マイクロサービスを用いたフロー型データ構造の研究

◯森 毅（東京工業大学） 出口 弘

Study on flow type data structure by using microservice

∗T. Mori (Tokyo Institute of Technology) and H. Deguchi

概要– 多品種少量生産をメインとする日本の製造業において製造する製品や使用する機械などは頻繁に変わりうるものであり, 変更に合わせて生産管理のシステムも変更する必要が生じる. 今日の大規模システムの大半はリレーショナルデータベースが用いられているが, リレーショナルデータベースは ER 図と呼ばれるデータの関連を表した図に基づいて作成されたモデル図に沿って厳密に設計されるため, システム作成後の変更に弱く変更には多大なコストがかかることが問題としてあげられる. 本研究ではリレショナルデータベースを用いたシステムよりも変更に強いシステムとしてデータフロー型のシステムを作成し, 変更に対するコストやシステムデザイン, 開発手法についての比較・検討を行った.

キーワード: データフロー, マイクロサービス, FalconSeed, 疎結合

1 初めに

1.1 研究背景

コンピューターがビジネス利用され始めた当初, 企業の情報管理は各部門が独立したシステムで管理しており, 部門間でのデータのやりとりは行われて居なかった. しかし, コンピュータの性能向上・普及や「クライアント・サーバ型」環境の登場によりシステムのデザインは大きく変わり, 情報をはじめとし, 金, 人などの企業の資源を一元管理するシステムである ERP が普及し始めた. ERP 以外にも多くの大規模システムが現在でも利用されているが, それら大規模システムの大半はリレーショナルデータベースと呼ばれるデータベースを用いて全ての機能を一つのアプリケーションとして実行されるモノリシックなデザインで作成されている. リレーショナルデータベースはモデルの平明性や非手続き性の保証などの性質があげられるが, その一方でスキーマ (データベースの構造) の定義をデータベース作成前に厳密に行う必要があり, スキーマの定義・変更にかかるコストが問題としてあげられている.

また, 製造業は日本の企業の中でも特に IT 化が進んでおらず,ERP パッケージの導入率も低いが, それらの要因として前述したリレーショナルデータベースを用いたシステムの開発, 管理費用が高いことがあげられる. 製造業は他の業種に比べ作業工程が複雑なため, 生産管理のシステムを設計する際のスキーマ定義が難しく,ERP の初期開発費用がかかってしまう. また, 日本の製造業の大半は多品種少量生産と呼ばれる高付加価値の商品を少量生産する生産方式をとっているおり, 製造する製品や生産ライン, 使用する機械まで頻繁に変わるため, そのたびにスキーマの変更やシステムの改変を行うと変更に弱いモノリシックなシステムでは莫大なコストが生じてしまう. 日本の GDP の約 2 割を占める [1] 製造業において IT 化が行われていないことは大きな問題であり, この問題を解決するような変更に強いデータ構造・システムデザインが普及することで製造業の生産管理はより効率的に行われるようになり, 日本の製造業の競争力の増加に繋がると考える. また, 製造業のみならず, 今日日本にも普及されつつあるシェアリングエコノミーや座組み型のビジネスなどの会社に縛られないビジネスモデルやネットビジネスなどの刻々と変わりうるビジネスモデルのシステムなどにも広く利用可能だと考える.

1.2 研究目的

業務工程が複雑で頻繁に変化しうるような仕組みの管理に用いるシステムに適するデータ管理手法としてデータフロー型のデータ構造を提案する. それに共ない, 変更につよいシステムデザインや適した開発手法に求められる要件を明らかにする.

1.3 本論文の構成

本論文では, まず従来使用されてきた, また現在研究が行われているデータ構造やシステムデザイン, 開発手法についてそれらの長所や問題点, 適するシステムについて検討する. その後新しいデータ構造としてデータフロー型のデータ構造を導入し, 従来のデータ構造の問題をどのように解決出来るのか, また新しいシステムデザインや開発手法との適応性について述べる. 実際にリレーショナルデータベースを用いた従来のシステムとデータフローを用いた同程度のシステムの両方を作成し, それぞれのデータ構造を変更した際の変更にかかったコストの計測, 新たなデータ構造の優位性を評価する. またそれらの結果から変更に強いシステムに求められる要件について明らかにする.

第15回社会システム部会研究会, 第59回システム工学部会研究会 (2018年3月14日-16日・沖縄) PG0002/18/0000-0107 © 2018 SICE

2　先行手法

2.1　データ管理手法

2.1.1　リレーショナルデータベース

リレーショナルデータベースは 1970 年代に Codd[2] により提案されたリレーショナルデータモデルに基づいて設計, 開発されるデータベースである. リレーショナルデータベースが普及する以前はネットワークデータモデルやはいアラキカルデータモデルなどのデータモデルを元にデータベースが開発されていたが, リレーショナルデータベースは数学的にフォーマルな発想からデータベースを作成しており, モデルの平明性やデータの非手続き性に優れたデータベースであったため, 今日まで一般的なソフトウェアから大規模システムまでほとんどがリレーショナルデータベースを用いて開発されている.

