情報処理学会論文誌　データベース　Vol.9　No.1　1–19 (Mar. 2016)

クラウド環境におけるデータベース ライブマイグレーションミドルウェア

三島 健[1,a)]　藤原 靖宏[1,b)]

受付日 2015年9月27日, 採録日 2015年11月7日

概要：クラウドコンピューティングにおけるマルチテナント型データベースはテナント間で市中製品を共有できるためコスト削減が実現でき，Database-as-a-service として人気が上昇している．しかしながら，資源を共有することによって，予想外に負荷が増えることなどの理由でホットスポットを生み出すことがある．不幸なことに，ホットスポットはサービスレベルアグリーメントを満たせなかったり，顧客の満足度を下げたりすることにつながる．そこで，ホットスポットを解決するために，我々はデータベースライブマイグレーションを実行できる Madeus と呼ぶミドルウェアを提案する．効率良くマイグレーションを実現するために，我々は lazy スナップショット分離ルール（LSIR）と呼ぶルールを導入する．これは，スレーブデータベースとマスタデータベースをコンシステントにするためのクエリの集合を効率良くスレーブへ並行転送するルールである．我々のアプローチの優位性を示すために，PostgreSQL のミドルウェアを作成し TPC-W ベンチマークで実験を行った．その結果，Madeus は既存アプローチをベースとした 3 つのアプローチよりも短時間でマイグレーションが行えることが分かった．特に Madeus は高負荷の場合ほど有効であるため，ホットスポットを解決することができる．

キーワード：クラウド，ライブマイグレーション，ミドルウェア，並列転送，ホットスポット

Database Live Migration Middleware in Cloud Environment

Takeshi Mishima[1,a)]　Yasuhiro Fujiwara[1,b)]

Received: September 27, 2015, Accepted: November 7, 2015

Abstract: Database-as-a-service has been gaining popularity in cloud computing because multitenant databases can reduce costs by sharing off-the-shelf resources. However, due to heavy workloads, resource sharing often causes a hot spot. Unfortunately, a hot spot can lead to violation of service level agreements and destroy customer satisfaction. To efficiently address the hot spot problem, we propose a middleware approach called Madeus that conducts database live migration. To make efficient database live migration possible, we also introduce the lazy snapshot isolation rule (LSIR) that enables concurrently propagating syncsets, which are the datasets needed to synchronize slave with master databases. Unlike current approaches, Madeus is pure middleware that is transparent to the database management system and based on commodity hardware and software. To demonstrate the superiority of our approach over current approaches, we experimentally evaluated Madeus by using PostgreSQL with the TPC-W benchmark. The results indicate that Madeus achieves more efficient live migration than three other types of middleware approaches, especially under heavy workloads; therefore, it can effectively resolve hot spots.

Keywords: cloud, live migration, middleware, concurrent propagation, hot spot

1. はじめに

近年，マルチテナントモデルによってコストを削減できることからクラウドコンピューティングの活用がますます進んでいる [1]．このモデルでは，複数のテナントを 1 つ

[1] 株式会社 NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center, Musashino, Tokyo 180–8585, Japan
[a)] mishima.takeshi@lab.ntt.co.jp
[b)] fujiwara.yasuhiro@lab.ntt.co.jp

のノードへ集約し，ノードやオペレータを共有することでハードウェアやオペレーションコストを節約できる．さらに，市中製品を使うことでエンジニアリングやテスティング，保守コストを削減できる．リソースの使用率を向上させるために，クラウドプロバイダは 1 つのノードにできるだけ多くのテナントを集約しようとする．しかしながら，ノードに多すぎるテナントを集約させてしまったり，あるテナントが想定外の高負荷となってしまったりした場合，そのノードはホットスポット[*1]になってしまう．ホットスポットによりサービスレベルアグリーメント（SLA）を守れなかったり，顧客のサービスの満足度を下げたりすることになる．一般的に顧客はインターネットサービスは 2 秒以内で応答することを期待しているからである [2], [3]．不幸なことに，1 ノードに搭載する最適なテナント数を算出することは非常に難しい．さらに悪いことに，負荷は不安定で予測が困難である．このホットスポットによる問題を解決するために，クラウドプロバイダはしばしばライブマイグレーションを利用する．これはライブマイグレーションにより，ノード間の負荷のバランスを改善できるからである．

今日のクラウドデータベースサービスはデータベースインスタンスを走らせた VM 単位で貸し出す運用である [4], [5], [6]．しかしながら，VM 自体オーバヘッドが大きくスループットの低下を招く [7]．特にデータベースサーバを動作させたときの性能低下は大きい．その理由の 1 つは，個々のデータベースインスタンスがログファイルを持つため，1 つのログファイル自体はシーケンシャルアクセスであるが，全体としてランダムアクセスになってしまうからである．Curino らが提案した共有プロセスモデルはこの問題を解決している [8]．データベース 1 つをテナントとし，複数のテナントで同じデータベースサーバプロセスを共有する．これによりトランザクションログは 1 つとなり，ランダムアクセスを避けられる [8]．したがって，我々はこのモデルを前提とする．

共有プロセスモデルを前提としたデータベースマイグレーションは 2 つの既存研究が存在する [9], [10]．これらは，ソースノード上の 1 つのテナントをデスティネーションノードへマイグレートするものである．ソース上のマイグレートするテナントをマスタと呼び，デスティネーション上に新たに作るテナントをスレーブと呼ぶ．Das らは共有ディスクを前提として，スレーブテナントにウォームキャッシュを持たせるアプローチを提案している [9]．Elmore らは分散アーキテクチャを前提として，インデックスの情報を利用したアプローチを提案している [10]．不幸なことに，どちらの提案もデータベースサーバの改造をともなう．データベースサーバのソースコードは巨大で複雑であることを考えると，市中製品を無改造で利用しコストを抑えるクラウドサービスなどには適していない．

低コストでライブマイグレーションを行うために，我々はミドルウェアによるアプローチに着目する．すべてのマイグレーションの機能をミドルウェアで実装すれば，市中製品を無改造で利用できるからである．Barker らは一部の機能をミドルウェアで実装したアプローチを提案している [11]．彼らのアプローチの利点は VM を使わないうえにデータベースサーバの改造も必要ない．しかし，我々のアプローチと違って，スレーブをマスタに同期させるためにトランザクションログを用いるがログフォーマットはデータベースサーバの種類やバージョンによって異なるため，彼らのアプローチはある特定のデータベースサーバペアに特化した実装になってしまう [12]．また，トランザクションログからある特定のテナント情報だけを抜き取ることは困難かつ非効率なため，共有プロセスモデルの適用は困難である．さらに，彼らのアプローチは特定のバックアップツール（Percona XtraBackup [13]）に依存している．

ライブマイグレーションはマスタとスレーブ間でレプリケーションをともなう．レプリケーションアプローチは eager と lazy に分類できる [14]．eager レプリケーションは 1 つのトランザクション内ですべてのレプリカ（データベース）を更新する厳密な同期をとる手法である．lazy レプリケーションでは，マスタのトランザクションがコミットした後でスレーブに非同期に更新を転送する手法である．ライブマイグレーションでは，最初はスレーブは空であり，スナップショットからデータベースを構築する必要がある．その間にもユーザのトランザクションを処理する必要があるため，ライブマイグレーションには lazy レプリケーションが適している．しかしながら，既存の lazy レプリケーションアプローチ [15], [16], [17] をライブマイグレーションに利用することは非効率である．既存手法 [15], [16] ではスレーブトランザクション[*2]をシリアルに転送するアプローチであるため同期完了までの時間が長くなってしまう．既存手法 [17] は更新クエリを平行に転送できるが，コミットクエリはシリアルに実行しなければならないという制約がある．この制約は平行実行を鈍らせるだけでなく，グループコミットの恩恵を利用できなくなるという欠点がある．

本論文では効率的なデータベースマイグレーションを実現する Madeus と呼ぶミドルウェアアプローチを提案する．Madeus は共有プロセスモデルを前提とする [8]．このモデルを採用すれば，VM は必要ないので VM によるスループットの低下を防ぐことができる．また，トランザクション分離モデルは，スナップショット分離 [18] を前提とする．これは強い分離モデルであるだけでなく，高い性能を期待できる [18]．スナップショット分離では，参照はト

[*1] ホットスポットとはあるノードが高負荷によってオーバロードになったが，他のノードは低負荷のままであることをいう．

[*2] スレーブをマスタとコンシステントにするためのトランザクション

ランザクション T_i が開始される前にコミットされたデータ，すなわちスナップショットから読み込む．更新はそのスナップショットに対して行われる．トランザクション T_i がスタートした後にコミットされたトランザクションの変更をトランザクション T_i は検知しない．スナップショット分離では，参照オペレーションは更新オペレーションをブロックしないし，その逆も真である．

Madeus はデータベースサーバに透過であり，コモディティなハードウェアやソフトウェアを変更する必要もない．しかもなんらかの特殊なツールも必要ない．Madeus のキーアイデアは，更新クエリと同様にコミットクエリも平行に転送できることである．Madeus は特に高負荷でライブマイグレーションを行うためにデザインしている．我々の貢献は以下のとおりである：

- 効率良いマイグレーションのために lazy スナップショット分離ルール（LSIR）を提案する．また，データベースライブマイグレーションにおいてスレーブをマスタに同期させる最小限のクエリ集合を示す．
- 効率良くデータベースマイグレーションを実現する Madeus を提案する．Madeus は LSIR を使うことで更新トランザクションのみならずコミットクエリも平行に転送する．したがって，他の既存アプローチとは異なり，グループコミットの恩恵を受けることができる．
- TPC-W ベンチマークを使って Madeus の評価を行った．Madeus は他の 3 つのミドルウェアアプローチよりも短時間でマイグレーションできることが分かった．特に，高負荷の場合の効果が大きい．さらに，我々は，ホットスポットを解決するためには低負荷のテナントをマイグレーションすべきか，高負荷のテナントをマイグレーションすべきか，という問いに対する答えを導いた．

この論文は SIGMOD2015 で速報的に発表した内容 [19] をもとにして完成版としたものである．この論文の構成は以下のとおりである．2 章ではこの論文のバックグラウンドを説明する．3 章ではデータベースモデルを理論的に議論し LSIR を導く．4 章では Madeus について述べる．5 章では実験結果を述べ，6 章では関連研究について述べる．7 章で本論文の結論を述べる．

2. 背景

2.1 節は我々のアプローチで使うデータベースモデルを説明し，2.2 節では 6 つの dependency を説明する．2.3 節ではスナップショット分離の特徴を議論する．

2.1 データベースモデル

データベースはデータアイテムの集合から構成される．各データアイテムは値を持つ．あるときの値の集合はデータベースの状態をなす．データアイテムは，たとえば x, y, z のように小文字で表す．データベースサーバはデータベースにアクセスするコマンドをサポートするハードウェアとソフトウェアの集まりである．これらのコマンドはオペレーションと呼ぶ．トランザクションは参照，更新，終了（コミットまたはアボート）オペレーションのシーケンスである．下付き文字 i はトランザクションの i 番目のバージョンを表し，他のトランザクションと区別する．したがって，x_i はトランザクション T_i によって更新されたデータアイテム x を表し，$w_{i,p}(x_i)$ はトランザクション T_i によるデータアイテム x_i に対する p 番目の更新オペレーションを表す．$r_{i,q}(x_j)$ はデータアイテム x_j に対する q 番目の参照オペレーションを表す．c_i と a_i は T_i のコミットとアボートオペレーションを表す．

もし，トランザクション T_j が始まる前にトランザクション T_i がコミットした場合，トランザクション T_i と T_j $(i < j)$ はシリアルであるという．もし，トランザクション T_i が始まってコミット前にトランザクション T_j が始まった場合，トランザクション T_i と T_j は平行であるという．複数のトランザクションが平行に実行されると，オペレーションはインタリーブされる．データベースサーバのスケジューラが決めたオペレーションの実行順序をスケジュールと呼ぶ．また，実際に実行された順序をヒストリと呼ぶ．スケジュールは未来のプランであるのに対してヒストリは実際に行われた順序である．したがって，スケジュールとヒストリが同じとは限らない．

