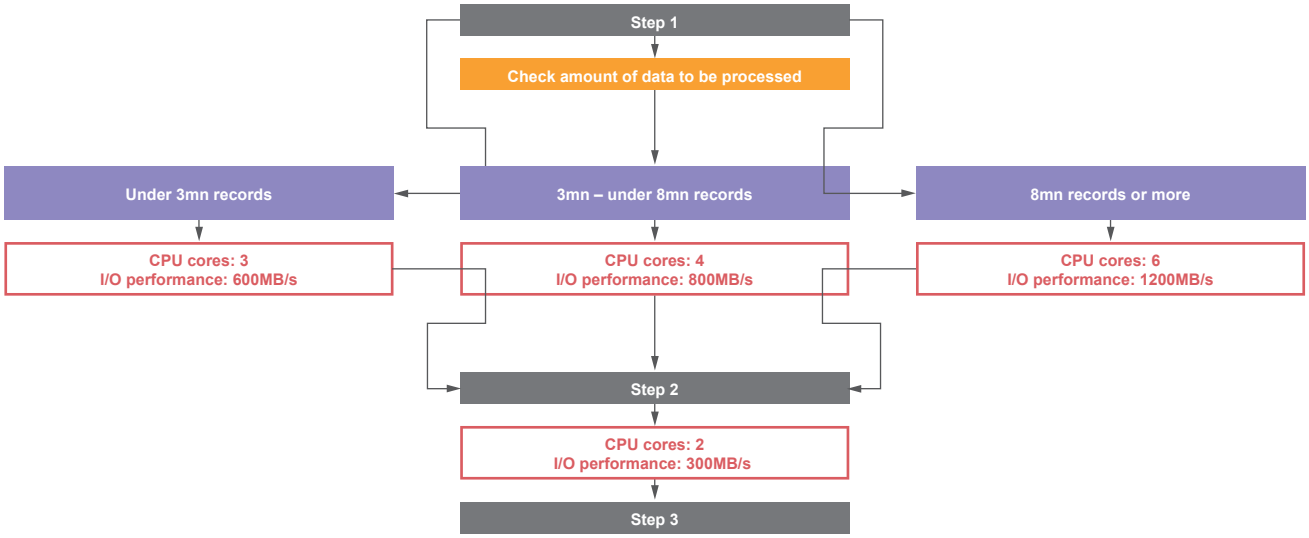

Figure 4: Programmable Resource Control

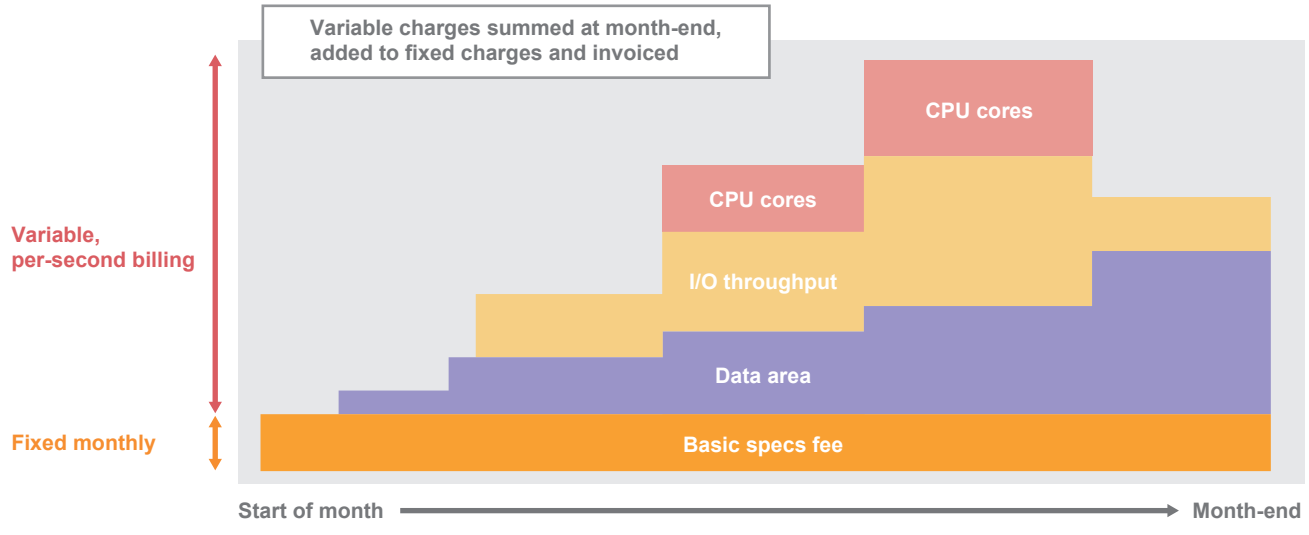

Figure 5: How Query Service Billing Works

If this can be combined with parallel database processing to balance processing speed and increases in resources, then we can keep the impact on overall charges small even as the per-second charge rises (Figure 6). I believe that more flexible billing functionality is what will encourage people to use the online resources feature.