系	類
数理・計算科学系	1 類
情報工学系	5 類

コース	系
数理・計算科学コース	数理・計算科学系
知能情報コース	数理・計算科学系
知能情報コース	情報工学系
情報工学コース	情報工学系

研究室	コース
研究室 A	数理計算科学コース
研究室 B	知能情報コース
研究室 C	知能情報コース
研究室 D	情報工学コース

Table 1: リレーショナルデータモデル

長年システムに用いられてきたため, リレーショナルデータベースの管理システム (RDBMS) についても時代と共に改良されており, その中でもデータベース RDBMS のトランザクション処理は特に有用性の高いものである. トランザクションとは「データベースに対するアプリケーションプログラムレベルでの 1 つの原始的な作用」のことを指し, SQL での複数の命令が合わさってアプリケーションレベルで意味のあるデータベースへの作用があるとして, それを適切に処理するための仕組みがトランザクション処理である. データベースに複数人が同時にアクセスし, 処理を行うと一貫性が失われる恐れがあるので RDBMS では一人がデータベースにアクセスしている際に他の人がデータベースの該当行, または該当テーブルのデータを変更することを禁止する仕組みが導入されている. これは金融システムなど処理の一貫性を厳密に満たす必要があるシステムに置いて必須の技術であり, このような一貫性の保証のためにリレーショナルデータベースを用いる場合も少なくないが, この仕組み自体はリレーショナルデータベース本来の有用性とは異なる. また, トランザクション処理によってリレーショナルデータベースでは更新頻度が高いと処理性能が落ちるという問題点があり, 同様の理由からスケーラビリティに適さない.

コスト面について, リレーショナルデータベースは数学的にフォーマルな発想でデータスキーマを設計するため設計にコストがかかる. さらに初めに定めたモデルに不都合が生じると変更をする必要があるが, 変更についても多大なコストが必要になるためスキーマ作成の段階で変更の余地が内容なモデルを決定する必要がある.

これらの特徴から銀行や交通機関などの業務の流れが頻繁に変わらず, 厳密なトランザクション処理が求められるような業種において適したデータ構造と言えるが, 変更に対する柔軟性は低いため頻繁にスキーマが変わるような業種に置いてリレーショナルデータベースを用いるのは適切ではない.

2.1.2　NoSQL

NoSQL とは 2000 年代に提案された次世代のデータベースのことである. Not only SQL の略であり, その定義は「非リレーショナル, 分散, オープンソース, 水平スケーラビリティに取り組む次世代のデータベース」[3] である. NoSQL の特徴として

- 明示的なスキーマを必要としない
- join 操作を必要としない
- 水平スケーラビリティ (スケールアウト) が確保されている
- 単一障害点がない
- ACID を必ずしも要求しない
- 大量データ, 分散処理に向いている
- 何らかの処理に特化している

ことがあげられ [4], リレーショナルデータベースで用いられてきた厳密な一貫性を捨て結果整合性を保証することでスケーラビリティに適したデータベースとなっている. NoSQL の例としてはオープンソースデータベースである MongoDB や Amazon 社で利用されている Amazon Dynamo,Google 社で利用している Google Bigtable などがあげられる. ビッグデータの処理などスケールアウトに適したデータベースであるが変更への柔軟性について強いデータベースは未だ普及されておらず,NoSQL についての研究は活発に行われている.

2.1.3　データウェアハウス

データウェアハウスとは企業の複数の基幹システムに蓄積されたデータを統合し, データサイエンスや意思決定などに再利用するための統合データベースのことをいう. それぞれの基幹システムは生産管理や売上管

理など業務における処理を効率化する目的で利用されており, それぞれのデータはそれぞれのシステムで使用しているデータベースに格納されている．それらの独立したデータ同士を統合することでデータ分析の幅が広がるなどのメリットがあげられる．データウェアハウスについてもリレーショナルモデルを用いたリレーショナルデータウェアハウスが用いられることが多いがリレーショナルデータベースで挙げたスキーマ定義のコストが問題である．複数のシステムで用いられているデータベースを統合する際に同じ対象を表すラベルがデータベースによって異なっていると統合ができないためデータウェアハウス専用のラベルを作成するなどして対応する必要があるが元のデータベースのスキーマ変更やデータウェアハウスのスキーマ変更に膨大なコストが生じてしまうなどの問題があげられる．

2.1.4 データレイク

データレイク[5]はデータウェアハウスと同様データ分析に用いられるデータの管理手法である．データウェアハウスは時系列に貯められたデータを構造化して蓄積していくものであるが, データレイクは構造化せずに保管し, 必要なデータを抽出する際にスキーマを定義する遅延バインドを採用している. これにより, 分析の必要が生じた際にそれに合わせたデータモデルを作成して必要なデータを抽出することができ, データウェアハウスの問題として挙げられていたスキーマ変更にかかるコストがデータレイクでは抑えることができる．IoT などでセンサーを用いて集められたスケーラブルなビッグデータを活用するにはスキーマレスなデータレイクの方が適しており, 導入コストもデータウェアハウスと比べて 10 100 倍縮小されると言われている．

2.2 システムアーキテクチャデザイン

2.2.1 モノリシックデザインアーキテクチャ

モノリシックデザインアーキテクチャは複数の機能を一つの大きなモジュールとして実装するシステムのデザインである．直接データを受け渡ししないような機能同士も一つにまとめられているためシステムの一部を変更しようとしてた際に全体に影響がないかどうかを確認した上で全体をデプロイする必要がある．拡張についても同様であり, このような一つにまとまったシステムの中で変更を繰り返していくと変更に連れてモジュール構造が曖昧になりより変更にかかるコストが大きくなるという悪循環が生じてしまう．また, 複数ある機能のうち一部の機能の需要が大きくなった際にモノリシックデザインの場合はシステム全体をスーケルする必要があり, 一部の機能のみを必要な分スケールするというようなリソース管理ができない．

2.2.2 マイクロサービス

マイクロサービスとは 2011 年にベネチア近郊で行われたソフトウェアアーキテクトのワークショップで議論され、2014 年に Jemes Lewis によって提唱されたソフトウェアの開発手法である.[6] マイクロサービスアーキテクチャスタイルは単一のアプリケーションを小さなサービス群の組み合わせとして構築する手法であり, ここのサービスは独立して動作し, サービス同士の通信は HTTP のリソース API などの軽量の機構により行われるというものである．マイクロサービスにより, 従来のモノリシックなサービスとは異なりアプリケーションの小さな部分の変更に対してプロジェクト全体の工程や予算の承認を得て全体のデプロイを行う必要がなくなる．

