情報処理学会論文誌　データベース　Vol.8　No.2　68–77 (June 2015)

Webアプリケーション向け結合ビューライブラリにおける更新内容の即時反映機能とその実装

畑谷 卓哉[1,a)]　熊谷 良夫[1,†1]　鎌田 十三郎[1,2,b)]

受付日 2014年12月21日, 採録日 2015年4月6日

概要：現在，多くの利用者が各種クラウドサービスを用いて，自分たちのスケジュール情報や位置情報を登録・利用している．いくつかのサービスはWeb APIも提供しており，複数の関連情報を統合利用することで，利便性向上も期待できる．ただし，更新をともなうデータを扱う場合，データ更新に応じて関連情報の再取得を行うなど，アプリケーション実装上の手間も多い．本論文は，クライアントサイドWebアプリケーションフレームワークに対し，データ更新を即時反映可能な結合ビューライブラリを提供することで，情報統合アプリケーションの開発を容易にすることを目指す．データ変化に応じた結合ビューの更新はシステムが自動的に行い，ユーザによる結合ビューを介したデータ編集も可能である．また，Web APIプロキシにおいて，選択クエリ結果の再利用や論理和による複数クエリの取りまとめが行える仕組みを提供することで，結合演算のためのWeb API問合せ回数の削減を可能とした．

キーワード：情報統合，マッシュアップライブラリ，結合ビュー

Client-side Joined View Library for CRUD Web Service Operations and Its View Renewal on Data Modification

Takuya Hatadani[1,a)]　Yoshio Kumagai[1,†1]　Tomio Kamada[1,2,b)]

Received: December 21, 2014, Accepted: April 6, 2015

Abstract: In recent years, we are using many kinds of cloud services to put various data entries (e.g., schedules, blog entries, or location tracking data) and share among groups. Some of these services offer Web APIs, and a developer can utilize these APIs to build new integrated Web applications. However, when the developer wants to treat data association between mutable data, the developer has to carefully prepare routines to watch such data changes and request associative data for updated entries. This paper aims to provide a client-side JavaScript library to join Web service that process CRUD operations. Even when users add/modify source elements, the library continuously offers the renewed view reflecting the modification. In order to reduce the number of Web API requests needed for joined view construction/maintanance, our library prepare element cache to eliminate duplicated requests, adajust Web API queries taking care of client-side data modification, and provide a mechanism for developers to register query mergence processes.

Keywords: information integration, mashup library, joined view

1. はじめに

現在，多くの利用者が各種クラウドサービスを用いており，クラウド上に自分たちのスケジュール情報などを登録し，グループ間の情報共有や，PC/モバイル端末間での情報共有などに利用している．

これらのクラウドサービスのいくつかはWeb APIも提

[1] 神戸大学
Kobe University, Kobe, Hyogo 657–0013, Japan
[2] 独立行政法人科学技術振興機構，CREST
CREST, JST, Kawaguchi, Saitama 332–0012, Japan
[†1] 現在，株式会社アルファシステムズ
Presently with ALPHA SYSTEMS INC.
[a)] takuya.cs26@gmail.com
[b)] t_kamada@acm.org

供しており，アプリケーション開発者は，複数の関連サービスを統合的に利用することで，アプリケーション上で多くの関連情報を提供することもできる．たとえば，上映中の映画情報を検索するサービスに，感想の投稿/閲覧サービスを結合することで，ユーザは手軽に感想を発信したり，友人の感想を参考にしながら見る映画を決めたりすることができる．また，カレンダーサービスとリンクさせれば，上映時間とスケジュールの重複情報を付加することもできる．これらのサービスのうち，感想投稿サービスやカレンダーサービスは，ユーザによる情報の登録・修正が可能であり，アプリケーションは，情報の検索・閲覧だけでなく，登録・更新処理への対応が求められる（スケジュールの変更に応じた，時間重複情報の修正など）．

近年の Web アプリケーションは，Ajax 技術などを用いたインタラクティブ性の高いものが多い．ライブラリ整備も進んでおり，地図や表・グラフなどの各種 GUI コンポーネントなども提供されている．一方で，Ajax や GUI アプリケーションにおいては，非同期処理・イベント駆動処理が基本であり，アプリケーション開発は，必ずしも簡単なものではない．データ登録や更新などの CRUD 操作が可能な Web サービスを他の Web サービスと連携して利用する場合，ユーザによるデータ更新処理に応じて，関連データを随時取得するといった処理が加わるため，さらに実装上の手間が増えてしまう．

本論文は，クライアントサイド Web アプリケーションフレームワークに対し，データ更新を即時反映可能な結合ビューライブラリを提供することで，情報統合アプリケーションの開発を容易にすることを目指す．開発者は，各 Web サービスを提供データ要素群を保持したテーブルとみたて，ライブラリの提供するある種の左結合演算によって結合ビューを作成することができる．取得データ要素に更新操作が行われると，システムが自動的に関係する結合ビューへの反映処理を行う．また，結合ビューに対する更新操作は，元となる取得データ要素への更新操作として処理し，状況に応じて結合ビューの編集箇所以外の部位にも反映処理が行われる[*1]．Web アプリケーションフレームワークには，洗練されたデータモデルおよび MVC（Model-View-Controller）モデルに基づいた GUI を有した Sencha Ext JS [2] を用いており，結合ビューの内容を各種 GUI コンポーネント（地図や表）を通じて表示可能である．結合ビュー上の変化は，MVC モデルを介して随時 GUI コンポーネントに反映できる．

更新操作が可能な結合ビューに関する基本提案は，先行研究 [3] において行っている．ただし，単一結合ビュー内の更新反映のみを目的としたため，結合条件に関わる右側要素属性については更新操作を禁止するなど，一般ライブラリとして利用するには強い制約を想定していた（7.4 節）．

本論文の貢献は，汎用的に利用可能で効率的に動作する結合ビューライブラリを実現するため，要素データ更新に応じて結合ビューを部分的に変更する手法を提案・実装し，また，結合演算のために発行される Web API クエリ数を抑制するための Web API プロキシ機構の実装手法を示したことにある．提案システムでは，検索クエリの結果をプロキシで管理し，その更新操作をフックすることで，検索結果の再利用によるクエリ数抑制と，更新操作の結合ビューへの即時反映を実現している．問合せに際しては，クライアント側のデータ更新を考慮したサーバに対するクエリ補正が行われ，また，複数の検索クエリに対して論理和による取りまとめを行うための仕組みも提供されている．本論文で提案するライブラリは，文献 [4] において公開中である．