2.2 Dependency

本論文では，マスタがトランザクション T_i と T_j を実行して得られた結果と，マスタとは異なる順序でスレーブが実行した場合に得られた結果が異なる場合，トランザクション T_i と T_j は dependency があるという．スレーブとマスタをコンシステントにするためには，マスタのすべての dependency をスレーブで再現しなければならない．データアイテム x_i を参照または更新し，その後 x_j が上書きされ，x_i と x_j の間に他の上書きがない場合，x_j を直後の更新者と呼ぶ．2 つのオペレーション o_i と o_j のうち少なくとも 1 つが更新であり，実行順序が異なると異なる結果になる場合は o_i と o_j は dependency がある．3 つの dependency を以下のように定義する．

定義 1（dependency） 2 つのオペレーション間で次の 3 つの dependency が存在する．

- もし，オペレーション o_i がデータアイテム x を x_i に更新して，オペレーション o_j がこの x_i を参照したとき，オペレーション o_i と o_j には wr-dependency がある．
- トランザクション T_k によって更新されたデータアイテム x をオペレーション o_i が参照した後，オペレーション o_j が直後の更新者であってデータアイテム x を x_j に更新したとき，オペレーション o_i と o_j には

rw-dependency がある．

- オペレーション o_i がデータアイテム x を x_i に更新した後，直後の更新者であるオペレーション o_j がデータアイテム x を x_j に更新したとき，オペレーション o_i と o_j には ww-dependency がある．

2 つの参照オペレーションの順序は結果になんら影響を与えないため，以後の議論から rr-dependency は省略する．先の 3 つの dependency の観点とは別に，intra と inter の 2 つの dependency が存在する．もし o_i と o_j が同じトランザクションに含まれる場合，intra-dependency がある．オペレーション o_i と o_j が異なるトランザクションに含まれる場合，inter-dependency がある．これら 2 つのカテゴリは直行しているため，intra-wr-dependency，inter-wr-dependency，intra-rw-dependency，inter-rw-dependency，intra-ww-dependency，inter-ww-dependency の 6 つの dependency がある．

表 1　主要な記号の定義
Table 1　Definition of main symbols.

Symbol	Definition
T_i	i 番目のトランザクション
T^m	マスタデータベースのトランザクション
T^s	スレーブデータベースのトランザクション
x_i	トランザクション T_i によって更新されたデータアイテム x
$r_{i,p}(x_j)$	T_i におけるデータアイテム x_j への p 番目の参照オペレーション
$w_{i,p}(x_i)$	T_i におけるデータアイテム x_i への p 番目の更新オペレーション
c_i	トランザクション T_i のコミットオペレーション
a_i	トランザクション T_i のアボートオペレーション
o_i	トランザクション T_i の参照または更新オペレーション
R^m	マスタデータベース
R^s	スレーブデータベース
H^m	データベース R^m のヒストリ
H^s	データベース R^s のヒストリ
S^s	データベース R^s のスケジュール
$\mathcal{T}^m$	トランザクション T^m の集合
$\mathcal{T}^s$	トランザクション T^s の集合

2.3　inter-ww-dependency の解決方法

inter-ww-dependency に関して，スナップショット分離では次のように first-updater-wins ルール [20] [*3] に従う．もし，トランザクション T_i がデータアイテム x を更新した場合，データアイテム x に更新ロックを設定する．続いて，もし，トランザクション T_j がデータアイテム x を更新しようと試みた場合，トランザクション T_j はそのロックのためにブロックされる．もし，トランザクション T_i がコミットした場合，トランザクション T_j はアボートされる．トランザクション T_j がデータアイテム x を更新するためには，トランザクション T_i がアボートによってデータアイテム x のロックを解放しなければならない．他方，もし，トランザクション T_j がデータアイテム x を更新しようと試みる前にトランザクション T_i がデータアイテム x を更新しコミットした場合，ロックによるブロッキングは発生しない．トランザクション T_j がデータアイテム x を更新しようとしたとき，即座にトランザクション T_j はアボートされる．

3.　効率的な lazy レプリケーション

我々のゴールは高負荷のもとで短時間でライブマイグレーションすることである．このために，スレーブをマスタとコンシステントにするための最小限のクエリ集合を示すとともに，それらクエリを効率良くスケジュールする方法を示す．

もし，マスタデータベースとスレーブデータベースで実行する 2 つのオペレーション間でまったく dependency がない場合，実行順序にかかわらず 2 つのオペレーションは同じ結果を出力する．いい替えると，2 つのオペレーション間で dependency がある場合，異なる出力結果になる可能性がある．この章では，スレーブをマスタとコンシステントにするデータベースライブマイグレーションにおける最小のクエリ集合を示す．

R^m と R^s を lazy レプリケーションシステムにおけるマスタデータベースとスレーブデータベースとする．もし，データベース R^m におけるすべての dependency をデータベース R^s で再現できれば，データベース R^s はデータベース R^m とコンシステントになる．データベース R^m と R^s は最初はコンシステントであると仮定する．3.1 節では我々が仮定するスナップショット作成ルールを説明する．3.2 節ではスレーブをマスタに同期させるために必要な dependency と不必要な dependency を説明する．3.3 節ではライブマイグレーションをミドルウェアで実現するために必要な dependency を説明する．3.4 節ではライブマイグレーションでスレーブをマスタとコンシステントにするために必要な LSIR を説明する．**表 1** は本章で扱う主要な記号と定義を表す．

3.1　スナップショットの作成

本論文ではブラインド更新オペレーション [21], [22] はないものと仮定する．つまり，あらかじめデータアイテムを読むことなしに更新しない．先行研究 [17] では，スナップショットはスタートオペレーションによって明に作成されることになっている．現実的なデータベースサーバ（Oracle や SQL Server，PostgreSQL など）では，最初のオペレーションを実行する直前に暗に作成される．我々はこの現実的な動作を仮定する．ブラインド更新オペレーションがないことを考えると，トランザクションの最初のオペ

[*3] 2 つのトランザクションが同じデータアイテムを更新しようとしたとき，最初の更新だけが成功しその他はアボートする．first-updater-wins ルールは標準のデータベースサーバ，たとえば Oracle や SQL Server，PostgreSQL に実装されている．

レーションは参照オペレーションである．したがって，最初の参照オペレーション $r_{i,1}(x_p)$ が実行される直前にトランザクション T_i のスナップショットが作成される．

3.2　再現実行が必要な dependency

lazy レプリケーションでは，マスタデータベースにおけるすべての dependency をスレーブデータベースで再現実行できれば，スレーブデータベースはマスタデータベースとコンシステントになる．しかし，このナイーブなアプローチは明らかに非効率的である．マスタデータベースとスレーブデータベースのコンシステンシを崩さずに効率の良いライブマイグレーションをするためにはスレーブが再現実行する必要がない dependency を見分ける必要がある．

2.2 節で述べたように，intra-wr-dependency と inter-wr-dependency，intra-rw-dependency，inter-rw-dependency，intra-ww-dependency，inter-ww-dependency の 6 つの dependency があった．このうち，inter-ww-dependency と intra-wr-dependency はスナップショット分離のもとではスレーブデータベースが再現実行する必要がない．

すべての証明は付録に記載する．

補題 1（不必要な inter-ww-dependency）lazy レプリケーションではスナップショット分離のもとでスレーブとマスタをコンシステントにするためには inter-ww-dependency を再現実行する必要はない．

補題 2（不必要な intra-wr-dependency）lazy レプリケーションではスナップショット分離のもとでスレーブとマスタをコンシステントにするためには intra-wr-dependency を再現実行する必要はない．

補題 1 と 2 より，次に示す残り 4 つの dependency が再現実行を必要とする．

補題 3（再現実行が必要な dependency）lazy レプリケーションではスナップショット分離のもとでスレーブとマスタをコンシステントにするためには inter-wr-dependency と inter-rw-dependency，intra-rw-dependency，intra-ww-dependency を再現実行する必要がある．

3.3　dependency とクエリの関係

3.2 節ではデータベースライブマイグレーションに必要な 4 つの dependency を示した．本節では，ミドルウェアレベルでトランザクション間の dependency を再現実行するのに使う特徴を述べる．

我々のアプローチでは，最初の参照オペレーションがスナップショットを作成し，他の参照オペレーションはそのスナップショットからデータアイテムを参照する．さらに，更新オペレーションはそのスナップショットを更新する．したがって，最初の参照オペレーションが wr-dependency と rw-dependency を引き起こす．

補題 4（Inter-wr-dependency）もし T_i が更新トランザクションであり $c_i < r_{j,1}(x_i)$ であるならば，トランザクション T_i と T_j は inter-wr-dependency を持つ．

補題 5（Inter/intra-rw-dependency）もし T_j が更新トランザクションであり $r_{j,1}(x_k) < c_i$ であるならば，トランザクション T_i と T_j は inter-rw-dependency または intra-rw-dependency を持つ．

補題 6（Intra-ww-dependency）もし T_i が更新トランザクションであり $w_{i,p}(x_i) < w_{i,p+1}(x_i)$ であるならば，トランザクション T_i は intra-ww-dependency を持つ．

3.4　lazy スナップショット分離ルール（LSIR）

本節では，コンシステンシを崩すことなく効率良くデータベースライブマイグレーションを行うための我々のコアのアイデアである LSIR を示す．3.2 節で，スナップショット分離下でデータベースライブマイグレーションを行うためには再現実行の必要がないトランザクションがあることを述べた．したがって，スレーブとマスタをコンシステントにするミドルウェアレベルで効率良いライブマイグレーションを実行するために LSIR は 3.3 節の特徴をベースとする．LSIR は再現実行が必要なオペレーションの最小集合を定義する．LSIR の詳細を論じる前に，LSIR で使うマッピング関数を導入する．マッピング関数はスレーブに転送すべきトランザクションを検出する．このマッピング関数は次に示すようにマスタデータベースのトランザクションからスレーブデータベースへ転送すべきスレーブトランザクションを出力する．

定義 2（Mapping Function）$\mathcal{T}^m$ をマスタトランザクションの集合とし，各マスタトランザクション T_i^m はオペレーション $o_i^m \in \{r_{i,1}, r_{i,2}, \ldots, r_{i,p}, w_{i,1}, w_{i,2}, \ldots, w_{i,q}, c_i, a_i\}$ を持つ．$\mathcal{T}^s$ をスレーブトランザクションの集合とする．マッピング関数 $\mathcal{F}$ は次のようにマスタトランザクションの集合 $\mathcal{T}^m$ からスレーブトランザクションの集合 $\mathcal{T}^s$ を出力する．

(1) もしマスタトランザクション T_i^m が参照のみのトランザクションまたはアボートトランザクションであるならば，マッピング関数 $\mathcal{F}$ は空集合を出力する．つまり，$\mathcal{F}(T_i^m) = \emptyset$.
(2) もしマスタトランザクション T_i^m がコミットされた更新トランザクションであるならば，マッピング関数 $\mathcal{F}$ はマスタトランザクション T_i^m の最初の参照オペレーション $r_{i,1}^m$ をスレーブトランザクション T_i^s の最初の参照オペレーション $r_{i,1}^s$ にマップする．また，マスタトランザクション T_i^m の他の参照オペレーション $r_{i,2}^m, \ldots, r_{i,p}^m$ は捨てる．
(3) もしマスタトランザクション T_i^m がコミットされた更新トランザクションであるならば，マッピング関数 $\mathcal{F}$ はマスタトランザクション T_i^m の更新オペレーション $w_{i,1}^m, w_{i,2}^m, \ldots, w_{i,q}^m, c_i^m$ をスレーブトランザクション