3.2.3 Autoscaling Feature

The autoscaling feature works in conjunction with the online resource reallocation feature and constantly provides optimum performance by automatically controlling the system on behalf of the user. The online resource reallocation feature is useful, but if someone has to make a decision to

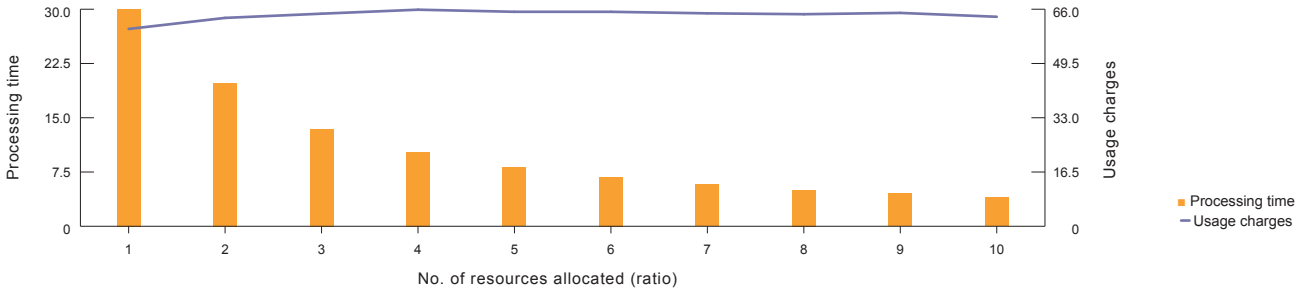

Figure 6: Online Resource Reallocation Feature and Per-second Billing

Table 4: Autoscaling Feature

Resource	Minimum	Maximum	Increment/decrement	Criteria for increase	Criteria for decrease	Decision interval
CPU	1 core	6 cores	1 core	Average usage of all allocated CPU cores over the last 5 minutes		1 minute
				70% or higher	65% or lower	
I/O throughput	100MB/s	2000MB/s	100MB/s	Average usage of allocated I/O throughput over the last 5 minutes		1 minute
				80% or higher	75% or lower	