マイクロサービスを用いたシステムとして IBM 社が開発した Node-RED が挙げられる．

Node-RED　Node-RED は Node.js で動くオープンソースソフトウェアであり，「ハードウェアデバイス,API, オンサインサービスを新しく興味深い方法で結びつけるプログラミングツール」である.[7] サイドメニューから使用するノードを選択しパレット内にドロップすることで使用するノードを追加していく．ノードにはセンサーやアクチュエーターなどの IoT で用いられるエッジデバイスから受け取ったデータの処理方法など様々なもが定義されており, 複雑な処理などは JavaScript で自作のノードを作成し使用することが出来る．パレット上でノード間を結びつけてフローを作成することで一連のデータ処理を定義し Node-RED のウィンドウでコーディングをほとんど行うことなしにデプロイや実行が可能である．元々 Node-RED は IoT システムの作成をサポートするツールとして開発されており, IoT における通信プロトコルとしてメジャーな MQTT をサポートしているという特徴がある．また, フローベースのプログラミングでそれぞれのノードが独立して存在するものをパレット上で結びつけることで一連の処理を定義しているため, あるノードを変更する必要が生じた際に入力と出力さえ以前のものと合わせればフロー全体に影響がないことが保証されている．IBM では IoT センサーシミュレータというセンサーからのデータ送信をブラウザでシミュレーションするツールをサポートしており, これにより IoT のエッジデバイス側のアプリケーションを作成する前にシミュレータで挙動を確認し, その後シミュレータが配置されていた箇所にアプリケーションを作成したエッジデバイスを置き換えるという柔軟な開発が可能になる．

Node-RED はノードというマイクロサービスをクラウド側で管理し, エッジデバイスからの入力ををクラウドでフローを元に処理しエッジデバイスに出力するという流れになっているがクラウド側で管理を行うと開発がトップダウンになってしまい, 製造業などの現場の知識が活かされないこと, クラウド側の処理性能によってスケーラビリティが制限されることが問題として挙げられる．

Real World Operating System　Real World Operating System(以下リアルワールド OS) とは「実世界でネットワークで結ばれた, 人・もの・こと (ソフトエージェント) が自律分散協調的に, 機能を遂行することを, その設計・実装・管理の全フェーズに渡って支援する, 実世界のエージェント・プロセスに対するオペレーティング・システム」[8] のことであり, 東京工業大学情報理工学院教授出口により提唱された. コンピュータのオペレーティングシステムがプロセスを管理するように, リアルワールド OS は実世界の人, もの, ことをプロセスとみなし, それらの状態や通信を管理することが出来る. リアルワールド OS や Node-RED の他にも CISCO 社の IOx や Amazon の AWSIoT など様々な IoT デバイスを管理する環境が開発されていますが, リアルワールド OS は「自律分散的である」という点で他の環境と異なる. 「RWOS 以外のシステムは全て, クラウドや SQL データベースに代表される中央のサーバーに一度全てのデータを集中させ, そこで計算, 司令を行うという中央集約的な処理が前提」であるが, リアルワールド OS はコンポーネント化された個々のエッジデバイスが必要なデータをエッジデバイスに直接送信し, それぞれのエッジデバイスが必要な処理を行うという中央での管理を必要としないストリーム型のデータ処理になっている. これにより Node-RED で述べたノードの変更への柔軟性の他に,IoT システムのスケーラビリティへの頑健性や工場の現場のような末端での改善や変更への柔軟性などの特徴があげられる. 日本の製造業などのサプライチェーンでは, 現場での改善や変更による生産性の工場が大き強みとなっており, これらの強みを活かした IoT システムの開発にはリアルワールド OS が適している.

2.3　システム開発手法

2.3.1　ウォーターフォール型開発

ウォーターフォール型開発とは要件定義→基本設計→詳細設計→製造 (プログラミング) →単体テスト→結合テスト→総合テスト→リリース→運用・保守という開発の流れを固定して開発し, 前の工程に戻らないような開発手法である. ウォータフォール型では開発中の仕様変更などを考慮しておらず, 仕様変更に対して前工程に戻る必要が生じたり長期間の開発に対してスケジュール通りに進むことを前提にしているため仕様変更が起こった際に開発が頓挫する可能性があるなどの問題がある.

2.3.2　アジャイル型開発

アジャイル型開発とはウォーターフォール型開発の代替手法として考案された開発手法であり, アジャイルの意味の通り素早い開発に焦点を当てており, 短い期間で設計→実装→テストを繰り替えすことで優先度の高い機能から実装し, 開発期間における仕様変更についても対応出来るような開発手法が例として挙げられる. スクラム開発やエクストリームプログラミングがアジャイル型開発として知られており, それぞれシステムの分割方法や開発状況の共有, コーディングの方法やリファクタリングなど細かなプラクティスが異なっているがアジャイル型開発は運用方法には言及しておらず, あくまで迅速な開発を目的としているため, 開発部門と運用部門の間に起こる機能の追加と可用性の対立に対して解決作を与えない.

2.3.3　DevOps

上記二つの開発手法はシステムの開発のみについての手法であったが,DevOps とは機能の追加等の価値を素早くリリースすることを目標としている. DevOps はアジャイル型開発の中で生まれ, 開発チームと運用チームの関係を密にすることでシステムの機能の追加とシステムの可用性というそれぞれ対立していた立場を機能の価値の工場を素早くユーザーに届けるという一つの目標に一致させた.