T_i^s の更新オペレーション $w_{i,1}^s, w_{i,2}^s, \ldots, w_{i,q}^s, c_i^s$ にマップする．

直観的には，定義 2 より，マッピング関数は次のようにしてスレーブトランザクションを得る．参照のみのトランザクションとアボートトランザクションは捨てる．スナップショット分離ではデータベースのスナップショットを最初の参照オペレーションが作成するうえに intra/inter-rw-dependency を引き起こすため，コミットされた更新トランザクションの最初の参照オペレーションを保存する．他の参照オペレーションはスナップショットからデータアイテムの状態を参照するだけなので，なんの dependency も生じない．したがって，最初の参照オペレーション以外は捨てる．すべての更新オペレーションとコミットオペレーションはデータベースを変更するため保存が必要である．結果としてマッピング関数は，マスタトランザクションのオペレーション $o_i^m \in \{r_{i,1}, r_{i,2}, \ldots, r_{i,p}, w_{i,1}, w_{i,2}, \ldots, w_{i,q}, c_i, a_i\}$ はスレーブトランザクションのオペレーション $o_i \in \{r_{i,1}, w_{i,1}, w_{i,2}, \ldots, w_{i,q}, c_i\}$ を出力する．このマッピング関数をベースとして，スレーブデータベースをマスタデータベースとコンシステントにする LSIR を導入する．LSIR は以下のように定義する．

定義 3（LSIR） S^s をスレーブデータベース R^s のスレーブトランザクションの集合 $\mathcal{T}^s$ に対するスケジュールとする．スケジュール S^s のための LSIR は以下のように定義する．

(1) もし T_i^m をマスタデータベースにおけるコミットした更新トランザクションだとすると，スレーブデータベースは次のようなルールで制御する．
  (1-a) $c_i^m < r_{j,1}^m \in H^m \Rightarrow c_i^s < r_{j,1}^s \in S^s$
  (1-b) $r_{j,1}^m < c_i^m \in H^m \Rightarrow r_{j,1}^s < c_i^s \in S^s$
(2) 更新オペレーションはマスタのヒストリ H^m と同じ順序で実行するようにスレーブトランザクションを制御する．
  $w_{i,p}^m(x_i) < w_{i,p+1}^m(x_i) \in H^m \Rightarrow$
  $w_{i,p}^s(x_i) < w_{i,p+1}^s(x_i) \in S^s$

LSIR は次のような特徴を持つ．

定理 1（LSIR） もしスレーブデータベース R^s のスケジュール S^s がスナップショット分離のもとで LSIR により決定されるならば，スレーブデータベース R^s はマスタデータベース R^m にコンシステントである．

4. Madeus

我々が提案する lazy レプリケーションミドルウェアである Madeus を説明する．Madeus は DBaaS でホットスポットを効率良く解決するためにデザインした．データベースライブマイグレーションができるように既存の lazy レプリケーションプロトコル [15], [16] を利用することも可能ではあるが，クエリをシリアルに転送しなければならないプロトコルであり効率が悪い．本章では，クエリを並行に転送できる新たなプロトコルを提案する．4.1 節ではオペレーション（クエリ）を並行に転送する Madeus について述べる．4.2 節では Madeus のアーキテクチャについて述べる．4.3 節では，Madeus のライブマイグレーションについて概説する．4.4 節では，Madeus のアルゴリズムの詳細について説明する．付録 A.3 では Madeus の動作例について説明する．4.5 節では Madeus が，マスタテナントとスレーブテナント間のコンシステンシを犠牲にすることなく，データベースライブマイグレーションできることを理論的に説明する．

4.1 効率的な並行転送

定義 3 に示すように，LSIR は 2 つのオペレーションペアが実行されるべき順序を定めているルールである．2 つのオペレーションペアとは (1)コミットオペレーション c_i^s と最初の参照オペレーション $r_{j,1}^s$ のペアと (2) 1 つのトランザクション内の更新オペレーション $w_{i,p}^s(x_i)$ と $w_{i,p+1}^s(x_i)$ のペアである．データベースサーバはオペレーションの実行順序を制御する機能がないため，実行順序を一意に決めるためにはオペレーションを 1 つずつシリアルに送信するしかない．逆にいえば，上記の 2 つのペア以外のオペレーションペアに対して LSIR はなんら実行順序を定めていない．これは，上記 2 つのオペレーションペア以外はどんなオペレーションペアでも同時並行に転送できることを意味する．特に，次のオペレーションペアを同時並行に転送できることは有益である．

- 最初の参照オペレーション：LSIR は最初の参照オペレーション間になんの関係も規定していないことから，複数の最初の参照オペレーションを同時並行に転送することができる．たとえば，もし $r_{i,1}^s < c_k^s$ と $r_{j,1}^s < c_k^s$ であるならば，$r_{i,1}^s$ と $r_{j,1}^s$ は同時並行に転送することができる．
- 更新オペレーション：LSIR は異なるトランザクションの更新オペレーション間になんの関係も規定していないことから，異なるトランザクションに属する更新オペレーションを同時並行に転送することができる．たとえば，更新オペレーション $w_{i,p}^s(x_i)$ と $w_{j,q}^s(x_j)$ を同時並行に転送できる．
- コミットオペレーション：既存方式 [17] とは違って，複数のコミットオペレーションを同時並行に転送することができる．これは LSIR はコミットオペレーション間になんの関係も定めていないことに由来する．たとえば，もし $c_i^s < r_{k,1}^s$ と $c_j^s < r_{k,1}^s$ であるならば，c_i^s と c_j^s は同時並行に転送することができる．

上記 3 つのオペレーションペアの中でも，コミットオペレーション（3 番目のペア）は最も重要である．なぜならば，もしコミットオペレーション c_i^s と c_j^s を同時並行に転送

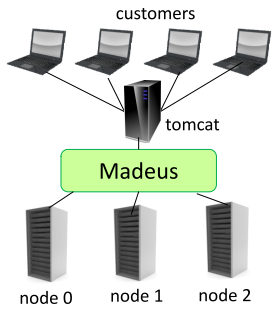

図 1　Madeus のモデル
Fig. 1　Model of Madeus.

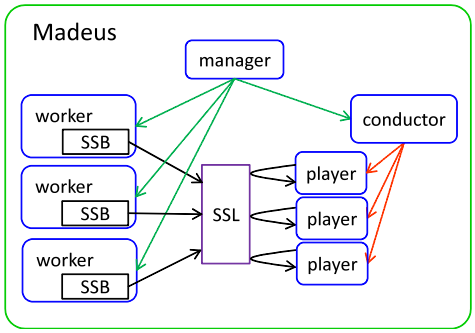

図 2　Madeus のアーキテクチャ
Fig. 2　Architecture of Madeus.

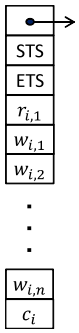

図 3　SSB
Fig. 3　SSB.

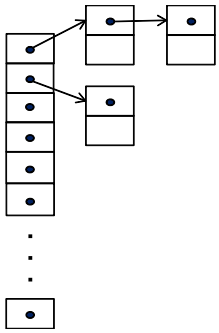

図 4　SSL
Fig. 4　SSL.

したならば，データベースサーバはグループコミット[*4]を実行する．I/O コストが高いことを考えると，グループコミットはデータベースライブマイグレーションのオーバヘッド削減に大きく寄与する．

4.2　アーキテクチャ

図 1 は Madeus のモデルを表す．データベースサーバから送信されたオペレーションを受信するために，Madeus はユーザとデータベースサーバの間に位置する．Madeus は市販の製品を無改造で使う．Madeus は共有ディスクを持たず，いわゆるシェアードナッシングアーキテクチャである．各ノードは 1 つのデータベースサーバインスタンスを動作させ，各インスタンスは複数のデータベース（テナント）を持つという，いわゆるシェアードプロセスモデルである [8]．ユーザは特定の 1 つのデータベースを使う．たとえば，3 つのノード 0，1，2 があり各ノードは 1 つのデータベースサーバインスタンスを動作させている．さらに，ノード 0 はテナント A を持っており，ノード 1 はテナント B と C を持っており，ノード 2 は 1 つもテナントを持っていない．この例では，テナント B と C は同じデータベースサーバインスタンスを共有する．ユーザがオペレーションを送信すると，Madeus はそれを受け取りオペレーションを解析して必要な情報を拾い上げる．そして，Madeus はユーザのテナントへ転送する．テナントからの応答は Madeus が受け取りユーザへ転送される．ユーザは Madeus から応答を受け取ると次の動作へ移り新たなオペレーションを送信する．

図 2 は Madeus のアーキテクチャを示す．Madeus は四種類のスレッドから構成される．つまり，1 つの *manager* と複数の *workers*，1 つの *conductor*，複数の *players* である．Madeus のすべてのスレッドは 1 つのノード上で動作する．ユーザがオペレーションを送信すると，*worker* がそれを受け取りユーザのデータベース（テナント）があるノードへ転送する．*worker* はスレーブトランザクションの要素としてそのオペレーションを保存するために動的に *syncset buffer*（*SSB*）を作成する．SSB は *syncset list*（*SSL*）に接続される．*manager* はすべてのスレッドを制御するアドミニストレータである．オペレータがマイグレーションの命令を指示すると，*manager* はそれを受け取りすべてのスレッドへ指示する．複数の *players* はオペレーションを並行に転送するために重要な役割りを果たす．*conductor* は，*player* がスレーブトランザクションを転送するタイミングを知らせる．

図 3 は SSB のアーキテクチャを示す．SSB は *start timestamp*（*STS*）と *end timestamp*（*ETS*）を保存するエリアを有する．また，SSB はスレーブトランザクションの要素を入れるための複数のエリアを有する．クエリの順序を保存するために，複数のエリアは first-in, first-out（FIFO）キューとなっている．**図 4** に SSL のアーキテクチャを示す．SSL の各要素は同じ STS を持つ SSB のリストへのポインタとなっている．*players* は同じ STS を持つ SSB を並行に転送する．

単一故障点を避けるためにアクティブな Madeus に加えてスタンバイの Madeus を用意することができる．Madeus はわずかな状態情報しか持っていないため，アクティブな Madeus からスタンバイの Madeus へスムーズにスイッチできる．マイグレーションを障害などを原因として止まらないようにするために，1 つのマスタテナントから複数のスレーブテナントを同期させてもよい．この場合，もし 1 つのスレーブテナントが障害となっても，残りのスレーブテナントで処理を継続できる．Pangea のプロトコル [23] を使えば，複数のマスタテナントを同期することもできる．

4.3　データベースライブマイグレーション

Madeus はソースノードからデスティネーションノードへ次のステップでテナントをマイグレートする．

ステップ 1：スナップショットの作成

オペレータがデータベースライブマイグレーションを Madeus に指示すると，Madeus はデータベースをダンプするトランザクションをマスタテナントに発行しマスタテ

[*4] 複数のコミットオペレーションから発行される複数の I/O アクセスを 1 つの I/O アクセスにまとめて高速化をはかる．

ナントのスナップショットを作成する．スナップショット作成中にもマスタテナントはユーザのオペレーションを実行しなければならない．作成するスナップショットにはスナップショット作成中に実行したオペレーションは含まれない．したがって，Madeus はスレーブトランザクションとしてそれらオペレーションを保存する．

ステップ 2：スレーブテナントの作成

スナップショットの作成が完了したら，Madeus はデスティネーションノードにスレーブテナントを作成する．この間もマスタテナントはユーザのオペレーションを実行し続ける．したがって，Madeus はスレーブトランザクションとしてそれらオペレーションを保存し続ける．

ステップ 3：スレーブトランザクションの転送

スレーブテナントの作成が完了した後，Madeus はスレーブトランザクションをスレーブテナントに実行させる．このとき，最小限のクエリ集合をシリアル転送が最小限になるように転送する．同時に，Madeus はユーザのオペレーションをマスタテナントへ転送し，同時にスレーブトランザクションとして保存する．

ステップ 4：スイッチオーバの実行

Madeus がすべてのスレーブトランザクションをスレーブテナントへ転送し終えたら，マスタテナントとスレーブテナントはコンシステントである．このとき Madeus はユーザのオペレーションの転送先をマスタからスレーブへ切り換えるスイッチオーバ動作を行う．

既存の転送アプローチ [15], [16], [17] と違って，ステップ 3 で LSIR を適用することで，最小限のクエリ集合を最小限のシリアル転送を実行することが我々のキーアイデアである．既存のアプローチはすべてのクエリをシリアル転送するものであったり [15], [16]，コミットクエリをシリアル転送するものであったりする [17]．コミットをシリアル転送すると，ロックのオーバヘッドが生じたり，グループコミットの恩恵を受けられなかったりという重大なデメリットを生じる．