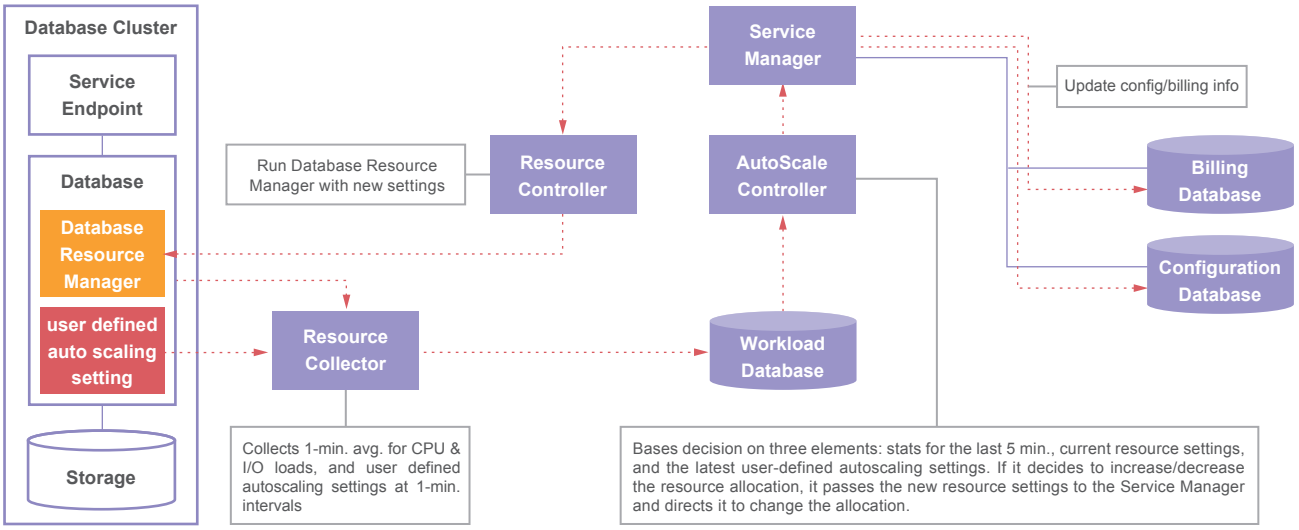

Figure 7: Internal Workings of the Autoscaling Feature

run it manually each time, and if it requires a complicated setup to combine it with links to an external monitoring system and so forth, this would greatly reduce the benefits of having it online and diminish the appeal of the feature. To automate system operations and facilitate the use of the query service's convenient features, I also developed an autoscaling feature to automatically control the online resource reallocation feature (Table 4).

Figure 7 shows the internal workings of the autoscaling feature. The autoscaling feature collects database operating status information, and in response to load levels, it automatically allocates, via the online resource reallocation feature, resources that affect database performance, namely CPU cores and I/O data throughput. It works constantly to keep resource loads within the criteria for either increasing or shrinking the resource allocation.

Autoscaling is a very useful feature with various use cases. One situation in which it is probably highly effective is that of dealing with performance degradation. If you run systems, you will come up against sudden slowdowns in database performance several times a year. This can have a variety of causes. Sometimes it suddenly slows down even though you haven't done anything (often, the database execution plan has suddenly changed), and otherwise, the trigger is always something different: a new program has just been resourced, or the volume of data has increased substantially, or you've run an event like a bargain sale, for example. When performance degradations visit, you face a battle against many conflicting forces.

FIrstly, it can be difficult to detect an application slowdown via resource monitoring on the infrastructure side, meaning that the operations team can be slow to detect what is happening. In many cases, user complaints are the trigger, and you face a high bar in dealing with the issue from the outset. Also, in some cases the operation causing the problem can be stopped at the process level, but stopping is not an option in other cases, such as nighttime batch jobs that have a significant business impact if not completed by the start of business hours. This is a tough situation: you cannot stop the operation, but you have to deal with it immediately. The root cause of these slowdowns is almost always on the application side and can include data spikes, queries running based on improper execution plans, and searches on items with no index. But even if you can identify the cause, dealing with it without stopping the system is either impossible or a highly difficult operation. And what's worse is that these sorts of disruptions often occur on holidays or late at night when no one is around.

While we might not be able to fundamentally resolve performance degradations riddled with conflicting issues like this, if the service could provide a solution, this would make users, operators, and developers all happy, and this is what I had in mind when developing the autoscaling feature.

■ Measuring the Effects of Autoscaling

Deliberately causing a performance degradation to see what effect the autoscaling feature has is difficult. Instead, I executed large operations on the database that were beyond the capabilities of the current specs to see how the autoscaling feature would react.

In this test, I joined several tables together—including an orders table with over 100 million records, a product master, and a customer master—and executed a query. Since this test makes it easy to tell what effect autoscaling has, I cleared the shared memory buffer each time a query finished executing.

When the query is executed, all records in the orders table are accessed sequentially. Running queries that cause sequential access and flush shared memory every time result in a large number of blocks located in storage being read out. So the queries require a lot of I/O resources.

When initialized, the query service has the basic specs (it's minimum configuration), which means a modest 100MB/s

of I/O throughput performance, so executing a sequential scan of over 100 million records drives the I/O throughput resource usage up sharply as soon as it is started (Figure 8). The autoscaling feature constantly collects and evaluates database usage statistics, and directs the system to increase or decrease the resource allocation. The queries used in the test were I/O-bound, so the autoscaling feature continued to increase I/O throughput performance only, up to 900MB/s.

What's interesting is that when I/O throughput reaches 900MB/s, the CPU reduces the I/O wait time, so the process becomes CPU-bound. When this happens, the service tries to increase the degree of parallelism by increasing the CPU core count, but since the queries were not the sort that raise CPU usage all that much, the autoscaling feature began to let go of the CPU cores again. This can be called online scaling up, but with containers on Kubernetes, it looks like this implementation has not yet reached a stable version. While interesting to watch, this sort of CPU core catch-and-release behavior will cause performance fluctuations, so there is certainly room to improve. Yet I don't think it's that big of a problem. With this sample database, the autoscaling feature is able to increase resources up to at most six CPU cores and 1000MB/s, but it determines that one or two CPU cores and I/O throughput of 900MB/s is adequate. Bumping it up to 1000MB/s or more and increasing the CPU core count does indeed increase performance, but the change is not remarkably large.

