クエリサービス
～柔軟なマネージドデータベースサービスへの挑戦

3.1 はじめに

IIJでは、2019年度よりエンジニアが日頃考えている技術や新サービスのアイデアを具現化する機会として「テックチャレンジ」という制度が始まりました。この度、第1回テックチャレンジに筆者の企画が採択され、去る2020年9月末日までの1年間にわたり、サービスの仕様検討・設計、サービスのプロトタイプ開発を１人で行ってきました。

採択された開発テーマは「クエリサービスの開発」です。昨今、アプリケーションがKubernetes上のコンテナへデプロイされることが当たり前の時代となっているものの、データベースに関してはデータの永続性、可用性、パフォーマンスの面でKubernetesのコンテナ利用にまだ最適解といえる手法がないのが実状です。そこで、コンテナを利用した際と同等の柔軟性に加えデータの永続性、可用性を持つマネージドデータベースサービスであるクエリサービスを、Kubernetesに隣接する外部サービスとして開発することで、これらの課題を解決しようというのが今回の狙いです。

3.2 重点開発した機能

テックチャレンジとして「クエリサービスの開発」という大きな開発テーマが設定され、解決するべき課題は見えていましたが、具体的に開発する機能は着任後に改めて考える必要がありました。機能の開発に当たっては、IIJのサービスを開発・運用している管理者の方々からデータベースにまつわる課題をヒアリングしました。話を聞くと、それらは筆者が開発担当者であった当時抱えていた課題と重なるものがあり、大変共感を覚えました。開発者・運用者の視点で「こんな機能があれば助かる」という機能を複数挙げ、要件の整理を行い、クエリサービスの設計・開発へ進みました。

データベースはサービスを開発・運用する上で必須の構成要素であるものの、サービスのアプリケーション開発や運用が主務の部門が少人数のチームでデータベースの構築や運用を担当するのは重荷となります。サービス開発・運用者にとって使いたいのは、クライアントから接続してデータを格納し、CRUD操作ができる「データベースの機能」であって「データベースサーバ」ではないのです。更に言えば、その非機能要件になると「あってほしいけれど関わりたくない」という存在になっています。従って、データベースを使い始めるまでのデプロイ、データベースの高可用性構成やバックアップ、セキュリティ構成、パフォーマンスを考慮したインスタンスの設計は利用者からは完全に隠蔽されるサービスであることが必要だと考えました。ただ、これはデータベースのサービスとしては言わば当たり前の機能であり、クエリサービスならではという新規性は感じませんでした。

一方、IIJのサービスを開発・運用している管理者の話を聞くうちに筆者も大きく共感したことがありました。それは、サービスのアプリケーションの実行速度が遅いときは「力業」で何とか乗り切りたいということです。開発者にとってデータベースのパフォーマンスが上がらないときはSQLのチューニングや索引を追加するなどの手法が正攻法ですが、アプリケーションが急にスローダウンしたときには、まずはアプリケーションを止めずにデータベースのパワーが上がる「魔法」のような機能があればとても助かるというのです。筆者自身が開発担当だった頃でもあれば助かる機能でしたので、クエリサービスにはこの魔法のような機能を実装しようと考えました。これはKubernetesのデータベースOperatorでも提供されていない機能で、技術的にチャレンジし甲斐があるというのもモチベーションとなりました。

課題としてよく話に挙がってきたことがもう1つあります。「データはシステムの寿命より長いのでデータベースは長く使いたいが、システムのリプレースの度にデータ移行を行わなければならない。更に、データベース自体もバージョンアップしないといけないのに詳しい人が部門にいないので塩漬けされた状態で使い続けている」という課題でした。筆者もシステムインテグレーション部門でお客様のシステムのリプレースに伴うデータベースの移行やバージョンアップを長年にわたり多くの案件で技術支援し、その対応に疲弊するプロジェクトの現場を経験してきました。

クエリサービスでは同じデータベースを使い続けられるのはもちろんのこと、そのデータベースが稼働するハードウェアもソフトウェアも常にアップグレードしていくものを目指すべきと考えましたが、これらが利用者の負担にならないことが非常に重要だと理解しました。このようなことを目指しているのがKubernetesのローリングアップグレードであり、似た機能になると思うのですが、ここはクエリサービス完全オリジナルの実装を目指して開発を行いました。

1年間という時間はあれど、現実的な話として実際に開発を行うのは筆者1人です。いろいろ手を出しても中途半端になる可能性はありました。しかし、やはり現代のデータベースサービスにおいて最低ラインとなる高可用性構成やバックアップといった非機能要件の機能開発はもちろん、クエリサービスの独自性として身近なエンジニアや自分自身が抱いていた課題解決、また当初のコンセプトのとおりKubernetes上のコンテナのデータベース同等またそれ以上の機能を提供するべきと考えました。そこで、以下を重点開発項目に掲げて開発を行いました。

① データベースを止めることなくユーザが自由にデータベースのパフォーマンスをコントロールできる機能を開発
② ①の機能をアプリケーションから簡単かつ自由に使えるインタフェースの開発
③ ①の機能を使いやすくする柔軟な課金機能の開発
④ ①の機能をユーザに代わりシステムが自律制御し、常に最適なパフォーマンスを提供する機能
⑤ クエリサービスを継続利用してもらうためサービスをアップデートする機能

開発した機能群を概念図で示したのが図-1です。なお、今回はコアとなるデータベースにはOracle Databaseを採用しています(他のデータベースへの対応も検討中です)。

次項から、①②については「オンラインリソース機能」、③は「秒課金機能」、④は「オートスケーリング機能」、⑤は「サービス更新機能」と分けて、クエリサービスのコア機能を紹介します。