本論文の構成は以下のとおりである．まず，2 章で提案ライブラリが想定する Web アプリケーションについて例を用いた説明を行う．3 章では，ベースとなる Ext JS フレームワークについて説明し，4 章で提案ライブラリの概要とその実装方針について述べる．5 章で実装法を説明し，6 章で提案ライブラリの性能と有用性について評価する．7 章で関連研究紹介，8 章でまとめを行う．

2. 対象とするアプリケーション例

提案ライブラリは，データ登録や更新などの CRUD 操作が可能な Web サービスを他の Web サービスと連携利用するような Web アプリケーションを想定したものである．**図 1** は，スケジュールアプリケーションの例であり，ユーザは各種イベントとその関連情報を見ながら予定を調整することができる．左上の表は，ユーザが選択した日の自分の予定を Google Calendar から取得・表示したもので，時間帯が重複した予定があれば表内右側に表示している．また図の右側には，映画の上映情報（上）やイベント開催情報（下）を表示しており，たとえば，映画情報サービスから取得した当日の上映予定一覧を表示する際，対象映画に対する感想や，各上映時間とユーザの予定との重複情報を付加している．提案ライブラリは，このような複数のサービスの情報を結合した結合ビューを容易に作成可能とする[*2]．

ユーザは，関連情報を見ながら見る映画やイベントを選び，「＋」ボタンにより自分の予定表に仮登録することができる．また既存の予定と重複している場合は，その予定を削除したり時間を変更したりすることで，予定の調整を行うこともできる．ユーザの更新操作に応じて，結合ビュー中の時間帯重複情報なども自動的に更新される．更新操作

[*1] 結合ビューの更新については，更新可能性問題 [1] が知られているが，我々のライブラリとは問題設定が異なる（7.3 節）．

[*2] なお，このアプリケーションでは，Google Calendar 以外は，ダミーの映画/イベント/感想投稿サービスを自作して利用しているが，提案ライブラリでは一般のサービス（Yelp や Foursquare など）の利用実績もある．

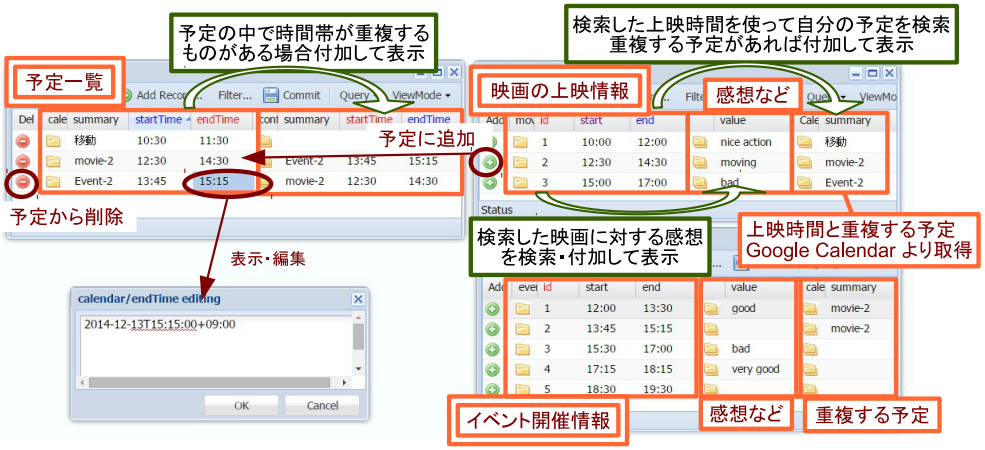


図 1　アプリケーション事例
Fig. 1　Sample application.

は結合ビューを介しても可能であり，システムは結合演算の元となった要素データに対する更新操作と，更新内容の結合ビューへの反映を行う．たとえば，左上の予定一覧表において，ある予定の時間帯を変更した場合，その予定が複数箇所に出現していれば，それらがすべて更新され，場合によっては別の予定との重複の解消（重複表示が消える）や，新たな重複の発生などが即時反映される．このように，ユーザは結合ビューを通して関連する予定の状況を確認しながら，その調整をすることができる．ユーザは，作成/修正した予定を随時サーバに登録することもできるし，一連の予定を仮決めしてから，予定間の時間衝突を確認したうえで，まとめて確定させるといった利用法も可能である．

3. Sencha Ext JS

Ext JS [2] は，インタラクティブな Web アプリケーションを構築するための JavaScript フレームワークである．主な特徴として，開発者は操作性の高い各種コンポーネントを利用でき，MVC モデルに従った実装が可能である．利用可能なコンポーネント部品として，テーブルビューア，ウィンドウ，メニューバー，ボタン，チャートが提供され，他の Javascript GUI ライブラリなどとの併用も可能である．一般的な Ajax 環境では，Web 問合せの結果に基づいて結果の再描画を行う場合，callback 関数を用いた明示的再描画を行うことになるが，Ext JS の場合，Web 問合せ結果をデータモデルに格納することで，MVC モデルに基づいた再描画をシステムが自動的に行うことができる．

以下，提案ライブラリとの関係性が深いため，Ext JS のデータモデルについて解説する（図 2）．データ要素は Model とそのインスタンス（以下，要素と呼ぶ）として扱われる．開発者は，要素データの構造（保持する属性のリストなど）を Model として定義する．要素のリストを扱う場合，Store というデータ構造を用いる．Model 定義時に Web サービスにアクセスするためのプロキシを指定することができる．開発者は，プロキシにサーバの URL や返り値

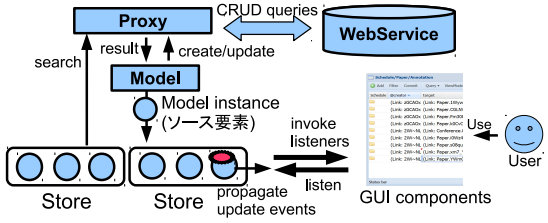


図 2　Ext JS のデータモデル・MVC モデル
Fig. 2　Data model and MVC model of Ext JS.