4.4 アルゴリズム

本節では，*workers* と *manager*，*conductor*，*players* の詳細な動作を説明する．

4.4.1 Workers

worker は *master logical clock*（*MLC*）を管理することでオペレーションの相対順序を決める．*worker* は最初の参照オペレーションを SSB へストアする際に，MLC を STS として SSB にストアする．*worker* はコミットオペレーションを SSB にストアする際にも，MLC を ETS として SSB へストアする．

アルゴリズム 1 は更新トランザクションを受信したときの *worker* の動作を表す．トランザクションの最初の参照オペレーションとコミットオペレーションは，実行順序を決めるために，相互排他で実行される（lines 2–9 と lines 17–28）．*worker* が最初の参照オペレーションを受信したとき，*worker* は自分専用の SSB を作る（line 7）．inter-rw-dependency と wr-dependency，intra-rw-dependency を再現するために *worker* は最初の参照オペレーションをスレーブトランザクションの要素として保存する（line 8）．MLC はコミットオペレーションが実行されるたびに 1 増やす（line 21）．スレーブトランザクションを含んだ SSB はマイグレーション中に限り SSL へリンクされる（lines 23–27）．

もしトランザクションが参照のみであるならば，Madeus はどのオペレーションも捨てる（Algorithm 2）．

Algorithm 1 Worker for update transaction

```
 1: if first read operation then
 2:     enter critical region;
 3:     /* there is no commit operation executed */
 4:     send operation to master;
 5:     receive response from master;
 6:     STS := MLC;
 7:     allocate SSB;
 8:     save first operation to SSB;
 9:     leave critical region;
10:     send response to customer;
11: else if write operation then
12:     send operation to master;
13:     receive response from master;
14:     send response to customer;
15:     save operation to SSB;
16: else if commit operation then
17:     enter critical region;
18:     /* there is no first read operation executed */
19:     send operation to master;
20:     receive response from master;
21:     ETS := MLC++;
22:     save operation to SSB;
23:     if during migration then
24:         link SSB to SSL;
25:     else
26:         discard SSB;
27:     end if
28:     leave critical region;
29:     send response to customer;
30: else
31:     send operation to master;
32:     receive response from master;
33:     send response to customer;
34: end if
```

Algorithm 2 Worker for read-only transaction

```
1: send operation to master;
2: receive response from master;
3: send response to customer;
```

4.4.2 Manager

アルゴリズム 3 は *manager* の動作を表す．*manager* はデータベースライブマイグレーションのステップ 1 と 2，3，4 を制御する（4.3 節を参照）．クリティカルリージョンではコミットオペレーションを実行しない．これにより，MLC は変更されないことを保証する（lines 1–5）．ステップ 1 では，CreateSnapshot() 関数がマスタのスナップショットを作るトランザクションを発行してすぐに制御を戻す（line 3）．*manager* はスナップショットが出来上がるまで待つ（line 6）．ステップ 2 では，スナップショットが出来上がった後，*manager* はそのスナップショットを使ってデスティネーションノードにスレーブデータベースを作

Algorithm 3 Manager

```
 1: /* Step 1 */
 2: enter critical region;
 3: CreateSnapshot();
 4: MTS := MLC;
 5: leave critical region;
 6: WaitUntilSnapshotCreated();
 7: /* Step 2 */
 8: CreateDatabase();
 9: WaitUntilDatabaseCreated();
10: /* Step 3 */
11: StartConductorAndPlayers();
12: WaitUntilAllSyncsetsExecuted();
13: /* Step 4 */
14: SuspendAllTransactions();
15: WaitUntilAllSyncsetsExecuted();
16: SwitchOverFromMasterToSlave();
17: ResumeAllTransactions();
18: EndConductorAndPlayers();
```

Algorithm 4 Conductor

```
 1: SLC := GetSmallestSTS();
 2: /* step 3 */
 3: while SLC ≤ MLC do
 4:     OrderToPropagateTheFirstOperation();
 5:     WaitUntilAllTheFirstOperationPropagated();
 6:     oldSLC := SLC;
 7:     SLC := GetSmallestSTS();
 8:     CCN := GetConcurrentCommitNumber();
 9:     OrderToPropagateCommitOperation();
10:     WaitUntilAllCommitOperationPropagated();
11: end while
12: WakeUpManager();
```

る（line 8）．ステップ 3 ではスレーブトランザクションを並行に転送する．このために，*manager* は *conductor* と複数の *players* を作ってステップ 3 を *conductor* と *players* に任せる（lines 11–12）．スレーブがマスタに追いつくと，つまり，すべての SSB がスレーブへ転送が完了すると，*manager* はステップ 4 へ移る．スレーブがマスタとコンシステントであることを保証するために，*manager* はオペレーションの送信を中断させる（line 14）．すべてのトランザクションが停止しすべてのスレーブトランザクションがスレーブへ転送された後（line 15）*manager* はスイッチオーバを実行する（line 16）．*manager* はオペレーションの送信を再開する（line 17）．

4.4.3 Conductor

アルゴリズム 4 は *conductor* のアルゴリズムを表す．*conductor* はスレーブトランザクションを並行に転送するために重要な役割りを果たす．*conductor* はコンシステンシを崩さずにオペレーションの転送を制御するための *slave logical clock*（*SLC*）を管理する．*conductor* は *concurrent commit number*（*CCN*）という重要なローカル変数を使う．これは，並行に実行できるコミット数を表す．*conductor* は SSL をスキャンして最小の STS を探し出して SLC にセットする（line 1）．条件 SLC ≤ MLC が成り立つ間，次のステップを繰り返す（lines 3–11）．*conductor* は，STS が SLC と等しい最初の参照オペレーションをスレーブへ転送するように，*players* に指示する（line 4）．*conductor* はすべての最初の参照オペレーションがスレーブへ転送されるのを待つ（line 5）．*conductor* は次の SLC を得て

Algorithm 5 Player

```
 1: /* propagate first operation */
 2: WaitUntilOrder();
 3: SendOperation();
 4: RecvResponse();
 5: InformToConductor();
 6: /* propagate write operations */
 7: while write operation exists do
 8:     SendOperation();
 9:     RecvResponse();
10: end while
11: /* propagate commit operation */
12: WaitUntilOrder();
13: SendOperation();
14: RecvResponse();
15: InformToConductor();
```

（line 7）CCN を算出する（line 8）．*conductor* は ETS が次の式を満たすコミットオペレーションを *players* に並行に送信するように指示する（line 9）．

$$oldSLC \leq ETS \leq oldSLC + CCN - 1 \tag{1}$$

たとえば，現在の SLC が 3 で次の SLC が 5 の場合，CCN は 2(= 5 − 3) となる．このとき，式 (1) は $3 \leq ETS \leq 3 + 2 - 1 = 4$ となる．したがって，*conductor* は ETS が 3 または 4 である SSB を持っている *players* にコミットオペレーションを並行に送信するように指示する．これはグループコミットの恩恵を受けることができる．*conductor* はすべてのコミットオペレーションが転送されるのを待つ（line 10）．*conductor* がループを抜けると，*manager* を呼び起こす（line 12）．

4.4.4 Players

アルゴリズム 5 は *player* のアルゴリズムを示す．*players* はスレーブにスレーブトランザクションを並行に転送する．*conductor* はスレーブトランザクションを転送する順番を *players* に指示するだけである．*workers* はユーザとマスタテナントの間でパケットをやり取りし，かつ，スレーブトランザクションを作る．*player* は，*conductor* が最初の参照オペレーションを送る指示を出すのを待つ（line 2）．もし *player* が指示を受け取ったら，最初の参照オペレーションをスレーブへ転送する（line 3）．*player* は応答を受け取ったら（line 4），*player* は *conductor* にその旨を通知する（line 5）．*player* はすべての更新オペレーションを転送する（lines 7–10）．*player* は，*conductor* がコミットオペレーションを転送するように *player* に指示するのを待つ（line 12）．*player* は指示を受け取ると，*player* はスレーブにコミットオペレーションを転送する（line 13）．複数の *players* がコミットオペレーションを転送するため，我々はグループコミットの恩恵を受けることができる．*player* が応答を受信すると（line 14），*player* はその旨を *conductor* へ通知する（line 15）．Madeus の動作例は付録 A.3 を参照願いたい．

4.5 理論的解析

本節では Madeus の理論的な特徴を議論する．

4.5.1　ライブマイグレーションコンシステンシ

定理 2（ライブマイグレーションコンシステンシ）
Madeus はスナップショット分離下でマスタテナントとスレーブテナントがコンシステントになることを保証してデータベースライブマイグレーションを実行する．

LSIR は弱いルールであるため，Madeus はスレーブトランザクションを並行に転送できる多くのチャンスをもたらす．もっと重要なことは，Madeus はコミットオペレーションを並行に実行させるため，我々はグループコミットの恩恵を受けることができる．

4.5.2　LSIR の有効性

本項では，LSIR の有効性を理論的に議論する．C_r と C_w，C_c を参照オペレーション，更新オペレーション，コミットオペレーションのコストとする．N_r と N_w を一トランザクション内の参照オペレーションの数，更新オペレーションの数とする．総トランザクション数を N_{total} とする．C'_c をグループコミットのコスト，N' をグループコミットオペレーションの数とする．Madeus のトータルコストを C_{madeus} とし，LSIR のどのルールも含まない場合のトータルコストを C_{ALL} とする[*5]．

$$C_{madeus} = N_{total}(C_r + N_w C_w) + N' C'_c + (N_{total} - N')C_c$$

$$C_{ALL} = N_{total}(N_r C_r + N_w C_w + C_c)$$

したがって，C_{ALL} と C_{madeus} の差は以下のようになる．

$$C_{ALL} - C_{madeus} = N_{total}(N_r - 1)C_r + N'(C_c - C'_c) \quad (2)$$

$N_r \geq 1$ と $N' \geq 0$，$C_c > C'_c$ であることを考えると，$C_{madeus} - C_{ALL} \geq 0$ となる．これは C_{madeus} は決して C_{ALL} よりも大きくならないことを意味する．したがって，Madeus はどのような N_{total} または N_r, N' でも有効である．また，N_{total} または N_r，N' が大きいほど $C_{madeus} - C_{ALL}$ は大きくなり，Madeus の有効性は大きくなる．

5.　評価

我々は Madeus を実装し，TPC-W ベンチマーク [24] を使って性能評価を行った．5.3 節では様々なミドルウェアのマイグレーション時間を評価し，5.4 節では 800-MB のデータベースの評価を行った．5.5 節では，データベースサイズを変えた場合の評価を行った．5.6 節ではマルチテナント環境において高負荷のテナントをマイグレートすべきか低負荷のテナントをマイグレートすべきかという疑問に対する答えを導いた．

[*5] 本来はアボートオペレーションの影響も考えられるが，その出現頻度は低いため，そのコストを無視するものとする．

5.1　TPC-W ベンチマーク

クラウドコンピューティングサービスとして意味ある評価を行うためには実際のエンタープライズアプリケーションを使った評価を行うことが重要である．そこで，我々は TPC-W ベンチマーク [24] を使った評価を行った．このベンチマークはオンラインブックストアにアクセスするお客をモデル化したものである．このベンチマークはブラウジングリクエストの割合が異なる，ショッピングミックス，ブラウジングミックス，オーダリングミックスの 3 つのメニューがある．3 つすべてのミックスは同じトランザクション集合を持つが，参照のみのトランザクションの割合が異なる．ブラウジングミックスは 95%，ショッピングミックスは 80%，オーダリングミックスは 50%である．更新が多い方がレプリケーションに大きな負荷がかかるため，我々はオーダリングミックスを選んだ．

TPC-W ベンチマークはエミュレートするブラウザ (EB) の数を変えることで負荷を変えることができる．EB の数を増やすと負荷は増える．各 EB はアプリケーションサーバとリクエストをやりとりをシミュレートする．EB は応答を受け取った後，何秒か待ったのちに，次の新しいリクエストを発行する．

5.2　実験環境

我々の実験環境ではマスタテナント用 PostgreSQL とスレーブテナント用 PostgreSQL それぞれに 1 台のマシンを割り当てた．ボトルネックを避けるために Tomcat と負荷発生プログラムはそれぞれ 3 台のマシンを割り当てた．ミドルウェアはすべての実験において `vmstat` コマンドを用いて CPU の使用状況を確認したところ，どの実験においてもアイドルタイムがほぼ 100%であったため，1 台で十分であった．すべてのマシンは，1 つの 3.1-GHz Xeon E3-1220 CPU，4 つの core，4 つの thread，8-MB キャッシュ，16-GB RAM，1 つの 250-GB SATA HDD というスペックのマシンを使った．すべてのマシンは 1-Gbps Ethernet LAN で接続した．Linux の kernel のバージョンは 2.6.32，PostgreSQL のバージョンは 9.2.6，Tomcat のバージョンは 7.0.27 を用いた．TPC-W ベンチマークはオープンソースで公開されているもの [25] を PostgreSQL のクエリに書換えて使用した．

ミドルウェアでは受信したパケットから必要な情報を取り出す必要があるため，PostgreSQL のプロトコルを理解できる必要がある．我々のプロトタイプでは，`libpq` プロトコルと type 4 JDBC プロトコルをミドルウェアに実装した．我々のプロトタイプは実用的な実装であるにもかかわらず，C 言語でたったの 5,000 行足らずである．これは，LSIR が単純であるためである．

我々の環境でどのくらいの EB 数で低負荷，中負荷，高負荷を発生させることができるかを知るために 1 つのテナ

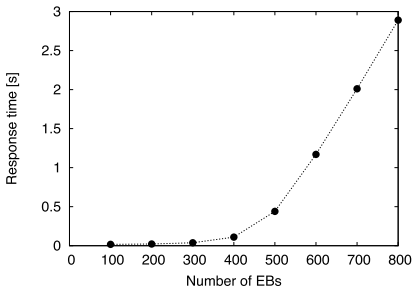

図 5　予備実験の結果
Fig. 5　Results of preliminary experiment.