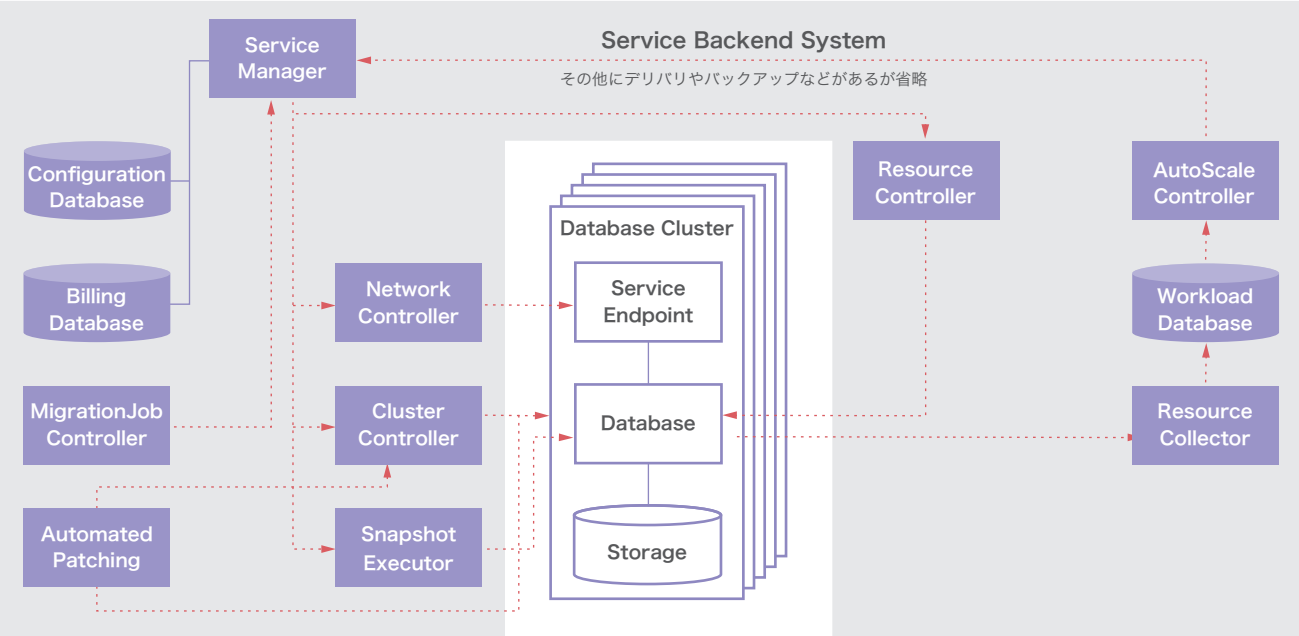

図-1 開発したオーケストレーションシステム

3.2.1 オンラインリソース変更機能

重点開発項目の1つ目に、データベースを止めることなくユーザが自由にデータベースのパフォーマンスをコントロールできる機能として「オンラインリソース機能」を開発しました。

データベースは、Web/アプリケーションサーバのように負荷の上昇に併せた分散構成を容易に構成することが難しいとされており、スケーラビリティ構成としてレプリケーションやシャーディング、または整合性を犠牲にした構成が用いられることが多くあります。しかしながら、それらは読み込み処理に限定された効果である、またはトランザクション処理に制限が生じるなどの制約が付きまといます。特にデータベース特有の大きな1つの処理にパフォーマンスの問題が生じた場合は負荷分散でのスケーラビリティでは効果が得られないと言えます。

また、データベースのパフォーマンスの特性としてマルチワークロードと言われており、同じデータベースに対しても使い方により必要となるリソースが大きく変わってくることも特徴として挙げられます。データベースの処理は主に①SQL構文②処理対象のデータ量③処理の同時実行数の組み合わせにより大きく異なります。

これら要素の組み合わせは1日の中でも大きく変化します(表-1)。例えば、日中帯は多くのユーザにより同時に利用されるのですが、1つの処理が小さい。一方で夜間は日次ジョブが実行され、同時に実行される数は少ないが、1つの処理が大きいことが珍しくありません。これに季節変動や突発的なイベントの発生などが重なると更にワークロードが複雑化していきます。

	日中	夜間
SQL構造	単純	複雑
1処理あたりのデータ量	小	大
同時実行数	多	少
求められる性能特性	レスポンス	スループット
より重視するリソース	CPU/メモリ	I/O

表-1 データベースの処理特性

従来のオンプレミスと呼ばれるコンピューティング環境では、異なる特性の論理積を弾力性のないリソースで実現するため、データベースには常にシステムの中で最も大きなリソースを用意してきました。

この10年でコンピューティングリソースはクラウドサービスを利用することが当たり前の時代になりました。かつてクラウド上でデータベースを動かすことに懐疑的であった時代が嘘のような話となり、当たり前のようにクラウドサービス上でデータベースが動いています(今日ではKubernetes上でデータベースを動かす取り組みが似ているように見えます)。

クラウドサービスが提供する多種多様なコンピューティングリソースにより、ユーザのリソース選択肢は大幅に拡大していますが、そんなクラウドサービス上のデータベースにも課題はあります。例えば、IaaS上にデータベースを構築・運用している場合は、仮想マシンが起動するロケーションがクラウドに変わっただけでオンプレミスとの運用における根本的な課題は解決されていないと言えます。また、クラウドベンダーがPaaSとして提供するデータベースのサービスの多くは、特にCPUのリソースを変更する際に再起動が必要となります。データベースの停止はサービス全体の停止となるため、気軽にリソースの変更が実行できない状況です。

昨今、注目度が急上昇のkubernetesとそのOperatorによるデータベースのスケールアウトも有効ではあります(レプリケーションを使用する場合、負荷分散は実行数が多い場合にはシステム全体の処理量を上げることが有効な手段であると思いますが)。しかし、前述のとおり1つの処理が遅い場合にはスケールアウトよりスケールアップの方が有効な手法であると言えます(NewSQLという選択肢もありますが、まだまだ一般的であるとは言えないと個人的には思っています)。「そんな遅いSQLを書いていてはダメだ」と指摘したくなるかもしれませんが、実際によく発生する事象ではあるという点は同業者として共感いただけるのではと思います。なお、スケールアウトを批判しているわけではなく、寧ろ筆者はスケールアウトが好きで、本稿では触れませんが、クエリサービスではスケールアウト“にも”対応しています。

話題が少し逸れてしまいましたが、クエリサービスではデータベースを止めることなく処理性能を上げることに取り組み、コンピューティングリソースを抽象化した“オンラインリソース変更機能”を開発し、以下の機能を実現しています。

① データベースのパフォーマンスに影響を与えるCPUとI/Oのデータスループットを、データベースを止めることなく必要なときに必要なだけ利用可能
② リソースをリクエストする発生源であるアプリケーションから簡単にリソースの変更ができるようSQLでのインタフェースを提供

①はクエリサービスの仕様として表現すると表-2のようになります。

CPUコアとI/Oのデータスループットのリソースはそれぞれ個別にリソースを基本スペックから最大スペックまで増加減させることができます。リソースの変更に要する時間は数秒以内で後述しますが課金対象も即時反映されます。

②ですが、オンラインリソース機能はAPIの他にSQLから実行できることが特徴の１つとなります。オンラインリソース機能は手動で制御する場合も分かりやすく、何といってもリソースをリクエストする発生源であるアプリケーションから簡単にリソースの変更ができるように開発されていますので、プログラム内に埋め込むことが容易にできます。例えば、大きな処理を実行する前後にCPU数を変更するなど、従来はできなかったリソースの使い方が可能となり、開発者なら簡単に実装することができます(図-2)。

	固定	可変	
リソース	基本	オプション	
		最大スペック	追加単位
データベース	1	ー	ー
CPUスコア	1	6	1
データスループット(MB/s)	100	2000	100
同時接続数	50	300	CPUスコア数連動
データ領域(GB)	50	8000	自動拡張

表-2 クエリサービスのスペック

```
        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;
/
```

CPU数を6へ変更
CPU数を2へ変更
並列処理による性能向上

図-2 CPUの並列処理

次にオンラインリソース機能の動きを解説します。有償のサービス提供を想定している以上、ユーザが直接CPUコア数やI/Oスループット性能を変更させてしまってはサービスとして破綻してしまいます。そのためユーザには通常のSQLのプロシージャとしてインタフェースを提供しているものの、システム変更のSQLコマンドがラップされた単純なプログラムを実行して変更しているわけではなく、外部プログラムを経由しバックエンドシステムのService Managerへリソース変更依頼を投げる方式を採用しています(図-3)。