のフォーマットを指定することで，それぞれの Web サービスのためのプロキシを簡単に準備できる．また，プロキシには Ajax や JsonP などの種類が用意されている．Model や Store は，プロキシを介して CRUD 要求を対象サーバの Web API に対して行い，Web サービスに対する検索クエリの発行や，Model 要素の追加・修正内容のコミット操作が可能となる．サーバへのデータの登録・修正は，まず，クライアント上で Model 要素の作成や状態修正を行い，その後，サーバに対してコミットしたい時点で，`save` メソッドなどを実行することになる．

Store が，サーバに Read 要求（`load()` 命令）を行う際，要素に対する検索条件（filter）を指定することができる．この際，`limit` option により，要求する要素数も指定可能である．検索条件や `limit` option は，プロキシにおいて Web API への Request パラメータに変換され，Web サーバに対する XMLHttpRequest が発行される．結果の受け取りは非同期に行われ，Store に結果が格納される．また，Store には，読み込まれた Model インスタンス群に対して filter をかける機能（localFilter）も提供される．

Ext JS の GUI は，データモデルとして Store を使い，MVC モデルを実現する．Model インスタンスに変化があった場合は，Store にデータ更新イベントが通知される．GUI コンポーネントは，Store に対してイベントリスナを登録することで，データ更新に応じた再描画を実現する．

このように，Ext JS のデータモデルは，Web サービスと

Model を 1 対 1 に対応づけたものであり，GUI コンポーネントは，Model 要素の配列である Store を対象とすることが多い．このため複数のサービスにまたがった結果を GUI などで扱う場合は，それぞれの Web サービスからの検索結果を Store に取得してから，自分で結合演算を施す必要がある．単に，2 つのサービス $S1$，$S2$ から検索結果をリストとして取得し，クライアントで組み合わせ処理ができる場合は，プログラミングも簡単である．一方で，$S1$ の検索結果の各要素に対して，$S2$ に検索クエリを発行する場合は，個々の問合せ応答の管理が必要となる．

このようなケースでは，開発者は `HasManyAssociation` 機能を用いることで，$S1$ の各要素に対して，$S2$ に検索クエリを発行するための Store を作成することができる．ただし，$S1$ と $S2$ を併合したような Model を作成するのとは違い，たとえば，図 1 のような結合情報の一覧において，要素の状態更新を即時反映させたい場合，プログラマ自身が更新の影響範囲の把握や GUI 描画機構との連携などを実現する必要が生じる．

4. 提案結合ライブラリ

4.1 結合ビューの概要と実装方針

本節では，提案ライブラリのデータモデルについて概説し，その実装上の課題と対策について述べる．データモデルの定式化と詳細な実装技術の紹介は，5 章で行う．

提案している結合ビューライブラリは，複数の Model（ソース Model と呼ぶ）とその結合条件を与えると，対応する結合要素を表現するための JoinedModel（結合 Model と呼ぶ）を作成する．結合 Model は，結合条件を満たしたソース Model の要素（ソース要素と呼ぶ）の組を表現する．提案ライブラリはある種の左結合演算を想定しており，1 つの最左ソース要素に対して，結合要素を 1 つ生成し，各入力ソースに対応したソース要素をたかだか 1 つ検索する．**図 3** は，ソース要素と結合要素の関係を図示したものである．`A` の各要素に対して，対応する `B`，`C` 要素が検索され，結合要素が構成される．`a3` のように，対応する `B`，`C` 要素が存在しない場合もある．図中の `B`，`C` などはソース Model を識別するための *alias* であり，また，ソース Model L の属性 *attr* は，結合 Model の属性 L/*attr* として表現される．また，結合要素の属性に対する更新操作は，対応するソース要素に対する更新操作として扱われる．

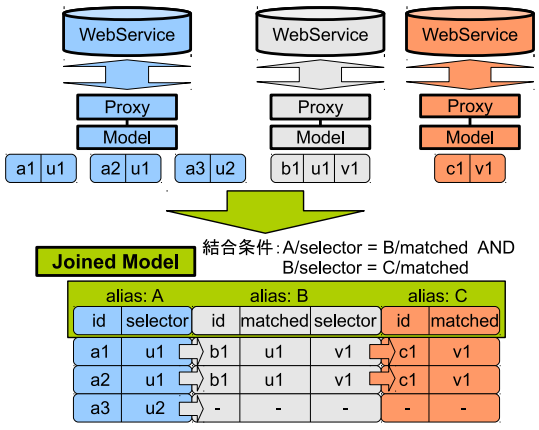


図 3　結合要素とソース要素の関係
Fig. 3　Relation between source and joined elements.

結合ビューの実装法について考えると，まず最左ソース要素に関する検索を行い，その後，各最左要素に対して結合条件を満たした右側ソース要素の検索を行うというのが，単純な実装法である．サーバに対して，それぞれ要求要素数（limit）1 のクエリを発行する．ただし，素直に最左要素の個数に応じた検索要求を Web API に対して行うのでは，無駄が多い．一方で，結合演算ごとに Web API への問合せ方法をカスタマイズするのは大変である．各 Model のプロキシにおいて，クエリの取りまとめや，結果の再利用を行うことで，Web API に対する検索クエリ数を抑制する機構を目指す．

次にソース要素の更新を，結合ビューに即時反映させるための方法についてであるが，あるソース要素の更新が結合要素に与える影響は，
(ア) 左側ソース要素の属性更新により結合条件が崩れる，
(イ) 右側ソース要素の属性更新により結合条件が崩れる，
(ウ) 更新の結果，他のソース要素との間で結合条件が成立
の 3 種類である．たとえば，図 3 のソース要素 `b1` において，属性更新 `v1`→`v2` が起こった場合，(ア) より対応する `C` 要素の検索が必要となる．一方，`b1` において属性更新 `u1`→`u2` が起こった場合は，(イ) より 1，2 行目に対する B，C 要素の検索が必要となる一方で，(ウ) より 3 行目に `b1` 要素が出現することになる．このように，更新属性に応じて，その影響範囲の再計算を行う必要がある．また，我々のシステムでは，ソース要素の生成・更新は，明示的にコミット操作を行うまで，サーバには反映されない．このため，その間はサーバとクライアントで当該要素の状態が異なることになり，ライブラリはクライアントにおける更新状態を優先して検索・結合処理を行う必要がある．