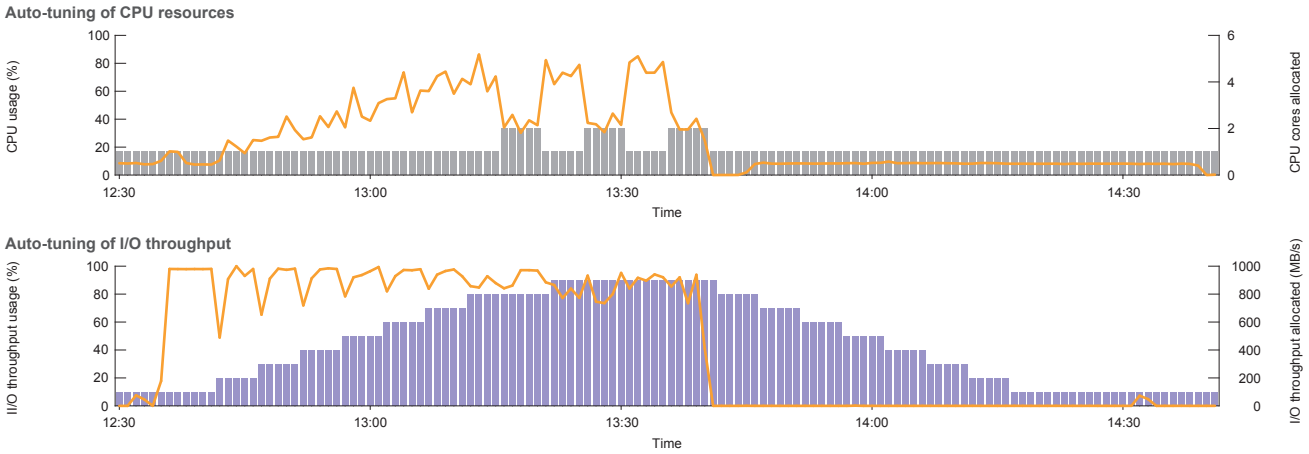

Figure 8: Example of Changes in Resources Controlled by Autoscaling Feature

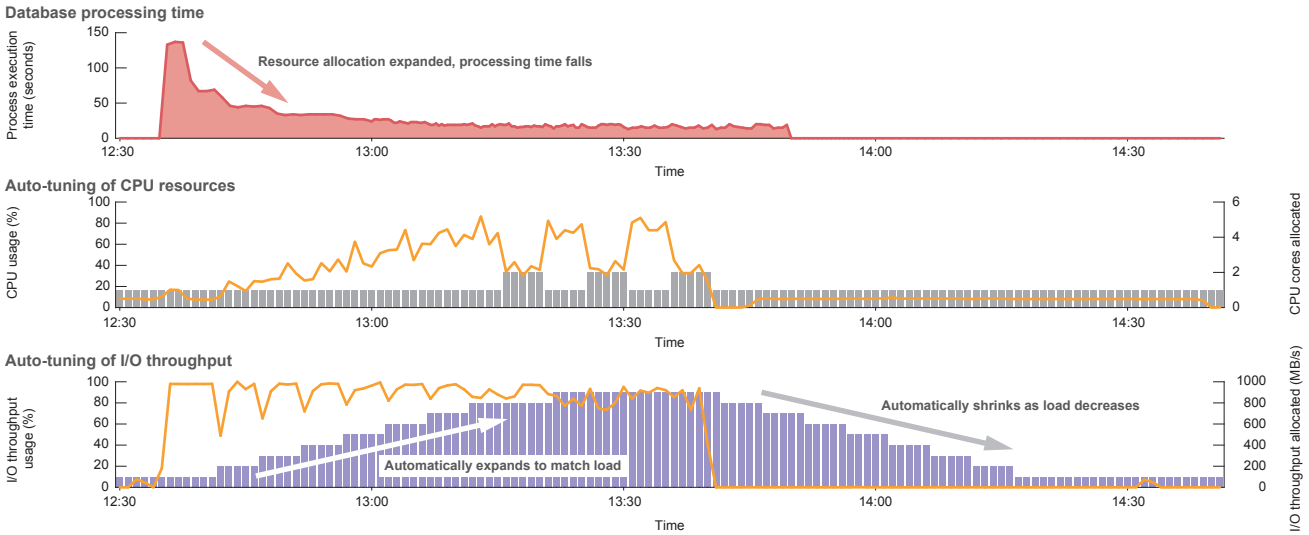

Figure 9: Example of Changes in Database Processing Time due to Autoscaling

Figure 9 shows the impact on actual processing performance. The initial increase in I/O throughput has a huge effect, and processing time certainly does drop.