従って、ユーザがSQLやAPIで実行に用いられるプロシージャはユーザリクエストを受け付け、設定可能な値であることをユーザデータベース上で判断させ、外部プログラムへ渡すだけの簡素なプログラムとなっています。

外部プログラム経由でリクエストを受け取ったサービスマネージャは、バックエンドシステムの構成管理データベースの対象ユーザデータベースの情報を変更し、課金システムで対象ユーザデータベースに関する課金条件を変更した後に、リソースコントローラを経由して、データベースのリソースマネージャを実行することで対象ユーザデータベースのCPUコア数やI/Oのスループットの設定を変更します。これらの処理が完了した後に、ユーザデータベース上のプロシージャを介して設定が完了したことを接続中のユーザセッションへ戻しています(表-3)。

詳細なプログラムの実装を説明するとページが全く足りないのですが、これらの処理は数秒以内に完了するため、ユーザはほぼリアルタイムでリソースの増加減を実行ができるよう

	日中	課金単位	使用料
基本	IIJクエリサービス基本	1	月額固定
オプション	CPUコア追加分	1コア	
	データスループット追加分	100MB/s	秒課金
	データ容量51GB以上の拡張分	1GB	

表-3 クエリサービスの課金

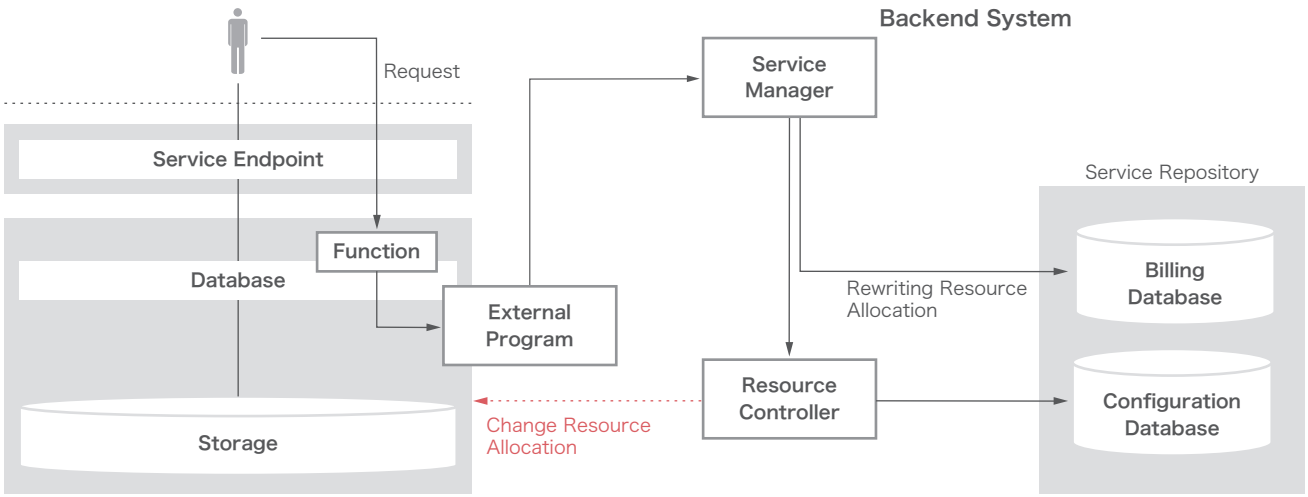

図-3 リソース変更処理の裏側

になります。これは極端な例かもしれませんが、オンラインリソース変更機能をプログラム内に組み込むことで、アプリケーションが動的に処理対象のデータ量を計測することによってデータ量に合わせたリソースを処理の都度、動的に変更するといった全く新しいデータベースサービスの使い方が可能になります(図-4)。

3.2.2 秒課金機能

オンラインリソース機能により好きなタイミングでリソースの拡張・縮小が可能になりましたが、リソースの利用料金が硬直的であっては使い勝手が悪いと言えます。クエリサービスでは柔軟な課金体系を目指し、基本スペックを超えるCPUコアやデータスループットなどのリソースについては秒課金とする機能も併せて開発しています。

図-5はクエリサービスで実装されている使用料金のイメージです。クエリサービスでは基本スペックを超えるCPUコアやデータスループット、データ領域がそれぞれ独立して秒課金されます。そのため必要になったときに使ったリソースに対し「使った分だけ課金」が可能になっています。