4.2 結合ライブラリの利用法

図 4 は，LJoinManager（JoinedModel の manager）の作成例であり，2 章で紹介した事例に相当する．Model を `Event`，`Memo`，`Calendar` の順に結合している．

結合条件は，2 番め以降のソース Model に対する `selections` 属性として記述される．いくかの記法が準備されているが，図 4 の例では，`event/id` の値を用いて `memo` の `key` の値を指定し，また，`conflict` においては，`event/end` および `event/start` の値を用いて，`startTime` と `endTime` の上限と下限を指定している．我々のシステムでは，結合条件に利用される左側要素属性（`event/id` 相当）を「selector attribute（選択属性）」と呼び，結合条件

中で明示されるデザインとしている．

結合条件は，オプション指定された型や operator に応じたフィルタが適用される．結合条件を関数記述したい場合は，17–28 行めのようなローカルフィルタを付加的に記述することができる．左辺要素属性値の参照は，value または values を介して行う．ローカルフィルタは，サーバから取得したソース要素に適用するフィルタであり，operator などに応じて定まるフィルタと重複して適用される．

5. 実装法

5.1 データモデル

本節では，実装法の説明に先立って，データモデルの定式化を行う．

入力となる Web サービス相当をテーブルと呼び，alias L_i で識別する．また，結合要素 u の L_i に相当するソース要素を $u.src(L_i)$ で表記する．また，alias L_i に対する結合条件を $selections(L_i)$，選択属性の集合を $selectors(L_i)$ とする．また，結合要素の属性は，alias L_i の属性 a を L_i/a などの形で表記する．ある結合要素の L_i, L_j（L_i が左側とする）について，$L_i/a \in selectors(L_j)$ ならば，u の L_i/a 属性値が変化すると L_j の結合条件も変化するため，$u.src(L_j)$ の再検索が必要となる．

選択属性 L_i/a の更新により再検索を行わなくてはならない入力テーブルを L_i/a の**影響範囲**と呼び，$Dominated(L_i/a)$ で表す．$Dominated(L_i/a)$ は以下の 2 式で定められる．

$$L_i/a \in Selectors(L') \Rightarrow L' \in Dominated(L_i/a) \quad (1)$$

$$L'/a' \in Selectors(L'') \wedge L' \in Dominated(L_i/a) \Rightarrow L'' \in Dominated(L_i/a) \quad (2)$$

提案ライブラリでは，$selectors(L_i)$ は，LJoinManager 定義時に定まるため，その際，$Dominated(L_i/a)$ などの計算も行っている．

```
1  new LJoinManager([ // JoinedModel の manager
2    {alias:'event', tableID:'Event'},
3    {alias:'memo', tableID:'Memo',
4     selections:[
5       {property:'key',
6        value:{table:'event', property:'id'}}]},
7    {alias:'conflict', tableID:'Calendar',
8     selections:[
9       {property:'startTime',operator:'<',
10       value:{table:'event',property:'end'},
11       type:'time'},
12      {property:'endTime',operator:'>',
13       value:{table:'event',property:'start'},
14       type:'time'}]}
15 ]);
16 // 結合条件の関数記述（ローカルフィルタ）の例
17  selections = [
18  { property: .., value: .., type: ..,
19    // 属性単位のローカルフィルタ
20    localPropertyFilter: function(value) {
21      return value != this.value; }},
22  { // 複数の左側要素属性を利用
23    type: 'multi',
24    values: [ {table:.., property: ..}, ..],
25    // 要素単位のローカルフィルタ
26    localFilter: function(item) {
27      return item.get(..) < this.values[0]
28           && ...; }}]
```

図 4　結合条件記述（例）
Fig. 4　Sample settings of LJoinManager.

5.2 結合要素のビュー管理

結合要素は，既存の Model クラスの構造を維持するためソース要素の全属性に相当する属性を保持しており，ソース要素更新のタイミングで，コピーが行われる．Ext の Model は，どの属性がクライアント側で更新したものであるかといった情報も保持しているが，この種の情報も結合要素へのコピーが行われる．現在の実装では，ソース要素の更新を結合要素に伝搬するため，ソース要素から所属結合要素への逆向き参照を保持し，set メソッドへのフック処理により情報伝搬を実現している．

結合要素に対する更新操作においては，禁止すべき更新であるかの判定（5.3 節）を行った後，対応するソース要素への属性更新と，結合ビューへの反映を行う．

ソース要素 s の属性 a が更新された際，結合要素 u に行うべき更新処理は，以下のとおりである．

(1) $u.src(L) = s$ であった場合，u の属性 L/a の更新を行う．もし，L/a の更新によって結合条件 $selections(L)$ が崩れた場合は，プロキシに対して再検索を要求する（4.1 節 (イ) 相当）．L の結合条件が崩れなかった場合も，$L/a \in selector(L')$ であるような $j.src(L')$ に対して再検索が必要となる（同 (ア) 相当）．

(2) $u.src(L) = null$ であったが，s の更新の結果，s が検索条件を満たすようになった場合は，$u.src(L)$ を s に変更する（4.1 節 (ウ) 相当）．これにともなって，$L/a' \in selector(L')$ であるような $u.src(L')$ に対しても再検索が行われる（同 (ア) 相当）．

現在の実装では，(1) についてはソース要素から結合要素への逆向き参照を用いて確認を行っている．現在の結合条件記述では，ソートオーダ指定を許していないため上記 (1)，(2) のみの対応で十分であるが，今後ソート指定に対応する場合は，更新の結果 s が $u.src(L) = s$ の最良解になるような結合要素 u をスキャンする必要が生じる．

選択属性に変更があった場合は，上述のようにソース要素の再検索が行われるが，結果が確定するまでの間は，影響範囲を「検索中」状態に変更する．この種の結合要素の状態変化についても，ソース要素と同じく状態更新イベン

トが発行され，結合要素を含む Store へのイベント伝搬や，GUI コンポーネントが登録したイベントリスナの起動などが Ext の MVC フレームワークに則って実行されることになる．