2.4　変更にかかる工数の計測手法

2.4.1　ファンクションポイント法

ファンクションポイント法とはソフトウェアに実装される機能の複雑さを基準に重み付けしてポイントを設定し, システムの全ての機能のポイントから開発されるシステムの規模や開発にかかるコストを予測する手法のことである. 現在では IFPUG[9] と呼ばれる国際団体が標準化した IFPUG 法が広く用いられている. IFPUG 法ではシステムのあるアプリケーションについてアプリケーションがアクセスする内部論理ファイルと外部インターフェースの数から計測されるデータファンクションと外部からのデータ入力, 外部への出力, 外部からの照会などの情報の出し入れの機能数から計測されるトランザクションファンクションの二種類から求められるファンクションポイントを基準に計算される. しかしファンクションポイント法が予測するのはあくまでシステムの規模であり, それ以外の要素である使用言語などの手法の差や変更に対する柔軟性が評価出来ないため手法ごとの変更の柔軟性を評価するための手法として適切ではない.

3　データフロー型開発手法

従来のデータ管理手法では変更に弱いという問題に対する新しいデータ構造としてデータフロー型のデータ構造を提案する. データフロー型のデータ構造とは互いに疎結合なデータ同士を加工するサービスの組み合わせで新しいデータに加工し, 上流からデータの加工を連続して行っていき, その一連の流れそのものをデータ構造とみなすデータ構造である.

3.1 FalconSeed

データフロー型のデータ構造として本研究では FalconSeed を用いる．FalconSeed は出口研究室で開発されたデータ加工ツールであり, ビッグデータからデータの抽出・削除等の加工の操作単位をフィルタとして定義し, フィルタを組み合わせたものを一つの大きなフィルタとして再定義するマクロフィルタという昨日も存在する．FalconSeed が対応しているデータの形式は csv,xml, 交換代数がある．

交換代数　交換代数というのは出口 (2000)10) により経済的な財の交換を公理化した代数系である．交換代数は value¡name,unit,time,subject¿と言う形式で, 財の名前, 単位, 時間, 主題を基底に持つ．交換代数を用いることで複式簿記の形式で与えられる取引を代数系で記述することができ, 複式簿記のような金額ベースの取引記述だけでなく実物の取引や実物と金額の振替などが可能になる．交換代数の特徴としてマイナスの数が用いられていない (冗長代数) ことがあげられる．現実世界で本を 100 円で買った, と言うイベントは

$x = 100$ <本, 円,*,*>+ 100^<現金, 円,*,*>(*はワイルドカード)

と言うように記述される．現金が 100 円出て行くという事実は 100^<現金, 円,*,*> というように表現されている．

現実において,100 円の売上原価で 120 円を売り上げることと 10,000 円の売上原価で 10,020 円を売り上げることは同じ利益 20 円を発生させるイベントだとしても内容が異なるが, 交換代数を用いることでこの二つのイベントを明確に区別することができ, 自然な処理が可能になる．この冗長性を除去するには ˜(bar) 演算と言うものが用いられ,

$x = 120$<現金, 円,*,*>+ 100^<現金, 円,*,*>

に bar 演算を行うことで

$\tilde{x} = 20$<現金, 円,*,*>

と言うように冗長性が除去される．交換代数という材の交換を公理化したことで, 会計データを自然に処理することが可能になりアドホックな実装を避けることができる．FalconSeed のフィルタは Java をベースとして交換代数の計算を可能とした AADL(Algebraic Accounting Description Language) を用いて実装される．

3.2 データフロー

従来のデータ管理手法ではデータは 1 つのデータベースに集められていたがデータフローによるデータの管理では各デバイスで必要なデータをデバイス同士で直接受けわたし, 処理を行う．リアルワールド OS ではデバイスが自立分散的に動作しており, 自立分散的なデバイス同士のデータの送受信も自立分散的に行われる方が自然である．FalconSeed は csv,xml, 交換代数に対応しているがこれは企業で用いられるデータが会計データと csv のような表データが大半を占めているためである．FalconSeed では会計データと表データをそれぞれ交換代数, データ代数としてそれらの基本的な演算をフィルターとして定義した．データフローのデータ構造は生産管理を行うデバイスで扱うデータを直接やりとりするものであり, データの一元管理のためなど, アーカイブとしてデバイスで処理をしたデータをデータベースに集約することはデータフローの理念とは反さない．マイクロサービスであるデータ処理の組み合わせとしてデータの流れを定義しているので各データを入れ替える際にも入れ替える部分のデータのみをデプロイすればよく, データ処理についても該当データと直接送受信を行っているデバイスの処理のみを考慮すればよい．

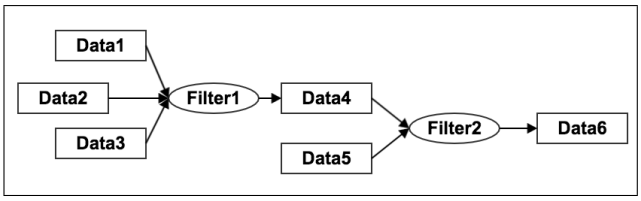

Fig. 1: データフロー図

Fig.1 において既存のデータと新しいデータを合わせて新しいデータを出力したい場合, データフローの該当箇所にそのままノードとフィルターを追加する．追加した図は Fig.2 のようになり,Filter3 で Data4 と Data7 を受け取って Data8 を出力するデータ処理を行ってもデータ構造の他の部分には影響がないことは明らかである．