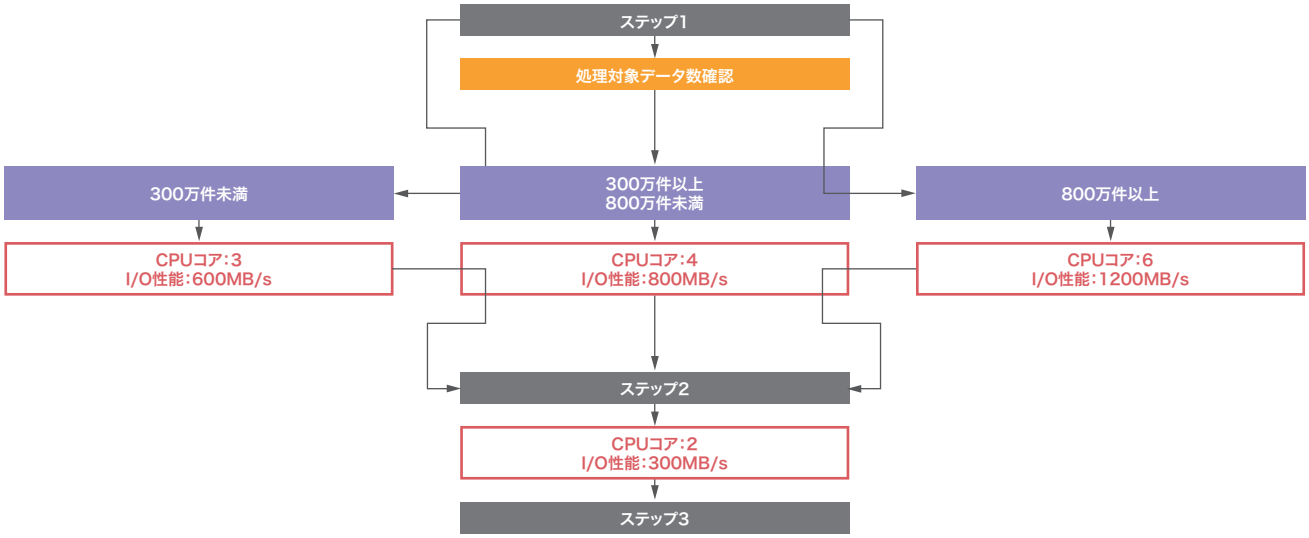

図-4 プログラマブルなリソース制御

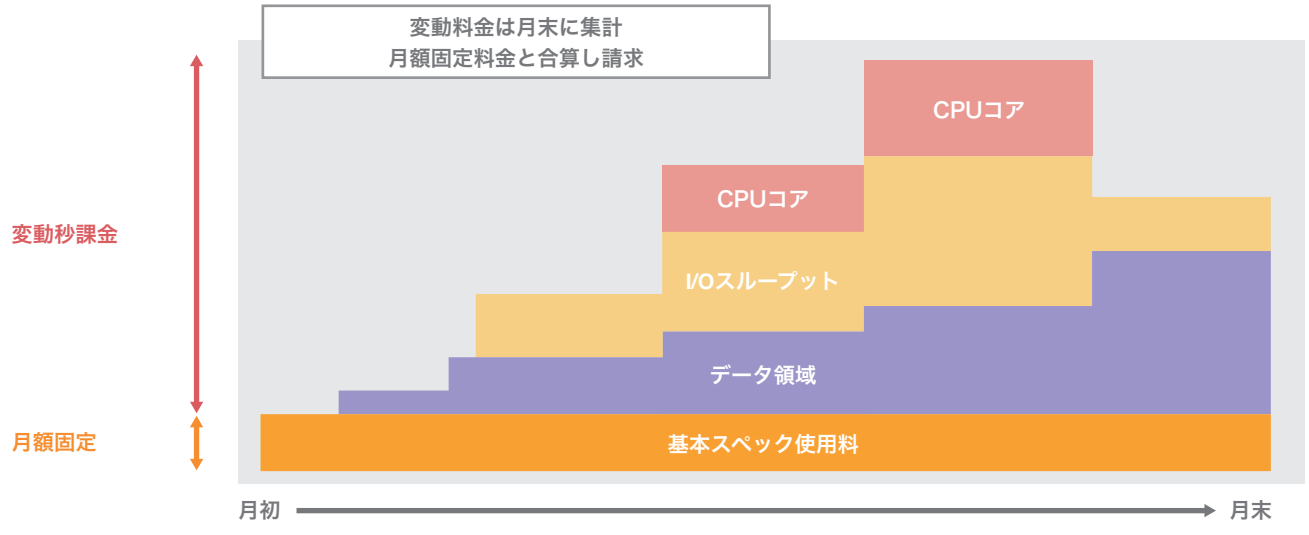

図-5 本サービスにおける課金の考え方

データベースの並列処理を組み合わせることでリソースの増強と処理速度のバランスが取れるようになれば、1秒当たりの単価が上がっても総利用料金へのインパクトは小さく抑えることができると言えるのです(図-6)。より柔軟な課金機能があってこそオンラインリソース機能が活用されると考えています。

3.2.3 オートスケーリング機能

オートスケーリング機能は、オンラインリソース変更機能と連動し、ユーザに代わってシステムが自動制御し、常に最適なパフォーマンスを提供する機能です。オンラインリソース変更機能は便利な機能ではありますが、その実行に際し人が都度判

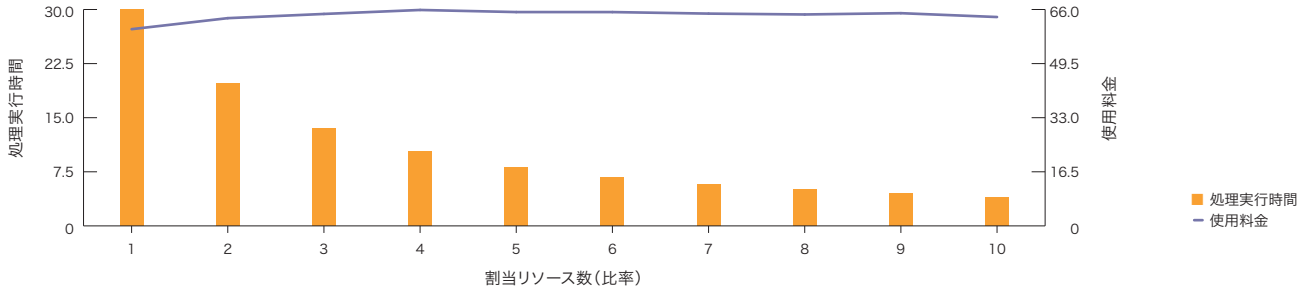

図-6 オンラインリソース変更機能と秒課金の考え方

リソース	最小値	最大値	増減値	拡張条件	縮小条件	評価間隔
CPU	1コア	6コア	1コア	割当済み全CPUコア 直近5分間の平均使用率を評価		1分間隔
				70%以上	65%以下	
I/Oスループット	100MB/s	2000MB/s	100MB/s	割当済みI/Oスループット 直近5分間の平均使用率を評価		1分間隔
				80%以上	75%以下	

表-4 自動スケーリング機能

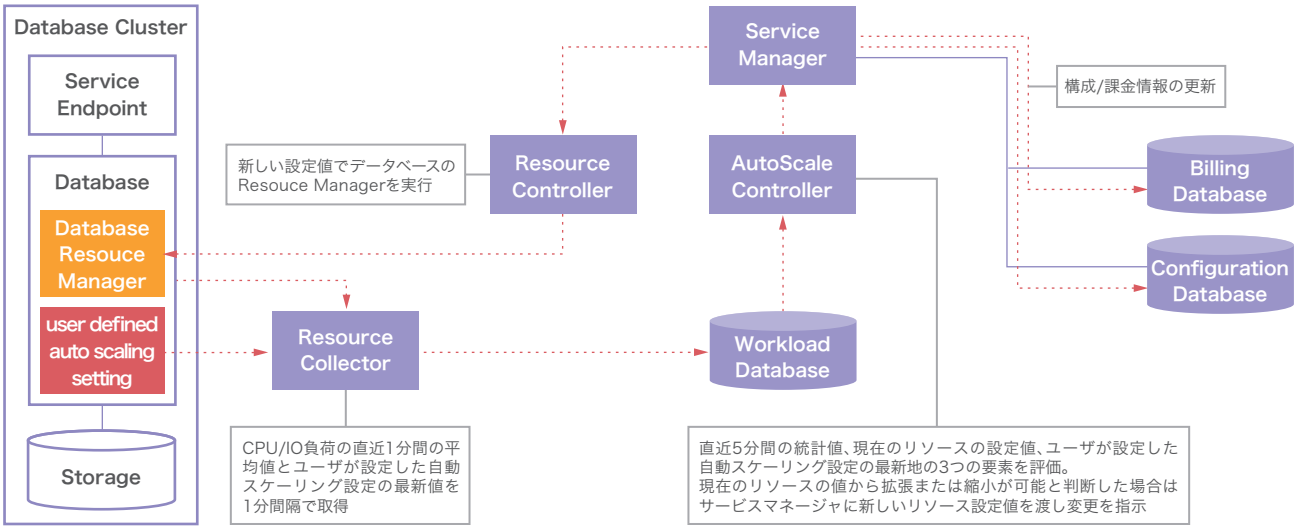

図-7 自動スケーリング機能の内部動作

断するような手動実行や外部の監視システムとの連携などを組み合わせた複雑な構成が必要になってしまうと、オンラインである利点や機能としての魅力が半減すると考えられます。クエリサービスの便利な機能をシステム運用の自動化でより活用してもらうために、オンラインリソース変更機能を自動制御するオートスケーリング機能も開発しました(表-4)。

