UNIVERSITY OF CALIFORNIA SAN DIEGO

Data Management Issues and Optimizations in an Ajax Application Framework

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Keliang Zhao

Committee in charge:

Professor Alin Deutsch, Chair
Professor Yannis Papakonstantinou, Co-Chair
Professor Michael Carey
Professor Sujit Dey
Professor Ranjit Jhala

2018

The Dissertation of Keliang Zhao is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California San Diego

2018

DEDICATION

To my parents Fangqiang Zhao and Ling Fang, my family and friends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Chapter 1 contains material from “The SQL-based all-declarative FORWARD web application and development framework” by Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Michalis Petropoulos, The Fifth Biennial Conference on Innovative Data Systems Research, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 2 contains material from “Utilizing IDs to Accelerate Incremental View Maintenance” by Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao, ACM SIGMOD Conference, 2015. The dissertation author was the primary investigator and author of this paper.

VITA

2007	Bachelor of Science, The Hong Kong University of Science and Technology
2007-2012	Graduate Student Researcher, University of California San Diego
2011	Master of Science, University of California San Diego
2018	Doctor of Philosophy, University of California San Diego

ABSTRACT OF THE DISSERTATION

Data Management Issues and Optimizations in an Ajax Application Framework

by

Keliang Zhao

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Alin Deutsch, Chair
Professor Yannis Papakonstantinou, Co-Chair

Since 2005, with Ajax functionality becoming a de-facto requirement of cloud-based applications, a developer needs to integrate a plethora of languages in order to deliver even simple data-centric web applications: SQL to access the database, HTML and JavaScript for browser-side interactions, and yet another programming language (e.g. Java) for server-side business logic. Even worse, most of the JavaScript and Java is tedious, imperative code to perform mundane, low-level data integration, coordination and access optimization problems, which result either from language and data heterogeneities, or from the distributed programming necessitated by multiple computation points (browser, application server, database server).

FORWARD is a data-centric, declarative web application development framework that uses SQL++, a minimally enhanced version of SQL as its core programming model. Data from SQL queries can be styled directly with a data-driven page configuration that allows HTML templates and pre-packaged Ajax units (e.g. maps, charts, calendars), thus leading to Ajax pages that are automatically rendered and refreshed. In the majority of applications, no Java or JavaScript is needed.

A central contribution is the provision of a unified, SQL-accessible application state that captures the application's database along with typical web application programming data spaces, such as session and request data. The unified state includes a logical representation of the core page data pertinent to the business logic, thus allowing the SQL-based business logic to easily access page data seamlessly with the rest of the state.

This dissertation also discusses a few key optimization and automation components of FORWARD, which are enabled by FORWARD's declarative approach. FORWARD mapping framework can express SQL++ queries found common in transforming data from one schema to another, and is used extensively in the page layer. A novel ID-based incremental view maintenance (IVM) system takes an algebraic approach and expresses data diffs with IDs, so that diff computation is accelerated as data access is minimized. Furthermore, the IVM system is extended to handle SQL++ query language and schema-less data model.

Chapter 1

FORWARD Web Application Framework

1.1 Introduction

The vast majority of database-driven web applications perform, at a logical level, fundamentally simple INSERT / UPDATE / DELETE commands. In response to a user action on the browser, the web application executes an action (i.e., a program) that transitions the old state to a new state. The state is primarily persistent and often captured in a single database. Additional state, which is transient, is maintained in the session (e.g., the identity of the currently logged-in user, her shopping cart, etc.) and the pages. The action performs a series of simple SQL queries and updates, and decides the next step using simple if-then-else conditions over the state. The changes made on the transient state, though technically not expressed in SQL, are also computationally as simple as basic SQL updates.

Despite their fundamental simplicity, creating web applications takes a disproportionate amount of time, which is expended in mundane data integration and coordination across the three layers of the application: (a) the visual layer on the browser, (b) the application logic layer on the server, and (c) the data layer in the database.

Challenge 1: Language heterogeneities.

Each layer uses different and heterogeneous languages. The visual layer is coded in HTML / JavaScript; the application logic layer utilizes Java (or some other language, such as PHP); and the data layer utilizes SQL. Even for pure server-side / pure HTML-based applications,

the heterogeneities cause impedance mismatch between the layers. They are resolved by mundane code that translates the SQL data into Java objects and then into HTML. When the front end issues a request, code is again needed to combine memory-residing objects of the session and the request with database data. Consequently, developers write a lot of "plumbing" code.

Challenge 2: Updating Ajax pages with event-driven imperative code.

Since 2005, Ajax led to a new generation of web applications characterized by user experience commensurate to desktop applications. The heavy usage of JavaScript code for browser-side computation and browser / server communication leads to superior user experience over pure server-side applications, comprising

- performance gains through partial updates of the page
- more responsive user interfaces through asynchronous requests, and
- rich functionality through various JavaScript component libraries, such as maps, calendars and tabbed dialogs.

For example, consider the web application page of Figure 1.1, where an Ajax maps component shows markers of restaurants. When a user clicks on a marker, a dialog shows up for the user to submit restaurant reviews. The user can also read other users' review comments and ratings which are displayed in bar charts. The interactive maps component is not available for pure server-side applications, since it requires client-side computation and asynchronous requests. The best that a single server-side model can provide for this kind of applications is showing restaurants as a list or table. Also, in the pure server-side model, submitting a review for a single restaurant will cause the entire page to be recomputed. Indeed, the server may issue queries for all the restaurants and reviews on the page, not only for the particular restaurant being reviewed. Furthermore, the browser will block synchronously and blank out while it waits for the new page from the server. Finally, various aspects of the browser state, including data of

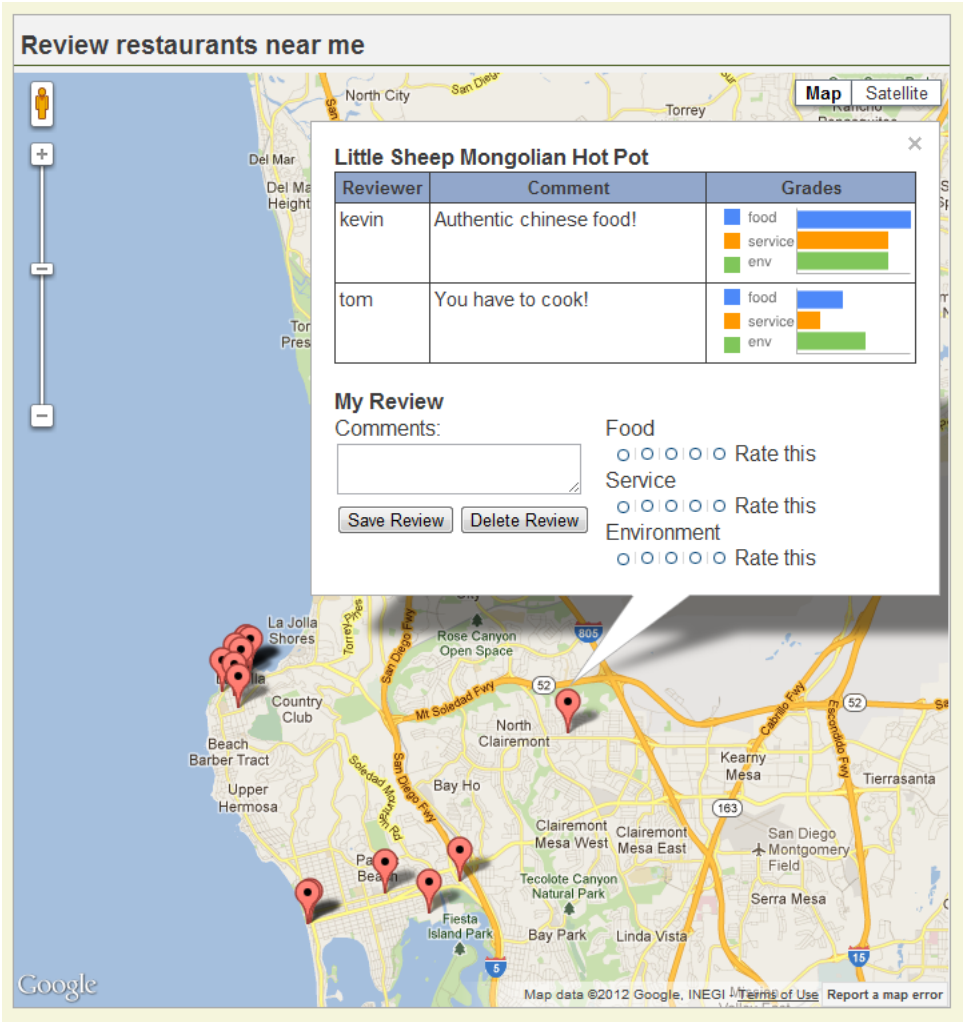


Figure 1.1. The `ReviewRestaurants` page of the running example

non-submitted form elements, cursor positions, scroll bar positions and the state of JavaScript components will be lost and re-drawn, thus disorienting the user.

For an Ajax page, however, a developer will typically optimize his code to realize the benefits of Ajax and solve the pure server-side problems listed in the previous paragraph. The same user action (e.g., the review submission) causes the browser to run an *event handling* JavaScript function collecting data from the page's components relevant to the action (i.e. the restaurant id, the review comment and ratings), and send an asynchronous *Xml Http Request (XHR)* with a *response handler* callback function specified. On the server, the developer implements queries that only compute the changed data (i.e. the newly created row of review with a bar chart), to take advantage of more efficient queries, as well as to conserve memory and bandwidth. While the asynchronous request is being processed, the browser keeps showing the old page instead of blanking out, and even allows additional user actions and consequent requests to be issued. When the browser receives the response, the response handler uses it to partially update the page's state. The partial update retains non-submitted forms, scroll bar positions, etc., therefore allowing the user to retain his visual anchors on the page.

The page state primarily consists of the browser DOM, which captures the state of HTML form components such as text boxes and radio buttons, and the state of the JavaScript variables, which are often parts of third-party JavaScript components. Therefore, the developer implements the response handler by writing imperative code that navigates the DOM and JavaScript components, and invokes JavaScript methods causing the DOM and components to incrementally render to the browser.

The Ajax optimizations demand a serious amount of additional development effort. For one, realizing the benefits of partial update requires the developer to program custom logic for each action that partially updates the page, which was not the case in pure server-side programming. In particular, in a pure server-side implementation, the developer needs to write code for the effect of each individual action on the database, but writes only one piece of code that generates the page according to the database state and session state. For example, notice that

the page of Figure 1.1 also provides a "Delete Review" button. The developer will have to write two pieces of code that modify the database when "Submit Review" is clicked and when "Delete Review" is clicked, respectively. The former issues either an INSERT or an UPDATE command while the latter issues a DELETE command. Both of them share the same piece of code that generates the page showing the list of restaurant markers, their review comments and ratings. This piece of page computation code is independent of what user action caused the re-generation of the page.

In contrast, in an Ajax application, each user action needs its own code to partially update the page. This piece of code consists of server-side code that retrieves a subset of the data needed for the page update, and browser-side JavaScript code that receives the data and re-renders a sub-region of the page. In the running example of Figure 1.1, different pieces of code would be needed for the "Submit Review" and the "Delete Review" buttons.

Such event-driven programming (which also occurs in Flash etc.) is well-known to be both error-prone and laborious [40], since it requires the developer to correctly assess the data flow dependencies on the page, and write code that correctly transits the application from one consistent state to another. Moreover, in a time-sensitive collaborative application (similar to Google Spreadsheet) where many users work concurrently, these dependencies may extend beyond the page of the user who submitted the review, and into the pages of other users who are viewing the same restaurants on their browsers.

Further compounding the custom logic required for each action is the amount of imperative code that needs to be implemented on the browser. Whereas the developer of a pure server-side application needs to understand only HTML, the developer of an Ajax application needs to integrate JavaScript as yet another language, understand the DOM in order to update the displayed HTML, and write code that refreshes the JavaScript components' state based on the nature of each partial update. Since there is no standardization between the component interfaces of different third-party libraries, the developer is left to manually integrate across these disparate component interfaces.

Challenge 3: Distributed computations over both browser-side and server-side states.

In response to a user action, the browser sends an HTTP request to the server and activates an action, which needs access to relevant data on the page in order to perform computations that involve such page data and the database. Writing code that involves both page data and server-side data was already mundane and time-consuming in pure server-side applications and became even more so in Ajax and Flash.

In a pure server-side application, the browser is essentially stateless since all state is lost when the new page is loaded. Using HTML markup such as `<input type="text" name="save_comment" />`, the developer declaratively specifies that the value collected by the text box will appear as the parameter `save_comment` in the HTTP request. The good news about pure server-side programming is that when a user action causes an HTTP request, the browser is responsible for navigating the DOM, and marshalling the request parameters according to the HTML specification. The bad news is that, on the server-side, the application first unmarshals the request parameters by using Java (or PHP, etc.), and then typically issues SQL queries where the request parameters become parameters of an SQL statement. Overall, lines of Java code are expended in such trivial "extract parameter from the request, plug it in the query" tasks. With Ajax it gets much worse, as discussed next, since the marshalling of request parameters is no longer automatic.

In particular, in an Ajax application the browser maintains its state across HTTP requests. Consequently, the state of the web application becomes distributed between the browser and the server. The developer is responsible for defining a custom marshalling format for the XHR request, typically in XML or JSON, and for writing imperative code to navigate over the DOM and marshal the relevant page data that must be sent to the server along with each HTTP request. The usage of JavaScript components (calendars, etc.) on the page further complicates the issue by requiring custom code that converts between the state of the component and the marshalling format. On the server-side, the developer writes custom code to unmarshal the request parameters, and then continues along the usual path, plugging such parameters into SQL statements.

A select few web application frameworks, such as Echo2 [19] and Microsoft's ASP.NET [4], mitigate the issue of distributed application state by automatically maintaining a mirror of the browser state on the server. Such synchronization can occur efficiently, using reduced bandwidth, by having the server send to the browser only the difference between the previous page state and the new page state. Yet, mirroring is only a half-solution, since the mirror made available on the server contains the exact and full state of the browser, despite the fact that each request cares about a different subset of page data. For example, the submission of a review for one restaurant of Figure 1.1 requires the data collected by the text box and sliders in the restaurant's pop-up dialog, while the submission of a review for another restaurant requires the data from another pop-up dialog. Furthermore, the mirrored browser state include visual styling details, which do not matter when logical form data is collected, yet they trouble collection. For example, to read the rating values of food, service and environment from the three sliders in Figure 1.1, using a mirror-based Ajax framework, the developer will need to navigate through its ancestor HTML elements (e.g., `<div>` and `<table>`) that are used purely for visual layout. The net effect of these issues is that the developer's code includes many mundane lines that navigate around the extraneous information in order to obtain the relevant data for the request.

1.1.1 FORWARD's Declarative Solution

The data management field has recently applied with success and great promise declarative data-centric techniques in network management and games. In a similar fashion, FORWARD adopts an SQL-based, declarative approach to Ajax web application implementations, going beyond prior approaches such as Strudel [21] and WebML [14] that focused on pure server-side data publishing applications. In particular, FORWARD removes the great amount of Java and JavaScript code, which is written to address the challenges above, and replaces them with the use of SQL-based languages to facilitate integration and enable automatic optimization. The

objective is to "make easy things easy and difficult things possible".[1]

FORWARD is a rapid web application development framework. The web application's pages are declaratively specified using *page configurations*. The actions that run when a request is issued are also declaratively specified, using *action configurations*. Both page configurations and action configurations are based on a minimally enhanced version of SQL, called *SQL++*, which provides access to the *unified application state* virtual database that, besides the persistent database of the application, includes transient memory-based data (notably session data and the page's data). The application runs in a continuous action-page cycle: An HTTP request triggers the FORWARD interpreter to execute an action configuration. The action reads and writes the application's unified state and possibly invokes functions that have side effects beyond the unified application state. (e.g., send an email). The action typically ends by identifying which page will be displayed next. FORWARD's interpreter creates a new page according to the respective page configuration. The page configuration also specifies actions that are invoked upon specified user events. The invocation of an action restarts the action-page cycle.

The key contributions of FORWARD are:

- The use of SQL++ allows unified access to browser data and server-side data, including both the database data and the application server's main memory data (e.g., session data). In conventional web application programming, such access would require Java and JavaScript. FORWARD eliminates Java and JavaScript from the majority of web applications, therefore resolving Challenges 1 and 3.

- The *page configurations* are essentially rendered views that visualize dynamic data generated by SQL++ and are automatically kept up-to-date by FORWARD. The configurations enable Ajax pages that feature arbitrary HTML and (pre-packaged) Ajax / JavaScript visual units (e.g. maps, calendars, tabbed windows), simply by tagging the data with tags such as

[1] This objective is not followed by today's web application development frameworks. Paradoxically, powerful Turing-complete low-level imperative languages (such as Java and PHP) accomplish tasks that can be easily accomplished by appropriate SQL-based declarative languages.

`<maps>`, `BarChart`, etc. The AJAX pages are automatically and efficiently updated by the FORWARD interpreter by appropriately extended use of incremental view maintenance. The FORWARD developer need not worry about coordinating data from the server to the page's Ajax components, which resolves a Challenge 2 problem.

- The business logic layer is specified by *action configurations*, where business logic decisions and the transfer of data between the database and services are expressed in FORWARD's *action language*, which is a subset of PL/SQL with SQL++ extensions. The action configurations have easy unified access to both the browser data and the database, because FORWARD guarantees that the browser's page data is correctly reflected into the unified application state so that it can be used by the action. The automatic reflection resolves Challenge 3.

- The page and action configurations have unified access to the persistent database, the page and the session via a single language (SQL++), therefore resolving Challenge 1. JavaScript needs to be written only if one needs to create a custom visual unit. Java needs to be written only for computations not easily expressible in SQL. For example, one can build the entire Microsoft CMT using the SQL++ based page and action configurations, except for the reviewer/paper matching step, which requires a Java-coded stable matching algorithm to assign papers to reviewers according to their bids.

1.2 Creating an Application

A developer creates an application by providing to the FORWARD interpreter source & schema configurations, page configurations and action configurations. Figure 1.2 is a high-level architecture diagram of FORWARD where developer-provided application specifications are marked in blue color.

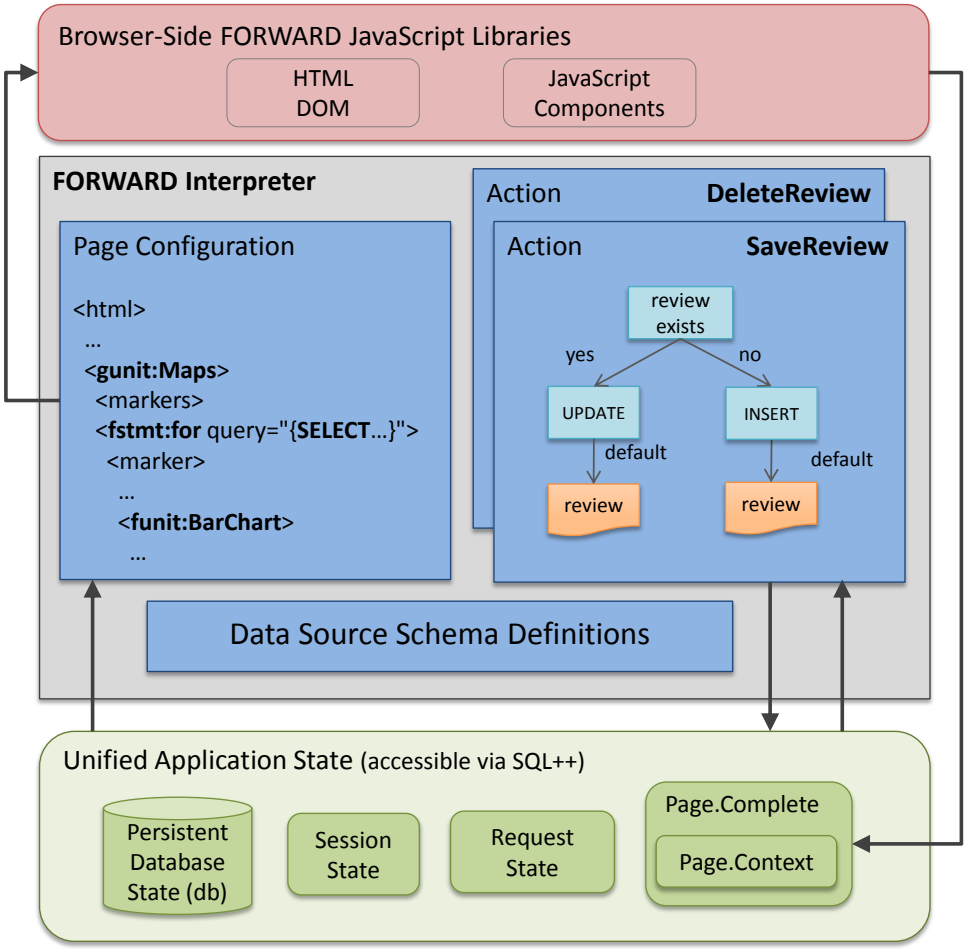


Figure 1.2. High level architecture of FORWARD interpreter and application specifications

Action-Page Cycle

In the spirit of MVC-based frameworks such as Struts and Spring, a FORWARD application's operation is explained by action-page cycles. Figure 1.3 shows the action-page cycle at a high level. An HTTP request triggers the interpreter to run the action configuration that is associated with the request's URL. Before the action is run, if it is triggered from an existing page, the page's visual state is synced at the server side and becomes part of the unified application state (step 1). The visual page state is used to derive the latest logical page states. Then the action reads and writes the application's unified state (e.g., database, session data, page data, etc.) and possibly invokes functions that have side effects beyond changing the application's state (step 2). The action's run typically ends with identifying which page p will be displayed next. Conceptually, FORWARD's interpreter creates a new page according to p's page configuration, which may be thought of as a rendered SQL++ view definition, and displays it on the browser (steps 3 and 4). A displayed page typically catches browser events (such as the user clicking on a button, mousing over an area, etc.; or a browser-side timer leading to polling) that lead to action invocations (via http runs), therefore continuing the action-page cycle. More details of each step will be explained in Section 1.3.1.