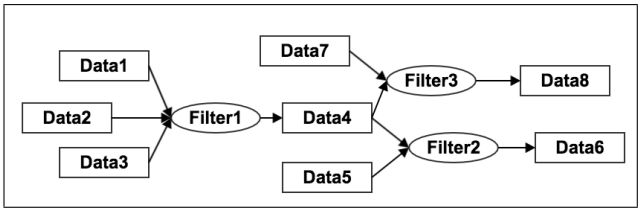

Fig. 2: データを追加したフロー図

3.3 システムデザインと開発手法

システムはエッジデバイスごとにコンポーネント化されたシステムになり, 各デバイスがマイクロサービスとして独立して動作する．企業の業務の流れをワークフローの形で記述することで業務の最適化を行うビジネスプロセスリエンジニアリングというものがあるが, データフロー型のシステムデザインにおいてもビジネスの流れをワークフローとして定義し, ワークフローにそってシステムを開発していく．は製造業における製造プロセスの一例であるが, 前述したように各システムは自身のタスクにおけるデータの取得や処理を独立して行い, 一連の流れで必要なデータ処理が行われ, 全体のデバイスの管理をリアルワールド OS が監視する．交換代数は振替テーブルと呼ばれる振替処理の内容を定義したデータを元に電気代を損益に振替たり実物のデータとして格納されていたデータを金額に振替たり

などの処理を行うが csv 形式で格納されている振替処理の定義を変えるだけでシステムの変更が可能な宣言型プログラミングに近い仕組みになっているためコードを変更する必要がなく現場知にとる絶えざる変更が可能になる.

3.4　従来手法との比較

従来手法と提案手法の差を以下に記す.

	従来手法	提案手法
構成要素	データベース,Web サーバー, サーバーにアクセスするためのデバイス等	データの処理, 送受信を行うデバイス
特徴	システム利用者全体が一体型のシステムにアクセスし, データベースにより一元管理されているデータを取り出しサーバー側で必要な処理をおこない, データを入手する	各部門で使用するデータを保管しており, 必要な処理を行いたい場合, 必要なデータを保有する部門からデータを受け取り, 処理を行う.
変更に対する処理	スキーマの変更と変更に伴うサーバー側の処理の変更を管理者がシステムを利用する各部門との調整をしながら行う	変更に影響を受ける部門で調整を行い, フィルターの追加や変更で対応する.
メリット	多くの事例が存在するため新たに考える必要が無い, トランザクション処理などが RDBMS に最初から　実装されている	全体の調整を必要としない, システムの管理者を介せず変更を行うことが出来る, フィルターの組み替えで　処理の変更が実装出来るため, 最低限のコーディングで実装が可能.
デメリット	小さな変更であっても全体の調整が必要, システムの変更が管理者を通してのみ可能	

Table 2: 手法別の開発スキーム

工場の現場知による変更を可能にするには小さな変更を部門内で行えるようなシステムが必要であり, データフロー型のシステムはそれに適していると言える.

4　システム事例

本研究では同程度のシステムをリレーショナルデータベースと交換代数を用いたデータフロー型のデータ構造で作成し, データ構造を変更する際のコストの計測や開発におけるデータフロー型の優位性の実証, 変更に強いシステムに必要な要件について検討する. また, ビジネスのデータ処理において交換代数を用いることの優位性について検討するため交換代数を用いないデータフロー型のシステムも同時に作成し, 比較を行う.

4.1　事例のシステムモデル

各開発手法を用いたシステムの事例として「レストランの予約・在庫管理」を行うシステムを作成した. 各システムでは「予約状況・在庫状況・売上」をビューとしで閲覧できる機能を実装する. 実際に製造業に導入してコストを計測することが望ましいが, 今回は製造業と似たようなフローでデータ処理を行う小規模なシステムを経営部門や経理部門, 生産部門など複数の部門が使用することを想定し, データ構造を変更した際の変更にかかるコストを検討する. 複雑でないデータ構造を用いた小規模なシステムにおいても優位性が現れれば大規模システムにおける優位性になるのは明確である. 変更の内容としては「予約の人数分同じメニューで固定していたものを予約の個人でメニューを選べるようにする」と言う内容の変更を行った.