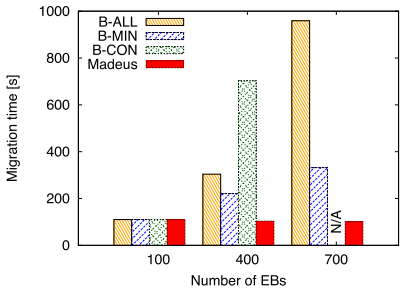

図 6　各アプローチのマイグレーションタイム
Fig. 6　Migration time of each approach.

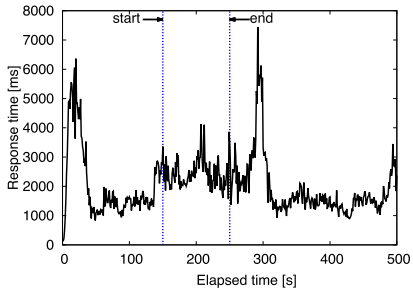

図 7　800-MB データベースのレスポンスタイム
Fig. 7　Response time of 800-MB Database.

ントを使った予備実験を行った．**図 5** が示すように，EB 数が 100 と 200，300 では平均レスポンスタイムが 100 ミリ秒以下である．また，EB 数が 400 と 500，600 では平均レスポンスタイムが 100 ミリ秒以上であるが 2 秒以下である．さらに EB 数が 700 と 800，900，1,000 では平均レスポンスタイムが 2 秒以上である．顧客は 2 秒以内にレスポンスが返ってくることを期待していること [2], [3] を考えると，EB 数が 700 と 800，900，1,000 では「高負荷」と定義する．さらに，EB 数が 100 と 200，300 では「低負荷」と定義し，EB 数が 400 と 500，600 では「中負荷」と定義する．

5.3　マイグレーションタイム

並行転送をともなうデータベースマイグレーションを実行する Madeus の有効性を検証するために，3 つの既存の転送方式を備えたミドルウェアとマイグレーションタイムを比較した．

5.3.1　3 つのベースラインミドルウェアアプローチ

我々の知る限りではデータベースライブマイグレーションを実行できる既存のミドルウェアアプローチは存在しない．データベースライブマイグレーションの重要なキーはオペレーションの転送アプローチである．この転送アプローチという観点では 3 つの既存アプローチが考えられる [15], [16], [17]．そこで，これらを使用した 3 つのベースラインミドルウェアを実装した．第 1 のミドルウェアを *baseline middleware with all serial propagation*（*B-ALL*）と呼ぶ．B-ALL はすべてのトランザクションをシリアルに転送する．マスタテナントが複数のトランザクションを並行に実行するのに対して，B-ALL はそれらトランザクションをコミット順にシリアルに並べ，スレーブテナントへ 1 つずつ転送する．スレーブはコミット順にシリアルに 1 つずつトランザクションを実行することになる．第 2 のミドルウェアを *baseline middleware with minimum serial propagation*（*B-MIN*）と呼ぶ．B-MIN は B-ALL と同じくコミット順にシリアルに転送するが，B-ALL と違ってすべてのトランザクションではなくてコンシステントにするために必要な最小限のスレーブトランザクションを転送する [15], [16]．第 3 のミドルウェアを *baseline middleware with concurrent propagation*（*B-CON*）と呼ぶ．B-CON は Madeus と同じく最初の参照オペレーションと更新オペレーションを並行に転送するが，コミットオペレーションはシリアルに転送しなければならない [17]．したがって，Madeus と違って B-CON はグループコミットの恩恵を受けることはできない．

表 2　ミドルウェアアプローチ間の違い
Table 2　Difference among middleware approaches.

	MIN	CON-FW	CON-COM
B-ALL			
B-MIN	√		
B-CON	√	√	
Madeus	√	√	√

B-ALL と B-MIN，B-CON，Madeus の違いを**表 2** に示す．もし最小のクエリ集合を転送する機能を有するならば，MIN という省略文字で表す．もし最初の参照オペレーションと更新オペレーションを並行に転送する機能を有するならば，CON-FW という省略文字で表す．もし，コミットオペレーションを並行に転送する機能を有するならば，CON-COM という省略文字で表す．

5.3.2　結果

従来の研究 [9], [10], [17] を参考にして，データベースサイズは 800 MB（288,000 人のお客，100,000 個のアイテム）で実験した．**図 6** は Madeus と 3 つのミドルウェアのマイグレーションタイムである．X 軸は負荷（EB の数）を表し Y 軸はマイグレーション時間を表す．低負荷（100 EBs）では，B-ALL と B-MIN，B-CON，Madeus のマイグレーションタイムはほぼ同じで約 110 秒であった．中負荷（400 EBs）では，B-ALL と B-MIN，B-CON，Madeus のマイグレーションタイムはそれぞれ 304 と 221，703，104 秒であった．高負荷（700 EBs）では，B-ALL と B-MIN，B-CON，Madeus のマイグレーションタイムはそれぞれ 959 と 332，N/A，101 秒であった．B-ALL と B-MIN の

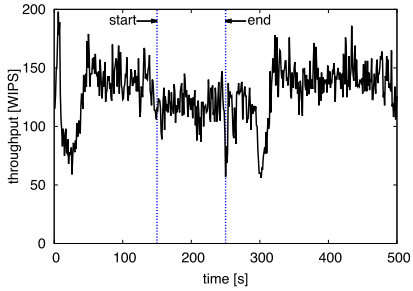

図 8　800-GB のデータベースの場合のスループット

Fig. 8　Throughput of 800-GB Database.

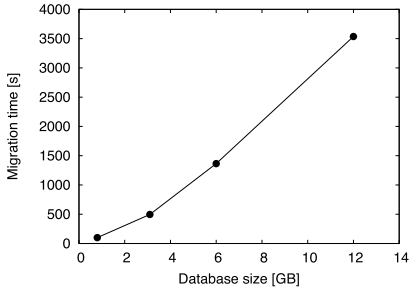

図 9　データベースサイズを変えたときのマイグレーションタイム

Fig. 9　Migration time with changing database size.

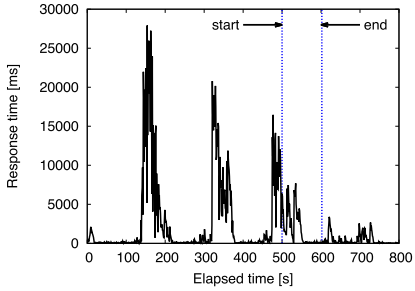

図 10　テナント B をマイグレートしたときのテナント A のレスポンスタイム

Fig. 10　Response time of tenant A when tenant B was migrated.

マイグレーションタイムはリニアに増えている．B-CON のマイグレーションタイムは中負荷で極端に増えている．高負荷での B-CON は，スレーブがマスタに追いつけずステップ 3 がずっと続き，マイグレーションタイムを計測できなかった．驚いたことに Madeus のマイグレーションタイムは負荷が増えるにつれて小さくなった．

Madeus では高負荷のときのマイグレーションタイム（およそ 101 秒）が低負荷や中負荷のときのマイグレーションタイム（およそ 110 と 104 秒）よりも小さくなった．この理由として 2 つ考えられる．第 1 に，高負荷の場合はより多くのクエリを実行する．これにより，より多くのデータアイテムがメモリにキャッシュされることになり，「ウォーム状態」を作り出す．第 2 に，Madeus はコミットを並行実行できることに起因する．コミットを並行に実行すれば，グループコミットにより高速化が実現できる．負荷が高いほど並行に実行できるコミットクエリも多くなる．

中負荷の B-ALL と B-CON を比較すると，B-CON のマイグレーションタイムの方が大きい．B-CON はコミットオペレーションを順番にシーケンシャルに実行するために mutex lock を使った排他制御を行っている．B-CON はコミットのたびにすべてのスレッドで lock を奪いあう．このオーバヘッドが大きいために，最初の参照オペレーションと異なるトランザクションの更新オペレーションを並行実行できるメリットを失ってしまった．B-ALL と B-MIN，Madeus はこのコードがないため，このオーバヘッドは生じない．

5.4　性能

本実験では，テナントが 1 つの場合の Madeus のレスポンスタイムとスループットを測定した．テナントのサイズは 800 MB（288,000 人のお客，100,000 個のアイテム）とした．**図 7** は X 軸は経過時間，Y 軸は高負荷時（700 EBs）のレスポンスタイムを示す．実験開始から 50 秒のあたりまでレスポンスタイムの悪化が見られるが，これはベンチマークのウォーミングアップによるものである．データベースライブマイグレーションは 150 秒付近で開始し，250 秒付近で終了している．マイグレーションの開始時にレスポンスタイムが増加しているが，これは *manager* が MTS を得るためにクリティカルリージョンを実行するためである（アルゴリズム 1 の 17–28 行目とアルゴリズム 3 の 2–5 行目）．マイグレーションの終了時にもレスポンスタイムが増加しているが，これは *manager* がスイッチオーバを実行するためにすべてのトランザクションをサスペンドしているからである（アルゴリズム 3 の 14–15 行目）．幸運なことにマイグレーション中のレスポンスタイムは通常時と比べてわずかな増加である．290 秒付近に大きなレスポンスタイムの増大がみられるが，これは PostgreSQL のチェックポイントによるものである．マイグレーションによるレスポンスタイムの増大の方がチェックポイントによるレスポンスタイム増大よりも小さいことを考えると，マイグレーションによるオーバヘッドは十分に小さいといえる．

図 8 は X 軸は経過時間，Y 軸は高負荷時（700 EBs）のスループットを示す．実験開始から 50 秒のあたりまでスループットの低下が見られるが，これはベンチマークのウォーミングアップによるものである．データベースライブマイグレーションは 150 秒付近で開始し，250 秒付近で終了している．上記で述べた理由で，マイグレーションの開始時と終了時にスループットの悪化がみられる．マイグレーション中のスループットは，通常時と比べてわずかに低下するのみである．290 秒付近に大きなスループットの低下がみられるが，これは PostgreSQL のチェックポイントによるものである．マイグレーションによるスループット低下の方がチェックポイントによるスループット低下よりも小さいことを考えると，マイグレーションによるオーバヘッドは十分に小さいといえる．

5.5　データベースサイズの変更による影響

図 9 は高負荷時（700 EBs）におけるデータベースサイズを 0.8 と 3.1，6.2，12 GB に変えたときの Madeus のマイグレーションタイムを表す．データベースサイズは**表 3** に示すように TPC-W ベンチマークの 2 つのパラメータで

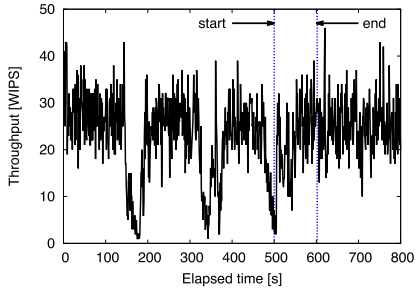

図 11　テナント B をマイグレートしたときのテナント A のスループット
Fig. 11　Throughput of tenant A when tenant B was migrated.