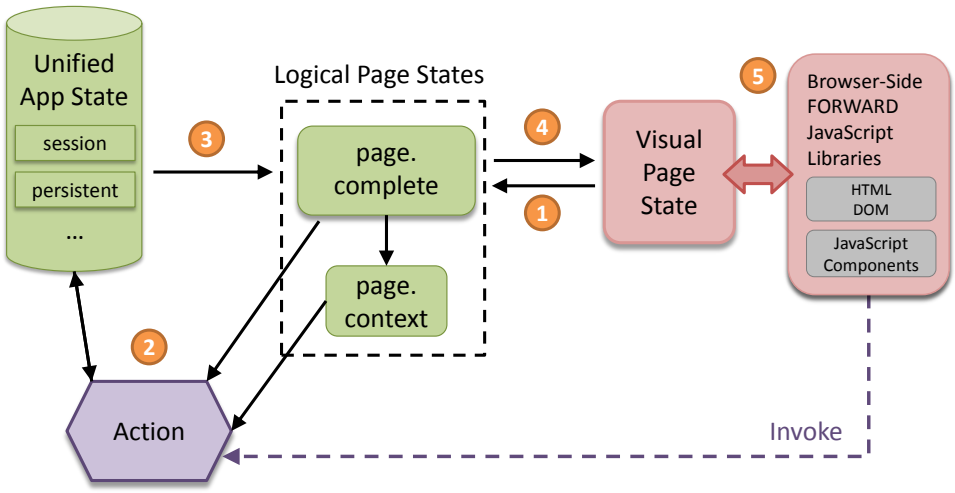


Figure 1.3. Action-page cycle (without incremental view maintenance illustrated)

Notice that FORWARD enforces the full separation of the Controller functionality from

the View functionality, which current MVC frameworks only encourage but do not enforce: The page configurations are literally views, unable to side effect the application state. This is marked in Figure 1.2 by a single arrow from the unified application state to the page configuration meaning that page computation can read from (but not write to) the unified application state, unlike action configurations which have both to and from arrows with the unified application state.

In the rest of this section, we will introduce the running example in Section 1.2.1. Unified application state, as one of the corner stones of FORWARD, is formally introduced in Section 1.2.2 with data source and schema configurations. The page configuration which can access any part of the unified application state to compute a FORWARD page is discussed in Section 1.2.3. Section 1.2.4 explains an important integration feature of the automatic inference of several page data objects. These page data objects could be accessed by page and action configurations. Finally, the action configuration is explained in Section 1.2.5.

1.2.1 Example Application

The running example of this section is a restaurant reviewing Ajax application. The paper focuses on its `ReviewRestaurants` page (see Figure 1.1), where the user submits reviews that consist of a comment collected using a text box, and food, service and environment ratings collected using Javascript components. For each existing review, a bar chart component is used to display the three rating values.

The full version of the reviewing application is available online at [22]. Online instructions on demo.forward.ucsd.edu also teach how to create your own FORWARD application.

The implementation of the discussed `ReviewRestaurants` page requires 179 lines of FORWARD page configuration and the implementation of `SaveReview` action requires 27 lines of FORWARD action configuration.

1.2.2 Unified Application State: Sources and Schemas

The FORWARD *unified application state* (UAS) includes data objects from different data sources. A *data source* is configured with the type of the source (e.g., relational database, LDAP server, spreadsheet) and how to connect to them. As a convenience for the development of cloud-based applications and databases, FORWARD implicitly provides to each application an SQL++ database source named `db`. In the running example, `db` provides the full persistent data storage of the application.

A FORWARD application's actions and pages also have access to the session, request and page sources which are main-memory based. In the spirit of the in-memory data storage provided by application servers, data in these data sources have limited lifetimes and there may be more than one instances of them at any time. For example, the `session` source lives for the duration of a session. All the pages of a browser session and all the actions initiated by HTTP requests of the browser session have access to the same instance of the `session` source, while pages and actions of other browser sessions have their own `session` instances. Similarly the `request` source lives for the duration of processing an HTTP request by an action (as discussed in Section 1.2) and each in-progress action has its own `request` instance. A `page` source lives for the duration that a page instance is displayed on a specific browser window during a specific session.

Each source stores one or more objects. Each object has a *name*, a *schema* and *data*. The schema may be explicitly created with the Data Definition Language (DDL) of SQL++ or may be imported from the source. For example, the schema of a relational database source is imported from its catalog tables. In order to accommodate the needs of pages, of session data and of other data typically occurring in web application programming, SQL++ is a minimal extension of SQL, whereas the root of a schema is either a tuple or a table. An attribute of a tuple may be either a scalar type or a table, whose tuples have attributes that may recursively contain nested tables. Notice that standard SQL corresponds to the case where a schema is simply a

table and the table's tuples may only have scalars (as opposed to nested tables). In order to allow variability in the spirit of XQuery and OQL, SQL++ also supports a *switch* type, which can contain multiple *case* tuples of different types. An instance of a switch type can have at most one case tuple instantiated.

An example of a schema that uses the extra features of SQL++ is the session, which in the running example is simply a tuple containing only the standard scalar attributes `session_id`, `user_id` and `role` that are set by FORWARD's session management and authentication and authorization utilities (not shown). A key integration contribution of FORWARD, which attacks Challenges 1 and 3, is the ability of SQL++ to combine persistent data with transient data using just a single SQL++ query. An example query will be introduced in Section 1.2.3 as part of the page configuration.

1.2.3 The Page Configuration

A page configuration is an XHTML file with added:

1. FORWARD *units*, which are specified as XML elements in the configuration and are rendered as maps, bar charts, and other Javascript-based components. Internally FORWARD units use components from Yahoo UI, Google Visualization and other libraries and wrap them so that they can participate in FORWARD's pages without any Javascript code required from the developer.

2. SQL-based inlined *expressions* and `for` and `switch` *statements*, which are responsible for dynamic data generation.

Figure 1.4 and Figure 1.5 provide the page configuration of the running example's `ReviewRestaurants` page. Figure 1.5 excludes a few parts, which are marked by `<!-- skipped -->` and can be found on the online demo. The complete page configuration's size is 179 lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns:funit="http://forward.ucsd.edu/units"
      xmlns:fstmt="http://forward.ucsd.edu/statements">
 <head>
  <title>Review Restaurants</title>
 </head>
 <body>
  <fstmt:switch>
   <fstmt:case condition="{session.current.role='admin'}">
    <a href="AdminHome">Go to Admin Home</a>
   </fstmt:case>
  </fstmt:switch>
  <div id="main-content">
   <h2>Review restaurants near me</h2>
   <gmaps:Maps height="700px" width="700px">
    <options pancontrol="false">
     <zoom name="zoom_input" carryover="true"/>
     <center>
      <direct>
       <latitude name="lat_input" carryover="true"/>
       <longitude name="long_input" carryover="true"/>
      </direct>
     </center>
    </options>
    <markers>
     <fstmt:for>
      <fstmt:query name="restaurants">
       SELECT p.id AS pid, p.name AS pname, p.geometry.location.lat AS lat,
              p.geometry.location.lng AS lng
       FROM db.places AS p
      </fstmt:query>
      <marker draggable="false">
       <position>
        <direct>
         <latitude>{lat}</latitude>
         <longitude>{lng}</longitude>
        </direct>
       </position>
       <infowindow>
        <content>
         <fesc:html>
          {hidden save_pid : pid}
          <div>
           <b>{pname}</b>
          </div>
          <table>
           <tr>
            <th>Reviewer</th>
            <th>Comment</th>
            <th>Grades</th>
           </tr>
           <fstmt:for>
            <fstmt:query>
             SELECT r.comment AS comment, r.uid AS uid, r.place AS rpid,
                    r.food as food, r.service as service, r.env as env
             FROM db.reviews AS r
             WHERE r.place = pid AND r.uid=session.current.user_id
            </fstmt:query>
            <tr>
             <td>{uid}</td>
             <td>{comment}</td>
             <td>
```

Figure 1.4. The page configuration of `ReviewRestaurants` - Part 1

```
62              <td>
63               <funit:BarChart><bars>                                    (A)
64                <bar bar_value="{food}" label="Food" />
65                <bar bar_value="{service}" label="Service" />
66                <bar bar_value="{env}" label="Environment" />
67               </bars></funit:BarChart>
68              </td>
69             </tr>
70            </fstmt:for>
71           </table>
72           <br />
73           <div>
74            <b>My Review</b>
75           </div>
76           <div>
77            <div>
78             <div>
79              Comments:
80              <br />
81              <hunit:textarea rows="2" columns="20">
82               <value name="save_comment">{null}</value>
83              </hunit:textarea>
84             </div>
85            </div>
86            <div>
87             Food
88             <cunit:Ratings>
89              <selected_value name="save_food">{null}</selected_value>
90              <ratings_ranks>
91               <rank>
92                <value>1</value>
93                <label>Awful</label>
94               </rank>
95               <rank>
96                <value>2</value>
97                <label>Bad</label>
98               </rank>
99               <rank>
100               <value>3</value>
101               <label>Average</label>
102              </rank>
103              <rank>
104               <value>4</value>
105               <label>Good</label>
106              </rank>
107              <rank>
108               <value>5</value>
109               <label>Excellent</label>
110              </rank>
111             </ratings_ranks>
112            </cunit:Ratings>
113            Service
114            <cunit:Ratings>
115             <selected_value name="save_service">{null}</selected_value>
116             <ratings_ranks>
117              <!-- skipped -->
118             </ratings_ranks>
119            </cunit:Ratings>
120            Environment
121            <cunit:Ratings>
122             <selected_value name="save_env">{null}</selected_value>
123             <ratings_ranks>
124              <!-- skipped -->
125             </ratings_ranks>
126            </cunit:Ratings>
127            <hunit:button onclick="AJAX SaveReview()" value="Save Review" />
128            <hunit:button onclick="AJAX DeleteReview()" value="Delete Review" />
129 <!-- skipped closing tags -->
```

Figure 1.5. The page configuration of `ReviewRestaurants` - Part 2

Lines 13-15 of the page configuration list the HTML that generates the top of the page and contains the FORWARD unit `gmaps:Maps`. FORWARD units are encapsulated and expose their states in the SQL++ data model. For page configuration, a unit's interface is represented in an XML format so that it can be easily incorporated with the rest of the page's XHTML content. Figure 1.6 shows the DTD of the Maps unit's definition. The details of unit definitions and their DTD are discussed in Section 1.3.3. Notice that a unit may contain other units and XHTML, which may, in turn, contain other units. For example, the `gmaps:Maps` in Figure 1.4 contains XHTML (line 41) for the content of the popup dialog of its markers. The XHTML contains several other units, including `funit:BarChart` (line 63), `hunit:textarea` (line 81), and `cunit:Ratings` (line 90).

```
<!DOCTYPE Map [
 <!ELEMENT Map ( options, markers? )>
  <!ATTLIST Map style string NULL>
  <!ATTLIST Map height string NULL>
  <!ATTLIST Map width string NULL>

  <!ELEMENT options (zoom?, center)>
    <!ELEMENT zoom (integer) 1>
    <!ELEMENT center (direct, indirect?)>
      <!ELEMENT direct (latitude?, longitude?)>
        <!ELEMENT latitude (double) 0.0>
        <!ELEMENT longitude (double) 0.0>
      <!ELEMENT indirect (address?, region?)>
        <!ELEMENT address (string) NULL>
        <!ELEMENT region (string) NULL>

  <!ELEMENT markers (marker*)>
    <!ELEMENT marker (position, infowindow?)>
      <!ELEMENT position (direct, indirect?)>
        <!ELEMENT direct (latitude?, longitude?)>
          <!ELEMENT latitude (double) 0.0>
          <!ELEMENT longitude (double) 0.0>
        <!ELEMENT indirect (address?, region?)>
          <!ELEMENT address (string) NULL>
          <!ELEMENT region (string) NULL>
      <!ELEMENT infowindow (content)>
        <!ELEMENT content ANY_UNIT>
]>
```

Figure 1.6. DTD syntax of Maps unit (a minimized version)

To generate dynamic data for FORWARD pages' XHTML and units, a `fstmt:for`

statement evaluates its query and for each tuple in the result (conceptually) outputs an instance of its body configuration, i.e., of the XHTML or unit tags within the opening and closing `fstmt:for` tags. For example, the `fstmt:for` on line 26 outputs for each restaurant one instance of the `marker` element on line 32 and its data. As another example, the `fstmt:for` on line 52 outputs for each nested review one HTML table row (i.e., a `<tr>` element) that contains the review's information.

A `fstmt:switch` statement brings variability and outputs the content of the first `fstmt:case` whose condition (specified in SQL++ again) evaluates to true. For instance, the example page uses a `fstmt:switch` at Line 8 to display a link only when the current login user is an admin.

FORWARD's page configurations enable nested, correlated structures on the pages. In particular, the query of a `fstmt:for` statement found within the body configuration of an enclosing `fstmt:for` may use attributes from the output of the query of the enclosing `fstmt:for`. For example, the table `db.reviews` has a foreign key `restaurant`. For each "restaurant" instance the correlated query on line 53 produces a list of its reviews by using the `pid` attribute of its enclosing query.

Expressions, which are enclosed in { }, can reference attributes of the query's output. Furthermore, an expression may be itself a query. In the interest of flexibility, which has been the norm in tools and languages for web page creation,[2] FORWARD coerces the types produced by the expressions to the types required by the FORWARD units or XHTML, depending on where the expression appears. Therefore, the developer need not worry about fine discrepancies between the types used in the database (which are dictated by the business logic and are often constrained) and the types used for rendering, which are often as general as they can be. For example, the expression that feeds the `bar_value` attribute of the `funit:BarChart` on line 64 is an integer attribute. However, it is coerced into a float, which is the type of argument that the `funit:BarChart` expects.

[2]For example, JSP pages convert JSP expressions into strings whenever possible.

Query expressions can access any data objects in the unified applications state. The expression {`session.current.role`} on line 9 is a SQL++ query accessing the attribute `role` of the `current` tuple-typed data object in the `session` scoped data source. For example, consider the query on lines 54 that combines `session.current.user` with the table `reviews` of the persistent schema `db` to produce the reviews of the currently logged-in user in just four lines of SQL. More integration contributions are made by the page data objects, discussed in Section 1.2.4.

The syntax and semantics of the `fstmt:for` and `fstmt:switch` statements are deliberately similar to the `forEach` and `choose` core tags of the popular JSP Standard Tag Library (JSTL) [35]. The same applies for FORWARD expressions and JSTL expressions. However, FORWARD's `fstmt:for` iterates directly over a query, whereas JSTL's `forEach` iterates over vectors generated by the Java layer, which are in turn produced by iterating over query results. Besides the obvious code reduction resulting from removing the Java middleware, FORWARD analyzes the queries behind the dynamic data of the page enables opportunities for automatic heterogeneity resolution and performance optimization, which are elaborated in later chapters.

The page as an automatically updated rendered view

Conceptually, the page configuration is evaluated after every action execution. While such an explanation is simple to understand, it is only conceptual. If the page that is displayed on the browser window before and after an action's execution is the same, then FORWARD will incrementally update only the parts of it that changed, therefore achieving the user-friendliness and efficient performance of Ajax pages, where the page does not leave the user's screen (and therefore avoids the annoying blanking out while waiting for the page to reload) but rather incrementally re-renders in response to changes. [24] explains how FORWARD utilizes incremental view maintenance in order to efficiently and automatically achieve pages as incrementally rendered views.

Summary

The page configuration is essentially an SQL view embedded into a visual template, consisting of HTML and unit tags. Therefore the page configuration resolves Challenge 1, since it enables the production of pages without requiring Java and Javascript code in addition to SQL. Furthermore, it is implemented as an Ajax page that is automatically updated to reflect the database state, therefore resolving Challenge 2 of Ajax application programming.

1.2.4 Page Data Objects in the Unified Application State

Transient data is heavily used in web application programming. A typical use case of it is to store a logical-level representation of page data, which captures user inputs (collected by HTML forms or Javascript components). In FORWARD, this logical-level representation is automatically created as part of the unified application state and belongs to the `page` data source.

A *page.complete* schema captures in an SQL++ schema the subset of page data that has been named by developers, using a special XML attribute `name` in the page configuration. Therefore the page.complete enables the developers to resolve part of Challenge 3, by enabling him to create a data structure encompassing the data of interest to the actions as opposed to, say, visual details. FORWARD infers the page.complete schema by inspection of the page configuration. In the running example, the page.complete simply contains a table of the restaurants that appear on the screen along with their reviews and ratings (see Figure 1.7). In particular, it is a table named `restaurants` (due to the `fstmt:for` on line 11) with a string attribute named `save_comment` (due to line 33) and numerical attributes `save_food`, `save_service` and `save_env` due to the ratings units (the last two not shown in Figure 1.4). The table `restaurants` also has the key attribute `restaurant_id` so that one can associate the data collected by the multiple instances of the textareas and the ratings units with restaurants. Mechanically, FORWARD infers this attribute to be the key of the query that feeds `restaurants` using a key inference algorithm (see Appendix B). Notice that such inference relies on the underlying `db.restaurants` table having a known key, which is an unavoidable assumption of

the running example, no matter what technologies one uses to implement it.

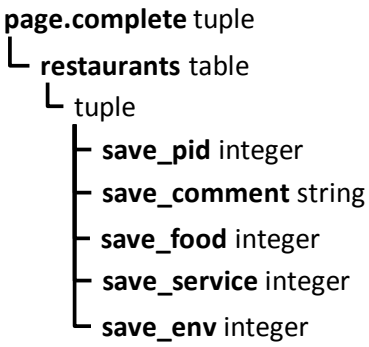


Figure 1.7. Page.complete schema of running example

The `name` attribute convention is reminiscent of the HTML standard's convention to allow a `name` to be associated with each form element and consequently generate request parameters with the provided names. Drawing further the similarities to HTML's request parameters, a *page.context* data object keeps only the tuple of the page.complete that corresponds to the invoked action. The page.context provides an important piece of data about which particular action was invoked. In the running example, the page invokes the action `SaveReview` (line 62), and therefore it is important to know upon invocation which one of the many instances of the `SaveReview` was invoked. There are as many instances as restaurants on the page. The page.context identifies the `restaurant_id` for the invoked `SaveReview` because all other restaurant tuples are excluded from it. Therefore in Figure 1.8, page.context is only a tuple, as opposed to the table in page.complete shown in Figure 1.7.

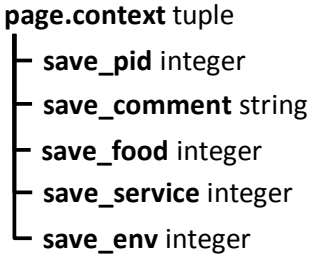


Figure 1.8. Page.context schema of running example

In fact, page.context can be computed from page.complete with action context informa-

tion. Figure 1.9 shows a mapping from page.complete to page.context as lines connecting source and target attributes (where dash arrow lines represent tuple mappings and solid lines represent scalar mappings). FORWARD uses an internal mapping framework tailored for such use cases during page compilation, and mappings are translated to SQL++ queries. The construction of the page.complete, the page.context and their mappings are discussed in Section 1.3.

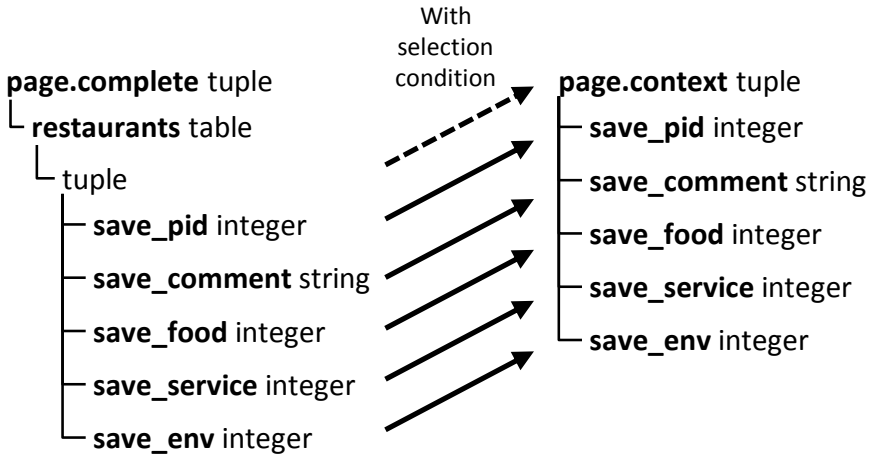


Figure 1.9. Internal mapping from page.complete to page.context

FORWARD guarantees that the page.complete and the page.context data is automatically up-to-date when an action starts its execution. This is a key contribution towards resolving Challenge 3. In a conventional Ajax application, Javascript and Java code has to be written to establish a copy of relevant page data on the server, in a way that they can be subsequently combined with the database.

Besides the difference between page.complete and page.context mentioned earlier, there is a notable heterogeneity between the logical page representation (i.e., page.complete and page.context) and the visual page representation. Notice that the page.complete and page.context schemas are decided by statements and queries of the page configuration, i.e. the page's logical aspects, while the unit structure and XHTML (the visual aspects) are not considered. The only unit aspect that matters is the types of data collected by the user, i.e., string from the `funit:textarea` and numbers from the `funit:Ratings`. This is a key advantage over page mirror-based frameworks, such as Microsoft's ASP.NET, that offer to the developer a server-side

mirror of the page data, so that the developer does not have to code in Javascript. Unfortunately, the structure of the mirror follows the (typically very busy) visual structure of the page, as opposed to the data structure that best fits the database (a typical "Challenge 3" problem). We will use an example to explain such structural heterogeneity.

Example showing logical and visual page heterogeneity

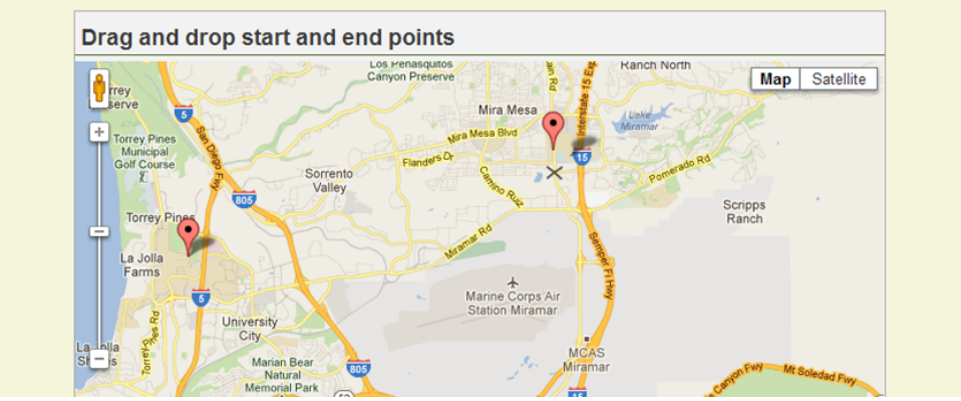


Figure 1.10. An application showing a heterogeneity between visual and logical page states

Consider an application that allows users to select a start and an end points by dragging and dropping two markers in the `Maps` component (see Figure 1.10). Logical representation of the page has exactly two markers (see page.complete in Figure 1.11), which are what the related actions care about (e.g., an action computing the distance between the two markers). The visual page, however, needs to conform to expected schemas of visual components. Here the map component can support a list of arbitrary number of markers, and hence the visual schema has to be in the form of a table of markers (see visual_page in Figure 1.11). This is a natural heterogeneity between logical and visual page that exist in web applications with visual components in general. In a manually built version, the developer would have to resolve it by specifying the computation to transform two distinct markers that are stored for business logic to a list of markers for visual components to display, and vice versa for collecting user input from the visual components and prepare it for related actions. In FORWARD, its resolution is handled by the framework automatically, again using the internal mapping framework. Figure 1.11 shows

the mapping from the page.complete to the visual page state at a high level. Essentially the mapping language supports mapping individual tuples to a table tuple. The mapping is invertible, meaning that changes to target data (i.e., visual state) can be propagated back to source data (i.e., logical state). Detailed explanation of this heterogeneity resolution is discussed in Section 1.3, where more page data objects will be revealed and the mappings are revised.

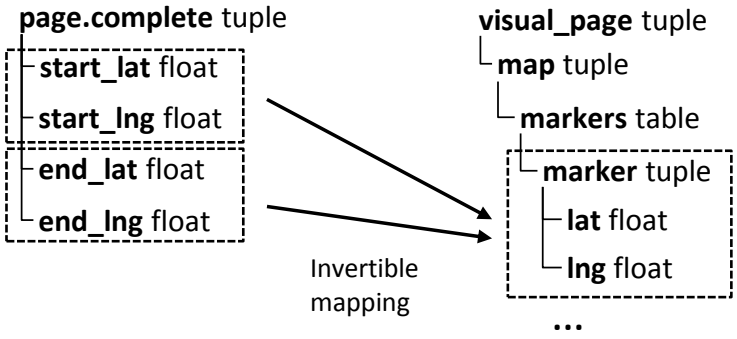


Figure 1.11. Logical & visual page states and their (abstract) mapping for the application shown in Figure 1.10

1.2.5 The Action Configuration

Actions of FORWARD applications can access and modify unified application state through SQL++ queries and commands, and invoke functions provided by the framework. Actions are specified in *FORWARD Action Language* which is a subset of Oracle PL/SQL. The action language supports If-Then-Else control flow logic, INSERT/UPDATE/DELETE DML statements, and calls to functions provided by the FORWARD framework.

In this section we illustrates a few key points of the action language's syntax and semantics using the `SaveReview` action configuration of the running example. Figure 1.12 shows the action configuration of `SaveReview`. The action issues either an UPDATE statement to update an existing review or an INSERT statement to insert a new review, depending on whether there already exists a review of the restaurant by the current user. Recall that the review form data in the context of the `SaveReview` action is retrieved by the framework and stored in page.context data object, which is part of the unified application state and has its schema inferred by the

framework. The UPDATE and INSERT statements in the action configuration can freely use any part of the unified application state, and in this example, they read from the page.context and modify data in the persistent database.

Besides issuing DML statements, actions can invoke functions. A very common FORWARD function used in actions is the built-in `next_page` function. Line 26 of Figure 1.12 specifies the next page to display at the end of the `SaveReview` action by calling the `next_page` function with a page name. This function call typically happens at the end of an action when the action leads to a (partial) page refresh, in which case the same page is specified, or navigation to another page. A function may also produce output in the SQL++ data model, although here the next_page function does not produce any output.