5.3　禁止すべき更新の判定

結合要素に対する更新操作における，更新を禁止する処理について述べる．本節の内容は，基本的には先行研究 [3] で提案済みのものである．

本章の最初で述べたように，我々のシステムでは，結合要素への更新は対応するソース要素に対して行われる．ただし，結合要素を介したソース要素に対する更新によって，当該ソース要素が結合条件を満たさなくなる可能性がある場合は，その更新処理を禁止する．図 3 の場合，結合ビュー上で属性 `B/matched` を `u1→u2` と変更するのは禁止される．システムが u の L/a への更新を禁止するのは，更新の結果，更新対象の $u.src(L)$ 自体が選択条件を満たさなくなる場合と，左側に同一ソース要素が存在し，書き込み対象がその影響範囲に入っている場合（$u.src(L') = u.src(L)$ かつ $L \in Dominated(L'/a)$）である．

5.4　プロキシ実装

4.1 節で述べたように，提案ライブラリでは，結合演算に関わる検索クエリの抑制は，各 Model に対するプロキシレベルで行う．各結合要素は，左ソース要素の属性値に応じて右ソース要素の検索要求をプロキシに対して行い，プロキシにおいて複数クエリの取りまとめや，以前のクエリ結果の再利用を行う．加えて，クライアント側で生成もしくは更新されたが，サーバにはまだコミットされていないソース要素に関する管理も行う．

提案ライブラリで提供する `CoherentProxy` は，一般のプロキシ（移譲先プロキシ）に対するアダプタとして利用可能で，当該 Web サービスから取得したソース要素の管理や，Web API に対するクエリの補正やクエリ結果の再利用などの処理を行う．

5.4.1　キャッシュの利用とクエリの補正

提案プロキシでは，Web API から取得したソース要素に関するキャッシュを保持する．Web API からの返り値がすでにキャッシュに含まれていた場合は，キャッシュ上のソース要素を代わりに返却し，同一 ID の要素が複数存在しないようにする．メモリ消費を抑えるため，プロキシは利用されていないソース要素を除去するためのメソッドを備えるが，ソース要素が Store に所属しているか（当初から判別可能），結合要素にソース要素として所属しているか，あるいは，未コミットの更新属性を持っている間は，キャッシュからクリアしない．

プロキシへの検索要求があった場合は，まずキャッシュに対してフィルタを適用し，検索条件に合致する結果がある場合は，その検索結果を返却するための callback 関数を実行する．もし，キャッシュ上に検索条件に合致するソース要素が指定された個数分なかった場合は，Web API に対して検索クエリを発行することになる．ただし，我々のシステムでは修正中データ（未コミット）に対する検索・結合を可能としているため，修正による検索クエリへの影響を加味する必要がある．提案ライブラリでは，最左ソース要素 1 個に対して，たかだか 1 個の右ソース要素検索を行っており，その際プロキシには `limit=1` という検索 option が渡される．システムは，修正前に検索条件を満たしていた要素数と，修正後に検索条件を満たすようになった要素数をカウントし，サーバに対する検索クエリの limit 数を補正する．**図 5** の例では，`value=v1` という検索条件について，データ修正によって，2 つの要素が検索条件を満たさなくなり，また新たに 1 つの要素が検索条件を満たすようになっている．このため，当該検索条件については $(+2)+(-1) = +1$ の limit 数補正を行う．

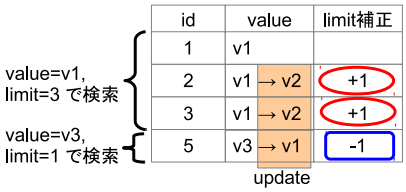


図 5　リミット補正
Fig. 5　Adjusting limit parameter of search query.

5.4.2　Web API へのクエリ発行

提案プロキシでは，アクセス対象のサーバに頻繁にリクエストを発行しないように，クエリのバッファリングを行う．同一サービスに同時に複数のリクエストを発行しないように制限を行っており，また，検索クエリの結果が返ってきた際は，その結果をキャッシュに格納したうえで，バッファリング中の検索クエリの中に，キャッシュを用いて対処できるものがないか確認する．この対応により，同一内容のクエリが複数回発行された場合も，Web API へのリクエストは 1 回に抑えることができる．

加えて，対象 Web サービスでは複数のクエリを 1 つにまとめて対処できると Web アプリ開発者が判断した場合に，取りまとめ処理をプロキシに登録し，ライブラリ側から利用するための仕組みも導入した．Web API の中には，複数要素の id を列挙することで，それら要素群の詳細情報を取得できるようなものも多い．また，クエリの種類によっては，意味的に検索条件の論理和をとることで，クエリを取りまとめられるケースも存在する．たとえば，今回利用した Google Calendar API は，指定した時間内の全スケジュールのリストを返すが，複数の時間帯を含む広い時間帯で検索することで，クエリの取りまとめが可能である[*3]．

[*3] 利用する API が，検索条件に limit を設定しているような場合，単純な取りまとめはうまく機能しないなど，注意点も存在する．

このような Web API を扱う場合，アプリケーション開発者は移譲先プロキシに，クエリの取りまとめを行うための関数 `mergeRequests(mergedRequest, request)` を定義することができる．提案プロキシでは，クエリバッファを確認する際，各クエリがキャッシュを用いて処理可能かチェックを行い，キャッシュ処理不能な先頭のクエリを Web API に対して発行することになるが，`mergeRequests` 関数が定義されている場合は，後続クエリのうちキャッシュでの処理ができないものについて，先頭クエリとの取りまとめ処理を行い，Web API に対してはマージされたクエリを発行する．返ってきたクエリ結果はまずキャッシュに格納され，その後，個々のクエリに対してキャッシュを用いた検索を行い，それぞれのクエリの callback 処理を行う．また，本クエリの取りまとめ機能を活用するために，提案プロキシが検索リクエストを受けた場合は，一定時間（デフォルトで 100 msec）待ったうえで，処理を行うようにしている．