Now, what happens if the user manually sets the I/O throughput performance value to 2000MB/s while the autoscaling feature is enabled? Manual settings take priority, so the database I/O throughput performance will be set to 2000MB/s. If autoscaling is disabled, the value will stay at 2000MB/s, but with it enabled, the system will assess operating conditions and shrink the resource allocation down toward 900MB/s.

The autoscaling feature could get in the way if it always reduces throughput to the minimum 100MB/s before then increasing it, resulting in delays in achieving the desired performance, so users are able to set a parameter range for the autoscaling feature to work within. For example, if it is configured to automatically adjust in the range of 2–4 CPU cores and 500–1500MB/s, it will not drop below 500MB/s. Users can also set autoscaling to adjust I/O throughput only and leave CPU core count unchanged. Changes to the settings can be made online, and these are reflected in the system after a delay of one minute (for reasons to do with the backend system).

■ Issues with Autoscaling

To ensure resources are used properly, the user cannot change the resource decision interval, thresholds, or increment/decrement values in the backend system, but I am looking at changing to an implementation that does allow the user to change these values as well.

And while autoscaling may seem useful, many issues remain. Especially in terms of resource allocation accuracy, I still face many difficulties from a developer standpoint.

As discussed, autoscaling operates on the basis of database usage statistics. Yet these figures are based on past occurrences, and the core autoscaling functionality, the autoscale controller, operates on the very simple assumption that past trends will persist for a while into the future, so it is not some sophisticated system capable of predicting future load levels ahead of time. Also problematic is that this clear and simple approach often deviates from desirable values immediately after a time-series regime shift. In specific terms, it sometimes increases resources despite loads being in decline. I'm hoping to make the autoscaling features even smarter via machine learning and so forth.

So while autoscaling has some remaining issues, I think it is one effective means of dealing with sudden performance degradations. In the end, since everything is controlled automatically, even if you are slow to detect an issue or it is discovered because of customer complaints, the autoscaling feature will be working in the meantime to expand the resource allocation in an attempt to maintain performance. And the resources allocation is shrunk once the issue is resolved, so you may not even notice it in some cases (although it will appear in your service charges). Configuration management and application deployments can be automated, and I think automating system performance maintenance as well will make system operations even easier.

3.2.4 Service Update Feature

While the query service espouses a serverless setup, it runs in the same system environment as a normal database, so the service platform will get old. Firstly, as the service platform's hardware ages, the incidence of mechanical failures rises. Security holes and bugs in the OS and database software stop being fixed because they are no longer supported or patches are no longer being released. So service platform renewals are crucial to continuously provide stable services.

Preparing a new query service platform is easy, but the current user database needs to be migrated to the new platform. In-place upgrades are the easiest way of dealing with simple version updates to the database alone, but version updates take considerable time, so lengthy service outages are unavoidable. There is also the possibility of OS

and hardware being inadequately renewed or of separate OS/hardware renewals being required, so this is not an efficient way of doing things. Another way is to create a separate instance from a backup, but this can involve changing the client's connection endpoint, which raises the possibility of the work required going beyond the scope of the query service.

The query service completes all of the following in a single process: switching over to new servers and storage, OS version updates, database software version updates, and upgrades to the database itself. This is similar to Kubernetes' rolling upgrades, but the query service does not use anything like replication. The mechanism implemented upgrades the entire service in a way that is faster and more transparent to the user.

(1) No data migration required
(2) All steps fully automated, including version updates and database switching that preserves data integrity
(3) A single click by the user will complete the entire process
(4) Takes at most 15 minutes to complete
(5) Can be run as many times as you like
(6) The query service connection endpoint does not change after the migration

The query service is equipped with functionality to perform the above service updates and allow continued use of the database. The service update feature does not require data to be migrated. Well, to be more precise, the in-use database is replicated on the new service platform under the hood, and the user does not need to think about it at all. Data integrity is automatically handled by transferring the transaction log.