```
1   CREATE ACTION SaveReview() AS
2   BEGIN
3       IF exists(
4           SELECT *
5           FROM db.previews
6           WHERE place=page.context.save_pid
7                 AND uid=session.session.current_user
8       ) THEN
9           UPDATE db.previews
10          SET    comment=page.context.save_comment,
11                 food=page.context.save_food,
12                 service=page.context.save_service,
13                 env=page.context.save_env
14          WHERE  place=page.context.save_pid
15                 AND uid=session.session.current_user;
16      ELSE
17          INSERT INTO db.previews
18          TUPLE(session.session.current_user AS uid,
19                page.context.save_pid AS place,
20                page.context.save_comment AS comment,
21                page.context.save_food AS food,
22                page.context.save_service AS service,
23                page.context.save_env AS env);
24      END IF;
25
26      next_page('ReviewRestaurants');
27  END;
```

Figure 1.12. Action configuration of `SaveReview`

1.3 Internal Architecture and Implementation

In this section we discuss the internal architecture of FORWARD. First, Section 1.3.1 explains the action-page cycle with more internal architectural details. Then, Section 1.3.2 and Section 1.3.3 discusses the visual page layer and how its data aspect is modeled by the visual page state. The rest of this section discusses the page compilation process step by step, covering the inference algorithms of various page states, and the resolution of heterogeneities between them.

1.3.1 Action-Page Cycle in Detail

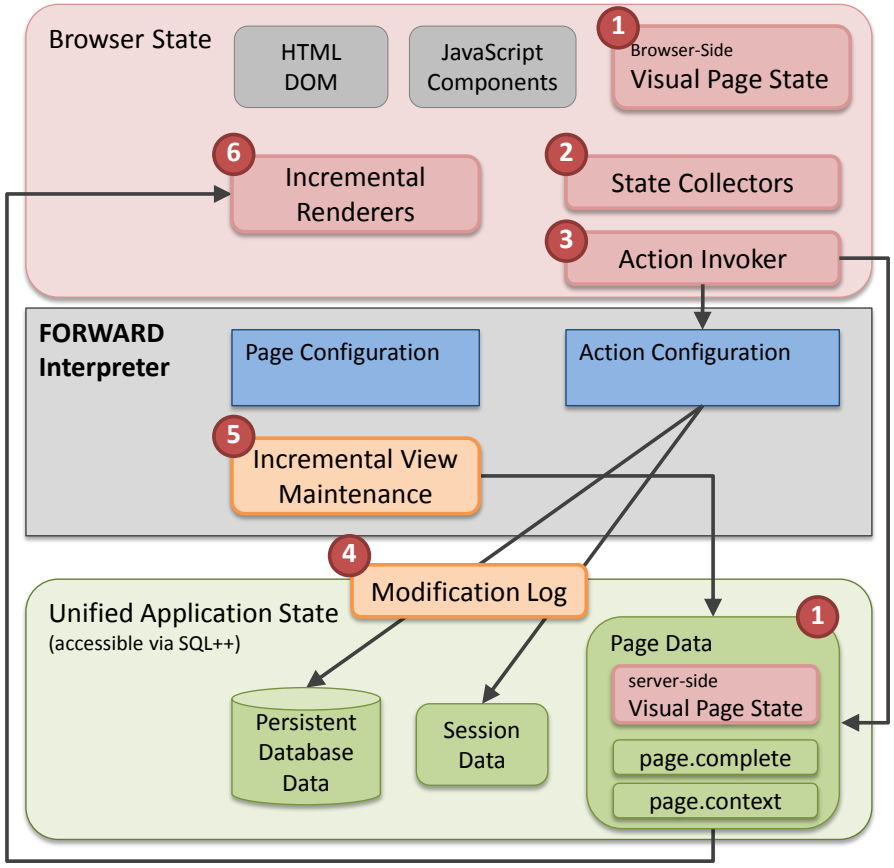


Figure 1.13. Internal FORWARD Architecture

The internal architecture of the FORWARD interpreter is illustrated in Figure 1.13, which

displays internal modules (labelled with red numbers) in association with developer-visible concepts of Figure 1.2. For efficiency of storage and communications, FORWARD maintains a *visual page state* (a.k.a. visual data object), which provides an abstraction over the browser state by including the externally visible state of visual units, but excluding their implementation details. Two copies of the visual page state lazily mirrored between the browser and the server.

When the user performs interactions such as typing in a text box, the HTML DOM and the JavaScript variables of visual components change in the browser state. The respective *state collectors* of each FORWARD unit synchronize the appropriate part of the browser state with the corresponding part of the *browser-side visual page state*. When the user eventually triggers an event that leads to invoking an action, such as clicking the "Save Review" button of Figure 1.1, the *action invoker* guarantees that the browser-side visual page state has been fully mirrored onto the server-side before the action executes. This guarantee is efficiently implemented via incremental writes to the prior visual page state.

Using the action invocation context and page configuration, the interpreter calculates (1) the page.complete data object consisting of named attributes of the visual page state, and (2) the page.context data object, the part of the page.complete that is in the context of the action instance. As services within the action read from and write to the unified application state, the system also uses a *modification log* to intercept all changes to the unified application state. By using the modification log in combination with the unified application state, the interpreter employs *incremental view maintenance* optimizations to incrementally maintain the page data to the next visual page state. Incremental view maintenance module is discussed in Chapter 2;

Finally, the interpreter uses data diffs to efficiently reflect changes back to the browser-side visual page state. The same data diffs are also provided to the respective *incremental renderers* of each visual unit, which, in turn, programmatically translates the data diff of the visual page state into updates of the underlying DOM elements or method calls of the underlying JavaScript components. Essentially, the incremental renderers modularize and encapsulate the partial update logic necessary to utilize Javascript components, so that developers do not have to

provide such custom logic for each page. Also illustrated in [24], in addition to performance gains due to less DOM elements / JavaScript components being initialized, incremental rendering also delivers a better user experience by reducing flicker and preserving unsaved browser state such as focus and scroll positions.

1.3.2 Rendered View Approach and Visual Schema

FORWARD treats Ajax pages as rendered SQL++ views consisting of units and XHTML segments that correspond to different parts of the view (see [24]). The visual data object (i.e., the "view") contains all information needed to render a browser page. Internally, the state of a visual unit is described by a SQL++ schema. Nodes of the visual schema (and hence of the visual data objects) may be marked with *unit annotations* that specify what visual unit is associated with the subtree rooted at each node. The unit annotation structure derived from the visual schema forms a *unit tree* structure of the page, where a parent unit functionally contains a child unit. For instance, the Maps unit in the running example is able to take an arbitrary child unit serving the content of each popup dialog, which may have its own child units recursively. As a result, the data of a child unit instance is nested under the data of its parent unit instance in the visual data object according to the unit tree structure. For example, the visual schema in Figure 1.14 shows that data for an XHTML unit (Line B) is nested under the Maps unit (Line C) as a child unit.

The rendering of the visual data object happens by first turning the visual data object into a unit instance tree, and then having each visual unit's renderer create corresponding HTML DOM or JavaScript components. In case of a page refresh, the framework figures out the difference of the visual data object using incremental view maintenance, and renderers may use the difference of the visual data objects to partially update visual units accordingly. On the other direction, the state collectors of visual units collect latest user input from the browser page and reflect it back to the visual data object, which is mirrored on both the server side and the browser side.

A technique commonly used to implement partial page update is to apply IDs to each

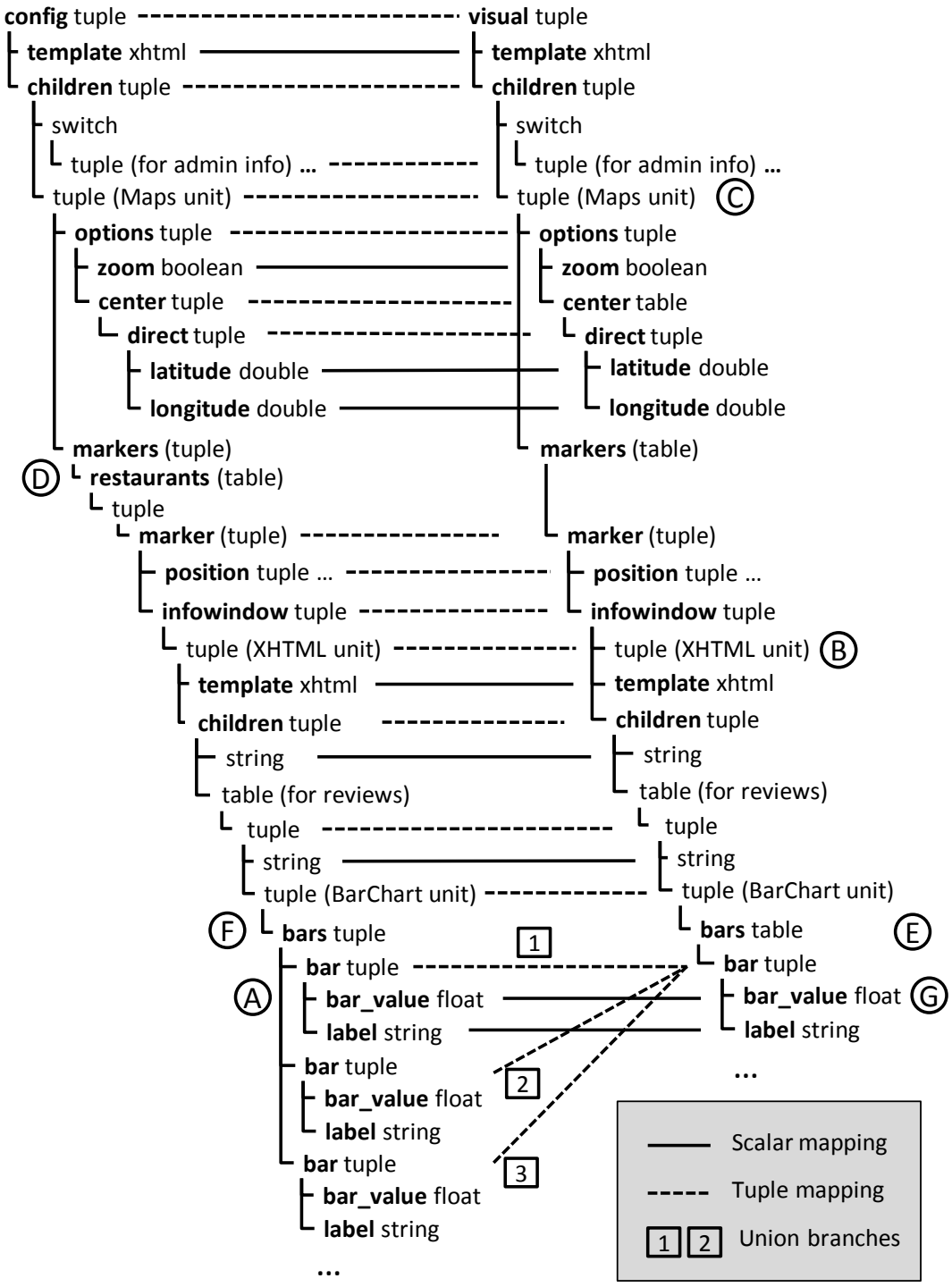


Figure 1.14. Config schema, visual schema, and c2v mapping (before mapping provenance IDs are inferred)

portion of the page so that page update can specify which part of the original page to change using IDs. In manual Ajax application development, developers manually specify DOM IDs in pages and have to figure out how to come up with the ID values that are unique and stable (meaning that an ID value is kept the same for the same entity at different timestamps). In FORWARD, IDs in the visual data object is automatically plugged in as keys that are inferred by the framework. Besides the partial page update feature, IDs and keys also help to describe the invocation context when action invocation happens. Key inference of FORWARD queries is discussed in Appendix B.

The visual data object including its unit annotations is all that the framework needs to render a page in the browser. The visual schema is completely driven by the requirement from visual units. Therefore, visual units need a way to formally declare the possible schemas of data that they can handle. Also notice that in FORWARD, schema and queries of visual data objects are internal and hidden from developers. Therefore the developers need not be aware of them or explicitly specify them. Instead, schemas and queries related to the visual state are automatically inferred from the XML page configuration.

1.3.3 Unit definitions and DTD

Visual units of FORWARD framework use *unit definitions* to model its external-facing state that is configurable in FORWARD pages. A unit definition describes three types of information:

1. Composability: A parent unit can specify a node in its schema to be a placeholder for a child unit. The page compiler will attach the schema of the child unit to the parent unit's schema in order to form a page's visual schema. For example, the Maps unit specifies that each marker's popup dialog contains a child unit of any kind.

2. Interaction: A node in a unit definition may have *event definitions* about possible events that can be triggered in the context of the node. Such events can be associated with actions.

For example, the button unit has an "onclick" event which is fired when the button is clicked.

3. Polymorphism: The unit definition may describe a class of schemas that the unit can be instantiated with, rather than a fixed SQL++ schema.

The last goal above is achieved by using *schema patterns* which is an extension to SQL++ schema tree. It has two notable features. First, attribute names can be open-ended wildcard or prefix patterns. Second, the list of attributes of a tuple type may be open-ended. The presence of an attribute may be quantified as either *exactly one*, *at least one (+)*, or *any (*)*. An attribute with + or * has to have an open-ended name. A scalar attribute that is quantified as exactly-one may carry a *default value*, which makes the attribute *optional (?)*.

Unit definitions are automatically translated to an XML model and in particular the Document Type Definition (DTD) in order to support the XML page configuration syntax. The framework does the translation as follows. The root becomes an element with the unit name. All tuple, switches and table patterns are translated to XML elements. Each scalar attribute, depending on how the unit developer specifies it, is translated to either an XML element or an attribute of the element corresponding to its parent tuple. Furthermore, the patterns of table, switch and open-ended attribute list of a tuple can be expressed using DTD syntax. Figure 1.6 shows the DTD syntax of the Maps unit. Figure 1.15 illustrate the pseudocode of the main block of unit definition-to-DTD translation.

Notice that the standard DTD syntax is less expressive than SQL++. For example, the DTD of Maps does not specify whether the markers (including nested XHTML and units under them) are of homogeneous type or not. Unit documentation covers such information. Therefore, validation of page configuration with respect to visual unit's schemas is performed via the unit definition (see Section 1.3.6), instead of via DTD validation.

Unit definitions and their DTD representations are what developers need to know in order to compose units into a page using XML page configuration. The page compiler then infers