オートスケーリング機能の内部動作は図-7のようになります。オートスケーリング機能はデータベースの稼働状況を収集し、負荷状況に応じたCPUコアとI/Oデータスループットといったデータベースの性能に関わるリソースを、オンラインリソース変更機能を介して自動で割り当てます。各リソースの負荷状況が拡張及び縮小の条件内で保つことを目標にして常時動作しています。プログラムとしてはこれら一連のジョブを制御するループ処理が実装されています。

オートスケーリング機能は非常に便利な機能でユースケースはいろいろありますが、こんなときにとても有効な仕組みだと思える例が“性能障害”が発生した際の対応ではないでしょうか。システムを運用していると、データベースの処理が突如スローダウンする現象に遭遇することが年に数回あります。その原因は様々で、よく言われる「何もしてないのに急に遅くなった」(多くの場合はデータベースの実行計画が突然変わった)というものもあれば、新しいプログラムがリリースされた直後や、データが大量に増えた、セールなどのイベントを実施した等々、いつもと何か違うことを契機に発生することもあります。このような“性能障害”といえる事象が発生すると、その対応は“多くの矛盾”との戦いとなっていきます。

まず、アプリケーションのスローダウンはインフラ視点でのリソース監視では検知しづらい場合があり、運用部門での検知が遅れることがあります。ユーザからのクレームが契機となる場合が多く、最初からハードルの高い対応となってしまいます。一方で障害の原因となっている実行中の処理は、プロセスレベルで止められる場合もあれば、夜間バッチのように翌日の営業開始時間まで完了しないと業務インパクトが大きいので止められないというものがあります。止められないがすぐに対応しなければならないという難しい局面です。これらのスローダウンの根本原因は、データの急増や不適切な実行計画に基づきクエリが実行されている、索引がない項目に対して検索が実行されてしまったなど、アプリケーション側に起因する原因がほとんどなのですが、仮に特定されたとしても、止めずに対応することは不可能または高難易度なオペレーションとなります。しかも何故かこのような障害が発生するのは人がいない休日か深夜の場合が多いのが厄介なところです。

上記のような矛盾に満ちた性能障害を、根本解決ではないものの、サービスが解決してくれたら、ユーザも運用者も開発者も幸せになれるだろうという思いからオートスケーリング機能を開発しています。

■ オートスケーリング機能の効果測定

オートスケーリング機能の効果を確認するために性能障害を起こすのは難しいのですが、代わりに使用中のスペックでは捌くことができないような大きな処理がデータベースに実行されるとクエリサービスのオートスケーリング機能がどのような動きを見せるのか、実際に処理を実行して計測してみました。

今回の試験では、1億件超のデータが格納された受注テーブルと商品マスタ、顧客マスタなど複数のテーブルを結合してクエリを実行しています。なお、今回はオートスケーリングの効果を分かりやすくする検証であるため、クエリの実行が完了するたびに共有メモリのバッファ領域をクリアしています。

クエリが実行されると受注テーブル内の全レコードがシーケンシャルにアクセスされます。シーケンシャルアクセスかつ、共有メモリは都度フラッシュしているクエリが実行されると、ストレージに配置されているブロックが大量に読み出されます。そのためクエリ実行に多くのI/Oリソースが要求されます。

初期値となっているクエリサービスの最小構成となる基本スペックのI/Oスループット性能は100MB/sと小さいため、億を

超えるレコード件数へのシーケンシャルスキャンが実行されると、開始直後からI/Oスループットリソースの使用率は急上昇します(図-8)。オートスケーリング機能は常時データベースの稼働統計情報を収集して診断を繰り返し、リソースの拡張または縮小を指示します。なお今回の検証で流したクエリはI/Oバウンドな処理であるため、オートスケーリング機能はI/Oスループット性能のみ900MB/sまで拡張を続けます。

面白いのは、I/Oスループットが900MB/sになるとCPUがI/Oの待機する時間を縮小するため、処理の特性がCPUバウンドへ変化を見せることです。そうなると今度はCPUコア数を増やして並列度を上げようと試みますが、CPUの使用率がそこまで上がるクエリではないので、オートスケーリング機能がCPUコアを再度手放す動きを見せました。いわゆる「オンラインスケールアップ」というべき動きになるのですが、Kubernetes上のコンテナではこの実装が今のところまだ安定版には至っていないようです。見ていると面白いのですが、CPUコアのキャッチ&リリースするような動きは性能にブレを生じさせる原因でもあるので改良すべき余地があります。もっとも、それほど大きな問題とは考えていません。このサンプルデータベースではオートスケーリング機能によりCPUコアは最大6コア、I/Oスループットは最大1000MB/sまで拡張可能なのですが、CPUコア数は1または2コア、I/Oスループットは900MB/sで十分と判断しています。実際、1000MB/s以