6. ライブラリの有用性と評価

6.1 性能評価

本節では，提案システムの反応速度および Web API へのクエリ発行回数削減効果について性能評価を行う．測定用に準備した Web サービス `L1`, `L2`, `L3` を結合するという実験を行い，結合条件は `L1/sel=L2/matched AND L2/sel=L3/matched` とした．各 Web サービス L の i 番目の要素データを $L(i)$ = {`id`: i, `matched`: i, `sel`: $i/2$ }（i は整数）として作成し，`L1` の {`L1`(i)$|0 \leq i < 30$} の 30 要素に対して結合演算を行った（**表 1** 相当）．測定対象ライブラリのバージョンは 0.1 である．測定用のサービス `L1`, `L2`, `L3` は，Google App Engine 上に配置されており，測定クライアントには一般の広域ネットワークに無線 LAN 接続したノート PC（Intel Corei-3 2350M 2.3 GHz，Windows 7）上の chrome（ver. 39.0.2171.95 m）を利用した．サービス単独でのレスポンスタイムは，`L1` が 350 msec，`L2`，`L3` が 250～300 msec 程度であった．

まずは，ソース要素修正時に，結合ビューへの反映が行われ，GUI コンポーネントのレンダラが呼ばれるまでの時間を測定する（**表 2**）．

表 1　実験 A1 で使用するデータ
Table 1　Benchmark data for Experiment A1.

id	sel	id	matched	sel	id	matched
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	1	1	1	0	0	0
...	...	...	...	...	...	...
29	14	14	14	7	7	7

実験 A1：ある L1(i) の `sel` 属性を i' に更新し，`L2`，`L3` のレンダラが「検索中」の表示，および検索結果の表示を行うまでの時間（それぞれ start，disp）を測定．i' 相当の L2 要素がすでにキャッシュにある場合（$0 \leq i' \leq 14$）と，ない場合（$i' \geq 15$）に分けて実験を行う．

実験 A2：事前に L2 テーブルから L2(i')（$0 \leq i' \leq 14$）要素を削除した状態で，L2(i) の `matched` 属性を i' に変更し，結合要素（L1/sel=i）からの L2(i) の消滅と，結合要素（L1/sel=i'）への出現に関して，「検索中」の表示，および検索結果の表示までの時間（それぞれ start，disp）を測定する．たとえば $i' = 1$，$i = 0$ の場合，事前に L2(1) を削除した状態で，L2(0).matched=1 を実行すると，0，1 行目から L2(0)，L3(0) が消滅し，2，3 行目へ L2(0)，L3(0) が出現する．

各実験は，それぞれ 5 回測定したうえで，平均値と最大/最小値（実験 A2 では，消滅と出現が 2 つの行で起こるが，すべてを対象とした最大/最小値）を表記した．実験 A1 で，すでにキャッシュ中に検索対象がある場合は，100～150 msec 程度で L2，L3 の結果表示が完了している．一方で，キャッシュには結果がなく Web API に対してクエリを発行する場合も，即座に L2，L3 とも「検索中」状態に移行し，その後は Web クエリの結果到着を待って順次結果の表示（L2: 700～900 msec，L3: 1,300～1,500 msec）が行われている．今回の Web サービスは，リクエスト発行から callback 関数の実行まで L2，L3 それぞれ 500～600 msec 程度要していることを考えると，クライアント側の処理に要した時間の合計は L2，L3 あわせて 300～400 msec 程度であったと思われる．この中には，クエリ取りまとめのための Web クエリ発行待ち時間 100 msec が 2 回分含まれる．次に実験 A2 についてであるが，「検索中」への移行は 80 msec 程度以内で完了しており，L2，L3 消失確定までに最大 1,300 msec 程度有している（後で検討）．全般的にみると，結合要素数が 30 個程度で，我々のシステムは高い

表 2　実験 A1，A2（実行時間，単位 ms）
Table 2　Experiment A1, A2 (Elapsed time in ms).

実験 A1	L2 start	L3 start	L2 disp	L3 disp
キャッシュを用いた検索結果取得が可能				
平均	1.6	1.8	90.6	90.6
max/min	2/0	3/0	103/77	103/77
L2,L3 にクエリ発行実施				
平均	2.0	2.4	747.4	1389.4
max/min	3/1	3/1	889/686	1517/1296

実験 A2				
L2(i) 消失 start	L3(i) 消失 start	L2(i) 消失 disp	L2(i') 出現 disp	L3(i') 出現 disp
平均 max/min				
51.5	51.5	1286.7	171.3	301.8
76/17	76/17	1636/933	251/96	363/225

表 3　実験 B1：WS 呼び出し回数（結合要素数 30）
Table 3　Experiment B1: # of WS calls for creating 30 joined instances.

クエリをマージできる最大値（M）	1	5	10	15	20
L1 への問合せ回数	1	1	1	1	1
L2 への問合せ回数	15	3	2	1	1
L3 への問合せ回数	8	3	2	1	1

表 4　実験 B2：WS 呼び出し回数（選択属性 20 カ所の更新）
Table 4　Experiment B2: # of WS calls on 20 updates on selector attributes.

クエリをマージできる最大値（M）	1	5	10	15	20
L2 への問合せ回数	20	4	2	2	1
L3 への問合せ回数	20	4	2	2	1

反応性を示している．

実験 A2 について，もう少し詳しく分析を行う．本実験では「消失」行が 2 つ「出現」行が 2 つ存在し，「出現」についてはキャッシュ処理が行われている．一方，「消失」について，行ごとに Web サービスリクエストが合計 2 回発行されており，結果，先の実行時間が必要となっている．これは，現在のシステムでは，対象 Web サービス上に `matched` 属性の値が i である L2 要素が 1 つ「しか」ないことをプロキシが認識できず，そのつど問合せが行われているためであり，今後，実装法を改善する予定である．

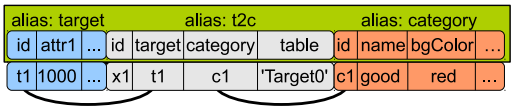
次に，結合演算時のクエリ取りまとめ効果について評価を行う．対象 Web API に対して，クエリをマージできる最大値（M）を 1，5，10，15，20 と変えて実験を行った．L1 については，一度の問合せで 30 件のデータを取得する API を仮定する．

実験 B1（表 3）：初期状態から結合ビューの表示を行う．内部では，まず L1 のクエリを実行し，L1 の各要素に対して L2，L3 へクエリを発行する．30 件の結合要素を表示するまでの，L2，L3 に対するクエリ発行回数を測定する．

実験 B2（表 4）：実験 B1 終了後，`L1/sel` 相当の属性を 20 個同時に「それぞれ異なる値」に修正し，これにより L2，L3 に対するクエリ発行回数を測定する．更新の結果取得される L2，L3 は，すべてキャッシュ上にないものとする．