```
function translateTuplePattern
input  : TuplePattern t
Output : Element declaration of t
begin
    e ← an XML element for t;
    foreach Attribute type v of t do
        if v is of scalar type and v.Xmlization == Attribute then
            a ← an XML attribute for v;
            Declare a as e's attribute.;
            if v has default value then
                Declare a's default value to be v.default;
        else
            c ← null;
            if v is a tuple pattern then
                c ← translateTuplePattern(v);
            else if v is a table pattern then
                c ← an XML element for v;
                vt ← table tuple of v;
                ct ← translateTuplePattern(vt);
                Declare ct as c's child element with * quantifier;
            else if v is a scalar pattern then
                c ← an XML element for v;
            else if v is a switch pattern then
                c ← an XML element with disjunctive child group;
                foreach case tuple pattern vc of v do
                    cc ← translateTuplePattern(vc);
                    Declare cc as c's disjunctive child element;
            else if v is an "any child unit" pattern then
                c ← an XML element for v;
                Declare c's element content as ANY_UNIT;
            dtd_quantifier ← none;
            if v.quantifier == + or * then
                dtd_quantifier ← v.quantifier;
            else if v is a scalar type and has default value then
                dtd_quantifier ←?;
            Declare c as e's child element with dtd_quantifier. ;
    return e;
```

Figure 1.15. Translating a unit definition to DTD

various schemas and mappings from the XML page configuration.

1.3.4 Overview of Page Compilation

FORWARD's page compilation engine achieves three goals.

Goal 1: Inferring visual and logical schemas from XML page configurations

Section 1.2 introduces several page-related state/data objects. To build a FORWARD page, however, developers need not specify each page state individually, which would incur redundant work and repetition. Instead, the page compiler infers all page states from the single XML page configuration that the developers have to write.

Goal 2: Resolving heterogeneities between the visual schema and the logical schemas.

One may have noticed that in the example illustrated in Figure 1.10 of Section 1.2.4, a page's visual schema that represents the visual structure of a page can be very different from the page's logical schemas, and in particular the page.complete schema. Indeed, the two are designed to serve different purposes to start with. At a high level, the visual schema reflects the visual structure of the page. Internally, the visual schema of a page is dictated by unit definitions of visual components, and the Maps unit actually requires data of markers to be formatted as tuples in a table. The page.complete schema, on the other hand, is driven by the page's logical structure rather than the visual structure. That is why despite the markers are modeled as a table of tuples visually, in the page.complete (Figure 1.11), there are standalone attributes for the coordinates of markers standing for start and end points. The page.complete schema follows the logical structure of the page implied by the XML page configuration, and provides what an action wants to access from the page in an intuitive way. Moreover, a page.context schema further simplifies the logical structure to contain only data that is in context of an action instance. The gap between the visual schema and the logical schemas is one major issue that FORWARD's internal page architecture is designed to deal with.

Goal 3: Resolving the heterogeneities between the database schema and the visual schema.

Besides the above-mentioned heterogeneities between a page's visual and logical schemas, there is another gap between the database schema and the visual schema. This is most notable in terms of differences between scalar types. For instance, in the running example, Area A of Figure 1.4 shows review ratings being retrieved from database and displayed in a bar chart. The ratings are stored as integers in database, but when drawn as bars, they are treated as double values which are expected by the BarChart unit.

Figure 1.16 shows the complete picture of page computation steps. Two internal page data objects are revealed: *Page query result* is the result of running the *page query*, a query obtained by combining all query expressions in the page configuration; *Config page state* is the representation of the entire page configuration in the SQL++ data model. Since the page query result is the result of all expressions (e.g., those within {}) in the page configuration, the difference between the page query result and the config page state is that the former does not contain constant non-expression data and static template data, while the latter does. The difference between the config page state and the page.complete is that page.complete contains only those named attributes in the page configuration.

Both page query result and config page state are internal concepts to FORWARD's page layer architecture. In principle they need not exist in implementation, since their computation can be chained together and merge into the computation for visual page state and page.complete etc., in order to reduce computation overhead. In this paper, we do use them to explain the page computation and compilation process, because they help to present the overall architecture and heterogeneity resolutions in a clear way.

As shown in Figure 1.16, the page data objects are transform from one to another using either queries or mappings. Schemas, queries and mappings are the end products of page compilation process. Figure 1.17 shows the internal flowchart of page compilation. The rest of this section will discuss each step in detail.

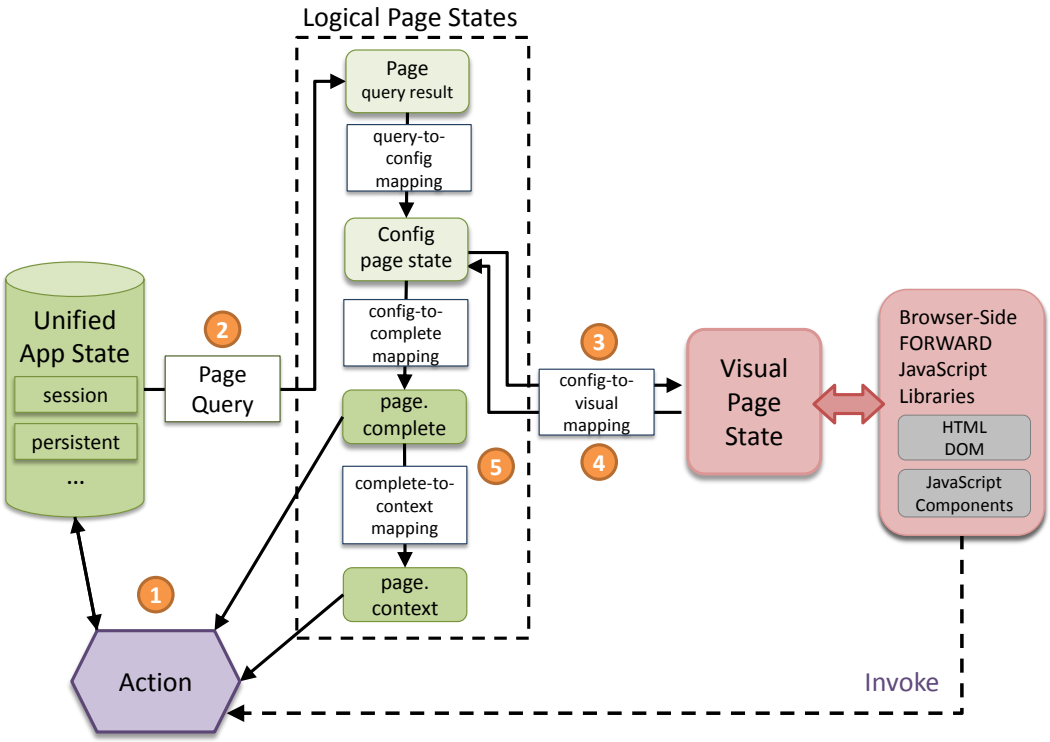


Figure 1.16. Internal page computation cycle (Incremental View Maintenance module is not shown)

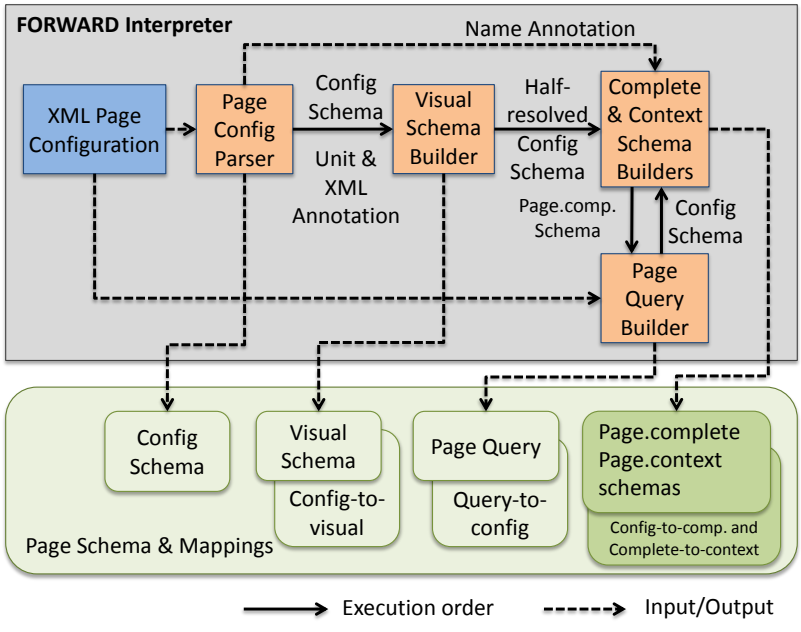


Figure 1.17. Page compilation

1.3.5 Page Config Parser

As the first step of page compilation, a *page config parser* takes an XML page configuration as input and utilizes a few general rules to turn it into a config schema: First, each XML element that has sub-elements or attributes (excluding the statement elements `fstmt:for` and `fstmt:switch`) corresponds to a tuple in the config schema. Second, the remaining XML elements and the XML attributes (excluding the special attribute `name`) correspond to attributes in the config schema. Finally, each `fstmt:for` corresponds to a table and each `fstmt:switch` corresponds to a switch. The resulting schema is *non-resolved* in the sense that its scalar types are still unknown. These decisions are resolved in later steps of page compilation. The config schema in Figure 1.14 is one that has been fully resolved.

Also produced by the page config parser are various annotations to the config schema, including *XMLization annotations* that tell whether a SQL++ attribute originates from an XML element or attribute, *unit annotations* that contain unit information and are used to build unit trees, *event annotations* that associate actions with events of visual units, and *name annotations* that carry the `name` information of the subset of config schema for building the page.complete and the page.context schemas.

The page config parser specially handles *HTML unit*'s parsing so that its configuration syntax can be aligned with XSLT style. Figure 1.18 shows the DTD of the HTML unit. The HTML unit's schema consists of a `template` scalar value representing the static part of the XHTML document, and a `children` tuple containing values corresponding to dynamic data and child units. These have been reflected in both the config and the visual schemas in Figure 1.14. The `template` value also contains *named placeholders*, which will be substituted with values from the `children` tuple during rendering. To use the HTML unit, developers need not explicitly use its DTD. Instead, they use the `fesc:html` tag within which they can write directly XHTML code with embedded `fstmt` tags and child unit configurations (e.g., Line 41 of Figure 1.4).

A config schema, even though not fully-resolved after this step, can be used to match

```
<!DOCTYPE Html [
 <!ELEMENT Html ( template, children )>

  <!ELEMENT template (xhtml)>
  <!ELEMENT children (ANY_UNIT*)>
]>
```

Figure 1.18. DTD of HTML visual unit

against unit definitions by the visual schema builder to produce the visual schema and a config-to-visual mapping.

1.3.6 Visual Schema Builder

Before we dive into the generation of the visual schema from the config schema, a subtle question to ask is why they both exist conceptually. It may seem that the config schema and the visual schema agree and all that we needed to do would be to add on the visual schema attributes that have default values and are skipped in the config schema. The answer is about the heterogeneity between visual and logical aspects of a page. Notice that the generation of table and switch constructs in the config schema is driven by `fstmt:for` and `fstmt:switch` statements, i.e., the logical structure of the dynamic data that will populate the page, rather than the needs of the units that will consume the data. For example, the table of markers of line D in Figure 1.14 is due to the fact that the query outputs a list of restaurants rather than the fact that a `Maps` unit consumes a list of markers. Another example shown by Figure 1.14 is that the config schema lines A corresponds to the visual schema lines E, where despite the fact that a `funit:BarChart` requires a list of bars we map to it from three individual tuples on the config schema, since the page[3] does not require a query-dependent number of bars but rather needs exactly three bars for displaying the review's ratings.

Because the visual schema is decided by unit definitions, the visual schema builder matches the config schema of a page to unit definitions in order to infer the page's visual schema.

[3]Though in the running example, the logical-visual heterogeneity has limited impact, Figure 1.10 has shown that in another example, it can become significant when the page.complete inherits the heterogeneity in order to serve actions.

Also inferred during the unit match is the config-to-visual mapping (which internally replaces the high-level complete-to-visual mapping introduced in Section 1.2.4). We use mapping instead of SQL++ query for this purpose because the computation power needed here (and in several other places of the framework for automatic data coordination and heterogeneity resolution as well) is much less than the one provided by SQL++. Appendix A describes the mapping framework in detail. For now it is fair to consider a mapping as a set of lines between two SQL++ schema trees with their most straightforward meanings. Here, the config-to-visual mapping is built by the visual schema builder through the unit match process.

The unit match algorithm of the visual schema builder fires a set of predefined rules to recursively match a node in the config schema to a schema pattern in the unit definition, and automatically loads the definition of a child unit when the matching dives into it. Successfully firing a rule results in a subtree of the visual schema corresponding to the schema pattern, and a subset of config-to-visual mapping towards the visual schema subtree. One important rule that deals with the logical-visual heterogeneity is *TupleToTable* match rule that matches a TupleType c in the config schema to a TablePattern p in the unit definition. For example, the `bars` attribute of BarChart unit is a table (line E of Figure 1.14) according to the unit definition. But `bars` is a tuple in the config schema since an XML element in the page configuration is always translated to a tuple (line F of Figure 1.14). Therefore the TupleToTable match rule tries to match the content of the `bars` tuple c in the config schema to the pattern of the `bars` table p in BarChart unit definition. Its pseudo-code is listed in Figure 1.19. The rule iterates each attribute of c and then (1) matches a TupleType attribute to the TuplePattern of p, since a TupleType corresponds to a hardcoded tuple in XML configuration (e.g., line A for BarChart unit), (2) matches a TableType attribute's TupleType to the TablePattern p recursively, since a table in config schema comes from a `fstmt:for` tag in XML configuration, whose semantics says it should be replaced by the evaluation of the for loop (e.g., line D for markers of Maps unit), or (3) matches each case of a SwitchType attribute to p recursively for a similar reason to the table case. At the end of this rule, sub-matching result for each attribute of c are checked for compatibility such that all of

them should lead to the same visual schema v. Finally, the mappings in different branches are merged using a union, which is then returned with v as the result of the current rule firing.

```
function matchTupleToTable
input  : TupleType c, TablePattern p
Output : Visual schema subtree v and Mapping subset m
begin
    subResults ← empty list of (v,m) pairs.;
    foreach attribute c_a of c do
        if c_a is annotated as logical-only then
            continue;
        else if c_a is a TupleType then
            p_t ← the TuplePattern of p;
            (v',m') ← matchTupleToTuple(c_a, p_t);
            subResults ← subResults + (v',m');
        else if c_a is a TableType then
            c_t ← the TupleType of table c_a;
            (v',m') ← matchTupleToTable(c_t, v);
            subResults ← subResults + (v',m');
        else if c_a is a SwitchType then
            foreach case tuple c_t of c_a do
                (v',m') ← matchTupleToTable(c_t, v);
                subResults ← subResults + (v',m');
    if all matching results in subResults are compatible then
        (v,m) ← merge (union) subResults;
        return v,m;
    else
        Fail;
```

Figure 1.19. Unit match rule for matching a tuple in page configuration to a table pattern in unit definition

Unit match also leads to type resolution in config schema. As the non-resolved config schema is matched against the unit definitions, concrete types are created in the visual schema, which are applied back to the config schema and result into precise type assignments to scalar nodes and tables. For example, the float type of `bar_value` (line A) in config schema is propagated from the visual schema. Some types in config schema may exist for logical page only

and hence not matched against any units[4]. These types will remain non-resolved until later steps of the page compilation.

1.3.7 Page.complete and Page.context Builders

As introduced earlier, page.complete and page.context data objects are part of the unified application state that developers can access from page and action configurations, unlike other data objects which are not exposed. Figure 1.16 shows that the page.complete and the page.context data objects are computed from the config page state using page-to-complete and complete-to-context mappings. These happen during the action-page cycle and after the data updates to visual page state has been propagated back to the config page state. Update propagation from visual to configuration is achieved by *mapping inversion* of config-to-visual mapping provided by FORWARD's mapping framework. To make it possible, mapping framework automatically plugs in provenance IDs in the target data so that nodes updated in the target can be traced back to their corresponding source nodes. The details of mapping inversion is explained in Appendix A.

The page.complete schema of a page can be defined as a projection of the config schema that retains attributes that correspond to (a) named expressions, (b) named form-collected data and (c) their enclosing tables and primary keys. Those unnamed non-table tuples are projected away and hence do not appear in the page.complete schema. The page.complete schema of the running example is shown in Figure 1.7, where `save_comment` etc. correspond to form field values, while `save_pid` corresponds to the primary key of the restaurants. The page.complete schema builder takes as inputs the half-resolved config schema after unit match and its name annotations, and produces the page.complete schema and a config-to-complete mapping that transforms data from the config page state to the page.complete. To do so, the builder traverses the config schema and copies the named attributes as well as their ancestors to the page.complete schema which is built on-the-fly. At the same time, the builder establishes config-to-complete mapping lines from the config schema to the page.complete schema.

[4]Such types are hidden from visual page and carry information for actions to access

Notice that the page.complete schema produced at this stage may still have non-resolved attributes, since they may be copied from non-resolved attributes in the config schema. The reason for inferring the page.complete schema before the config schema is fully resolved is that query in the XML page configuration may use the page.complete as part of the unified application state, and therefore the framework architecture needs the page.complete to be at least partially defined in order to compile queries in next step, while type resolution will be completed at the end.

The page.context data object with respect to a page action is defined to be part of the page.complete that is in the context of the page action. Its computation from page.complete is defined as follows. Let x be the tuple corresponding to the context of the action invocation. Then for any ancestor of x (including x itself) that is a table tuple, its sibling tuples are removed from request data object. Furthermore, the tables from root to x which are left with only one tuple are flattened out, leaving the singleton tuples pushed up and merged into their ancestor tuple. The same applies to switch nodes as well. That is, all switches from root to x are flattened out with the content of the switch case that contains x pushed up to its ancestor. For example, the page.context schema for the "SaveReview" action is shown in Figure 1.8. Comparing it to the page.complete schema, one can see that the table of restaurants is replaced by the exact single tuple corresponding to the invocation context. The page.context schema builder is in charge of inferring the page.context schemas and the complete-to-context mappings for every action in the page. The builder works by consulting the page.complete schema and config-to-complete mapping produced by the page-complete builder to decide their page.context counterparts.

1.3.8 Page Query

So far we have seen how various page data objects are computed around the config data object during an action-page cycle. The remaining question is how the config data object is computed at the first place. Recall that an XML page configuration contains embedded snippets of SQL++ queries and expressions. The *page query builder*, being part of the page compiler,

extracts these snippets and composes them into the page query with proper nesting. The builder constructs the page query by piecing together (a) the queries that appear in the queries of the `fstmt:for` statements, (b) the conditions of the `fstmt:switch` statements and (c) computations in the expressions.

The page query builder is also in charge of creating a query-to-config mapping that transforms the page query result to the config page state data object. To do so, the builder needs to deal with the heterogeneity in the database-to-visual direction. In that respect, the config schema's types are mostly decided by the visual unit definitions, therefore incurring heterogeneity with the query results that are driven by the database's types. In particular, the query-to-config mapping coerces the query results into the types needed by the config schema, or resolves the config schema types if they are still non-resolved. For example, the query snippet of line 64 produces an integer for the bar value but the query-to-config mapping converts it into a float expected by the BarChart unit (line G of Figure 1.14).

A complexity in type resolution and key inference is present here as the page query may access the page.complete data object which is part of the unified application state. This means that there may be a circular dependency of types and keys among the page query result, the config page state, and the page.complete. For key inference, recall that the framework tries to automatically infer keys for the visual page state. Consequently, in the current architecture and implementation, the framework needs to be able to infer keys for the precursors of the visual page state, including the page query result and the config page state. To limit the complexity of such circular dependency, the page compiler requires that once page query builder is run, the resulting page query should have all types resolved and keys successfully inferred. The page compilation process then ends with propagating this information to the previously non-resolved data objects (e.g., the config page state).

Chapter 1 contains material from "The SQL-based all-declarative FORWARD web application and development framework" by Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Michalis Petropoulos, The Fifth Biennial Conference on Innovative Data Systems Research,

2011. The dissertation author was the primary investigator and author of this paper.

Chapter 2

ID-Based Incremental View Maintenance

2.1 Introduction

Materialized views are widely used to speed up query evaluation by storing the results of commonly asked queries. Being materialized, these views have to be brought up to date when the underlying data change. This is typically done through *Incremental View Maintenance (IVM)*. Abstracting out the details of different IVM approaches, a typical IVM algorithm takes as input three diff tables $\mathcal{D}_R^+$, $\mathcal{D}_R^-$ and $\mathcal{D}_R^u$ per base relation R, containing the tuples that were inserted, deleted and updated in R and computes the corresponding diff tables $\mathcal{D}_V^+$, $\mathcal{D}_V^-$ and $\mathcal{D}_V^u$ for the view V, containing the changes that have to be performed on V to bring it up to date.

In prior IVM work, each diff table $\mathcal{D}_V^+$, $\mathcal{D}_V^-$ and $\mathcal{D}_V^u$ contains one diff tuple for each view tuple that has to be inserted, deleted and updated, respectively. This is why we refer to such diffs as *tuple-based diffs* (in short *t-diffs*). In this work we show that if the base tables contain keys, one can represent the view modifications in a much more compact way through a novel type of diffs, called *ID-based diffs* (in short *i-diffs*), which identify the to-be-modified view tuples through their IDs. In contrast to t-diff tuples, a single i-diff tuple can represent modifications to *multiple* view tuples. This difference is crucial, as i-diffs are more efficient to compute than t-diffs, requiring in general fewer base table accesses as we will explain next. This leads to more efficient ID-based IVM algorithms, under common assumptions. The following example demonstrates the difference between t-diffs and i-diffs. To differentiate between the two types

```
devices(did, category)
parts(pid, price)
devices_parts(did, pid)
```

(a) Database schema