4.2　交換代数を用いたデータフロー型のシステム

4.2.1　使用データ

作成するシステムのデータフローは以下の通りである.　各データの説明を以下に記す

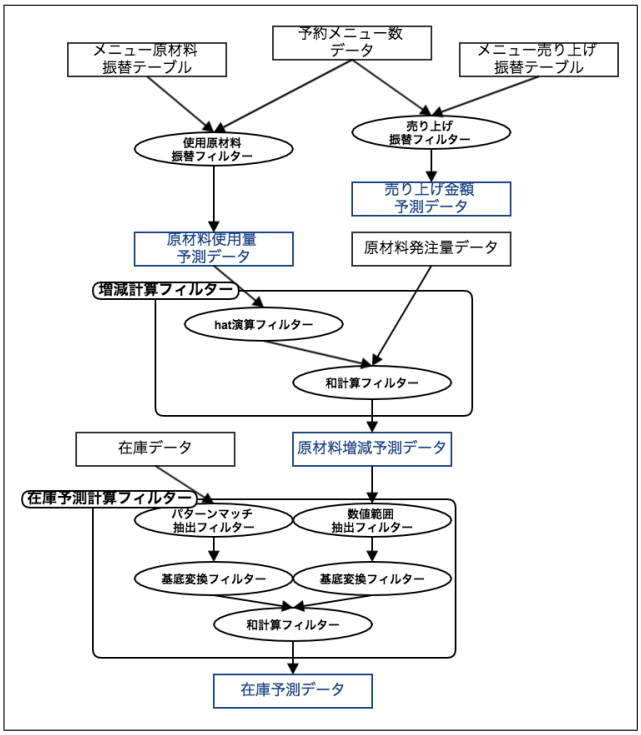

Fig. 3: データフロー図

- 予約メニュー数データ
  予約数<メニュー名, 品, 予約日, 予約 ID>
  日にち毎のメニュー数を格納するデータ. 予約 ID は予約を識別するために主題の基底に格納されているが今回のシステムではその後のフローで使用しないためワイルドカードに振替える処理をフローの始めで行っている.
- 売り上げ金額データ
  売上額<メニュー名, 円, 売上予定日,*>
  日にち, メニュー毎の売り上げ金額を格納するデータ.
- 原材料使用量予測データ
  使用量<原材料名, 使用単位, 使用予定日,*>
  日にち毎の各原材料の使用予測量を格納するデータ.
- 原材料発注量データ
  発注量<原材料名, 発注単位, 在庫追加日,*>
  原材料の発注量を格納するデータ.
- 原材料増減予測データ
  使用量<原材料名, 使用単位, 使用予定日,*>
  原材料使用量予測データと原材料発注量データを元に日にち, 原材料毎の在庫の増減予測量を格納するデータ.
- 在庫データ
  在庫量<原材料名, 在庫単位, 計測日,*>
  原材料毎の在庫量を格納するデータ.
- 在庫予測データ
  在庫予測量<原材料名, 在庫単位, 予測対象日,*>
  原材料毎の対象の日付の在庫の予測量を格納するデータ.
- メニュー売り上げ振替テーブル
  実物としてのメニューの勘定科目を金額としての勘定科目に振り替える定義を格納するデータ予約数<メニュー名, 品, 予約日,*>という基底の元を売上額<メニュー名, 円, 売上予定日,*>　という基底に振り替える.
- メニュー原材料振替テーブル
  メニューを貸方, 原材料を借方としてメニュー一品を原材料に戻す振り替え作業の定義を格納するデータ. (本来の製造業においては借方に製品 (メニュー), 貸方に原料で製造作業を仕分けるが今回はメニューの予約を原材料に分解しているため借方貸方が逆になっている)

4.2.2　各フィルターの説明

各フィルターの説明を以下に記す.

- 使用原材料振替フィルター
  予約メニュー数のデータをメニュー原材料振替テーブルの定義を元に原材料使用量のデータに振替るフィルター. 例としてメニューの一つであるハムサラダを原材料に振り替える仕訳を以下に示す.

借方	値	単位	貸方	値	単位
ハム	50	g	ハムサラダ	1	品
レタス	0.5	個			
トマト	1	個			
チーズ	50	g			

- 売り上げ振替フィルター
  予約メニュー数のデータをメニュー売り上げ振替フィルターの定義を元に売り上げ金額のデータに振替えるフィルター. 例としてハムサラダ 1 品を金額に振替える仕訳を以下に示す.

借方	値	単位	貸方	値	単位
ハムサラダ	1000	円	ハムサラダ	1	品

- 増減計算フィルター
  原材料使用量データと原材料発注量データの和から日付毎に原材料がどれだけ増減するかどうかを計算するフィルター. 原材料の使用は在庫量を減らすイベントであり, 発注は在庫量を増やすイベントのため総和を計算する前に原材料使用量データに hat 演算を用いて, その出力結果と発注量データの和 (足し合わせた後に bar 演算で冗長性を除去した結果) を出力するフィルターの組み合わせを増減計算を行うマクロフィルターとして定義した.

- 在庫予測計算フィルター
  在庫データから当日の在庫量を抽出し, 当日から在庫を知りたい日付までの期間の在庫の増減予測量を原材料増減予測データから抽出し, それらの和から指定した日付の在庫予測データを計算するフィルター. 在庫データ一覧からパターンマッチによって当日の在庫量を抽出し, 同様に原材料増減予測データから数値範囲抽出フィルターを用いて当日から知りたい期間までの増減予測量を抽出する. それらの交換代数元の基底の日付はそれぞれの日付になっているので, 在庫を知りたい日付に基底を変更する. その後二つのデータを計算することで在庫を知りたい日付の在庫データを交換代数形式で出力する.

4.3　交換代数を用いないデータフロー型のシステム

交換代数を用いないデータフロー型のシステムのデータフローは Fig.4 のようになる.

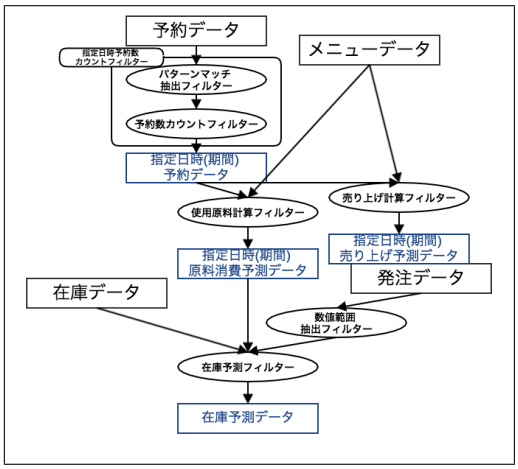

Fig. 4: データフロー図

4.3.1 使用データ

使用するデータの説明を以下に記す.

- 予約データ
  予約 ID, 予約の人数と日時, 注文メニューを格納するデータである.

予約 ID	人数	日時	メニュー
Y001	2	171130	ハムサラダ, ナポリタン, パンナコッタ
Y002	3	171204	タコサラダ, ペペロンチーノ, ティラミス

- メニューデータ
  各メニューのメニュー名, 単位, 原材料の情報を格納するデータである. 使用する原材料は各メニューの行のメニュー名, 単価の列に続いて [原料名, 値, 単位] のセットで格納されている.

メニュー名	単価	原料名	値	単位	原料名	値	単位	…
ハムサラダ	1000	ハム	50	g	レタス	0.5	個	…
タコサラダ	1200	タコ	50	g	セロリ	50	g	…

- 指定日時 (期間) 予約データ
  指定した日時 (期間) のメニューごとの予約数を格納するデータである.