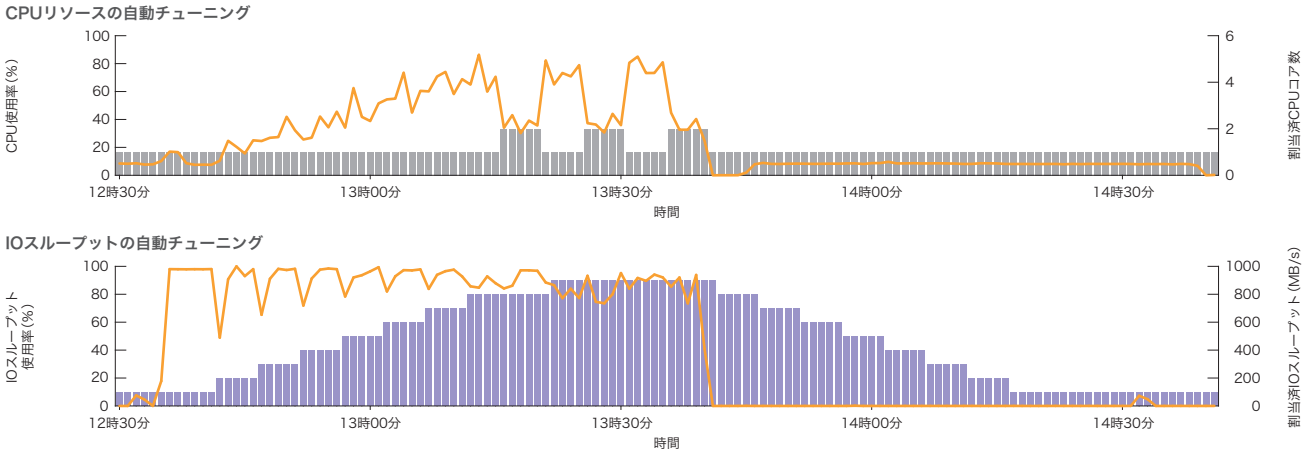

図-8 自動スケーリング機能により制御されるリソースの遷移例

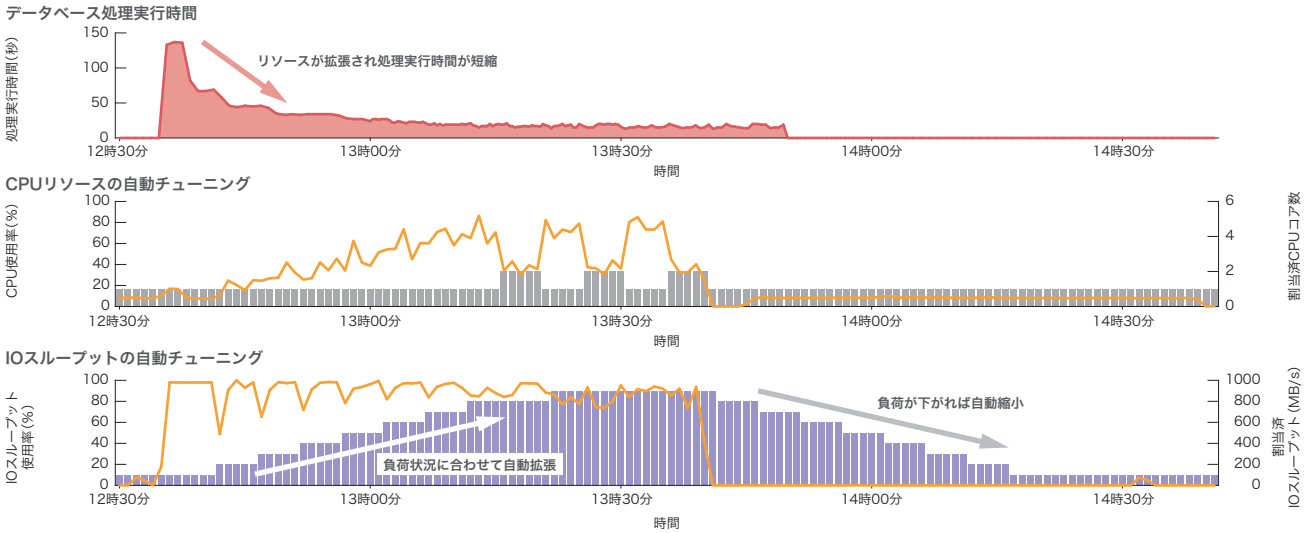

図-9 自動スケーリングによるデータベース処理時間の変化例

上やCPUコア数の設定を増やしても確かに性能は上がるのですが、目を見張るような大きな変化は見られませんでした。

実際の処理性能に対するインパクトは図-9のとおりです。初期のI/Oスループット性能の拡張の効果が非常に大きいのですが、処理時間は確実に短縮されていきます。

ところで、オートスケーリング機能が有効の状態でユーザが手動によりI/Oスループット性能の値を2000MB/sに設定した場合はどうなるのでしょうか。設定値自体は手動設定が優先されるためユーザのデータベースのI/Oスループット性能は2000MB/sに設定されます。オートスケーリングが無効の場合は2000MB/sが維持されますが、有効の場合はシステムが稼働状況を判断し、やはり900MB/sを目指してリソースが縮小していきます。

オートスケーリング機能で毎回100MB/sのような最小構成まで縮小してから拡張していては期待する性能を得るまでに時間がかかって困るという場合は、オートスケーリングの対象範囲をユーザが自由に設定できるようになっています。例えばCPUコアは2-4コア、I/Oスループット性能は500MB/sから1500MB/sの間で自動調整とすれば500MB/sを下回ることはありません。また、CPUコアはオートスケーリングさせずI/Oスループットだけという設定も可能です。設定の変更はオンラインででき、1分後(バックエンドシステムの都合)にシステムに反映されます。

■ オートスケーリング機能の課題

リソースの適正利用を図るため、リソースの拡張・縮小を判断する間隔や閾値、増加減値はバックエンドシステムの管理下でユーザが変更することはできないのですが、これもユーザが変更できる実装に変えていこうと考えています。

また、便利そうに見えるオートスケーリング機能ですが、まだまだ課題は多く存在しています。特にリソース割り当ての精度に関しては、開発している立場から見てもまだまだ難ありと言えます。

スケーリングの動作の根拠は前述のとおりデータベースの稼働統計値となるのですが、ここで得られる数値は過去に発生した事象に基づくものであり、オートスケーリング機能のコア機能であるオートスケールコントローラは「この傾向が今後もしばらく継続するであろう」という極めて単純な思考に基づき動作しているため、未来の負荷状況を先回り予測して動作ができるような格好の良いものではありません。また、この単純明快な思考は時系列における変化点の発生直後に期待値から外れることが多いのも問題です。具体的に言うと、負荷が下がっているにもかかわらずリソースを拡張することがあります。MLなどで更に賢いオートスケーリング機能を目指したいと考えています。

課題が残るオートスケーリング機能ではありますが、突発的に発生する性能障害のような事象に対しては有効な対応策の1つであろうかと思います。何といってもすべてが自動制御されているため、事象検知の遅延やクレームになって発覚するといった間にもデータベースのリソースが確実に拡張し性能を維持しようとします。事象が収まればリソースが縮小するので、場合によっては気がつかないかもしれません(使用料金には反映されますが)。自動化は構成管理やアプリケーションのデプロイだけではなく、システムの性能を自動的に維持するところまでやっていくと、運用はより楽になると思われます。

3.2.4 サービス更新機能

クエリサービスはサーバレスを標榜しているものの、実行環境は通常のデータベースと変わらないためサービス基盤は旧くなります。まずサービス基盤のハードウェアが旧くなり経年劣化に伴う機械的故障の発生率が高まります。OSやデータベースのソフトウェアも、サポート切れやパッチの提供が終わることでセキュリティホールやバグが修正されなくなります。このようにサービス基盤の更改は安定したサービスの継続提供には必要なイベントです。

新しいクエリサービスの基盤を用意することは簡単なのですが、利用中のユーザデータベースを新しい基盤へ移行することが必要です。データベースだけをバージョンアップするだけな

らインプレースアップグレードが最も簡単な方法ですが、バージョンアップに伴う時間が長くサービスの長時間の停止が避けられません。また、OSやハードウェアの更改が不十分、またはその更改を別途実施する必要が生じる可能性が出てくるため、効率の良い方法とは思えません。一方でバックアップから別のインスタンスを作成する方法もありますが、クライアントの接続先変更を伴う場合があり、作業範囲がクエリサービスの範囲内に収まらなくなってしまう可能性が出てきます。

クエリサービスでは、新しいサーバやストレージへの切替、OSのバージョンアップ、データベースソフトウェアのバージョンアップとデータベース自体のアップグレードをすべて1つの処理で完結させます。Kubernetesのローリングアップグレードに近いと思いますが、クエリサービスではレプリケーションのような技術を用いず、より高速かつユーザに透過的にサービス全体を更新する仕組みを実装しています。その特徴は以下のとおりです。

① データ移行が不要
② データの整合性を維持したデータベースの切替やバージョンアップなどすべての処理を完全自動化
③ ユーザが1クリックするだけですべてが完了
④ 最大15分で完了
⑤ 何度でも再実行可能
⑥ 移行後もクエリサービスの接続先は不変

クエリサービスには上記のようなサービスを更新し、継続的にデータベースを利用していただく機能が基盤に備わっています。サービス更新機能では、データ移行を不要としています。不要と言うと語弊がありますが、ユーザは全く意識することなく、利用しているデータベースが新しいサービス基盤へ複製されます。データの整合性もトランザクションログの転送で自動解決されます。