```
CREATE VIEW V AS
SELECT did, pid, price
FROM parts NATURAL JOIN
  devices_parts NATURAL JOIN
  devices
WHERE category = "phone"
```

(b) View definition

Figure 2.1. Database schema and view for running example

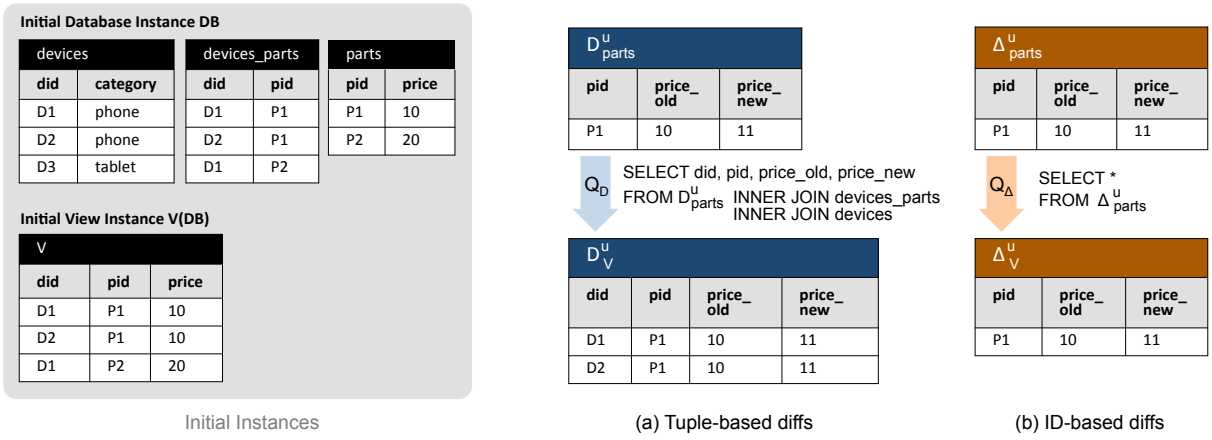


Figure 2.2. Example of tuple-based and ID-based IVM

of diffs, in the rest of the paper we will be using the standard symbol Δ to refer to the newly introduced i-diffs and the symbol $\mathcal{D}$ to refer to traditional t-diffs.

Example 2.1.1. Consider the database of an electronic device manufacturer, storing a list of devices and their parts. Figure 2.1a shows the respective database schema, consisting of the relations **devices**, **parts** and **devices_parts**. The key attributes of each relation are shown underlined. Also consider the view **V** of Figure 2.1b returning the list of parts for devices of type 'phone'.

Figure 2.2 shows an example of tuple-based and ID-based incremental maintenance of **V**. The initial database and view instance are shown on the left and t-diffs and i-diffs for a sample change in relation **parts** on the right in Figures 2.2a and 2.2b, respectively. Consider the action of updating the price of part "P1" from \$10 to \$11. This modification is represented through identical t-diff and i-diff tuples, as seen in the t-diff table $\mathcal{D}^u_{parts}$ and i-diff table Δ^u_{parts},

respectively. However, this is no longer true when we look at the diffs that represent the resulting updates that have to happen to the view. Note that the change of the "P1" price in **parts** has to be propagated as updates of all "P1" tuples in the view (which in this case are the first two tuples). While the t-diff table $\mathcal{D}_V^u$ describes these changes by two diff tuples, each describing an entire view tuple that has to be updated, the i-diff table Δ_V^u describes the same modifications through a single diff tuple (which intuitively instructs updating all "P1" tuples in the view).

Obviously ID-based diffs are more compact than their tuple-based counterparts. More importantly though i-diffs are in general more efficient to compute than t-diffs. Intuitively, the performance gains come from the fact that in contrast to t-diffs, i-diffs do not need to recreate the entire view tuples to be modified and thus can avoid some base table accesses. We use this observation to design an ID-based IVM algorithm, which is shown both analytically and experimentally to perform in most cases fewer base-table and view accesses than prior tuple-based approaches.

Example 2.1.2. Queries $Q_{\mathcal{D}}$ and Q_{Δ} show how the diffs for the view can be computed from the base tables and the diff for base relation **parts**. While computing the t-diff requires joining $\mathcal{D}_{parts}^u$ with the base tables **devices_parts** and **devices** (see $Q_{\mathcal{D}}$) to find all devices containing part "P1", producing the i-diffs simply amounts to finding the modified parts (and not the devices in which they are contained) and can therefore be accomplished by accessing only Δ_{parts}^u and avoiding all joins with the base relations (see Q_{Δ}).

Note that the fewer base table accesses of i-diff computations are not, just by themselves, an absolute proof of superior performance of the i-diffs, as maintaining the view should also count the cost of applying the i-diffs to the view. While both the ID-based and tuple-based approaches will have to join the resulting view diffs with the view, the ID-based approach has the drawback of potentially trying to join more diff tuples by creating *dummy* i-diff tuples, i.e., i-diff tuples describing changes for tuples that do not even exist in the view. For example, assume that **parts** included a tuple (P3, 20) and Δ_{parts}^u included a change of P3's price. Then Δ_V^u would

include a dummy P3 tuple, i.e., the system would pay the price of attempting to update the P3's in V, albeit there would be no P3 in V. We call this effect *overestimation*. Nevertheless, our theoretical and experimental analysis show that under common circumstances the i-diff approach is indeed superior. Furthermore, we show that the advantage of ID-based IVM grows as the queries become more complex.

Contributions. This paper makes the following contributions:

(a) An ID-based IVM system, called *idIVM*, applicable when the base relations have primary keys. The *idIVM* is based on a modular, algebraic approach, allowing one to extend the supported view definition language simply by adding one relational algebra operator at-a-time and providing *i-diff propagation equations* describing how ID-based changes are propagated through it.

(b) A set of i-diff propagation equations for a large subset of SQL (denoted by Q_{SPJADU}) that includes the algebraic operators select, project, join, grouping and aggregation with associative functions, generalized projection involving functions, antisemijoin[1] and union. Although the framework can be easily extended to more expressive view definition languages as described above, our analytical and experimental results focus on Q_{SPJADU}, as it is expressive enough to cover a large number of practical use cases.

(c) An efficient 4-pass algorithm that creates an IVM plan for a given algebraically-expressed view and a given set of modification types in four passes that are polynomial in the size of the view expression: The first pass computes the IDs of intermediate results. The second pass instantiates the operator IVM equations to the specifics of the view's operators. The third pass composes individual equations into the queries of the IVM plan. Finally, the fourth pass applies minimization and other optimizations particular to the IVM problem. Unlike general purpose minimization the considered minimization is polynomial.

(d) An algorithm that given a view expression decides what types of i-diffs should be mined from the modification log or captured from triggers. The problem is non-trivial since,

[1]Therefore capturing queries with negation. The difference operator is a special case of antisemijoin.

as we will see, the number of types of i-diffs that are applicable, given a base schema and a view schema, is exponential in the size of the schemas. The presented algorithm uses the view definition to decide the much smaller number of sufficient and efficient i-diffs.

(e) A formal analysis proving that for Q_{SPJADU} views in many use cases the ID-based IVM with the i-diff propagation equations described in the paper is more efficient than tuple-based IVMs. The analysis is based on a fine-grained cost model counting data accesses and includes a discussion under the specific conditions under which tuple-based IVMs can perform better.

(f) An experimental evaluation of the proposed IVM system for Q_{SPJADU} views indicating that in most cases it significantly outperforms traditional tuple-based approaches. The experimental results show speedups of 2 to more than 50 over tuple-based IVMs.

Note that ID-based IVM optimization is orthogonal and can be combined with many of the other IVM issues studied in the literature [28, 15], such as materialized view selection [6, 51, 41], self maintenance [9, 27] and compilation into code [2]. We briefly describe these prior IVM works and their synergies in Section 3.1.

Outline. The chapter is structured as follows: Section 2.2 defines ID-based diffs (i-diffs). Section 2.3 presents the architecture of *idIVM*. It consists of two main parts: The first transforms base table modifications to i-diffs and the second, given a set of i-diffs, creates a DML script for maintaining the view. For ease of exposition, we present them in the reverse order, i.e. Section 2.4 describes the algorithm for the DML script generation and Section 2.5 describes the transformation of changes to i-diffs. Sections 2.6 and 2.8 compare analytically and experimentally the efficiency of the generated script to those produced by tuple-based IVM approaches. Finally, Section 2.9 and later sections introduce the extension of *idIVM* to SQL++ query language.

2.2 ID-based diffs

For the following discussion we consider a relational database DB whose base tables contain keys and a relational view $V(\bar{I},\bar{A})$ over DB, containing a set of key attributes (which we will refer to as IDs) $\bar{I}$ and a set of non-key attributes $\bar{A}$.

Example 2.2.1. The view V of our running example contains IDs $\bar{I}$ = {did, pid} and non-ID attributes $\bar{A}$ = {price}. In the following we will be using the initial instance of V of Figure 2.2.

View definition language. Although, as we will see, the framework can be easily extended to more expressive view definition languages, unless otherwise stated, we consider views from the language Q_{SPJADU}, which contains all SQL queries that can be formulated using the algebraic operators Selection, Projection (involving functions), Join (with arbitrary join conditions), Aggregation with associative functions sum, avg and count, Antisemijoin (and thus Difference), and Union[2].

General Structure of an i-diff. Let $V(\bar{I},\bar{A})$ be a view with IDs $\bar{I}$ and non-ID attributes $\bar{A}$. An *ID-based diff* (in short *i-diff*) of type $t \in \{+,-,u\}$ for relation V is in its most general form a relation $\Delta^t_V(\bar{I}',\bar{A'}_{pre},\bar{A''}_{post})$ satisfying the following properties:

- It contains a subset $\bar{I}'$ of the view's IDs $\bar{I}$. These are used to identify the tuples to be modified.
- It may contain two sets $\bar{A'}_{pre}$ and $\bar{A''}_{post}$ of attributes, such that $\bar{A}',\bar{A}''$ are sets of non-ID attributes of V. An attribute A_{pre} and A_{post} intuitively stores the pre-state value (i.e., initial value before the change) and respectively post-state value (i.e., new value after the change) of attribute A of V.

Depending on their type, i-diffs may not contain both pre-state and post-state attributes. In particular, insert i-diffs (i.e, of type $t = +$), do not contain pre-state attributes, since they

[2]To maintain the IDs for the bag union, we employ a special union all operator, outputting a special attribute **b**, denoting which child branch (b = 0/1 for left and right, resp.) a tuple came from.

represent insertions of tuples that did not exist before. Similarly, delete i-diffs (i.e., of type $t = -$) do not contain post-state attributes. Only update diffs (i.e., of type $t = u$) may contain both old and new attribute values. We next describe the semantics for each i-diff type:

Update i-diff. An update i-diff instance Δ_V^u for view $V(\bar{I},\bar{A})$ is a relation instance with schema $\Delta_V^u(\bar{I}',\bar{A}'_{pre},\bar{A}''_{post})$, where $\bar{I}'$ is a subset of the IDs $\bar{I}$ of V and $\bar{A}'$, $\bar{A}''$ are potentially different subsets of the non-ID attributes $\bar{A}$ of V (with $\bar{A}'$ being potentially the empty set).

Intuitively, each tuple $(\bar{i}',\bar{a}'_{pre},\bar{a}''_{post})$ in Δ_V^u specifies that all tuples in V with values $\bar{i}'$ for their $\bar{I}'$ attributes should have the values of their $\bar{A}''$ attributes updated to $\bar{a}''_{post}$. Formally, applying Δ_V^u on an instance I_V of V is equivalent to applying the following DML statement on I_V:

APPLY Δ_V^u: UPDATE V
SET $\bar{A}'' = \bar{A}''_{post}$
FROM Δ_V^u
WHERE $V.\bar{I}' = \Delta_V^u.\bar{I}'$ [3]

In the rest of the paper, the instance I_V will be implied from the context and therefore for simplification we will simply refer to a diff as being applied on the view V.

Note, that although not affecting its semantics, an update i-diff may also contain pre-state values of some non-ID attributes of V. As we will see later, this additional information is leveraged by the IVM algorithm to reduce the number of accesses to the database.

Example 2.2.2. Applying the following update i-diff

Δ_V^u	pid	price$_{pre}$	price$_{post}$
	P1	10	11

leads to the update of the *price* of both tuples in V with pid = "P1" from 10 to 11.

[3] Note that for conciseness the UPDATE statement is written using PostgreSQL's special UPDATE FROM syntax. However, it could be equivalently written using standard SQL syntax.

Remark. In the following we consider only i-diffs where $\bar{I}'$ forms a primary key of the i-diff. The reason is that if $\bar{I}'$ is not a key, then update i-diffs are not well-defined and insert i-diff applications may lead to primary key violations.

Insert i-diff. An insert i-diff instance Δ_V^+ for a view $V(\bar{I},\bar{A})$ is a relation instance with schema $\Delta_V^+(\bar{I},\bar{A}_{post})$, or in other words a relation containing the post-state values for all attributes of the view and no pre-state values.

Intuitively, an insert i-diff instance Δ_V^+ contains a set of tuples that should be inserted into V. Formally, applying Δ_V^+ has the same effect as applying the following DML statement on V:

APPLY Δ_V^+: INSERT INTO V
SELECT $\bar{I}$, $\bar{A}_{post}$ AS $\bar{A}$ FROM Δ_V^+
WHERE ROW($\bar{I}$, $\bar{A}_{post}$) NOT IN
(SELECT $\bar{I}$, $\bar{A}$ FROM V)

Example 2.2.3. Applying the following insert i-diff

Δ_V^+	did	pid	price$_{post}$
	D3	P2	20
	D4	P3	30

inserts tuples ⟨D3, P2, 20⟩ and ⟨D4, P3, 30⟩ in V.

Remark. The WHERE clause in the above DML statement ensures that an attempt is made to insert a tuple into V only if it is does not already exist in V in the exact same form. This allows multiple insert i-diffs to try to insert the same tuple.

Delete i-diff. A delete i-diff instance Δ_V^- for a relation $V(\bar{I},\bar{A})$ is a relation instance with schema $\Delta_V^-(\bar{I}',\bar{A}'_{pre})$, where $\bar{I}'$ is a subset of the IDs $\bar{I}$ of V and $\bar{A}'$ is a potentially empty subset of the non-IDs $\bar{A}$ of V.

Intuitively, Δ_V^- specifies the tuples that should be deleted from V based on the values of the $\bar{I}'$ attributes. Formally, applying Δ_V^- has the same effect as applying the following DML

statement on V:

APPLY Δ_V^-: DELETE FROM V
WHERE ROW($\bar{I}'$) IN (SELECT $\bar{I}'$ FROM Δ_V^-)

Note that, similarly to update i-diffs, a delete i-diff may also specify the pre-state values of the deleted tuples, which are used to create more efficient IVM solutions.

Example 2.2.4. Applying the following delete i-diff

Δ_V^-	pid	price_{pre}
	P1	10

leads to the deletion of both tuples with pid = "P1" from V.

Effective i-diff instances. Given a set of i-diff instances $\bar{\Delta}$ for a relation V, applying them on V leads in general to different results depending on the order of application. However, in this work we only look at sets of i-diffs where any order of applying them on V yields the same result. To this end, we define the notion of *effective* i-diff instances. Given the pre-state V^{pre} and post-state V^{post} of a relation V, an i-diff instance Δ_V^t is said to be *effective* w.r.t. V^{pre} and V^{post} if for each value of a tuple of V it reflects its final value. Formally, it is effective iff it satisfies the following properties:

- If Δ_V^t is an insert i-diff: Every tuple inserted by the i-diff exists in the post-state (i.e., $\Delta_V^+ \subseteq V^{post}$).
- If Δ_V^t is a delete i-diff over schema $\Delta_V^-(\bar{I}', \bar{A}'_{pre})$: Every tuple deleted by the i-diff does *not* exist in the post-state relational instance (i.e., $\pi_{\bar{I}'}\Delta_V^- \cap \pi_{\bar{I}'}V^{post} = \emptyset$).
- If Δ_V^t is an update i-diff over schema $\Delta_V^u(\bar{I}', \bar{A}'_{pre}, \bar{A}''_{post})$: Every tuple updated by Δ_V^u that exists in the post-state instance, has all updated attributes $\bar{A}''_{post}$ set to the corresponding values in that instance (i.e., $\pi_{\bar{I}', \bar{A}''_{post}}\Delta_V^u \ltimes_{\bar{I}'} V^{post} \subseteq \pi_{\bar{I}', \bar{A}''}V^{post}$).

It can be shown that a set of effective i-diffs lead to the same result regardless of the order in which they are applied. In the following the i-diff instances we consider are assumed to be effective. We will discuss in Sections 2.4 and 2.5 how *idIVM* makes sure that it always operates on effective i-diff instances.

i-diff schemas. It should be obvious that a single modification could be represented through i-diffs of different schemas. In particular, one can include pre-state or post-state values for different sets of attributes. More importantly, different base table i-diffs may lead to IVM solutions of different efficiencies. For instance, an update of a t tuple of relation $R(\bar{I}, A_1, A_2)$ on attribute A_1 can be represented by either an i-diff that contains only the post-state of A_1, or both the post-state of A_1 and A_2 (even though the value of A_2 did not change). However, the first i-diff will in general lead to a more efficient solution, since for the second i-diff the IVM algorithm will have to account also for the change of A_2, although this is not needed. This generates a novel challenge of selecting which base tables i-diff schemas to create, as explained next.

IDs and functional dependencies. As discussed earlier, the set $\bar{I}$ of ID attributes of a view V forms a key of that view. Moreover, any i-diff Δ_V for the view V identifies the tuples of V to be updated, deleted or inserted through a subset of that key. However, this cannot be an *arbitrary* subset of the key. The key $\bar{I}$ of a view is split into components $(\bar{I}_1, \bar{I}_2, \ldots \bar{I}_n)$, such that for each component of the key there is a functional dependency $\bar{I}_i \rightarrow \bar{A}_i$ from this component $\bar{I}_i$ to a subset $\bar{A}_i$ of non-key attributes of the view. The i-diff can identify the tuples of the view to be modified through a subset of these key components.

Example 2.2.5. For instance, in our running example, V has ID/key $\bar{I} = \{did, pid\}$, which can be decomposed into two components $\bar{I}_1 = did$ and $\bar{I}_2 = pid$, since there is the functional dependency $pid \rightarrow price$ (in a more general view V' containing also attributes of the **devices** relation there would also be a functional dependency from did to those attributes). Thus the tuples of V can be identified by an i-diff either through did or through pid (as is the case in our example).

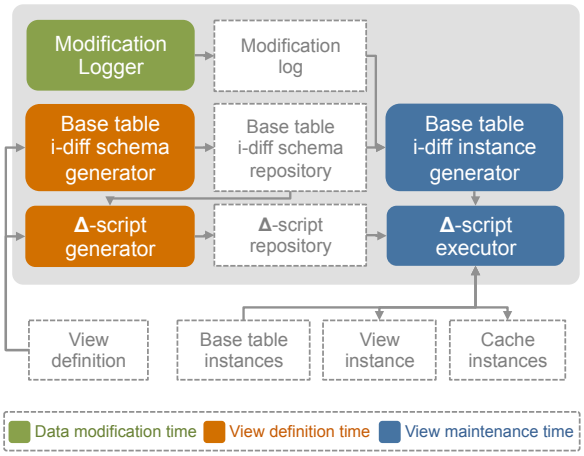


Figure 2.3. *idIVM* architecture

2.3 System Architecture

Figure 2.3 depicts the architecture of *idIVM*; an ID-based IVM system based on i-diffs and built on top of a relational DBMS. The modules of the system are shown as rounded boxes, while the system's data structures are depicted as white rectangles. *idIVM* can be setup to maintain the view either eagerly (i.e., whenever the base data change, known as *eager IVM* [10, 13, 26]) or lazily at some later point in time (known as *deferred* IVM [16, 31, 38]). In either case, the modification logging module of the *idIVM* remains the same. Furthermore, the only part of the architecture that is substantially different in the two approaches is the i-diff propagation rules and cache maintenance rules (see Figure 2.4). This paper describes the deferred IVM rules. *idIVM* contains modules executed at three different times (shown through color-coding in Figure 2.3): (a) when the views are defined (orange), (b) whenever the data in the underlying database change (green), and (c) whenever the views are maintained (blue). We present next briefly each of these stages:

View definition time. The most interesting and novel computations happen when a view is added to the system. At that point *idIVM* precomputes in the form of DML scripts how to translate i-diffs on the base tables to view updates. This computation happens through the synergy of two components: First, it employs a *base table i-diff schema generator* to decide

which i-diff schemas to generate for the base tables. As we have discussed in Section 2.2, this is a non-trivial problem, as the same update could be modelled through i-diffs of different schemas. Once the base table i-diff schemas have been decided, *idIVM* invokes the Δ-*script generator* creating a DML script that accesses the generated i-diffs, the base tables and the potential caches (which as we will see can be used to speedup the IVM) to maintain the view. The resulting Δ-script is stored in a repository to be used at view maintenance time.

Data modification & view maintenance time. Given this offline computation, the system's online component is simple: When the base data are modified, a *modification logger* logs these changes for later use. When the time comes to maintain the view, the *base table i-diff instance generator* consults the modification log and converts it to instances of the base table i-diff schemas precomputed at view definition time. A Δ-*script executor* then retrieves the Δ-script corresponding to the view from the Δ-script repository and executes it to propagate the changes represented by the base table i-diff instances to the view instance.

We next describe the Δ-script generation, leaving the discussion of how to convert base table modifications to i-diffs for Section 2.5.

2.4 Δ-script Generation Algorithm

Given a view definition and a set of base table i-diff schemas, the Δ-script generation algorithm creates a DML script that includes (a) queries over the base table i-diffs, the base tables and the auxiliary caches (which as we will see are used by *idIVM* to speed up the IVM) that compute the corresponding view i-diffs and (b) UPDATE / INSERT / DELETE statements of the form described in Section 2.2 that apply these i-diffs on the view.

The Δ-script generator is based on the algebraic IVM approach [46, 26, 47]: Each relational operator type (e.g., selection, projection, join, etc.) is annotated with a set of *rules*, describing how to transform an (effective) i-diff over its input schema to an (effective) i-diff

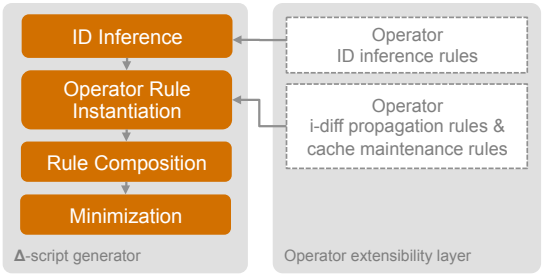


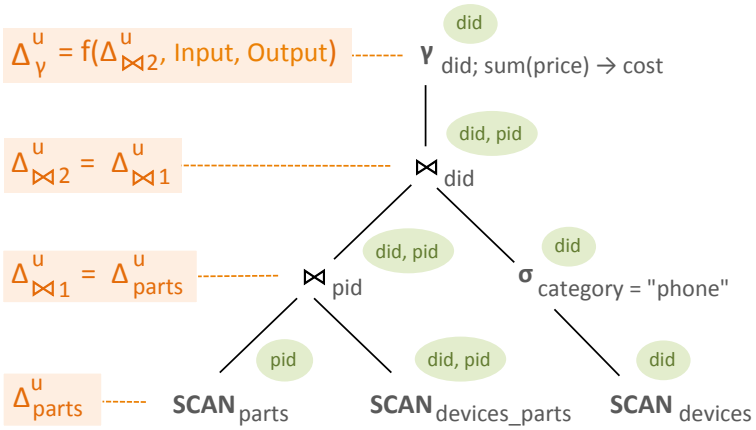
Figure 2.4. Δ-script generator architecture

over its output schema. Given this information, the Δ-script for a view V can be composed from the individual rules for each operator that appears in an algebraic plan of V. Intuitively, the algorithm is computing how to maintain the entire view by first computing how to maintain all intermediate subviews in the algebraic plan. This approach enables a modular implementation in which the supported view definition language can be easily extended by adding rules for additional relational algebra operators. In this work we present the rules for all operators included in Q_{SPJADU} (i.e., selection, join, generalized projection involving functions, grouping with aggregation, antisemijoin, and union). Similarly to prior algebraic IVM approaches, we assume that the algebraic plan of the view on which the algorithm operates is given as input.

Example 2.4.1. To showcase the algorithm, we extend the view of our running example to also perform an aggregation, returning the total cost of the parts for each device. Figures 2.5b and 2.5a show the view definition V' and a corresponding algebraic plan, respectively. The shaded components are annotations inserted by the algorithm, which we explain below.

idIVM performs the 4 efficient passes of Figure 2.4.

Pass 1: Inferring ID information for intermediate views. Since i-diffs determine the view tuples that have to be modified through their IDs, the view and all intermediate subviews should contain as part of their output schema a set of ID attributes that form a key of the corresponding view. *idIVM* determines the ID attributes that should be contained in the output schema of each subview through the use of *ID inference rules*. An ID inference rule is supplied for each operator type supported by the system and describes how the IDs of the view rooted at



(a) Algebraic plan (annotated by the Δ-script generator)