メニュー名	予約数
ハムサラダ	5
タコサラダ	10

- 指定日時 (期間) 売り上げデータ
  指定した日時 (期間) のメニュー毎の売り上げ予測金額を格納するデータである.

メニュー名	売上金額
ハムサラダ	5000
タコサラダ	12000

- 指定日時 (期間) 原料消費データ
  指定した日時 (期間) の原料毎の消費量を格納するデータである.

値	原料名	単位
500	ハム	g
5	レタス	個
…	…	…

- 在庫データ
  現在の在庫を原料消費データと同じ形式で格納するデータである.
- 発注データ
  発注した原料のデータを原料消費データに日付の列を追加した形式で格納するデータである.

値	原料名	単位	発注日時
2000	ハム	g	171204
10	レタス	個	171207

- 在庫予測データ
  指定した日時の在庫を原料消費デーア t と同じ形式で格納するデータである.

4.3.2 各フィルターの説明

- 指定日時予約数カウントフィルター
  予約データからパターンマッチ抽出フィルターを用いて指定した日付の予約データを抽出する. その後抽出した予約データから各メニューの出現数にその予約の人数を掛け合わせた数を足していくことでメニュー毎の予約数を計算し, その結果を出力する.
- 使用原料計算フィルター
  メニューデータに格納されている各メニューの一人前あたりの使用原材料にそのメニューの予約数を掛け合わせ, 原料毎に足し合わせた結果を出力する.
- 売り上げ計算フィルター
  メニューデータに格納されている各メニューの単価にそのメニューの予約数を掛け合わせた結果を出力する.
- 在庫予測フィルター
  各原料について現在の在庫データと現在から指定した日付までの期間の発注データを足し合わせたものから現在から指定した日付までの期間の原料の消費予測データを差し引くことで指定した日付の在庫予測を計算し, 出力する.

4.4 リレーショナルデータベースを用いたシステム

Tomcat と Eclipse を用いてリレーショナルデータベールを用いて作成する Web アプリケーションを作成する.

4.4.1 データスキーマ

リレーショナルデータベースの ER 図を Fig.5 に示す.

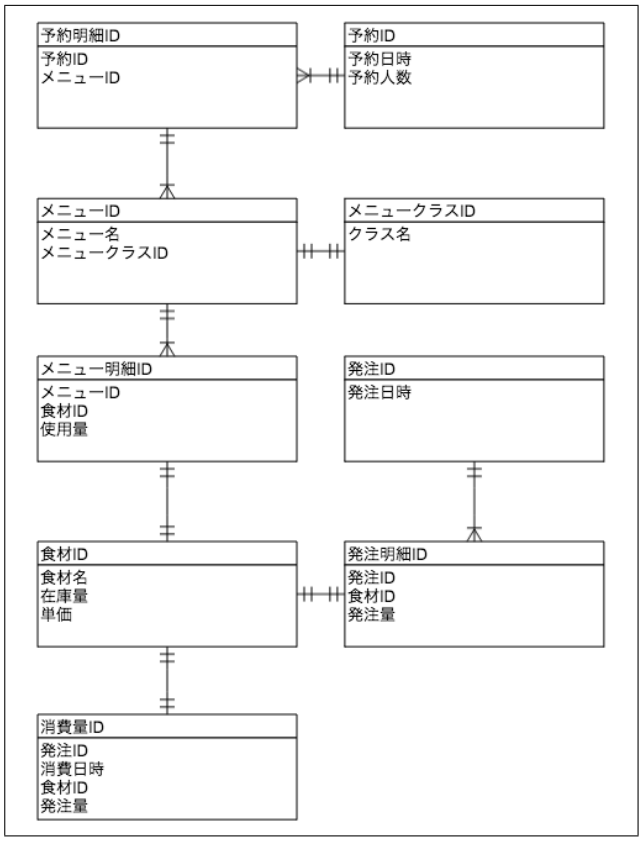

Fig. 5: ER 図

4.4.2 データの閲覧

データの閲覧の際にサーバー側で行う処理を以下に記す.

- 予約データの閲覧予約データは日にちや期間を指定することで予約テーブルから指定した期間の予約一覧データを抽出する SQL 文をリレーショナルデータベースに送り, 結果を jsp ページで受け取って表示する.
- 在庫データの閲覧在庫データは当日の在庫データは食材テーブルに格納されている各食材の在庫量から, 翌日以降の在庫データについては当日から指定した日付までの発注明細テーブルから抽出した発注量と消費量テーブルから抽出した消費量を元に各食材の増減予測を計算し, 結果を jsp ページで表示する.
- 売り上げデータの閲覧予約テーブルとメニュー詳細テーブルから指定した日付や期間のメニュー毎の売り上げ予測量を抽出する SQL 文を作成しリレーショナルデータベースから送られてきた結果を jsp ページで受け取って表示する.

5 手法の比較・考察

FalconSeed とリレーショナルデータベースにおけるデータ構造の変更点はそれぞれ以下の通りである.