サービス更新機能は大きく3つのフェーズに分かれています。フェーズ1として、まずユーザのデータベースのスナップショットが新サービス基盤へオンラインで作成されます(図-10)。スナップショットの作成はユーザが手動実行する必要はなく、スナップショットが取得可能な状態をモニタリングして自動生成されます。データベースが複数個存在していても処理は個々のデータベースごとに完全に独立して処理されます。

フェーズ2以降はデータベースが新しいサービス基盤へ切り替わるため、処理の開始はユーザにトリガーが移動します。ユーザは任意のタイミングで処理を開始することができます。フェーズ2が実行されると、対象のユーザデータベース用のサービスエンドポイントがクローズされます(図-11)。これにより、切替を切り戻す際の静止点ポイントが確定されます。静止点ポイントが確定したら、新しいサービス基盤上のスナップショットとの最終的な同期処理を実行します。最終のデータベース間の同期処理が完了した後に現行のサービス基盤上の

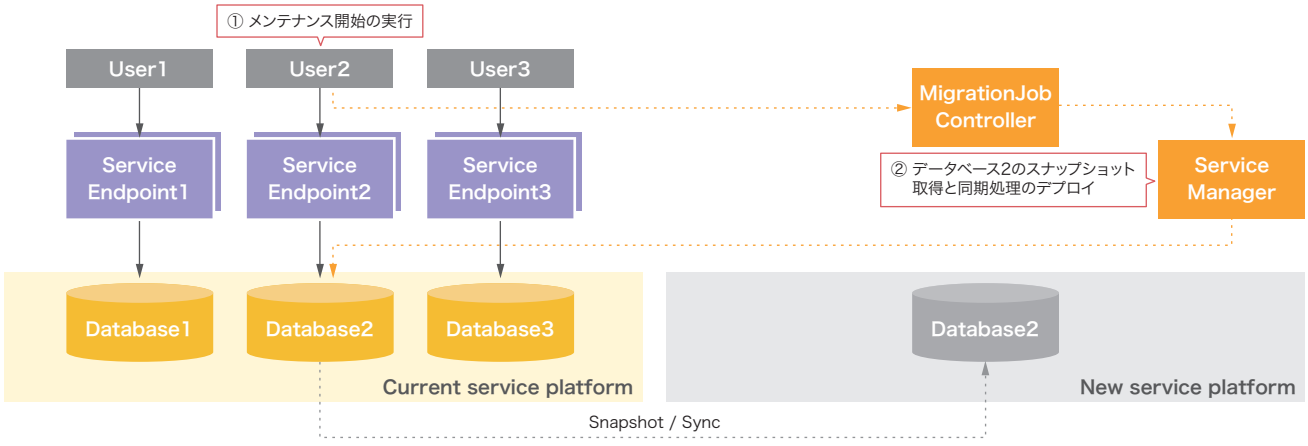

図-10 クエリサービス更新機能の内部動作1

データベースのステータスをinactiveへ変更し、オフライン状態へ遷移させます。オフラインとなってもデータベースの物理的な削除は行いません。これはサービス更新処理の実質的なバックアップという意味合いがありますが、ユーザが切戻しを実行した際にサービス更新処理をスナップショットから復元する時間を要しません。ステータスをactiveへ切り替えるだけで、極めて短時間で切戻しが完了できるからです。

新しいデータベースは現行のデータベースがinactiveになるまで処理を待機します。これは双方のステータスがactiveになることでスプリットブレインが発生する原因を排除するためです。そのため独立して処理は行われません。現行データベースが正常にinactiveとなった後に、新しいデータベースでサービスを継続するための処理が再開されます。データベースのバージョンをアップする場合はこのタイミングで実行します。また、新しいデータベースを高可用性構成へ構成変更します。フェーズ2の最終段階で、ユーザデータベース用サービスエンドポイントに登録されている経路情報を新サービス基盤側へ更新します。

フェーズ3では新しいサービス基盤上のユーザデータベースのステータスがactiveとなった時点でエンドポイントをオープンします(図-12)。フェーズ3での特徴は、サービスエンドポイントの経路情報が変更になっただけで、ユーザ側の接続ポイントが不変であることです。そのためユーザは処理完了の通知後に再接続を実行すれば、何も変更することなく新しいサービス基盤上のユーザデータベースへ接続されます。