```
CREATE VIEW V' AS
SELECT did, sum(price) AS cost
FROM parts NATURAL JOIN
     devices_parts NATURAL JOIN
     devices
WHERE category = "phone"
GROUP BY did
```

(b) View definition

Figure 2.5. View definition and plan for extended running example

an operator p can be computed from the IDs of the subviews rooted at p's children. Table 2.1 shows the ID inference rules for operators in Q_{SPJADU}.[4] *idIVM* uses these rules to perform a postorder traversal of the plan checking at each operator whether the IDs inferred by these rules exist in the operator's output schema. If this is not the case, *idIVM* automatically extends the plan to include the required ID attributes.

Example 2.4.2. Figure 2.5a shows the set of IDs for each operator in a shaded oval on the top right side of the operator.

Note that extending the view with additional ID attributes simply increases the width of the view instance (i.e., the number of columns) but does not affect its cardinality (i.e., the number of tuples). In particular, if V_{orig} is an original view with attributes $\bar{A}$ and V_{ID} is the view inferred

[4] Union refers to the *union all* operator described in Section 2.2.

Table 2.1. Operator ID inference rules

Operator	Output ID attributes
$SCAN(R)$	$key(R)$
$\sigma_\phi(R)$	$ID(R)$
$\pi_{\bar{D}}(R)$	$ID(R)$
$R \times S$	$ID(R) \cup ID(S)$
$R \bowtie_\phi S$	$ID(R) \cup ID(S)$
$R \triangleright_\phi S$	$ID(R)$
bag union $R \cup S$	$ID(R) \cup ID(S) \cup \{b\}$
group by $\gamma_{\bar{G},f(\bar{M})\ldots}(R)$	$\bar{G}$

by the ID-inference algorithm, then for all instances of the base tables the original view can be computed from the original view simply by projecting out the additional attributes introduced by the ID-inference algorithm (i.e., $V_{orig} = \pi_{\bar{A}} V_{ID}$). Given that the number of additional ID attributes is usually small compared to the number of attributes already in the view, we do not expect the extension of the view schema with ID attributes to significantly affect the query evaluation performance. Importantly, the above observations hold not only for operators in Q_{SPJADU} but *for any* SQL operator (for the reader wondering how this can be the case for the duplicate elimination operator δ, given that in general $\delta(\pi_{A,B}(R))$ is different from $\delta(\pi_{ID,A,B}(R))$, consider an implementation where the duplicate elimination operator δ above is replaced by the group by operator $\gamma_{A,B}$).

Pass 2: Instantiating rules for each intermediate operator. To construct the Δ-script for the view, the algorithm employs operator rules that describe how each operator can propagate i-diffs over its input to i-diffs over its output.

Operator rules. An operator describes how to transform an i-diff Δ^t_{input} over one of its input schemas to an i-diff Δ^t_{output} over its output schema through a set of queries known as *i-diff propagation rules*. These queries can access (a) the operator's input i-diff Δ^t_{input} and (b) the data corresponding to the subview rooted at the operator or at one of its child operators. The latter is the way in which *idIVM* allows operator rules to access data from the base tables. Since an operator does not have knowledge of the exact place in the query plan where it appears to

ask for a query result over the base tables, it can access the base table data only indirectly by asking for the subview rooted at one of its children through the use of the $Input_{i=l,r}$ (standing for left and right input, resp. for binary operators) or for the subview rooted at itself using the *Output* keyword, respectively. The input subviews can be requested either in their pre-state form (i.e., using the instances of the base tables before the diffs were applied to them) or in the post-state (i.e., using the final instances of the base tables after the diffs were applied to them). An i-diff propagation rule can specify which of the two versions of the input it needs through the subscripts *pre* and *post*. The output is always provided in pre-state.

Example 2.4.3. For instance, a general grouping and aggregate operator $V = \gamma_{\bar{G},f(\bar{X})\to c}(Input)$ contains among others the i-diff propagation rule: $\Delta^u_V = \gamma_{\bar{G},f(\bar{X})\to c}(Input_{post} \ltimes_{\bar{G}} \Delta^+_{Input})$, which semijoins the post-state of the subview rooted at the operator's child with an input insert i-diff to find all tuples that belong to groups affected by the insertions and use them to recompute the value of the aggregate function for these groups. The $Input_{post}$ keyword is the way in which the operator asks for the (post-state of) some base data (in this case the base data defined by the subview rooted at the operator's child, which for the aggregate operator of Figure 2.5a is the subview **parts** $\bowtie_{pid}$ **devices_parts** $\bowtie_{did} \sigma_{\text{category="phone"}}$ **devices**).

There are two different classes of operators in *idIVM*: The first consists of operators which can produce an output effective i-diff by looking at one input i-diff at a time. These operators are called *non-blocking* operators, in contrast to *blocking* operators which need to inspect the entire set of input i-diffs before creating an effective output i-diff. The operator type affects how the operator's i-diff propagation rules are expressed. For non-blocking operators, each rule is expressed over a single input i-diff, while for blocking operators, a rule is expressed over all input i-diffs.

Example 2.4.4. The general aggregate operator γ of Example 2.4.3 is a non-blocking operator, since it can decide how to propagate an input insert i-diff without looking at other input i-diffs (e.g., delete or update i-diffs). The reason is that for each insert i-diff tuple it recomputes the

entire affected group from the base data thus reflecting indirectly also the changes incurred by other input i-diffs. On the other hand, imagine a more efficient aggregate operator designed specifically for the SUM aggregate function. This operator avoids recomputing entire groups by combining all input i-diffs to figure out the amount by which the aggregate value of each group has changed. While it avoids some base table accesses, it requires knowledge of all input i-diffs and is thus a blocking operator.

Tables C.1-C.10 show the i-diff propagation rules for the operators considered in this work, including join, union, generalized projection involving functions, antisemijoin and aggregation. Rules for aggregation are provided in four different versions (see Tables C.4, C.6, C.8, and C.9); one for general aggregation functions and others for specialized functions, such as SUM, COUNT and AVG.

Some operator rules may also benefit from special caches to speed up IVM. For instance, an aggregate AVG operator in the presence of a COUNT and SUM cache can incrementally maintain its output without accessing the base tables. To accomodate such cases, *idIVM* allows operators to declare special *operator caches* and associated *cache maintenance rules*, describing how to compute the i-diffs that maintain the caches. The i-diff propagation rules can then be expressed also over the operator cache schemas and the operator cache i-diffs. Table C.9 shows the cache maintenance rules and i-diff propagation rules for the AVG operator.

Rule instantiation. In its second pass, the Δ-script generator algorithm employs the predefined operator rules to compute how each base table i-diff is propagated from operator to operator in the view plan. This is done as follows: For each base table i-diff schema Δ^t_R, the algorithm starts from the scan operator of the corresponding base table R and in a bottom-up fashion instantiates the rules for all operators in the path from the scan operator to the root of the plan.[5] The rule instantiation simply consists in selecting from all rules for the particular operator the ones that apply in the particular case (based on the input i-diff schema and other conditions)

[5]If the base table R appears with multiple aliases, this process is repeated for every scan operator of R.

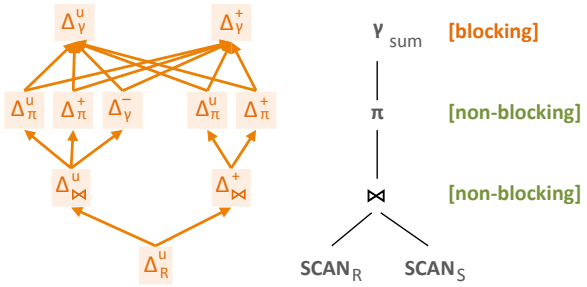


Figure 2.6. Rule DAG structure

and replacing the abstract schema used in the rules with the concrete schema of the particular operator instance (e.g., for an operator $V = \pi_x(R)$ the general projection i-diff propagation rule $\Delta_V^+ = \pi_{\bar{a},f(\bar{X})\to c}\Delta_R^+$ becomes $\Delta_V^+ = \pi_x\Delta_R^+$).

Example 2.4.5. Consider an update i-diff schema $\Delta^u_{parts}(pid,\ price_{pre}, price_{post})$ modeling updates on the price attribute of table **parts** of our running example. Figure 2.5a shows on the left the corresponding instantiated rules generated by the algorithm. The exact rule for the aggregate operator is omitted due to lack of space. However, it is important to note that it is a rule that mentions the input i-diff and the input and the output of the operator, respectively.

Note that for a single input i-diff an operator may create multiple output i-diffs. For instance, an update i-diff going through a selection operator may lead to insert, update and delete i-diffs, depending on whether a tuple satisfied the condition before and after the change. Whenever the rules of an operator create multiple output i-diffs, the above computation continues conceptually in parallel for each generated i-diff schema. This leads to a directed *rule DAG*, whose nodes are instantiated rules and whose edges point from a rule to all rules that were created using its output schema. Figure 2.6 shows such a structure. Note how blocking rules convert the structure that would otherwise be a tree into a DAG.

Pass 3: Composing operator-level instantiated rules into a Δ-script. Each rule in the DAG is a query expressed over the output schema of its parent rules (note that the DAG in Figure 2.6 is shown inverted with its root shown at the bottom). Thus each i-diff for the view (which corresponds to a leaf) can be computed by composing the instantiated rules of its ancestors. The

exact order in which these compositions are performed does not matter, since all considered i-diffs are effective. This is guaranteed by the fact that (a) the base table diff instance generator creates effective diffs (as we will discuss in Section 2.5) and (b) i-diff propagation rules transform effective input i-diffs to effective output i-diffs.

To make the generated plan more efficient, *idIVM* employs also additional caching, other than the caching used internally by operators. In particular, for aggregate operators, whose rules typically ask for the base data corresponding to their input/output (through the $Input_i$ and $Output$ keywords, resp.), *idIVM* attempts[6] to create an *intermediate* cache in which it materializes this result. This cache is treated as any other view and maintained during the IVM process. In particular, *idIVM* first composes all rules that create the i-diffs for the cache and then using them as input, composes the rest of the rules up to the next cache until it reaches the view.

Example 2.4.6. For instance, as we have seen in Figure 2.5a, the instantiated rule for the aggregation mentions both the input and the output of the operator. Thus, *idIVM* tries to generate two intermediate caches; one before the aggregate and another after the aggregate. Since however the output of the aggregate coincides with the view (which is already materialized), *idIVM* creates only the first cache and utilizes the already existing view as the second.

The result of this composition is a Δ-script, containing queries that compute i-diffs for an intermediate cache/view and APPLY operators that use the DML statements corresponding to each i-diffs type (shown in Section 2.2) to apply these i-diffs to the cache/view.

Example 2.4.7. In our running example *idIVM* employs an intermediate cache below the aggregate operator. Thus, it composes the rules up to that point, updates the cache and then uses it as input to compose the rules up to the view, which is subsequently updated. This leads to the Δ-script of Figure 2.7.

[6]Intermediate caches are not generated when they are expected to contain multi-valued dependencies (for instance due to a many-to-many join), since in that case reading the result from the cache would incur more tuple accesses than simply recomputing it on the fly from the base tables. *idIVM* exploits foreign key constraints to infer the absence of multi-valued dependencies.

1 $\Delta^u_{Cache} = \Delta^u_{parts}$;
2 APPLY Δ^u_{Cache};
3 $\Delta^u_{V'} = \pi_{did,cost \to cost_{pre},cost+cost_\Delta \to cost_{post}}(V' \bowtie$
$\gamma_{did,sum(price_\Delta) \to cost_\Delta}($
$\pi_{did,pid,(price_{post}-price_{in}) \to price_\Delta}($
$\Delta^u_{Cache} \bowtie \pi_{price \to price_{in}} Cache)))$;
4 APPLY $\Delta^u_{V'}$;

Figure 2.7. Δ-script for running example

Pass 4: Optimizing the generated Δ-script. As a last step, *idIVM* optimizes the Δ-script by performing semantic optimization, which minimizes each individual query included in the plan. In contrast to general minimization, this minimization takes into account the special semantics of i-diff tables. As described in Section 2.2, given a base table $R(\bar{I},\bar{A})$ in its post-state and i-diffs $\Delta^+_R(\bar{I}, \bar{A}_{post})$, $\Delta^-_R(\bar{I},\bar{A}'_{pre})$, and $\Delta^u_R(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$ over this table, the following constraints hold: (a) $C_1 : \Delta^+_R \subseteq R$, (b) $C_2 : \pi_{\bar{I}}\Delta^-_R \cap \pi_{\bar{I}}R = \emptyset$, and (c) $C_3 : \pi_{\bar{I},\bar{A}''_{post}}\Delta^u_R \ltimes_{\bar{I}} R \subseteq \pi_{\bar{I},\bar{A}''}R$. *idIVM* minimizes w.r.t. constraints C_1 - C_3 by employing on top of the standard relational rewrite rules also the rewrite rules presented in Figure 2.8. Semantic minimization is crucial in eliminating inefficiencies introduced by composing individual operator rules, improving in some cases performance by more than 50%.

Designing operator i-diff propagation rules. The efficiency of the Δ-script obviously depends on the provided i-diff propagation rule definitions. Reasoning about the efficiency of individual rules is hard, as rules affect each other (e.g., a rule avoiding base table accesses may not bring in some information that could be used by rules later in the plan, thus leading to higher access cost later).

However, in this work we show that we do not have to get into this reasoning process. Simply creating rules that individually avoid accessing the base tables when possible leads to efficient Δ-scripts, as shown by our analytical and experimental results. To avoid data accesses, the rules are even allowed to *overestimate*, i.e. skip some filtering that would require base table

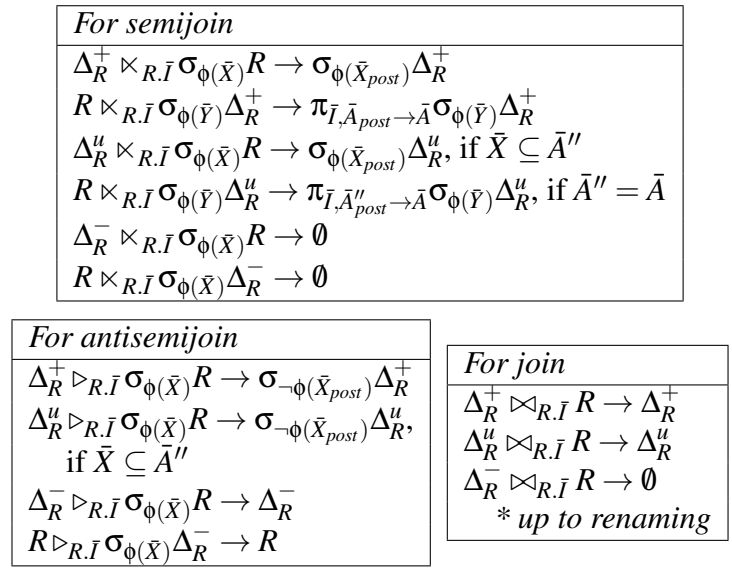


Figure 2.8. Rewrite rules for semantic optimization

accesses and propagate to their output i-diffs dummy tuples that do not affect the view.

Example 2.4.8. For instance, the selection operator allows a delete input i-diff to pass through the operator unmodified. However, this means that the output i-diff will also instruct the deletion of tuples that do not satisfy the selection conditions and thus do not exist in the view. Although this is an overestimated i-diff, it does not affect the correctness of the generated Δ-script, since the latter will simply try to delete some tuples from the view that do not exist. On the other hand, this rule locally minimizes the base table accesses, as it avoids accessing the base tables to filter out the tuples that do not satisfy the selection condition.

2.5 From modifications to i-diffs

We saw above how given a set of base table i-diffs, *idIVM* maintains the view. In this section we explain how these base table i-diffs are generated from base table modifications. This is a non-trivial problem, since as explained in Section 2.2, a single modification can be represented through i-diffs of different schemas, each leading potentially to Δ-scripts of different efficiencies.

idIVM solves the i-diff generation problem through the synergy of three components shown in Figure 2.3: (a) a modification logger recording the base table modifications at data modification time, (b) a base table i-diff *schema* generator deciding at view definition time which base table i-diff schemas to generate, and (c) a base table i-diff *instance* generator, translating at view maintenance time the modifications recorded in the log to instances of the pre-computed i-diff schemas. Logging changes to the base tables can be easily performed through known techniques, such as DBMS log inspections, timestamp queries or triggers (currently used by *idIVM*). Therefore we focus next on the other two components.

Generating i-diff schemas. Given a view definition V, *idIVM* generates suitable base-table i-diff schemas for all base tables mentioned in V. Insertions and deletions are straightforward cases: Consider a base table $R(\bar{I},\bar{A})$ with key attributes $\bar{I}$ and non-key attributes $\bar{A}$. For each such table, the i-diff schema generator creates a single insert i-diff schema $\Delta_R^+(\bar{I},\bar{A}_{post})$ (containing all attributes of R) and a single delete i-diff schema $\Delta_R^-(\bar{I},\bar{A}_{pre})$ (containing all non-ID attributes of R in pre-state form). This is based on the observation that pre-state values can lead only to a more efficient Δ-script as they may reduce overestimation and the respective view index lookups. For instance, as shown in Table C.3 with blue, a selection operator can exploit pre-state attributes to filter out the tuples of an incoming delete i-diff that do not satisfy the condition.

The same does not hold though for post-state attributes included in update i-diffs. Including more post-state attributes in an update i-diff schema leads to a generally less efficient Δ-script, as it has to account also for changes in these attributes. Creating one update i-diff schema for each subset of attributes of each base table is obviously not an option, due to the exponentiality involved.

In *idIVM* we solve this problem by observing that the base table attributes can be divided into sets of attributes whose updates lead to the same Δ-script and can thus be grouped together in a single i-diff schema. For each operator op in the algebraic view plan, let C_{op}, be the set of (non-key) base table attributes involved in any condition (e.g., selection, join etc)[7]. We refer to C_{op} as

[7]Base table key attributes do not need to be considered for updates as they are immutable.

the *set of conditional attributes for op*. The set of (non-key) base table attribute not included in any C_{op} for any operator op in the view's plan is referred to as the *set of non-conditional attributes NC*. Non-conditional attributes may still affect the view (since they could be included in the view's output), but intuitively they do not affect the generated Δ-script (up to projections). Updates on each set of conditional attributes C_{op} on the other hand may lead to a different Δ-script, since the updated values may affect whether the i-diffs make it past op's condition. Therefore for each base table $R(\bar{I},\bar{A})$, the i-diff schema generation algorithm creates (a) for each set C_{op} an update i-diff $\Delta^u_R(\bar{I},\bar{A}_{pre},\bar{A}'_{post})$, s.t. $\bar{A}' = \bar{A} \cap C_{op}$ and (b) an additional update i-diff $\Delta^u_R(\bar{I},\bar{A}_{pre},\bar{A}''_{post})$, containing the non-conditional attributes of R (i.e., $\bar{A}'' = \bar{A} \cap NC$).

Populating i-diff instances. Every time *idIVM* is invoked to maintain the view, the i-diff instance generator simply populates the i-diff tables created at view definition time. This is done by extracting the changes since the last view maintenance from the modification log and adding them as diff-tuples to all i-diff tables that contain at least one of the modified attributes (in the case of updates) and to the single insert and delete i-diff tables (in the case of inserts and deletes, respectively). Note, that when extracting the modifications from the log, the algorithm combines multiple modifications to the same tuple to a single modification, so as to generate *effective* diffs. As discussed in Sections 2.2 and 2.4, this is crucial for the algorithm's correctness.

2.6 Performance Analysis

We next analytically compute the speedup ratio of ID-based over tuple-based IVM (i.e., the ratio $\frac{\text{tuple-based cost}}{\text{ID-based cost}}$). A speedup ratio greater than 1 signifies that the ID-based approach has a lower cost than the tuple-based approach and thus is more efficient than the latter, while a speedup ratio lower than 1 signifies the opposite.

To compute the speedup we first compute the individual cost of each IVM approach. The cost of the ID-based/tuple-based approach is measured in the combined number of tuple accesses and index lookups incurred by the $\Delta/\mathcal{D}$-script, generated by the corresponding approach. For the

purposes of this analysis, we assume that both approaches have access to view indices on the view IDs and additionally the tuple-based IVM has access to appropriate base table indices (which are not required by the ID-based approach). We also try to be as general as possible regarding the query plan that the DBMS might choose to execute a particular $\Delta/\mathcal{D}$-script. However, since DBMSs employ complex optimizations that cannot be comprehensively accounted for in an analytical model, the computed cost and the associated speedups reported in this Section should only be used as rough estimates of the actual cost of performing IVM that illustrate the difference in performance between the two approaches. For an experimental comparison of the two IVM approaches, please refer to Section 2.8.