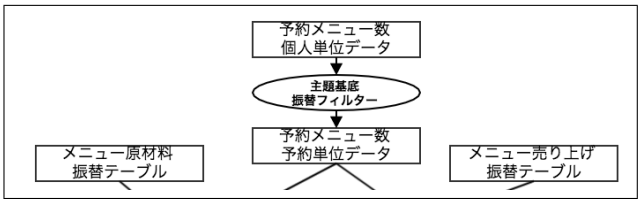

Fig. 6: データフロー図の変更箇所

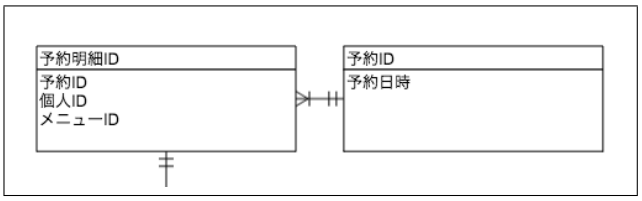

Fig. 7: ER 図の変更箇所

リレーショナルデータベースでは変更に対して ER 図を再度作成し直し, 今回であれば影響がある予約の閲覧機能に対して SQL 文を作成する部分のコードを変更した. 交換代数を用いたデータフロー型のデータ構造では主題の基底に予約 ID と個人 ID を格納する形に変更し, 規定の振替フィルターを用いて個人単位の予約メニューから予約単位の予約メニュー数の計算を行った. また, 予約単位の予約メニュー数を閲覧する部門があるとして, データ構造の変更ののち同じデータを表示できるようにデータフローにフィルターを挿入したが, 会計データを扱う中で今回の変更は対処を必要とするものではなく, 個人単位の予約メニューのまま予約メニュー数データに差し替えてもその後のフローで問題は生じない. 交換代数を用いないデータフロー型のデータ構造ではデータフロー図の変更はなかったが予約メニューの予約人数の列を削除して, あたらしく個人 ID の列を追加した. 予約数カウントフィルターにおいて, 予約人数を使用してメニュー数を計算していたため全ての予約について一人分として計算するようにフィルターを変更した. リレーショナルデータベースではシステムのどの部分で使うデータであっても一つのリレーショナルデータモデルの要素として扱われてしまうため, 複雑なシステムでは変更箇所が分かりにくくなるがデータフロー型のデータ構造では使用するデータがデータフロー図の中で把握できるため, システムのどの部分を変更すれば良いかが明確であり, 該当データとそのデータを送受信するフィルタのみを変更れば良いため全体のインテグレーションを必要としない. また, 交換代数を用いないデータフロー型のデータ構造については自作のフィルター内で処理する列を使用するデータを元に直接入力するなど, データの変更に対して自身が作成したコードについての記憶がないとフィルタ内で変更箇所を見つける作業が必要になる. それに対して, 交換代数を使用したデータフロー型のデータ構造では既存のフィルタを用いた交換代数の演算の組み合わせのみ

で全て作成されている. 演算に対応するフィルタが元々存在しているため, データ構造の変更に対して体系化されたデータの演算をどのように変えるかがそのままシステムの変更になり, データ構造の変更に合わせてフィルタを変更する必要がない. 小さなデータ処理をフィルターとして定義しているため自作のフィルタを変更することにかかるコストは大きくないが, データ構造の変更のたびにフィルタの内部を変更することは電子ブロックのような要素の組み替えでシステムを作成できるというデータフロー型のメリットを弱めてしまう. リレーショナルデータベース型についてはデータベース操作の知識,Web ページ作成の知識, サーバー側でのデータ処理の知識など幅広い知識とコーディングが必要であり, 交換代数を用いないデータフロー型のデータ構造ではフィルタを作成するための AADL の知識が必要になる. しかし, 交換代数を用いたデータフロー型のデータ構造では基本的な簿記の知識とそれを元に公理化された交換代数,FalconSeed の操作方法さえ理解していればコーディングの必要がないため現場での開発の環境として優れていると言える.

6 結論

本研究では日本の製造業など変更の激しいシステムに適したデータの管理手法が存在していない点を背景として, 従来のデータ管理手法とその問題点を検討した後に変更に強いデータ管理手法として交換代数を用いたデータフロー型のデータ構造を提案した. データフロー型のデータ構造は自律分散的に動作しているノードが必要なデータを受け取って処理をしたのち必要としているノードにデータを送信する流れの組み合わせで構成されているため, データ構造の変更があった際も変更があったノードとフローの中でそのノードとデータを送受信している部分のみについて変更を行えばよく, リレーショナルデータベースのように全体のインテグレーションを必要とせずに変更ができたことを示した. また, 交換代数を用いることでコーディングを一切行わずに一連の処理を行うシステムが作成可能であることを示した.

今後の展望として, 本研究では一つのコンピュータ上の FalconSeed で作成したが, 各ノードが独立している大規模なシステムにおいてデータ構造の変更を行った際のコストを計測することがあげられる.

また, データ構造の変更を含めたシステムの変更に対する柔軟性についての定量的な測定が挙げられる. 開発の工数計測としてあげたファンクションポイント法はシステムの規模を計測するものであり, コンピュータやシステムの定量的な測定としてはベンチマークテストと呼ばれる CPU の処理速度やデータ処理の速度などを計測するテストが一般に知られているがシステムの変更に対する柔軟性の評価を行うための手法について検討を行う必要がある.

7 謝辞

本研究を行うにあたり, 様々なご指導をいただきました指導教員である出口弘教授, 研究方法から論文の書き方まで親身に相談に乗っていただきました研究室の先輩方, FalconnSeed の使い方について丁寧に教えてくださった株式会社パイケークの皆様に御礼申し上げます.

参考文献

1) 経済産業省:製造業を巡る現状と政策課題, http://www.meti.go.jp/committee/sankoushin/seizou/pdf/005_01_00.pdf,(2017)
2) Codd : *A Relational Model of Data for Large Shared Data Banks* , Vol.2, No.6, 377/387(1970)
3) NoSQL Databases,http://nosql-database.org/
4) 松下 雅和:NoSQL の世界, 情報処理学会誌 Vol.51 No.10 ,1327/1331(2010)
5) pwc:technology forecast,Issue 1,(2014)
6) James Lewis:*Microservices*, https://martinfowler.com/articles/microservices.html, (2014)
7) Node-RED,https://nodered.org/
8) RWOS,https://realworldos.tumblr.com/post/135239706290/rwos%E3%81%A8%E3%81%AF
9) IFPUG,http://www.ifpug.org/
10) 出口 弘:複雑系としての経済学, 日科技連出版 (2000)