The service update can be broadly divided into three phases. In Phase 1, a snapshot of the user's database is created on the new service platform online (Figure 10). The user does not need to create the snapshot manually. The service keeps tabs on whether it is possible to take a snapshot and creates one automatically when it is. Even if the user has multiple databases, the process is performed completely independently for each database.

From Phase 2, the database is switched to the new service platform, so the user needs to trigger the process, which can be done at a time of the user's choosing. Once Phase 2 is started, the service endpoint for the relevant user database is closed (Figure 11). This determines the quiescent point for reverting the switch. Once the quiescent point is

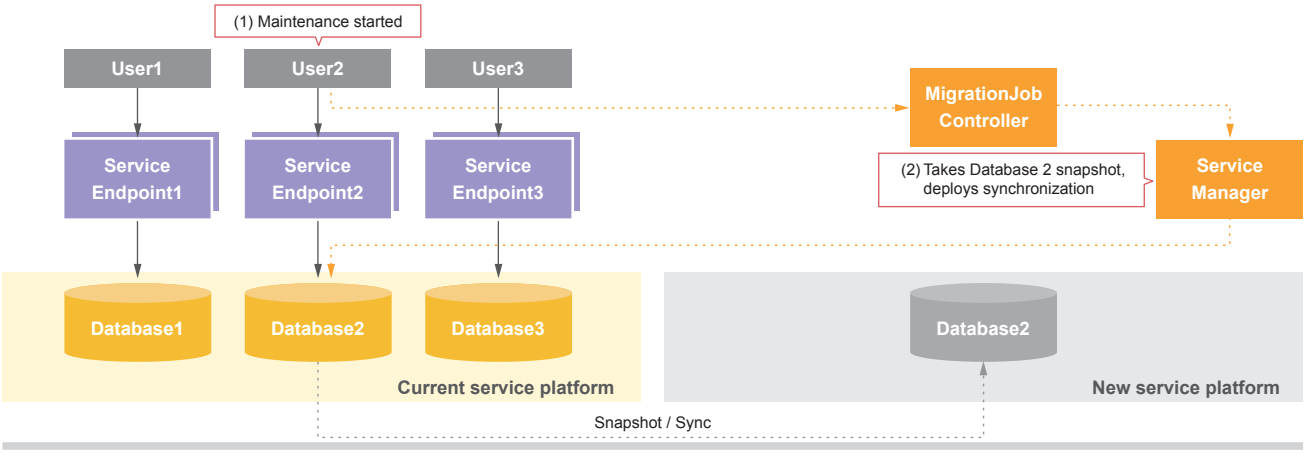

Figure 10: Internal Workings of Query Service Update Feature 1

determined, the final synchronization with the snapshot on the new service platform is done. Once the final synchronization process between databases is completed, the status of the database on the current service platform becomes inactive and the database goes offline, but it is not physically deleted. This means it is effectively a backup for the service update process. Plus, it avoids the time needed to restore from snapshot if the user rolls back the update. All that needs to be done is to change the status to active, so reverts happen very quickly.

The new database stands by on any processing until the current database becomes inactive. This is to avoid a split-brain scenario caused by both databases being active at the same time. So it doesn't process anything independently. Once the current database is properly inactive, processing resumes so that the service can continue on the new database. If the database version is to be updated, this is when it happens. The new database is also reconfigured for high availability. In the final stage of Phase 2, route information on the user database service endpoint is refreshed to reflect the new service platform.

In Phase 3, the endpoint is opened once the user database on the new service platform goes to active status (Figure 12). A notable aspect of Phase 3 is that the users connection endpoint does not change; only the route information on the service endpoint is changed. So once the user receives notification that everything is complete and reconnects, they are connected to the user database on the new service platform without having to change anything.