We next present the speedup for two representative cases: (a) SPJ views, which by default do not involve intermediate caches and (b) Aggregate views involving grouping and associative functions, which (by default) are supported by caches. For a detailed analysis explaining how this speedup was computed, please refer to Section 2.7.

2.6.1 SPJ Views

Consider the SPJ view V_{spj}:

$$\texttt{SELECT } \bar{S} \texttt{ FROM } R, R_1, \ldots, R_n \texttt{ WHERE } c$$

whose FROM clause involves a single alias of a table R, (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R. [8]

Parameters affecting speedup. The speedup of the ID-based approach over the tuple-based approach can be expressed in terms of two parameters: the *i-diff compression factor* p and the *tuple-based computation cost per base table diff tuple* a. The i-diff compression factor $p = |\mathcal{D}_{V_{\text{spj}}}|/|\Delta_{V_{\text{spj}}}|$ is the ratio of the size of the tuple-based diff to the size of the ID-based diff for the view. p may be less than 1 (when i-diffs summarize the modifications to the view in a more

[8] Recall that we use the symbols Δ and $\mathcal{D}$ to represent i-diffs and t-diffs, respectively.

compact way than t-diffs, as shown in Figure 2.2) but may also be greater than 1 (when i-diffs are overestimating and trying to modify tuples that do not exist in V_{spj}). The second parameter is the number of accesses a that the tuple-based approach has to perform on average to compute the t-diff tuples for the view that result from a given t-diff tuple for the base table. This cost will typically vary, depending on the plan chosen by the DBMS to evaluate the tuple-based $\mathcal{D}$-script.

Speedup ratio. The speedup ratio of the ID-based approach over the tuple-based approach is given by the following formula:

(a) if $\Delta_R/\mathcal{D}_R$ is an update i-diff/t-diff on attributes of R that are not involved in selection or join conditions in V_{spj}, then

$$\text{A: Speedup ratio} = \frac{a+2p}{1+p}$$

(b) else (i.e., if $\Delta_R/\mathcal{D}_R$ is any other update i-diff/t-diff or it is an insert or delete i-diff/t-diff)

$$\text{B: Speedup ratio} \geq min\left(\frac{a+2p}{1+p}, 1\right)$$

Discussion. Let us first explain why update diffs on attributes of R that are non-conditional may lead to a different speedup ratio than other types of diffs. Since the updates do not affect how a tuple behaves w.r.t. selections or joins, they are guaranteed to lead to updates (i.e., neither inserts, nor deletes) on the view. When this is the case (case (a) above), the ID-based IVM algorithm can simply propagate the base table i-diffs to the view without accessing the base tables, leading to a speedup ratio of $\frac{a+2p}{1+p}$. This speedup is in most practical cases greater than 1 (meaning that the ID-based is more efficient than the tuple-based approach). For the tuple-based approach to perform better, it should be the case that $a < 1 - p$, which can be satisfied only in the corner case when the following conditions simultaneously hold: (a) the tuple-based approach incurs $a < 1$ tuple accesses for each tuple in $\mathcal{D}_R^u$ on average (which can only happen if many tuples of $\mathcal{D}_R^u$ share the same join attribute values and thus the joined tuples can be retrieved once and reused for all of them) and (b) the ID-based approach is severely overestimating (i.e., $p \ll 1$).

We have experimentally verified that by following this pattern it is possible to create contrived scenarios in which the tuple-based IVM outperforms the ID-based approach. However, in all other cases, the ID-based approach performs better (with a difference that raises proportionally to p).

When the base table diffs are insert or delete diffs or they are updates on attributes involved in conditions (case (b) above), then they will lead in general to updates, inserts and/or deletes on the view. If they lead to updates and deletes only, then the resulting speedup is the same as in the first case (i.e., $\frac{a+2p}{1+p}$) and thus the ID-based approach is expected to perform better in most cases. However, if they lead to inserts, then the two approaches will perform identically and hence exhibit a speedup of 1. Finally, if the base table diffs lead to a combination of updates, deletes and inserts, then the speedup will be a linear combination of the above two speedups and thus will be greater than the smaller of the two (hence the use of inequality and the *min* function in the above formula).

Thus for SPJ queries the ID-based IVM will always (up to the corner case described above) perform at least as good as the tuple-based IVM and in most cases better then the latter. The two approaches will only perform identically if the IVM workload is heavy on modifications that lead to insertions to the view.

2.6.2 Aggregate Views

Consider the aggregate view V_{agg}:

`SELECT` $\bar{G}, f(\bar{X})$ `AS` g `FROM` $R, R_1, \ldots, R_n$
`WHERE` c `GROUP BY` $\bar{G}$

whose FROM clause involves a single alias of R, and f is an associative aggregation function such as `sum` (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R.

To ease exposition, we isolate the aggregation operator of the query, expressing it through the plan $V_{\text{agg}} = \gamma_{\bar{G}, f(\bar{X}) \to g} V_{\text{spj}}$, where V_{spj} is the plan for the SPJ query presented in Section 2.6.1.

We study the interesting case, where the ID-based IVM has identified that an intermediate cache storing the input of the aggregate operator, which is the result of the SPJ query, is beneficial (since without cache both approaches would perform identically). The tuple-based approach does not use a cache, since it cannot benefit from it.

Both approaches operate in two two stages: They first compute the diff to maintain the SPJ subview V_{spj} and then use it to maintain the final aggregate view V_{agg}. The second step is the same in both cases. Thus the difference in performance comes from computing the diff for the V_{spj}, which in the case of the ID-based approach is also used to maintain the cache. Let a and p be defined as above for the subview V_{spj} (i.e., let $p = |\mathcal{D}_{V_{\text{spj}}}|/|\Delta_{V_{\text{spj}}}|$ and let a be the average cost incurred by the tuple-based diff to compute for each base table diff tuple the corresponding diff tuples for the view V_{spj}).

Speedup ratio. The speedup ratio of the ID-based approach over the tuple-based approach is given by the following formula:

(a) if $\Delta_R/\mathcal{D}_R$ is an update i-diff/t-diff on non-conditional attributes of R, then

$$\text{Speedup ratio} = \frac{a+x}{1+p+x}$$

(b) else

$$\text{Speedup ratio} \geq min\left(\frac{a+x}{1+p+x}, \frac{a+x}{a+k+x}\right)$$

where x is a cost related to grouping that is shared by both approaches and thus can be safely ignored for the purposes of our discussion and k is a parameter concerning insert diffs that we will explain later.

Discussion. Similarly to SPJ views, we differentiate between cases where the base table diffs lead to update or delete diffs on the view V_{spj} and cases where they lead to insert diffs on V_{spj}.

In the first case the speedup ratio is $s_1 = \frac{a+x}{1+p+x}$. This speedup is always going to be at least 1, meaning that the tuple-based approach can never perform better than the ID-based approach. This happens because the cost a incurred by the tuple-based approach for each diff

tuple in $\mathcal{D}_R$ is at least $1+p$, since for each such tuple it will have to incur at least one index access (to find the tuple of the other relations it joins with) and p tuple accesses (to read the tuples it joins with to create the corresponding p tuples in the view). Note that these are lower bounds that apply when the view V_{spj} contains only one join. If it contains more joins, the speedup ratio and thus the performance benefit of ID-based IVM will be even higher.

On the other hand when base table diffs lead to insert diffs on the view V_{spj}, the ID-based approach will be performing the same plan with the tuple-based approach but will also be inserting tuples into the cache. Thus if k is the number of tuples created in V_{spj} on average as a result of a single diff in $\mathcal{D}_R$, then the speedup will be $s_2 = \frac{a+x}{a+k+x}$. This speedup is less than 1 (meaning that the tuple-based approach will be performing better), but now the loss is bounded, as it is always 1 per tuple inserted into V_{spj}.

Similarly to SPJ views, when the base table diffs are updates on non-conditional attributes (case (a) above), the generated diffs on the view V_{spj} are guaranteed to be update or delete diffs and thus the speedup ratio will be equal to s_1. In any other case (case (b)), the generated view diffs will in general be combinations of update, delete, and insert diffs and thus the speedup will be a linear combination of s_1 and s_2 (and thus bounded by the smaller of them).

To summarize, for all base table diffs that do not lead to inserts in the cache, the ID-based approach is guaranteed to perform not worse (and the more complex the query the better) than the tuple-based approach. It only performs worse in workloads heavy on modifications that lead to insertions to the view, because it has to maintain a cache, so that the update and delete diffs can benefit from it. However, even this loss is bounded and we expect it to not be significant in practice. Moreover, this loss will be balanced out by the speedup on diffs that lead to deletes and updates in the view, which benefit from the cache.

2.7 Detailed Performance Analysis

We next describe the detailed performance analysis of the ID-based and tuple-based approaches that led to the equations of Section 2.6. The analysis covers two representative cases: (a) SPJ views, which by default do not involve intermediate caches and (b) Aggregate views involving grouping and associative functions, which (by default) are supported by caches. In the following we assume that both approaches have access to view indices on the view IDs and additionally the tuple-based IVM has access to appropriate base table indices (which are not required by the ID-based approach).

2.7.1 SPJ Views

Consider (a) the SPJ view V_{spj}:

$$\texttt{SELECT } \bar{S} \texttt{ FROM } R, R_1, \ldots, R_n \texttt{ WHERE } c$$

whose FROM clause involves a single alias of a table R, (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R on R. We distinguish between two cases, depending on the type of the diff $\mathcal{D}_R/\Delta_R$.

Update diffs on non-conditional attributes

We first study base table update diffs on attributes of R that do not participate in any join or selection condition in V_{spj}. Consider such an update t-diff/i-diff on attributes $\bar{A''}$ of R:

$$\mathcal{D}^u_R = \Delta^u_R(\bar{I}, \bar{A'}_{pre}, \bar{A''}_{post})$$

where $\bar{I}$ is the key of R.

Since we are interested in the IVM of an update on R, we decompose the condition c into 3 subconditions c_R, c_{rest}, c_{R-rest} in conjunctive normal form, s.t. every one of their

conjuncts involves only attributes of R, only attributes of $R_1, \ldots, R_n$ and both attributes of R and $R_1, \ldots, R_n$, respectively. It is easy to see that V_{spj} can be computed through the algebraic expression $\pi_S(\sigma_{c_R} R \bowtie_{c_{R-rest}} E)$, where $E = \sigma_{c_{rest}}(R_1 \times \ldots \times R_n)$.

In general, the i-diff Δ_R^u (resp. t-diff $\mathcal{D}_R^u$) will lead to $\Delta_{V_{\text{spj}}}^u$, $\Delta_{V_{\text{spj}}}^+$ and $\Delta_{V_{\text{spj}}}^-$ in the ID-based approach (resp. $\mathcal{D}_{V_{\text{spj}}}^u$, $\mathcal{D}_{V_{\text{spj}}}^+$ and $\mathcal{D}_{V_{\text{spj}}}^-$ in the tuple-based approach). However, since the updated attributes $\bar{A''}$ of R do not participate in the join condition c_{R-rest} or the selection condition c_R, the update diff on R will only lead to an update diff on V_{spj}. Furthermore, since the selection σ_{c_R} simply filters out tuples of $\mathcal{D}_R^u$, it has the same effect as using a smaller initial diff, and is therefore ignored in the rest of the analysis.

$\mathcal{D}$/Δ-script. The scripts returned by the IVM algorithms are in Table 2.2. The ID-based approach simply propagates the base table diff by exploiting the fact that no tuples need to be inserted or deleted from the view, since the modified attributes $\bar{A''}$ do not participate in the join condition.

Table 2.2. Scripts returned by the IVM algorithms

ID-based approach	Tuple-based approach
$\Delta_{V_{\text{spj}}}^u = \mathcal{D}_R^u$ APPLY $\Delta_{V_{\text{spj}}}^u$	$\mathcal{D}_{V_{\text{spj}}}^u = \pi_{\bar{S}} \mathcal{D}_R^u \bowtie_{c_{R-rest}} E$ APPLY $\mathcal{D}_{V_{\text{spj}}}^u$

Cost analysis. Both the ID-based IVM cost and the tuple-based IVM cost are the sum of the *diff computation cost* of $\Delta_{V_{\text{spj}}}^u$ (respectively $\mathcal{D}_{V_{\text{spj}}}^u$) and the *view modification cost*, which is the cost of applying the modifications dictated by $\Delta_{V_{\text{spj}}}^u$ (respectively $\mathcal{D}_{V_{\text{spj}}}^u$) on the materialized view. We measure both costs in terms of block accesses to indices and tuples. Employing common assumptions on the index structures[9], the cost of retrieving (using an index) the m tuples whose $\bar{X}$ attributes have given $\bar{x}$ values can be approximated by $1+m$ (i.e., 1 index lookup and m tuple accesses). We next analyze each of the cost components.

[9] We assume that indices satisfy the following conditions:
1. An index is either a hash index, or a B-tree with leaf nodes in secondary storage and non-leaf nodes in memory.
2. The retrieved tuples, if any, are not clustered together.
3. Caching of index leaves and/or tuples has minimal effects on the overall cost, as the cache is significantly smaller than the database.

Diff computation cost. Since the ID-based approach simply propagates Δ^u_R to the view as is, its diff computation cost is zero. On the other hand, the cost of the tuple-based IVM varies widely depending on the computation of $\mathcal{D}^u_{V_{\text{spj}}} = \pi_{\bar{S}} \mathcal{D}^u_R \bowtie_{c_{R-rest}} E$.

We consider the common case where the join condition c_{R-rest} is a conjunction of equalities of the form $R.J = R_i.J_i$. Furthermore, we assume that the database is optimized for tuple-based IVM, having all necessary indices for the efficient computation of $\mathcal{D}^u_{V_{\text{spj}}} = \mathcal{D}^u_R \bowtie_{c_{R-rest}} E$. Since the diff-table $\mathcal{D}^u_R$ is considered in the IVM literature to be smaller than the base tables, the DBMS will typically execute the above query through a *diff-driven loop* plan: For each tuple t of $\mathcal{D}^u_R$ it executes the subplan $\sigma_{c'_{R-rest}} E$, where c'_{R-rest} is an instantiation of the c_{R-rest} condition where the attributes of R have been replaced with their values in t. Let us name a the average number of accesses performed for each tuple of $\mathcal{D}^u_R$, i.e., the average number of accesses in each execution of $\sigma_{c'_{R-rest}} E$. Then the diff computation cost of the tuple-based approach is $|\mathcal{D}^u_R|a$. The DBMS may also choose to evaluate the query with a plan other than a diff-driven loop, but this is expected to happen only when the diff tables are very large, when the use of an IVM approach becomes questionable.

View modification cost. To apply $\Delta^u_{V_{\text{spj}}}$ (resp. $\mathcal{D}^u_{V_{\text{spj}}}$) to the view, the DBMS will typically utilize the view index to locate the view tuples that need to be modified. In either of the approaches there will be as many view index lookups as tuples in the view diff (i.e., $|\Delta^u_{V_{\text{spj}}}| = |\mathcal{D}^u_R|$ lookups for the ID-based and $|\mathcal{D}^u_{V_{\text{spj}}}|$ lookups for the tuple-based approach, respectively). Once the to-be-modified view tuples have been identified (which are in both cases equal to $|\mathcal{D}^u_{V_{\text{spj}}}|$), both approaches will incur $|\mathcal{D}^u_{V_{\text{spj}}}|$ view tuple accesses to update them. Table 2.3 shows the view index lookups and view tuple accesses for each approach utilizing the i-diff compression factor $p = |\mathcal{D}^u_{V_{\text{spj}}}|/|\Delta^u_{V_{\text{spj}}}|$.[10]

Discussion. Table 2.3 summarizes the costs for the tuple-based and ID-based approach. Combining them leads to the following speedup ratio of the ID-based over the tuple-based

[10] In the unlikely case, where the DBMS chooses to identify the to-be-modified tuples by performing a full scan of the view instead of using the index, the view modification cost becomes the same for both approaches. In this case, the difference between the two approaches reduces to the difference between their computation costs.

Table 2.3. Costs of ID-based and tuple-based IVM on V_{spj}

Costs	ID-based	Tuple-based	
		Diff-driven loop plan	Other plan
Diff computation	0	$\lvert\mathcal{D}^u_R\rvert a$	E
View index lookups	$\lvert\mathcal{D}^u_R\rvert$	$\lvert\mathcal{D}^u_R\rvert p$	$\lvert\mathcal{D}^u_R\rvert p$
View tuple accesses	$\lvert\mathcal{D}^u_R\rvert p$		

approach (assuming a diff-driven loop plan for the tuple-based approach):

$$\text{Speedup ratio for } V_{\text{spj}} = \frac{\lvert\mathcal{D}^u_R\rvert(a+p+p)}{\lvert\mathcal{D}^u_R\rvert(1+p)} = \frac{a+2p}{1+p} \tag{2.1}$$

The relative performance difference between the two approaches varies depending on the value of the compression factor p. When $p \geq 1$, the ID-based approach is guaranteed to be more efficient with its absolute gain (i.e., the difference of accesses from the tuple-based approach) raising proportionally to p. When $0 < p < 1$, the ID-based approach is also more efficient in the typical case when the tuple-based approach has to do at least one access for each tuple in $\mathcal{D}^u_R$ (which means that $a > 1$). For the tuple-based approach to perform better it should be the case that $a < 1 - p$, which can be satisfied only if the following conditions simultaneously hold: (a) the tuple-based approach incurs $a < 1$ tuple accesses for each tuple in $\mathcal{D}^u_R$ on average (which can only happen if many tuples of $\mathcal{D}^u_R$ share the same join attribute values and thus the joined tuples can be retrieved once and reused for all of them) and (b) the ID-based approach is severely overestimating (i.e., $p \ll 1$).

Other diffs

Consider now insert and delete i-diffs/t-diffs on the base relation R, as well as update i-diffs/t-diffs on attributes of R that are involved in some join or selection condition in V_{spj}. Such base table diffs will lead in general to a combination of $\Delta^u_{V_{\text{spj}}}$, $\Delta^+_{V_{\text{spj}}}$ and $\Delta^-_{V_{\text{spj}}}$ in the ID-based approach (resp. $\mathcal{D}^u_{V_{\text{spj}}}$, $\mathcal{D}^+_{V_{\text{spj}}}$ and $\mathcal{D}^-_{V_{\text{spj}}}$ in the tuple-based approach). For the cases in which a base table diff is translated into an update or delete i-diff/t-diff on the view, the ID-based and tuple-

based algorithms will behave as described in Section 2.7.1 and thus the speedup ratio will be $\frac{a+2p}{1+p}$. On the other hand, in the case when a base table diff is translated into an insert i-diff/t-diff on the view, the ID-based and tuple-based algorithm will produce identical scripts, leading to a speedup of 1 (i.e., the ID-based algorithm will degenerate to the tuple-based algorithm but will not behave worse than the latter).

2.7.2 Aggregate Views

Consider (a) the aggregate view V_{agg}:

SELECT $\bar{G}, f(\bar{X})$ AS g FROM $R, R_1, \ldots, R_n$
WHERE c GROUP BY $\bar{G}$

whose FROM clause involves a single alias of a table R, and f is an associative aggregation function such as `sum`, (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R on R.

To ease exposition, we isolate the aggregation operator of the query, expressing it through the plan $V_{\text{agg}} = \gamma_{\bar{G}, f(\bar{X}) \to g} V_{\text{spj}}$, where V_{spj} is the algebraic plan for the SPJ query presented in Section 2.7.1.

We consider the case where the ID-based algorithm has determined that it is beneficial to create an intermediate cache storing the result of the SPJ subview V_{spj}, since otherwise both approaches will be performing identically. The tuple-based does not employ a cache, as it cannot benefit from it.

Similarly to the case of SPJ views, we distinguish between two cases depending on the type of the base table diff $\mathcal{D}_R/\Delta_R$.

Update diffs on non-conditional attributes

Consider an update t-diff/i-diff on attributes $\bar{A''}$ of R that are not involved in any condition in V_{agg}:

$$\mathcal{D}^u_R = \Delta^u_R(\bar{I}, \bar{A'}_{pre}, \bar{A''}_{post})$$

where $\bar{I}$ is the key of R.

Cache diff computation / modification cost. The ID-based approach maintains an intermediate cache, which is equivalent to V_{spj}. The cache incurs *cache diff computation* cost and *cache modification cost*, which are also equivalent to the diff computation and view modification cost of V_{spj}. There is no intermediate cache for the tuple-based approach.

View diff computation cost. We consider the case where f is *incrementally computable*. That is, there is an incremental function $f_{\mathcal{D}}$ that inputs $\mathcal{D}^u_{V_{\text{spj}}}(\bar{S}, \bar{X}_{pre}, \bar{X}_{post})$ where $\bar{S}$ is the key of V_{spj}, and outputs $\mathcal{D}^u_{V_{\text{agg}}}(\bar{G}, g_{pre}, g_{post})$. For example, when f is the `sum` function, $f_{\mathcal{D}}$ is also f, since `sum` is an associative aggregation function. Given $f_{\mathcal{D}}$, the tuple-based approach computes $\mathcal{D}^u_{V_{\text{agg}}} = \gamma_{\bar{G}, f_\Delta(\bar{X}) \to g} \mathcal{D}^u_{V_{\text{spj}}}$.