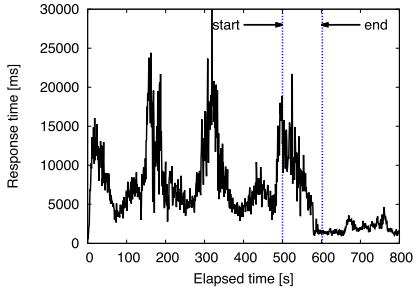

図 12　テナント B をマイグレートしたときのテナント B のレスポンスタイム
Fig. 12　Response time of tenant B when tenant B was migrated.

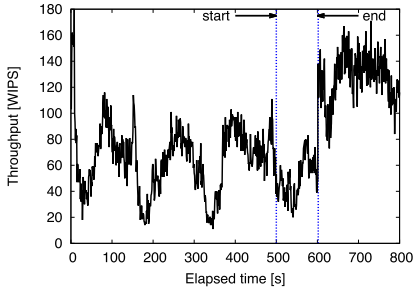

図 13　テナント B をマイグレートしたときのテナント B のスループット
Fig. 13　Throughput of tenant B when tenant B was migrated.

表 3　データベースサイズ
Table 3　Database size.

items	emulated browsers	database size [GB]
100,000	100	0.8
500,000	500	3.1
1,000,000	1,000	6.2
2,000,000	2,000	12

あるアイテム数とブラウザ数から決定される．マイグレーションタイムはそれぞれ 101 秒，496 秒，1,365 秒，3,536 秒となった．データベースサイズが増えるとマイグレーションタイムもほぼリニアに増加する．現実的なデータベースサイズを現実的な時間でマイグレーションできることを考えると，Madeus は有効性が高いといえる．

5.6　マルチテナント環境における性能

現実的な環境では，複数のテナントが同じノードに載っていて，そのうち 1 つのテナントが高負荷になり他のテナントを圧迫しホットスポットになりえる．データベースライブマイグレーションはこの問題を解決するために必要である．ここで湧いてくる疑問は低負荷のテナントをマイグレートすべきか，高負荷のテナントをマイグレートすべきか，というものである．我々はこの疑問に答えるために実験を行った．この実験で使ったノードはノード 0 と 1 の 2 つである．最初はノード 0 はテナント A と B，C の 3 つを収容しており，ノード 1 は 1 つもテナントを持っていない．また，テナント B は高負荷（700 EBs）でありテナント A と C は低負荷（200 EBs）である．テナント B が高負荷であるためにノード 0 はホットスポットである．我々は次の 2 つのケースでレスポンスタイムとスループットを計測した．

ケース 1：高負荷なテナントをマイグレートする

高負荷なテナント B をノード 0 から 1 へマイグレートし，レスポンスタイムとスループットを測定する．テナント A と B のレスポンスタイムを**図 10** と**図 12** に示し，スループットを**図 11** と**図 13** に示す．テナント C のレスポンスタイムとスループットはテナント A のレスポンスタイムとスループットとほぼ同じため，図示せず省略した．データベースライブマイグレーションは 500 秒付近で開始し，600 秒付近で終了している．したがってテナント B をマイグレーションするのに約 100 秒かかっている．

テナント A の負荷は低いにもかかわらず，図 10 に示すように，レスポンスタイムは小さくない．これは，高負荷なテナント B と同じノードで動作しており，テナント A が圧迫されているからである．図 10 と図 11 には，PostgreSQL のチェックポインティングによる性能低下も見ることができる．

図 10 よりテナント A のマイグレーション中の最大レスポンスタイムは通常時の最大レスポンスタイムよりも小さいことが分かる．これより，テナント A のレスポンスタイムはテナント B のマイグレーションからほとんど影響を受けないことが分かる．マイグレーションの後，テナント A のレスポンスタイムは小さくなっている．これは，高負荷のテナント B がノード 0 からノード 1 へマイグレートされたからである．図 11 よりテナント A のマイグレーション中のスループットは通常動作時のスループットとほとんど変わらない．

図 12 よりテナント B のレスポンスタイムは通常時でも大きい．これは，テナント B の負荷は高負荷であることが理由である．幸いなことに，マイグレーション中のレスポンスタイムは通常時と比較しても大きくない．図 13 に示すように，マイグレーション中のスループット低下も小さい．これらのことからマイグレーションのオーバヘッドはわずかであることが分かる．ノード 1 のスレーブはマイグレーション中にクエリを実行するため，マイグレーション終了後はノード 1 はコールドではない．したがって，マイグレーション直後の性能悪化はほとんどみられない．これは，文献 [9] で取り上げたマイグレーション直後のコールドキャッシュ問題を解決するものである．結果として高負荷なテナント B のレスポンスタイムはマイグレーション後に小さくなる．

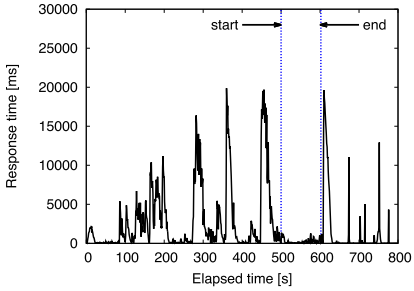

図 14　テナント C をマイグレートしたときのテナント A のレスポンスタイム

Fig. 14　Response time of tenant A when tenant C was migrated.

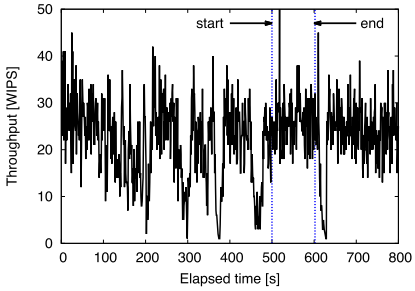

図 15　テナント C をマイグレートしたときのテナント A のスループット

Fig. 15　Throughput of tenant A when tenant C was migrated.

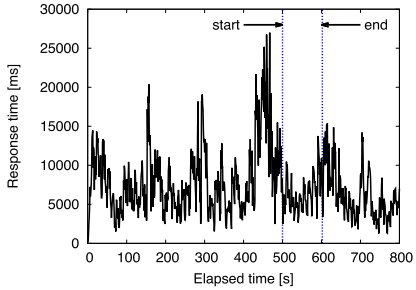

図 16　テナント C をマイグレートしたときのテナント B のレスポンスタイム

Fig. 16　Response time of tenant B when tenant C was migrated.

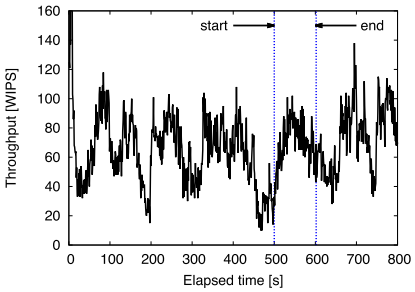

図 17　テナント C をマイグレートしたときのテナント B のスループット

Fig. 17　Throughput of tenant B when tenant C was migrated.

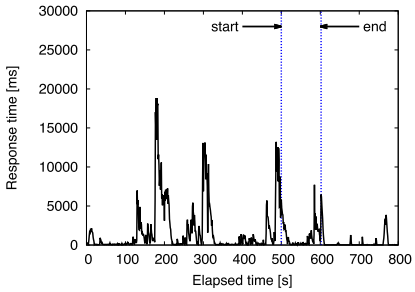

図 18　テナント C をマイグレートしたときのテナント C のレスポンスタイム

Fig. 18　Response time of tenant C when tenant C was migrated.

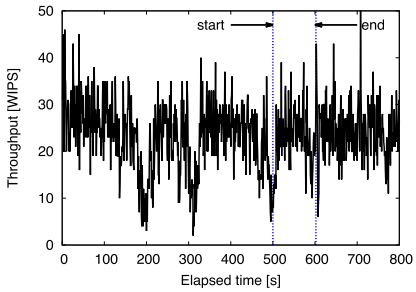

図 19　テナント C をマイグレートしたときのテナント C のスループット

Fig. 19　Throughput of tenant C when tenant C was migrated.

ケース 2：低負荷なテナントをマイグレートする

低負荷なテナント C をノード 0 からノード 1 へマイグレートする．**図 14** と**図 16**，**図 18** はそれぞれテナント A と B，C のレスポンスタイムである．**図 15** と**図 17**，**図 19** はそれぞれテナント A と B，C のスループットである．データベースライブマイグレーションは 500 秒付近で開始し，630 秒付近で終了している．したがってテナント C をマイグレーションするのに約 130 秒かかっている．

図 14 から分かるようにテナント A のレスポンスタイムはマイグレーションによって減らない．これには 2 つの理由がある．1 つ目は，マイグレーション後も高負荷のテナント B がテナント A を圧迫しているからである．2 つ目は，テナント C のマイグレーション時間はテナント B のマイグレーション時間よりも長い（ケース 2 は 130 秒 vs. ケース 1 は 100 秒）．これは 5.3.2 項の高負荷のテナントのマイグレーション時間の方が短いという実験結果（図 6）と一致する．さらに，図 16 から分かるように，テナント B のレスポンスタイムはマイグレーション後も減らない．なぜならば，ノード 0 はマイグレーション後も高負荷（200 EBs + 700 EBs = 900 EBs）である．図 18 はテナント C のレスポンスタイムである．マイグレーション前はテナント A と似ている（図 14）．マイグレーション後は，ノード 1 にはテナント C しか存在しないため，テナント C のレスポンスタイムは非常に短い．

どちらのテナントをマイグレートすべきか

低負荷のテナントをマイグレートすべきか，高負荷のテナントをマイグレートすべきか？　我々の答えは 2 つの理由から高負荷のテナントをマイグレートすべきである．第 1 に，高負荷のテナントをマイグレートした方が全体の負荷を減らしホットスポットを解消するには効果が高いからである．第 2 に，キャッシュ効果やグループコミットの恩恵を受けて，高負荷のテナントのマイグレーションタイムが短いからである．

6.　関連研究

我々の知る限りでは，DBaaS で使えるマイグレーションアプローチは存在しない．

仮想マシンのライブマイグレーション

Clark らは仮想マシンのライブマイグレーションを初めて提案した [26]．彼らのアプローチはソースマシンからデスティネーションマシンへメモリページを繰り返しコピーするという手法を使っている．Bradford らは WAN 環境でライブマイグレーションする提案である [27]．彼らのアプローチはメモリイメージのみならずディスクのデータもマイグレートする．Hines らはポストコピーアプローチを提案している [28]．Sväard らはマイグレーションスループットを向上させるために圧縮したメモリページを転送するアプローチを提案している [29]．Song らはページを並列に転

送するアプローチを提案している [30]．Mashtizadeh らはメモリとディスクイメージを LAN 環境と WAN 環境で効率良く転送する方式を提案している [31]．

データベースライブマイグレーションシステム

Minhas らは RemusDB と呼ぶ Remus [32] を改造して高可用性 DBMS システムを提案している [33]．これはデータベースシステムを Remus 用に改造する必要がある．2 つのビルトインレプリケーションアプローチが提案されている．Das らはキャッシュを繰り返しコピーする方式を提案している [9]．マイグレーション後にウォームキャッシュで動作を開始することをねらっている．Elmore らはインデックス構造を利用したレプリケーションアプローチを提案している [10]．マイグレーションの最中にはインデックス構造は変わらないという非現実的な仮定を必要とする．両方式ともに，Madeus とは違って，データベースサーバの改造をともなう．

Lazy レプリケーションミドルウエア

クエリの lazy シリアル転送プロトコルを持ったアプローチが提案されている [15], [16]．しかし，マスタはクエリを並行実行するため，スレーブがマスタに追いつけないという問題が発生する可能性がある．追い付けたとしても時間がかかる．