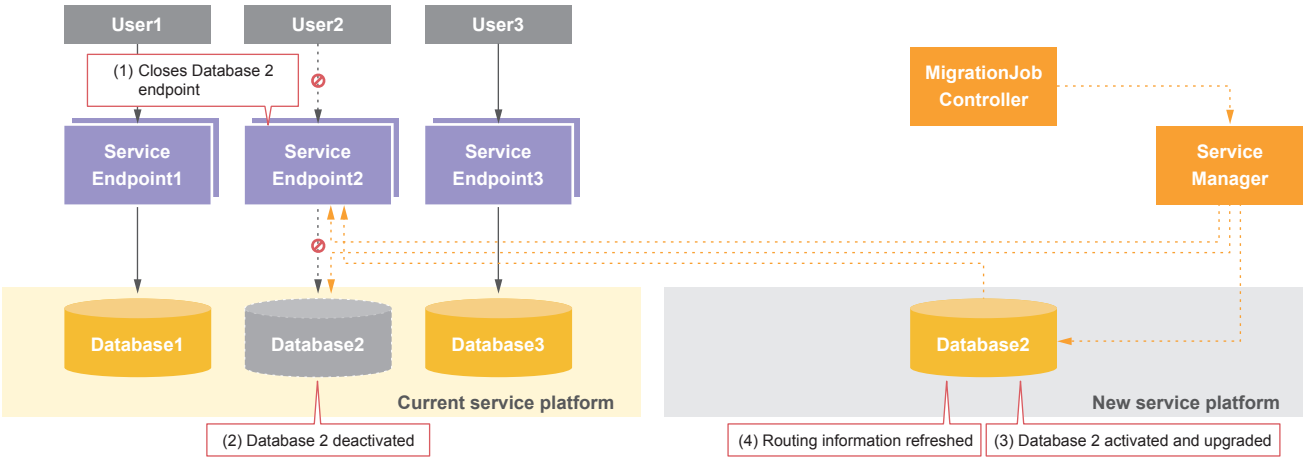

Figure 11: Internal Workings of Query Service Update Feature 2

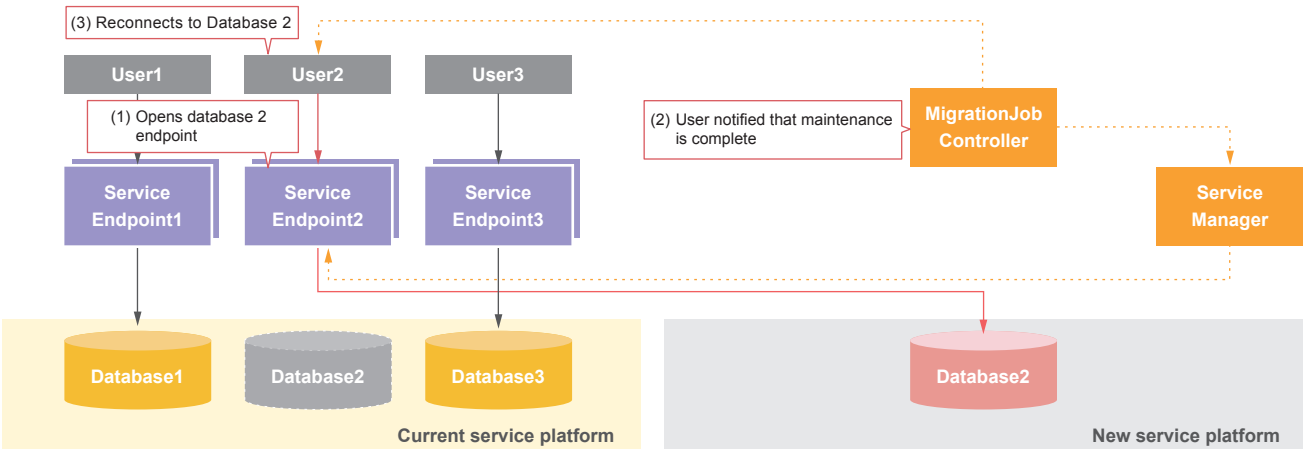

Figure 12: Internal Workings of Query Service Update Feature 3

When a switchover is executed, the service endpoint, which is a proxy between the client and the database, is blocked. Blocking the service endpoint eliminates the route between the client and the database, so user sessions are completely disconnected. Therefore, if any transactions were being processed in any sessions, these are rolled back on the database. This means it is best to initiate a switchover when no transactions are being processed. Once current database integrity is established with the service endpoint block in place, the final delta synchronization with the new database takes place.

The work done by the service update feature has traditionally been performed as a system integration project. Figure 13 shows what it takes to do this manually. Quite the laborious task. The query service automates all of this under the hood, so service users simply need to make a single click on the control panel.