実験結果をみると，実験 B1 についてはキャッシュを用いたクエリ発行の削減が十分に行われ，加えて L2 クエリに関して最大値の個数のクエリをマージしているが，$M > 1$ の場合，L3 クエリの発行回数は L2 と同じ回数となっている．これは，L2 への問合せ結果到着に応じて L3 へのクエリ発行が行われるが，マージ処理は一定時間（デフォルトで 100 msec）内に発行されたクエリに対してのみ行うため，結果として L2 への各クエリの結果（複数要素）に対する L3 クエリのみがマージされる．反応時間を重視するため，ライブラリが想定した動作である．

実験 B2 においても，意図したとおりのクエリの実行回数の削減が達成されている．本手法が実際に有効に機能するかは，実際の対象 API に対してクエリのマージが可能であり，サーバ側のスループット改善も期待できるという前提が必要となる．今後，実際の Web API に対する調査を行い，その有効性を評価していきたい．




```
new LJoinManager([
  // ベースとなるサービス
  {alias:'target', tableID:'Target0'},
  // カテゴリ分類
  {alias:'t2c', tableID:'Item2Cat',
   selections:[
     // target/id=t2c/target
     {property:'target',
      value:{table:'target', property:'id'}}
     // 分類対象の表が 'Target0' であることを指定
     {property: 'table', value: 'Target0'}]},
  // 装飾
  {alias:'category', tableID:'Category',
   selections:[
     // t2c/category=category/id
     {property:'id',
      value: {table:'t2c',
              property:'category'}}]}]);
```

図 6　結合条件記述（例 2）
Fig. 6　Sample settings of LJoinManager (2).

6.2　ライブラリの有用性

まず，本ライブラリを使用しない場合，結合ビュー相当の実装の手間について検討する．3 章でも述べたように，2 つのサービス $S1$，$S2$ からソース要素を取得し，クライアントで結合することができる場合は，結合ビューの作成自体は容易である．ただし，$S1$ の各ソース要素の属性に応じて $S2$ を取得しなくてはならない場合，5.4.2 項で行ったようなクエリの併合を行うか，$S2$ に対して個別的問合せに対する非同期処理を含めた管理が必要である．後者において，過去の問合せ結果の再利用を行う場合，さらに複雑な処理が必要となる．加えて，ソース要素更新に応じた差分的結合ビューの再計算を実現する場合，5.2 節相当の機構が必要となる．一方で，提案ライブラリを利用した場合，4.2 節で述べたように，数十行程度の設定のみで動作する結合ビューが実現可能であり，必要に応じて `mergeRequests` 関数（5.4.2 項）を定義することで，検索クエリの削減を図ることができる．

また，具体的なライブラリの応用事例を通して，ライブラリの記述性について検討する．2 章とは異なり，アプリケーションの内部機能実現にライブラリを用いるケースを紹介する．ある Web アプリケーションを機能拡張し，ユー

ザが興味を持った項目にカテゴリ情報を付与でき，カテゴリごとにGUI装飾も指定できるようにしたいとする．結合ビューを用いた場合，開発者はベースとなるWebサービスに対して，各カテゴリとその装飾を記録するサービス（Category）と，各ソース要素のカテゴリ分類を記録するサービス（Item2Cat）を準備すれば，20行程度で装飾情報を含む結合要素を実現できる（図6）．GUI部では，結合要素の属性に基づいた装飾を施すだけでよい．このように，様々な分野でライブラリが利用されることが期待される．

7. 関連研究

7.1 マッシュアップツール

多くのマッシュアップツールは，マッシュアップアプリケーションの作成補助をするために提案されている．しかし，Webサービスに対するCRUD操作を想定したものはほとんど存在しない．Yahoo! Pipes [5] やPopfly [6]，Plagger [7] は，Webサービスの接続関係をGUI操作などにより指定できるようにすることで，開発者が，より簡単にマッシュアップを行うことを目指したツールである．これらのツールにより，開発者はプログラミングを行うことなく新たなWebサービスやWebアプリケーションを簡単に作成できる．Popflyでは，ユーザが入力データに操作するためのUI Blockが提供されるが，CRUD Webサービスとのマッピングなどを指向したものではない．

Mash Maker [8], [9] は，エキスパートユーザが各Webページに対し，有用と思われるマッシュアップを作成し，ウィジェットとして公開することで，一般のエンドユーザが興味に応じて閲覧ページに対して付加可能なウィジェットを選択表示できるようにしたものである．木構造のデータモデルにより柔軟なウィジェット作成が可能である．我々のグループでもマッシュアップ環境 [10] を作成しており，Webサービスの接続関係を宣言的に記述するが，インタラクティブなビューアを備えることで，ユーザが画面上で表示させたい項目についてのみ要求駆動でデータ合成を行うという環境を実現している．いずれのツールも，ユーザ入力を入力ソースの一種として利用することは可能だが，Webサービスから取得したデータに対する修正は考慮しておらず，本実装で行っているようなクエリの補正（limitの補正）も行われない．また，Webクエリ数削減に対しては，同一クエリに対する結果の再利用は行われているが，それ以上の対策は行われていない．

7.2 結合データのためのエディタ

いくつかのデータベースシステムは，テーブルの結合やソースデータの編集を行うためのGUI環境を提供しており，Microsoft Accessは，結合ビューの閲覧や結合ビュー上でのデータ編集機能も提供している．結合条件を指定して結合ビューを作成し，すべてのテーブルを外部キーによりたどれるとき，ユーザは結合ビューから編集することができる [11] が，提案ライブラリで行っている修正中で未コミットなデータへの結合演算などは想定していない．

Google Fusion Tables [12] は，Google Docsに保管しているテーブル形式データを扱い，ユーザ間でのデータ共有も可能である．ユーザは，見ているテーブルに対して属性を選び，別のテーブルを左結合することができる．また，各種データ形式に応じて，地図や棒グラフなどの表示形式も選ぶことができる．結合データをテーブル形式で表示した際には，ビュー上において属性値のデータ編集が可能であるが，結合条件を指定した属性値の編集はできない．また，左テーブルの要素に関連する右テーブルの要素がない場合，その要素の生成を行うことができない．