Lazy レプリケーションシステム

Daudjee らは LSIR とよく似たルールを提案している [17]．しかし，彼らのアプローチはミドルウェアではなく，データベースサーバを改造するものである．また彼らは first-committer-wins ルール [18] [*6] を採用しているため，現実的ではない．これにより彼らのルールは制約が強すぎる．つまり，first-committer-wins ルールが原因でマスタとスレーブは同じ順序でコミットを実行しなければならない．実際のデータベースサーバ，たとえば Oracle や SQL Server，PostgreSQL などは first-updater-wins ルールを採用している．Madeus は first-updater-wins ルールを採用しているため，より制約が弱いルールである LSIR を提案できた．さらに，彼らの提案は非現実的な仮定をおいている．つまり，マスタのオペレーションはスレーブにタイムスタンプの順序でブロードキャストされると仮定している．また，スナップショットを作成するオペレーションの存在を仮定している．これらの非現実的な仮定から，彼らはプロトタイプを作成していない（シミュレーションによる評価を行っている）．

ログベースのレプリケーションシステム

Barker らは Slacker と呼ぶデータベースライブマイグレーションミドルウェアを提案している [11]．これはよく知られた PID 制御をベースとしている [34]．Slacker はスレーブへの更新をバイナリトランザクションログを解析して作っている．したがって，Slacker は特定のデータベースサーバの種類やバージョンに依存する．さらに，彼らの提案は共有プロセスモデルをベースせず，VM ベースのモデルと同じで各テナントごとにデータベースインスタンスを必要とする．それゆえに効率の良いリソース共有が実現できていない．PostgreSQL はログベースのレプリケーション機能を持っていてこれをスレーブへの更新とするアプローチが考えられる [35]．この場合，すべてのテナントでログを共有するため，共有プロセスモデルは利用できない．さらに，ログフォーマットはバージョンによって違うため，異なるバージョンからなるシステムは構成できない．

高可用性レプリケーションシステム

Oracle Real Application Cluster（RAC）[36] と IBM DB2 pureScale [37], Microsoft Windows Server Failover Cluster（WSFC）[38] は商用の高可用性システムである．Madeus と違って，それらは共有ディスクに依存している．

7. おわりに

DBaaS 環境で効率的なデータベースライブマイグレーションを行える Madeus と呼ぶミドルウェアを提案した．Madeus は市販のハードウェアとソフトウェアを無改造で使用でき，特殊なツールに頼ることなく，データベースサーバとクライアントに透過なミドルウェアである．Madeus はスナップショット分離下でスレーブをマスタとコンシステントにする並行転送プロトコルを備える．このプロトコルは最初の参照オペレーションや更新オペレーションのみならずコミットオペレーションも転送できる．結果として，Madeus は並行性だけでなくグループコミットの恩恵も受けることができる．我々の提案はシンプルであるため，Madeus を C 言語で 5,000 行足らずで作成した．TPC-W ベンチマークを使って性能評価を行ったところ，既存の転送プロトコルを実装した 3 つのミドルウェアよりも，Madeus は短い転送時間でマイグレーションを実行できた．特に，高負荷である場合にその効果は高かった．また，マルチテナント環境でホットスポットを解決するためには低負荷のテナントをマイグレートすべきか，高負荷のテナントをマイグレートすべきか，の実験も行った．その結果，ホットスポットのノードの負荷を大きく下げられること，マイグレーションタイムが短いこと，から高負荷のテナントをマイグレートすべきという知見を得られた．Madeus はとても実用的で DBaaS に効果的なアプローチであることが分かった．

[*6] 2 つのトランザクションが同じデータアイテムを更新しようとしたとき，最初にコミットしたトランザクションだけが成功し，もう一方のトランザクションはアボートする．

参考文献

[1] Multitenancy, available from ⟨http://en.wikipedia.org/wiki/Multitenancy⟩.
[2] A new 2 second rule, available from ⟨http://blogs.

keynote.com/the_watch/2009/09/
the-new-2-second-website-rule.html⟩.
[3] Akamai Reveals 2 Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times, available from ⟨http://www.akamai.com/2seconds⟩.
[4] Amazon RDS, available from ⟨http://aws.amazon.com⟩.
[5] Microsoft Azure, available from ⟨http://azure.microsoft.com⟩.
[6] Google Cloud SQL, available from ⟨http://cloud.google.com⟩.
[7] Koto, A., Kono, K. and Yamada, H.: A Guideline for selecting live migration policies and implementations in clouds, *CloudCom* (2014).
[8] Curino, C., Evan, J., Raluca, P., Nirmesh, M., Wu, E., Madden S., Balakrishnan, H. and Nickolai, Z.: Relational Clound: A Database-as-a-Service for the Cloud, *CIDR* (2011).
[9] Das, S., Nishimura, S., Agrawal, D. and Elabbadi, A.: Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration, *PVLDB* (2011).
[10] Elmore, A., Das, S., Agrawal, D. and Elabbadi, A.: Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms, *SIGMOD* (2011).
[11] Barker, S., Chi, Y., Moon, H., Hacigumus, H. and Shenoy, P.: Cut Me Some Slack: Latency-Aware Live migration for Databases, *EDBT* (2012).
[12] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. and Schwarz, P.: ARIES: A Transaction Recovery Method supporting Fine-granularity Locking and Partial Rollbacks using Write-ahead Logging, *TODS* (1992).
[13] Percona XtraBackup, available from ⟨http://www.percona.com/⟩.
[14] Gray, J., Helland, P., O'Neil, P. and Shasha, D.: The Dangers of Replication and a Solution, *SIGMOD* (1996).
[15] Röhm, U., Böhm, K., Schek, H. and Schuldt, H.: FAS – A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components, *VLDB* (2002).
[16] Plattner, C. and Alonso, G.: Ganymed: Scalable Replication for Transactional Web Applications, *Middleware* (2004).
[17] Daudjee, K. and Salem, K.: Lazy Database Replication with Snapshot Isolation, *VLDB* (2006).
[18] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P.: A Critique of ANSI SQL Isolation Levels, *SIGMOD* (1995).
[19] Mishima, T. and Fujiwara, Y.: Madeus: Database Live Migration Middleware under Heavy Workloads for Cloud Environment, *SIGMOD* (2015).
[20] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P. and Shasha, D.: Making Snapshot Isolation Serializable, *TODS* (2005).
[21] Blind write, available from ⟨http://en.wikipedia.org/wiki/Blind_write⟩.
[22] Jung, H., Han, H., Fekete, A. and Röhm, U.: Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios, *TODS* (2005).
[23] Mishima, T. and Nakamura, H.: Pangea: An Eager Database Replication Middleware guaranteeing Snapshot Isolation without Modification of Database Servers, *VLDB* (2009).
[24] available from ⟨http://www.tpc.org/tpcw/⟩.
[25] Java TPC-W implementation distribution, available from ⟨http://www.ece.wisc.edu/~pharm/tpcw.shtml⟩.
[26] Clark, C., Fraser, K., Hand, S., Gorm, H., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live Migration of Virtual Machines, *NSDI* (2005).
[27] Bradford, R., Kotsovinos, E., Feldmann, A. and Schiöberg, H.: Live Wide-Area Migration of Virtual Machines Including Local Persistent State, *VEE* (2007).
[28] Hines, M. and Gopalan, K.: Post-Copy Based Live Virtual Macine Migration Using Adaptie Pre-Paging and Dynamic Self-Ballooning, *VEE* (2009).
[29] Sväard, P., Hudzia, B., Tordsson, J. and Elmroth, E.: Post-Copy Based Live Virtual Macine Migration Using Adaptie Pre-Paging and Dynamic Self-Ballooning, *VEE* (2009).
[30] Song, X., Shi, J., Liu, R., Yang, J. and Haibo, C.: Parallelizing Live Migration of Virtual Machines, *VEE* (2013).
[31] Mashtizadeh, A., Cai, M., Tarasuk-Levin, G., Koller, R., Garfinkel, T. and Setty, S.: XvMotion: Unified Virtual Machine Migration over Long Distance, *ATC* (2014).
[32] Cully, B., Lefebre, G., Meyer, D., Feeley, M., Nutchinson, N. and Warfield, A.: Remus: High availability via asynchronous virtual machine replication, *NSDI* (2008).
[33] Minhas, U., Rajagopalan, S., Cully, B., Aboulnaga, A., Salem, K. and Warfield, A.: RemusDB: Transparent High Availabililty for Database Systems, *NSDI* (2011).
[34] PID controller, available from ⟨http://en.wikipedia.org/wiki/PID_controller⟩.
[35] PosgreSQL Global Development Group, available from ⟨http://www.postgresql.org⟩.
[36] Oracle RAC, available from ⟨http://www.oracle.com/technetwork/database/options/clustering/rac⟩.
[37] DB2 pureScale, available from ⟨http://www.ibm.com/developerworks/data/library/dmmag/pureScale⟩.
[38] Microsoft Windows Server Failover Cluster, available from ⟨http://www.microsoft.com/windowsserver2008/en/us/failover-clustering-main.aspx⟩.

付　　録

A.1 補題の証明

A.1.1 補題 1

証明. H^m をマスタデータベース R^m でコミットしたトランザクションのヒストリとする．もし，マスタデータベース R^m がスナップショット分離下で実行している場合，マスタデータベースのヒストリ H^m は 1 つも inter-ww-dependency を含まない．なぜならば，first-updater-wins ルールでは最初に更新したトランザクションだけがコミットでき，その他のトランザクションはアボートしなければならないからである．もし，最初に更新したトランザクションがアボートした場合，2 番目に更新したトランザクションがコミットでき，その他のトランザクションはアボートしなければならない．要するに，たった 1 つだけトランザクションがコミットできてその他のトランザクションはアボートしなければならない．したがって，マスタデータベースのヒストリ H^m は 1 つも inter-ww-dependency を含まない．それゆえに，スレーブデータベース R^s は inter-ww-dependency を再現実行する必要がない．　□

A.1.2　補題 2

証明. 最初の参照オペレーション以外の参照オペレーション $r_{i,p}$（$p > 1$）はデータベースの状態を得るためだけに必要である．これらの参照オペレーションはどのオペレーションともコンフリクトしない．また，データベースの状態を変更することもない．最初の参照オペレーション $r_{i,1}$ が作ったスナップショットを読むだけである．したがって，スレーブデータベース R^s は intra-wr-dependency を再現実行する必要はない．　□

A.1.3　補題 3

証明. 補題 1 と補題 2 から明らかである．　□

A.1.4　補題 4

証明. T_i は更新トランザクションであるため，$w_{i,p}(x_i) < c_i$ である．したがって，$w_{i,p}(x) < c_i < r_{j,1}(x_i)$．定義 1 より inter-wr-dependency である．　□

A.1.5　補題 5

証明. T_j は更新トランザクションなので，$r_{j,1}(x_k) < w_{i,q}(x_i) < c_i$ または $w_{i,q}(x_i) < r_{j,1}(x_k) < c_i$．もし，$i = j$ かつ $r_{j,1}(x_k) < w_{i,q}(x_i) < c_i$ の場合，$r_{i,1}(x_k) < w_{i,q}(x_i)$ となる．これは，トランザクション T_i と T_j は intra-rw-dependency である．もし $i = j$ かつ $w_{i,q}(x_i) < r_{j,1}(x_k) < c_i$ の場合，$w_{i,q}(x_i) < r_{j,1}(x_k)$ となる．これは，トランザクション T_i と T_j は intra-wr-dependency である．補題 2 より無視してよいケースである．もし $i \neq j$ かつ $r_{j,1}(x_k) < w_{i,q}(x_i) < c_i$ の場合，$r_{j,1}(x_k) < w_{i,q}(x_i)$ となる．これは，定義 1 よりトランザクション T_i と T_j は inter-rw-dependency である．もし $i \neq j$ かつ $w_{i,q}(x_i) < r_{j,1}(x_k) < c_i$ の場合，$w_{i,q}(x_i)$ の更新結果は c_i 以降に参照可能となる．しかし，$r_{j,1}(x_k) < c_i$ であり $r_{j,1}(x_k)$ は $w_{i,q}(x_i)$ の更新結果を参照できない．よって，$w_{i,q}(x_i) < r_{j,1}(x_k)$ は $r_{j,1}(x_k) < w_{i,q}(x_i)$ と等価である．これは，定義 1 より，トランザクション T_i と T_j は inter-rw-dependency である．以上より，intra-rw-dependency または inter-rw-dependency である．　□