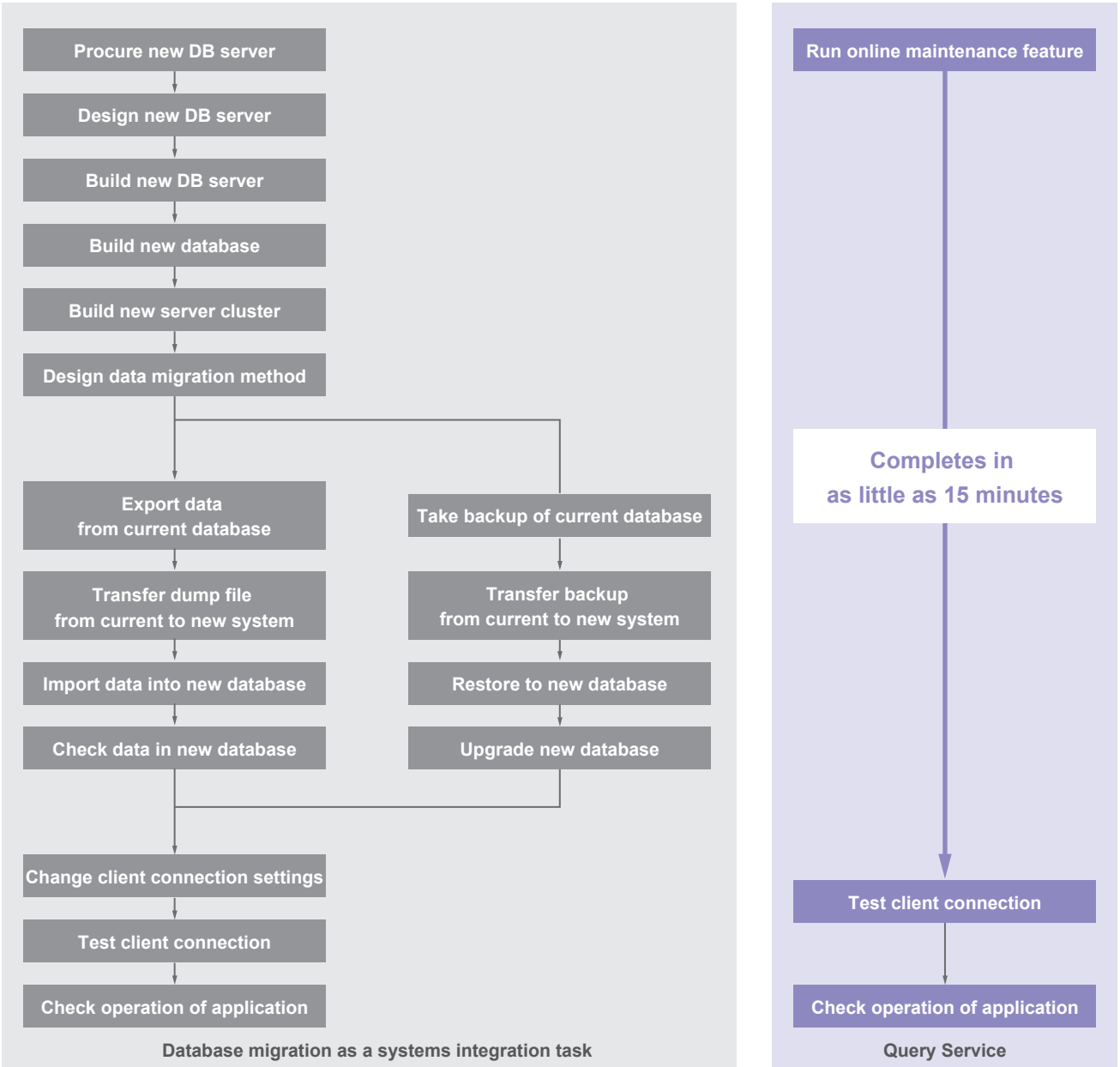

Figure 13: The Service Update Feature Greatly Streamlines the Platform Renewal Process

3.3 Conclusion

With the query service, my aim was not to develop a database itself, but to develop an orchestration system designed to make it easy for engineers to interact with databases, and while it is a prototype, I believe it has achieved that aim. Some readers may sense that I have a bone to pick with Kubernetes, but I actually rather like Kubernetes, and I would like to develop a query service using Kubernetes if I get the chance.

The Tech Challenge was different from the normal development routine guided by user requirements. I was able to bring my own ideas to life, and it was a most engaging and stimulating time for me as an engineer. It was a year that brought back all the simple joys of working with computers I had long forgotten since becoming a serious, working-age adult.

Tsutomu Ninomiya
Technical Manager, Service Planning Office, IIJ System Cloud Division.
Mr. Ninomiya is engaged in the planning and development of cloud services as well as technology support for projects in this area. He was originally a DWH/BI developer, and he likes SQL and parallel processing.