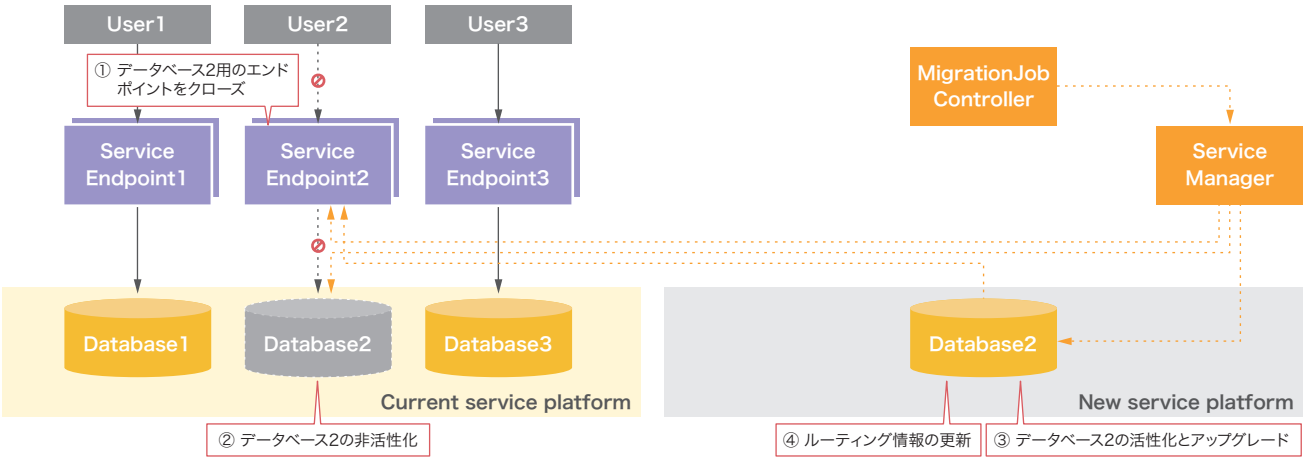

図-11 クエリサービス更新機能の内部動作2

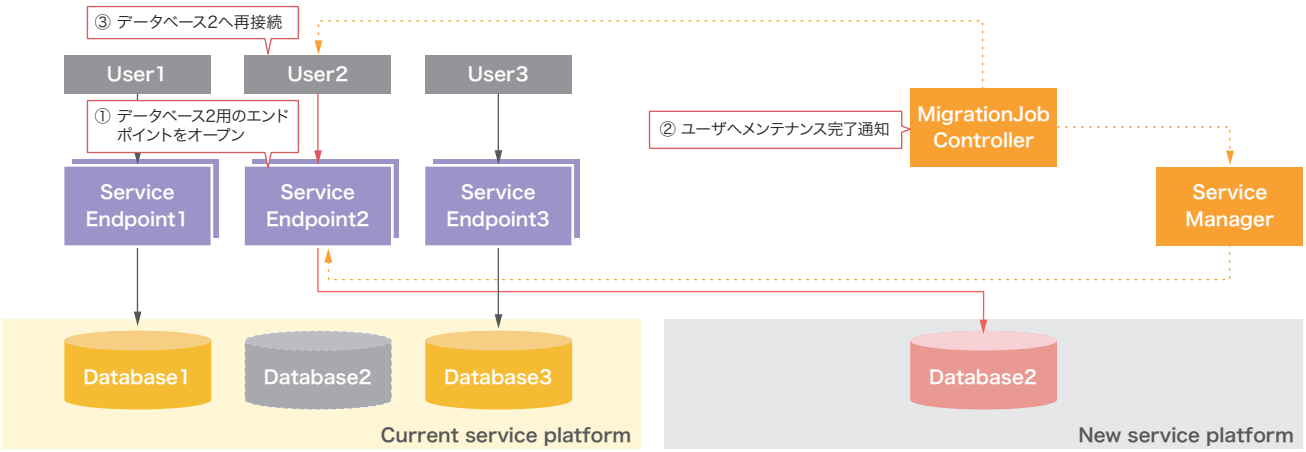

図-12 クエリサービス更新機能の内部動作3

切替が実行されると、まずクライアントとデータベース間に存在するプロキシであるサービスエンドポイント（以下、エンドポイント）を閉塞します。エンドポイントの閉塞によりクライアントとデータベース間の経路が消えるため、ユーザからのセッションは完全に切断されます。従って、もしトランザクション処理を実行中のセッションがある場合はデータベースにてロールバックされます。そのためトランザクションが実行されていない状態で実行されることが望ましいと言えます。エンドポイント閉塞完了後の現行データベースの整合性が確立された状態で、新データベースとの最終差分同期を実行します。

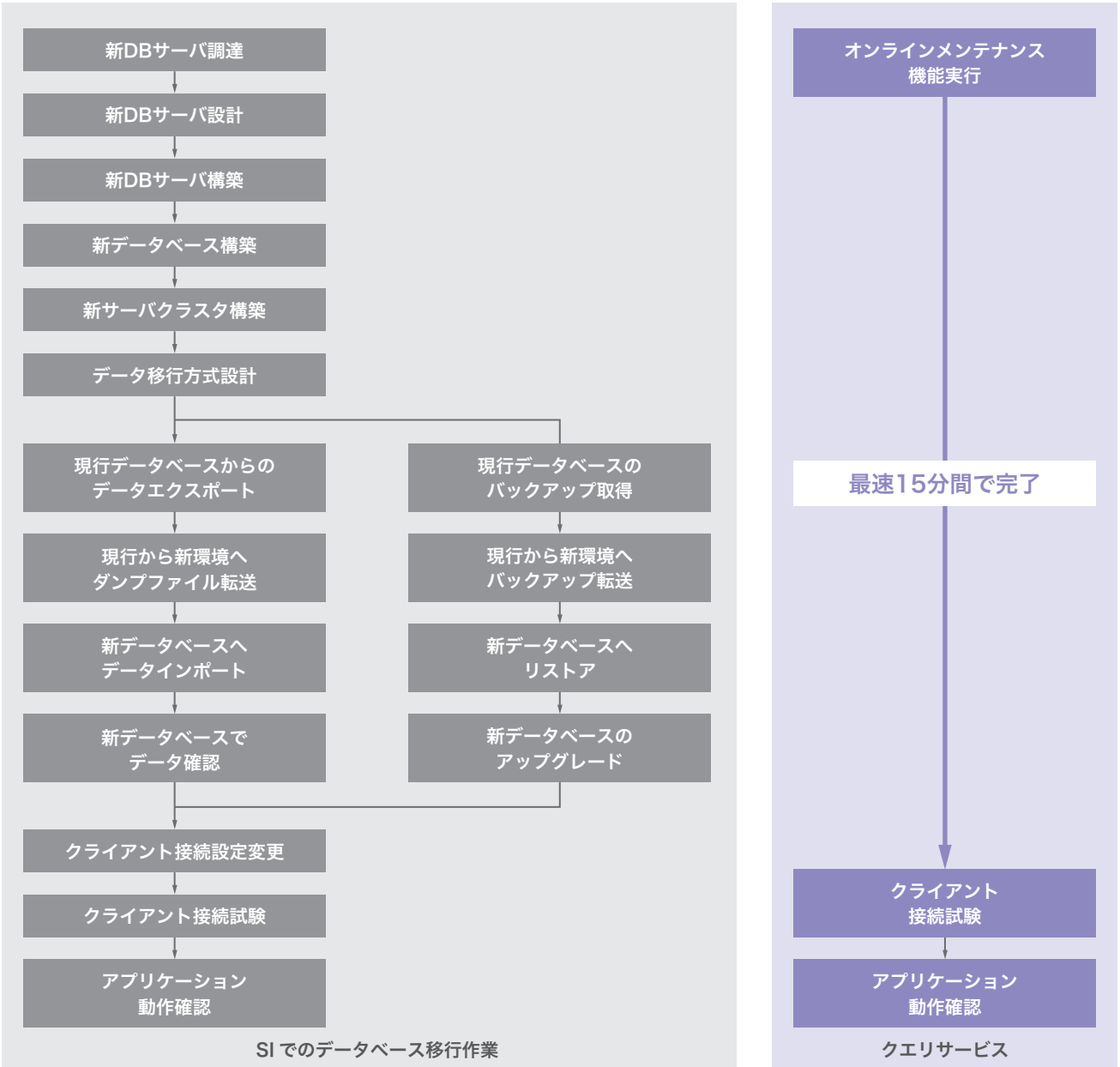

図-13 サービス更新機能による設備更改ステップの大幅な圧縮効果

サービス更新機能の処理は、従来はSIのプロジェクト作業として実施されてきました。これを手動で実行するとしたら図-13のようになります。なかなか骨の折れる手順だと思います。クエリサービスであれば裏側で自動処理されているので、サービスをご利用になる方はコントロールパネルから1クリックするだけで良いのです。

3.3 まとめ

クエリサービスはデータベースを作ったわけではなく、エンジニアが如何にデータベースと楽に付き合うかを求めたオーケストレーションシステムで、その目的はプロトタイプとはいえ達成できたように思います。また、本稿を読んだ方の中には、筆者がKubernetesに対抗心を持っているように感じ取られた方がいるかもしれませんが、実はKubernetes自体はとても好きで、機会があればKubernetesを使ってクエリサービスを開発してみたいものです。

テックチャレンジは、ユーザ要件がある従来の開発とは異なり、自分のアイデアを形にして良いという、エンジニアにとってこの上なく面白い時間で、社会人になってからすっかり忘れていた純粋なコンピュータの楽しさを味わった1年でした。

執筆者:
二ノ宮 務（にのみや つとむ）
IIJ システムクラウド本部 サービス企画室テクニカルマネージャー。
クラウドサービスの企画・開発やプロジェクトの技術支援に従事。元はDWH/BIシステムの開発者でSQLと並列処理を好む。