7.3 Materialized View

Materialized Viewとは，データベースに対するクエリ結果を実体化し，元のテーブルが更新されるたびに，実体化したビューの更新を行うというものであり，検索クエリの頻繁に行われる場合などに有効な手法である．古くから研究されている分野であり，文献 [13] では，ビューに対する更新差分の効率的な計算法などが示されている．

提案している結合ライブラリもある種のMaterialized viewであり，更新差分と入力テーブルとの結合演算をWeb API呼び出しとして行っているともいえる．一方で，本研究ではWeb API呼び出しを抑えることが重要となるため，内部にキャッシュを保持し，ランタイム処理によってクエリ発行回数を抑えるなどのクライアントサイドWebアプリケーション実行環境特有の問題に重点を置いている．

また，結合ビューには更新可能性問題が知られている [1]．ビューへの更新操作に対して，その更新内容を実現するような，結合元テーブルへの更新が存在し，かつ一意に定まるかという問題である．更新内容以上の変更をビューに対して加えないというのが原則である．一方で，提案ライブラリでは，結合ビューを介しての更新操作は，対応するソース要素への更新操作と意味づけており，その更新内容を結合ビューに反映させることで，結合ビュー上の操作対象部分以外も更新されることを前提としている．これは，一連の更新内容を実施するのではなく，細かな更新内容ごとに結合ビューの更新を行い，最後にサーバに対してコミットするというスタンスの違いに起因している．

7.4 Joined View Editor

我々のグループでは，文献 [3] においてJoined View Editorという，複数のWebサービスに対する左結合ビューを提供し，加えて結合ビュー上でのソース要素への更新操作を許容するライブラリを提案してきた．本研究とは，データモデルなどで共通性が高い．ただし，当初は単一の結合ビュー内の更新および即時反映のみを考慮しているため，

4.1 節のケース (イ) を想定しない（5.3 節の議論により禁止されている）デザインとなっている．また，ケース (ウ) も考慮されておらず，ケース (ア) への対処のみが行われていた．プロキシにおける Web API リクエストの抑制についても，5.4 節のような取り組みは行っておらず，各最左要素ごとに Web API リクエストを発行し，同一クエリに関してのみ前回の結果を再利用する実装となっていた．

つまり，文献 [3] で提案した更新可能な結合ビューのアイデアを，汎用的に利用可能なライブラリとするための実装上の課題を解決し，加えて，結合演算時の Web API リクエスト削減機能を導入したのが本論文の貢献となっている．

8. まとめ

本論文では，データ更新を即時反映可能な結合ビューライブラリの実現にむけ，ライブラリデザインの見直し，要素データ更新に対するモデル化およびプロキシデザインの見直しを行い，システム実装への反映・改善を行った．加えて，結合演算実現にあたって無駄な Web API アクセスを抑制するため，取得済みの検索クエリ結果の再利用や，複数の検索クエリに対して論理和による取りまとめを行える仕組みを，プロキシに導入した．性能評価では，データ更新時に新たな検索クエリを発行するまでの時間は 80 msec 程度以内であり，また，Web クエリの発行抑制の仕組みについても，十分機能していることが示された．

本論文では，主にクライアント側に起因するデータ更新を想定して，アプリケーション事例などの説明を行ってきたが，本技術自体は，プッシュ系 Web サービスによるデータ更新にも適用可能であり，今後，より多くのアプリケーション事例を通してその有効性を評価していきたい．

参考文献

[1] Furtado, A.L. and Casanova, M.A.: *Query Processing in Database Systems*, chapter Updating Relational Views, pp.127–142, Springer (1985).

[2] Sencha Inc.: Sencha Docs Ext JS 4.0, available from ⟨http://docs.sencha.com/ext-js/4-0/⟩.

[3] Kumagai, Y., Senba, M., Nagamine, T. and Kamada, T.: Joined View Editor for Mashups of Web Data Stores, *SNPD2012* (2012).

[4] tomiokamada: tomiokamada/ljoinJS Wiki (2014), available from ⟨http://github.com/tomiokamada/ljoinJS/wiki⟩.

[5] Yahoo!: Yahoo! Pipes (2007), available from ⟨http://pipes.yahoo.com/pipes/⟩.

[6] Microsoft: Microsoft Popfly (2007), available from ⟨http://www.popfly.com/⟩.

[7] Miyagawa, T.: Plagger (2006), available from ⟨http://plagger.org/⟩.

[8] Ennals, R., Brewer, E., Garofalakis, M., Shadle, M. and Gandhi, P.: Intel Mash Maker: Join the Web, *SIGMOD Record*, Vol.36, No.4, pp.27–33 (2007).

[9] Ennals, R. and Gay, D.: User-friendly functional programming for web mashups, *Proc. ICFP '07*, ACM, pp.223–234 (2007).

[10] Ikeda, S., Nagamine, T. and Kamada, T.: Application Framework with Demand-Driven Mashup for Selective Browsing, *Journal of Universal Computer Science*, Vol.15, No.10, pp.2109–2137 (2009).

[11] Microsoft: Edit data in a query (Applies To: Access 2007), available from ⟨http://office.microsoft.com/en-us/access-help/edit-data-in-a-query-HA010097876.aspx⟩.

[12] Google: Google Fusion Tables (2006), available from ⟨http://www.google.com/fusiontables/Home/⟩.

[13] Blakeley, J.A., Larson, P.-A. and Tompa, F.W.: Efficiently Updating Materialized Views, *Proc. 1986 ACM SIGMOD International Conference on Management of Data*, pp.61–71 (1986).

畑谷 卓哉

2014 年神戸大学工学部情報知能工学科卒業．現在，同大学大学院システム情報学研究科情報科学専攻在学中．Web アプリケーションにおける情報連携について研究を行う．

熊谷 良夫

2011 年神戸大学工学部情報知能工学科卒業．2013 年同大学大学院システム情報学研究科修士課程修了．現在，株式会社アルファシステムズに勤務．

鎌田 十三郎　（正会員）

1993 年東京大学理学部情報科学科卒業．1998 年同大学大学院理学系研究科情報科学専攻博士課程単位修得退学．博士（工学）．神戸大学助手，助教を経て，2010 年より同大学大学院システム情報学研究科講師．並列・分散計算，言語処理系等の研究に従事．

（担当編集委員　渡辺 陽介）