A.1.6　補題 6

証明. もし $w_{i,p}(x_i) < w_{i,p+1}(x_i)$ の場合，定義 1 よりトランザクション T_i は intra-ww-dependency であることは明らかである．　□

A.2　LSIR の証明

証明. 補題 3 より，スレーブをマスタとコンシステントにするためには，inter-wr-dependency と inter-rw-dependency，intra-rw-dependency，intra-ww-dependency の 4 つの dependency を再現実行すればよい．ルール (1-a) の $c_i^m < r_{j,1}^m \in H^m \Rightarrow c_i^s < r_{j,1}^s \in S^s$ はスレーブのスケジューラ S^s が補題 4 よりマスタの inter-wr-dependency を再現実行していることを意味する．さらに，ルール (1-b) の $r_{j,1}^m < c_i^m \in H^m \Rightarrow r_{j,1}^s < c_i^s \in S^s$ はスレーブのスケジューラ S^s が補題 5 よりマスタの intra-rw-dependency または inter-rw-dependency を再現実行していることを意味する．さらに，ルール (2) の $w_{i,p}^m(x_i) < w_{i,p+1}^m(x_i) \in H^m \Rightarrow w_{i,p}^s(x_i) < w_{i,p+1}^s(x_i) \in S^s$ はスレーブのスケジューラ S^s が補題 6 よりマスタの intra-ww-dependency を再現実行していることを意味する．したがって，補題 3 より，LSIR はスレーブをマスタとコンシステントにすることができる．　□

A.3　Madeus の動作例

本章では Madeus がどのようにオペレーションを処理するかの具体例を示す．例として取り上げるトランザクションは $\mathcal{T} = \{T_i = r_{i,1}(x_p), w_{i,1}(x_i), c_i,\ T_j = r_{j,1}(y_q), w_{j,1}(y_j), c_j,\ T_k = r_{k,1}(x_i), w_{k,1}(x_k), c_k\}$ である．最初の MLC は 3 とする．**図 A·1** に *workers* の動作例を示す．$worker_i$ が最初の参照オペレーション $r_{i,1}(x_p)$ を受け取ったとき，$worker_i$ は現在の MLC を STS として SSB に保存すると同時に $r_{i,1}(x_p)$ も SSB へ保存する．$worker_j$ が最初の参照オペレーション $r_{j,1}(y_q)$ を受け取ったとき，$worker_j$ は現在の MLC を STS として SSB へ保存すると同時に $r_{j,1}(y_q)$ も SSB へ保存する．$worker_i$ が更新オペレーション $w_{i,1}(x_i)$ を受け取ったとき，$worker_i$ は $w_{i,1}(x_i)$ を SSB へ保存する．$worker_j$ が更新オペレーション $w_{j,1}(y_j)$ を受け取ったとき，$worker_j$ は $w_{j,1}(y_j)$ を SSB へ保存する．$worker_i$ がコミットオペレーション c_i を受け取ったとき，現在の MLC を ETS として SSB へ保存すると同時にコミットオペレーション c_i も SSB へ保存する．$worker_i$ は MLC を 1 増やして 4 とする．$worker_j$ がコミットオペレーション c_j を受け取ったとき，現在の MLC を ETS として SSB へ保存すると同時にコミットオペレーション c_j も SSB へ保存する．$worker_j$ は MLC を 1 増やして 5 とする．$worker_k$ が最初の参照オペレーション $r_{k,1}(x_i)$ を受け取ったとき，$worker_k$ は現在の MLC を STS として SSB へ保存すると同時に $r_{k,1}(x_i)$ も SSB へ保存する．$worker_k$ が更新オペレーション $w_{k,1}(x_k)$ を受け取って続いてコミットオペレーション c_k を受け取ったとき，それらを SSB へ保存する．$worker_k$ が現在の MLC を ETS として SSB へ保存すると同時に MLC を 1 増加させて 6 とする．**図 A·2** に現時点の SSL を示す．トランザクション T_i と T_j は並行に実行されているのに対してトランザクション T_k はトランザクション T_i と T_j がコミットしてから実行される．

図 A·3 に *conductor* と *players* の動作例を示す．*conductor* は SSL の SSB を検索して最も小さい STS が 3 であることから SLC を 3 にセットする．*conductor* は $player_i$

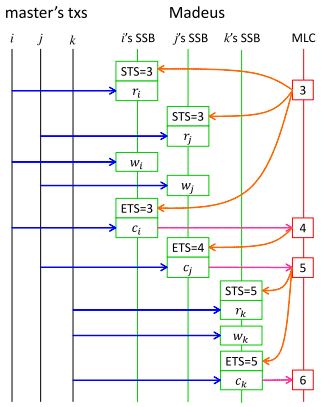

図 A·1　マスタのシーケンス
Fig. A·1　Master's sequence.

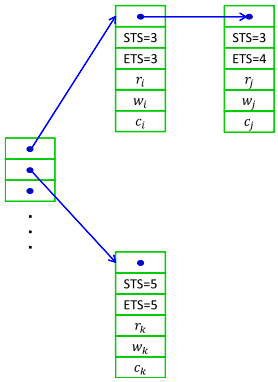

図 A·2　SSL の例
Fig. A·2　Example of SSL.

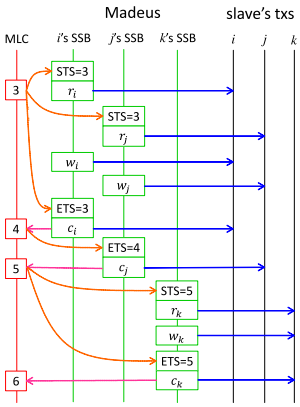

図 A·3　スレーブのシーケンス
Fig. A·3　Slave's sequence.

と $player_j$ に最初の参照オペレーション $r_{i,1}(x_p)$ と $r_{j,1}(y_q)$ を並行に転送するように指示する．続いて，$player_i$ と $player_j$ に更新オペレーション $w_{i,1}(x_i)$ と $w_{j,1}(y_j)$ を並行に転送するように指示する．現在の SLC は 3 であり次の SLC は 5 であることから CCN は 2 である．したがって，ETS が 3 と 4 のコミットオペレーションを並行に転送する．$player_i$ と $player_j$ はコミットオペレーション c_i と c_j を並行に転送する．コミットオペレーション c_i と c_j は同時に転送されるので，グループコミットの恩恵を受けることができる．$player_i$ と $player_j$ がそれぞれ SLC を 1 増加させるので，SLC は 5 になる．ここで $player_k$ の STS は 5 なので，$r_{k,1}(x_i)$ を転送する．次に $player_k$ は更新オペレーション $w_{k,1}(x_k)$ と続いてコミットオペレーション c_k を転送する．トランザクション T_i と T_j は並行に実行されることに注意されたい．また，グループコミットの恩恵を受けることができるという理由でコミットオペレーションを並行に転送していることにも注目したい．

A.4　ライブマイグレーションコンシステンシの証明

証明． *worker* は参照のみのトランザクションまたはアボートトランザクション実行中には SSB を作らない．*worker* は更新トランザクションの最初の参照オペレーションを受け取ると，*worker* はその最初の参照オペレーションを SSB に保存する（アルゴリズム 1 の 1–10 行目）．*worker* はその最初のオペレーション以外の参照オペレーションは捨てる（アルゴリズム 1 の 30–34 行目）．もし，*worker* が更新オペレーションを受け取ったら，*worker* は SSB へ保存する（アルゴリズム 1 の 11–15 行目）．したがって，*worker* はマッピング関数 $\mathcal{F}$ を提供する．Madeus はオペレーションの処理として次の 3 つの動作を行う．

(1) $c_i^m < r_{j,1}^m \in H^m$ の場合，MLC はコミットオペレーション c_i^m の後に 1 増えることから，$ETS_i < STS_j$ となる（アルゴリズム 1 の 21 行目）．*players* は最初の参照オペレーションとコミットオペレーションは小さい STS と ETS を持っているスレーブトランザクションから転送する（アルゴリズム 4 とアルゴリズム 5）．したがって，もしマスタデータベースで $c_i^m < r_{j,1}^m \in H^m$ ならば，スレーブデータベースで $c_i^s < r_{j,1}^s \in S^s$ である．

(2) $r_{j,1}^m < c_i^m \in H^m$ の場合，最初の参照オペレーション $r_{j,1}^m$ とコミットオペレーション c_i^m の間に他のコミットオペレーションがあるかないかで 2 つのケースが考えられる．もし，$r_{j,1}^m < c_k^m < c_i^m$ となるようなコミットオペレーション c_k^m がないならば，$STS_j = ETS_i$ となる．なぜならば，MLC はコミットオペレーション c_i^m が実行された後にのみ 1 増加するからである（アルゴリズム 1 の 21 行目）．マスタにおける最初の参照オペレーション $r_{j,1}^m$ とコミットオペレーション c_i^m はスレーブにおける最初の参照オペレーション $r_{j,1}^s$ とコミットオペレーション c_i^s にそれぞれ置換される（アルゴリズム 1 の 8 行目と 22 行目）．*conductor* は *player* に最初の参照オペレーション $r_{j,1}^s$ を転送させる（アルゴリズム 4 の 4 行目）．*conductor* は最初の参照オペレーション $r_{j,1}^s$ が実行されるのを確認する（アルゴリズム 4 の 5 行目）．その後，*conductor* は *player* がコミットオペレーション c_i^s を転送してよいと許可する（アルゴリズム 4 の 9 行目）．したがって，もしマスタで $r_{j,1}^m < c_i^m \in H^m$ であり，かつ，$r_{j,1}^m < c_k^m < c_i^m$ を満すコミットオペレーション c_k^m が存在しないならば，スレーブで $r_{j,1}^s < c_i^s \in S^s$ となる．

もし，$r_{j,1}^m < c_k^m < c_i^m$ なるコミットオペレーション c_k^m が存在するならば，MLC がコミットオペレーション c_k^m が実行された後で 1 増加するので，$STS_j < ETS_i$ となる（アルゴリズム 1 の 21 行目）．*players* は最初の参照オペレーションとコミットオペレーションを小さな STS と ETS を持つ SSB から転送していく（アルゴリズム 4 とアルゴリズム 5）．結果として，$r_{j,1}^m < c_i^m \in H^m$ かつ $r_{j,1}^m < c_k^m < c_i^m$ を満すコミットオペレーション c_k^m が存在するならば，$r_{j,1}^s < c_i^s \in S^s$ となる．

(3) SSB は FIFO キューであって *player* は順にオペレーションを転送するから，*player* は k 番目の更新オペレー

ションを $k+1$ 番目の更新オペレーションよりも先に転送する．したがって，$w^m_{i,k}(x_i) < w^m_{i,k+1}(x_i) \in H^m$ ならば，$w^s_{i,k}(x_i) < w^s_{i,k+1}(x_i) \in S^s$ となる．

Madeus は定義 3 で述べた LSIR に厳密に従ってオペレーションを処理する．定理 1 で示したように，もし，スレーブのスケジュールの動作が LSIR で決まるのであれば，スレーブはマスタにコンシステントである．したがって，Madeus はコンシステントなライブマイグレーションを実行する．

□

三島 健　（正会員）

1971 年生．1994 年筑波大学情報学類卒業．1996 年同大学大学院修士課程修了．同年 NTT 入社．2010 年東京大学大学院博士課程修了．博士（工学）．2010 年上林奨励賞授賞．日本データベース学会会員．

藤原 靖宏　（正会員）

2003 年早稲田大学大学院理工学研究科電気工学専攻修士課程修了．同年，日本電信電話株式会社入社．2011 年東京大学大学院情報理工学系研究科電子情報学専攻博士課程修了，2014 年ニューヨーク大学客員研究員．現在，日本電信電話株式会社 NTT ソフトウェアイノベーションセンタ特別研究員．博士（情報理工学）．グラフマイニングの研究開発に従事．第 27 回テレコムシステム技術賞，第 9 回上林奨励賞等受賞．電子情報通信学会，日本データベース学会各会員．

（担当編集委員　田村 孝之）