Given that $|\mathcal{D}^u_R|$ is much smaller than base tables, the number of groups in $\mathcal{D}^u_{V_{\text{agg}}}$ will be smaller than the number of groups in V_{agg}. The efficient implementation for γ is thus hash aggregation with in-memory buckets, which can be pipelined. Due to pipelining, no additional block accesses are incurred for γ. Thus, the tuple-based approach has the same diff computation cost for V_{spj} and V_{agg}.

As an optimization, the ID-based approach uses the UPDATE RETURNING statement to update the cache and return the result of the update in a single step. Thus, $\Delta^u_{V_{\text{spj}}}$ is obtained without additional accesses over cache modification costs. Similar to the tuple-based approach, $\Delta^u_{V_{\text{agg}}} = \gamma_{\bar{G}, f_\Delta(\bar{X}) \to g} \Delta^u_{V_{\text{spj}}}$, and the γ uses pipelined hash aggreation. Thus, the ID-based approach also has the same diff computation cost for V_{spj} and V_{agg}.

View modification cost. To apply $\Delta^u_{V_{\text{agg}}}$ (resp. $\mathcal{D}^u_{V_{\text{agg}}}$) to the view, both approaches will incur an index lookup and a tuple access per tuple in the i-diff (resp. t-diff). We denote the grouping compression factor $g = |\mathcal{D}^u_{V_{\text{agg}}}| / |\mathcal{D}^u_{V_{\text{spj}}}|$ in Table 2.4. **Discussion.** For V_{agg}, Table 2.4 summarizes the costs for both ID-based and tuple-based approaches. Combing them leads to the following speedup ratio of the ID-based over the tuple-based approach (assuming a diff-driven

Table 2.4. Costs of ID-based and tuple-based IVM on V_{agg}

Costs	ID-based	Tuple-based	
		Diff-driven loop plan	Other plan
Cache diff computation	0	–	
Cache index lookups	$\lvert\mathcal{D}^u_R\rvert$	–	
Cache tuple accesses	$\lvert\mathcal{D}^u_R\rvert p$	–	
View diff computation	0	$\lvert\mathcal{D}^u_R\rvert a$	E
View index lookups	$\lvert\mathcal{D}^u_R\rvert pg$		
View tuple accesses	$\lvert\mathcal{D}^u_R\rvert pg$		

loop plan for the tuple-based approach):

$$\text{Speedup ratio for } V_{\text{agg}} = \frac{a+2pg}{1+p+2pg} \tag{2.2}$$

For the tuple-based approach to perform better on V_{agg}, it should be the case that $\frac{a+2pg}{1+p+2pg} < 1$, which implies that $a < 1+p$. However, we will show that this is never possible. The average cost a spent by the tuple-based IVM for each diff tuple in $\mathcal{D}^u_R$ will always be at least $1+p$, as for each such tuple t the algorithm would have to perform (a) at least one index access to check whether t joins with some of the other relations in the FROM clause and (b) at least p tuple accesses to retrieve the tuples it joins with from the other relations to create p diff tuples in $\mathcal{D}^u_{V_{\text{spj}}}$. Note that these are the lower bounds for a that happen when the query contains just a single join $R \bowtie R_1$. For longer join chains the tuple-based IVM will have to perform additional index and tuple accesses, yielding even worse performance compared to the ID-based approach.

Note that the above analysis exploits the absence of multivalued dependencies in V_{spj} (which is a necessary condition for *idIVM* to create a cache). If there was a multivalued dependency, multiple tuples in $\mathcal{D}^u_R$ could share the same computation (and thus it would not be the case that each of them would incur at least $1+p$ accesses).

Other diffs

Similarly to the SPJ views, an insert and delete base table diff or an update diff on an attribute of R involved in a condition, might lead to insert, update and delete diffs on V_{spj}. For the cases that lead to deletes and updates on V_{spj} the cost will be the same as the one outlined in Section 2.7.2.

On the other hand when base table diffs lead to insert diffs on the view V_{spj}, the ID-based approach will be performing the same plan with the tuple-based approach but will also be inserting tuples into the cache. Let k be the number of tuples created in V_{spj} on average as a result of a single diff in $\mathcal{D}_R$. Then the cost of the tuple-based and ID-based approach is $|\mathcal{D}_R|(a+2pg)$ and $|\mathcal{D}_R|(k+a+2pg)$, respectively. Thus the speedup ratio is $\frac{a+2pg}{a+k+2pg}$. This speedup is less than 1 (meaning that the tuple-based approach will be performing better). However, this loss is bounded, as it is always 1 per tuple inserted into V_{spj}.

2.8 Experimental Evaluation

To compare the performance of ID-based and tuple-based IVM, we ran two sets of experiments. In our first set of experiments we studied the performance of both approaches on a diverse workload of views, by applying them on views commonly used in social networks. In our second set of experiments we studied the effect of varying different parameters on both the view and the data (such as selectivity, fanout, number of joins and base-table diff size).

In all cases we used *idIVM* to generate the Δ-script and $\mathcal{D}$-script (the latter was produced using our implementation of *idIVM* with tuple-based diff propagation rules instead of the standard ID-based propagation rules). We then measured the times to run each script in PostgreSQL. All experiments were run using Ubuntu LTS 12.04, OpenJDK JRE 7 and PostgreSQL 9.1, on top of an Amazon Web Services (AWS) m1.large dedicated instance configured with 1,200 input/output operations per sec (IOPS) for 16KB blocks. The configured IOPS provide predictable throughput for random block accesses. Each experiment was run with cold PostgreSQL page buffers and

Relation	Tuples
User	1M
FriendList	100M
Microblog (i.e. tweets)	20M
Retweets	4M (#tweets $\times$ 10% $\times$ 2 retweets per tweet)
Mentions	8M (#tweets $\times$ 20% $\times$ 2 mentions per tweet)
Rel_Event_Microblog	16M (#tweets $\times$ 40% $\times$ 2 events per tweet)

(a) BSMA relation sizes

Query	Description
Q7	Mentioned users within a time range
Q10	Users who are retweeted within a time range
Q11	Pair of retweeting users, grouped by retweeting times
Q15	Users talking about events within a time range
Q18	Pairwise count of mentions
Q*1	Aggregate of friends of friends within the same city
Q*2	Aggregate of retweeters for every user
Q*3	Aggregate of users who tweet about topics

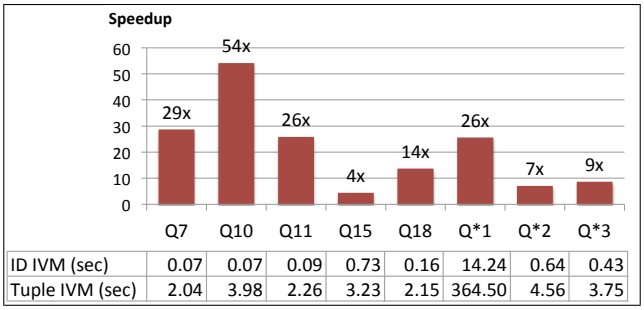
(b) BSMA queries and additional Queries

Figure 2.9. Configuration of social analytics experiments

Linux disk buffers, which is the common case when large number of views need to be maintained.

2.8.1 IVM in social analytics

To study the relative performance of ID-based and tuple-based IVM on a diverse set of real queries, we applied both IVM approaches to a workload of analytics views over social media. Maintaining analytics over social media is a primary use case for IVM techniques because: (a) large base tables are produced by social media such as Twitter and Facebook, (b) rapid, frequent



	Q7	Q10	Q11	Q15	Q18	Q*1	Q*2	Q*3
ID IVM (sec)	0.07	0.07	0.09	0.73	0.16	14.24	0.64	0.43
Tuple IVM (sec)	2.04	3.98	2.26	3.23	2.15	364.50	4.56	3.75

Figure 2.10. Speedup & IVM time for extended set of BSMA queries

updates occur on the base tables, and (c) analytic views that monitor metrics and trends need to be updated continuously.

To generate the workload, we utilized the Benchmark for Social Media Analytics (BSMA) [56]. Figure 2.9a shows the size of the relations generated, while Figure 2.9b provides a summary of the workload, which comprises:

- 100 update diffs on the **User** table for attributes *tweetsnum* (i.e number of tweets) and $favornum$ (i.e. number of favorites)

- Views corresponding to queries Q7, Q10, Q11, Q15 and Q18 from BSMA, which exhibit join chains and aggregates, hence resulting in high cost for view re-computation. These queries are also minimally extended to: (a) extend the `SELECT` clause with attributes *tweetsnum* and $favornum$ (b) remove the `ORDER BY` and `LIMIT` and the ID parameter in the WHERE clause in order to create larger views where the benefit of the IVM becomes apparent.

- An additional 3 aggregate views over the BSMA schema, labeled as Q*1, Q*2 and Q*3. Whereas queries Q7, Q10, Q11, Q15 and Q18 include aggregation, this aggregation is not affected by the updated attributes. Since, as we have discussed in Section 2.6 the ID-based and tuple-based approaches behave differently in the presence of aggregates affected by the updates, we designed views Q*1, Q*2 and Q*3 to include such aggregates.

Figure 2.10 shows the speedup ratio of the ID-based over the tuple-based approach for each of the views. The speedup varies widely between the 8 different views and its value does not seem to be determined by whether a view contains an aggregate affected by an update or not. From the reported views, it is interesting to look at the extreme cases with either very high or low speedup. Queries Q10 and Q*1 create a huge benefit for the ID-based IVM due to the fact that the tuple-based IVM has to incur a large number of data accesses, which can be avoided by the ID-based IVM. Interestingly, this need for data accesses is created in different ways by each of

Relation	Tuples	Size
parts	5M	170MB
devices	5M	170MB
devices_parts	50M	3GB

(a) Relation sizes

Parameter	Defaults
d: Diff size	200
s: Selectivity	20%
f: Fanout	10
j: Joins	2

(b) Parameters

$$\Delta^{u}_{parts} = \mathcal{D}^{u}_{parts}(\underline{pid}, price_{pre}, price_{post})$$

(c) Base-table Diff

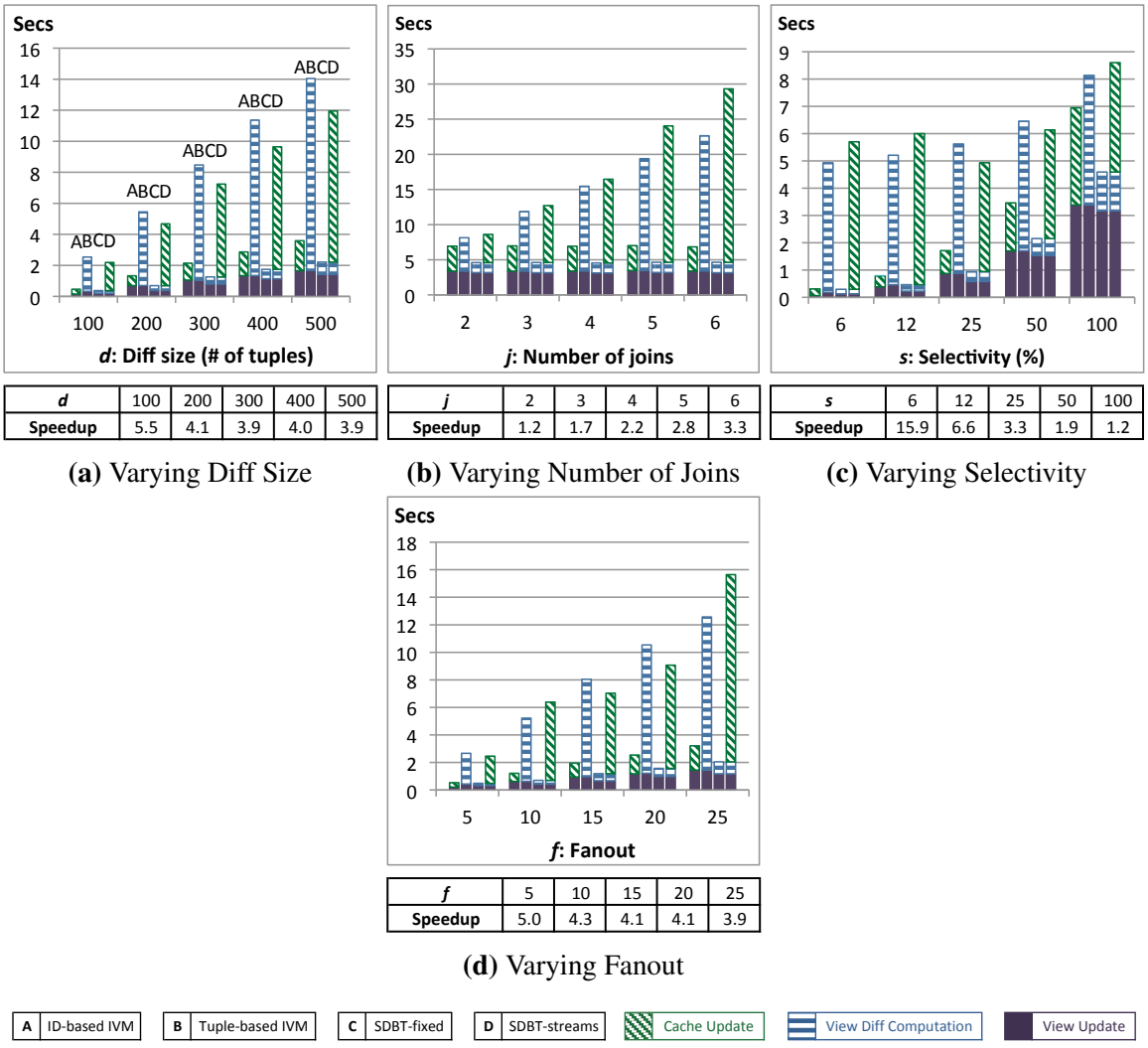
Figure 2.11. Configuration of varying parameter experiments

the two queries. In Q10 it is created by a long join chain (Q10 joins 4 relations), while in Q*1 it is created by a combination of a long join chain with a high selectivity that appears at the end of the join chain (so that the tuple-based has to perform a lot of data accesses before it can decide that a tuple will be dropped). Finally, Q15 displays a relatively low speedup because the resulting view and the number of tuples that need to be updated in the view is very large. This makes the view update time component (which is shared by both the ID-based and the tuple-based approaches) dominate the IVM cost, thus leading to a relatively small speedup. However, it should be noted that even in this case the ID-based approach outperforms the tuple-based approach by a factor of 4.

2.8.2 Effect of data & query parameters

To study in a more controlled fashion how the structure of the view and the data affects the ID-based and tuple-based IVM, we next ran a second set of experiments, in which we picked a single view and measured the performance of both ID-based and tuple-based IVM on maintaining the view, while varying different parameters of both the view definition and the underlying data. For ease of exposition we employed the view used in our running example and shown in Figure 2.5a.

Experimental setup. Figure 2.11 shows the properties of the dataset and the view used in the experiments. Figure 2.11a shows the sizes of the relations, Figure 2.11c presents the employed base-table diff (which captures updates on the prices of parts) and Figure 2.11b lists the



d	100	200	300	400	500
Speedup	5.5	4.1	3.9	4.0	3.9

(a) Varying Diff Size

j	2	3	4	5	6
Speedup	1.2	1.7	2.2	2.8	3.3

(b) Varying Number of Joins

s	6	12	25	50	100
Speedup	15.9	6.6	3.3	1.9	1.2

(c) Varying Selectivity

f	5	10	15	20	25
Speedup	5.0	4.3	4.1	4.1	3.9

(d) Varying Fanout

Figure 2.12. View maintenance time of ID-based IVM vs tuple-based IVM and two DBToaster-inspired systems for varying parameters

parameters that we varied in the experiments and their default values. We varied four parameters: (a) The size d of the base-table diff (i.e., the number of price updates that happened), (b) the number of joins performed by the view (as we will explain later we extended the view that by default performs two joins with additional joins), (c) The selectivity s of the selection condition *category="phone"* (i.e., the percentage of **devices** tuples that satisfy the condition), and (d) the fanout f from **parts** to **devices_parts**, i.e. the number of parts for each device. (Note that the fanout from association table **devices_parts** to entity table **devices** is always 1.) For each experiment varying a parameter, we used for all other parameters their default values shown in

Figure 2.11b.

Figure 2.12 shows the view maintenance times for ID-based IVM versus tuple-based IVM and the resulting speedup of ID-based over tuple-based IVM. The cost of each approach is broken down to its components. Column A represents ID-based IVM, with the top stack (diagonally striped) corresponding to cache update time (recall that the input of an aggregate is materialized as an intermediate cache), and the bottom stack (solid colored) corresponding to view update time. No stack is shown for the cache/view diff computation time as both are negligible. Column B represents tuple-based IVM, with the top stack (horizontally striped) corresponding to view diff computation time, and the bottom stack (solid colored) corresponding to view update time. No stack is shown for cache update time, since as we have explained the tuple-based approach does not use a cache as it cannot benefit from it. Columns C and D correspond to simulations of DBToaster [2], which we discuss in Section 2.8.3. We next explain the effect of varying each of the parameters.

Varying the base-table diff size. Figure 2.12a illustrates the effects of varying the diff size d linearly from 100 to 500 tuples[11]. As shown on the figure, the speedup stays within 4-5. The experiment also shows a slight downward trend on the speedup. This is because as d increases, the chance also increases for reading a block of the **devices** table (out of its 20k blocks) from PostgreSQL's page buffers. In the running example, this buffering benefits tuple-based IVM, but not ID-based IVM.

Varying the number of joins. Figure 2.12b illustrates the effect of joins on the speedup. For this experiment we consider more complex views by augmenting the original view definition with j additional joins as follows: (i) **devices_parts** is joined with tables $R_1 \ldots R_j$, such that each join is 1-to-1 on (pid, did). This simulates joins across vertically-decomposed tables, which is common practice in data warehousing. (ii) The selection $\sigma_{\text{category="phone"}}$ is disabled in order to focus on the effects of each additional join. Figure 2.12b shows that the total running time

[11] Similar trends can be observed for diff sizes up to 15,000 tuples. This is the point where it is beneficial to recompute the view rather than apply IVM, as discussed in prior work [29].

of ID-based IVM is unaffected by linearly varying j between 2 (i.e., the original view with no additional joins) to 6 (the view with four additional joins). On the other hand, the total running time of the tuple-based IVM increases with each additional join, making the speedup of ID-based IVM arbitrarily high, as the complexity of the query increases. This happens due to the fact that tuple-based IVM has to perform all joins in order to compute the entire view tuples that have to be modified, in contrast to the ID-based IVM that can simply propagate the base-table diff to the view and avoid performing any joins.

Varying the selectivity of the selection condition. Figure 2.12c shows how the speedup is affected by the selectivity s (i.e., the number of **devices** tuples that satisfy the condition). We vary s on a log scale from 6% to 100%. Allowing more tuples to pass through the selection adversely affects the performance of ID-based IVM. The reason is that more tuples of the **devices** table join with the other tables and thus the size of the intermediate cache employed by the ID-based approach increases. This in turn leads to a higher cost of updating the cache. It should be noted however that even at 100% selectivity, ID-based IVM is faster than tuple-based IVM, albeit at a lower speedup of 1.2. Thus, ID-based IVM is at least on par with tuple-based IVM, and performs better in the common case where the selection filters a subset of the base table tuples.

Varying the fanout. Finally, Figure 2.12d illustrates the effects of varying the fanout f of the join (**parts**, **devices_parts**) linearly between 5 to 25. For all values of the fanout, the ID-based IVM performs better than the tuple-based IVM by a factor of 4-5 times.

We highlight that the ID-based approach consistently outperforms the tuple-based approach. This is the case even though the experimental conditions were designed to explicitly benefit the tuple-based IVM. In particular, (a) we assumed the existence of appropriate base table indices to speedup tuple-based joins (without counting the associated index maintenance cost) and (b) we did not use a cache for the tuple-based IVM, to avoid the cache maintenance cost, since tuple-based maintenance of associative aggregate functions does not benefit from caches. When these optimizations are inadmissible, ID-based IVM will exhibit an even higher speedup.

2.8.3 Comparison to the state of the art

Finally, we compared *idIVM* to DBToaster [2]; the current state of the art IVM system, which, while being essentially a tuple-based system, has been shown to significantly outperform prior IVM approaches. DBToaster's performance is the result of five major optimizations: (a) performing IVM one diff tuple at a time (which leads to reducing $\mathcal{D}$-script joins with a diff table into selections), (b) compiling the $\mathcal{D}$-script into code, instead of SQL statements, (c) utilizing an in-memory implementation, (d) aggressively pushing aggregations down, and (e) materializing a large number of intermediate views (i.e., caches) that are used to maintain the original view and each other. On the other hand, *idIVM* benefits most from using (a) ID-based diffs and (b) update diffs (in contrast to DBToaster, where updates are simulated through inserts and deletes).

These differences make the two systems not directly comparable. Since *idIVM* could in principle also benefit from DBToaster's optimizations a-d, we next focus on comparing *idIVM* to the intermediate view materialization strategy used by DBToaster. To this end, we designed a DBToaster-inspired implementation (denoted as Simulated DBToaster or SDBT) that runs on top of a DBMS and uses the same intermediate views as the original DBToaster implementation (up to aggregation push-down). We then executed SDBT on all scenarios considered in Section 2.12. Since, in contrast to *idIVM*, DBToaster creates different intermediate views depending on the types of allowed base-table diffs, we ran two different versions of SDBT: one assuming that only diffs to the **parts** table are possible (referred to as SDBT-fixed) and one assuming that all base tables may change (referred to as SDBT-streams). Columns C and D in Figure 2.12 show the times of SDBT-fixed and SDBT-streams, respectively. We observe that *idIVM* in all cases significantly outperforms SDBT-streams, while it is in most cases slightly slower then SDBT-fixed. It should be noted however that we allowed both SDBT-fixed and SDBT-streams to employ update t-diffs. Had they simulated updates through inserts and deletes, as is the case in DBToaster, their performance would have been worse.

Note that SDBT captures only one of the optimizations used in DBToaster. Furthermore,

SDBT (alike *idIVM* and tuple-based IVM) operates on large data, residing in secondary storage and managed via a database (PostgreSQL in this case). Due to the mix of optimizations involved and its main memory orientation, DBToaster behaves differently from SDBT. Experiments we conducted with DBToaster showed for instance that the compilation to code and in-memory implementation lead to 50-300 times faster execution than the PostgreSQL-based SDBT-fixed. On the other hand, the in-memory execution severely limits DBToaster's scalability (allowing it to scale only up to 2% of the data size used in our experiments when diffs are allowed on all base tables). Moreover, the lack of set-processing makes DBToaster's performance deteriorate much faster than SDBT with increasing diff sizes (e.g, DBToaster's speedup over SDBT-fixed drops from 300x for a diff size of 100 tuples to 50x when the diff size becomes 500).

2.9 Generalization to SQL++

In subsequent sections, we generalize *idIVM* to handle SQL++, a nested and schema-less data model and query language. The main extensions and contributions are:

- Handling nested data: The i-diff format, system architecture and i-diff propagation rules are generalized to handle nested data structures in SQL++ with the inferred schema information from view definition. SQL++ data model is described in Section 2.10 and the generalized i-diff format is described in Section 2.12.

- Generalized provenance: Since SQL++ data model is schema-less and therefore has no notion of IDs, the ID-based information in early sections has been generalized as built-in provenance in SQL++ data model and query language (see Section 2.11).

- i-diff propagation rules for SQL++ operators: For the scope of SQL++ operators that we consider, i-diff propagation rules are listed. In particular, we show that rules for equivalent SQL++ query plans behave equivalently. This is described in Section 2.13.

- Schema vs schema-less: Although the SQL++ data model may not have any schema-

level information declared, from the view definition there is still information regarding the schema that can be statically inferred. *idIVM* utilizes this information to improve performance of IVM, including building indexes and delta scripts at view definition time, while at view maintenance time it also dynamically handles any i-diffs that show up deeper than the inferred schema. Finding and applying i-diffs deeper than the inferred schema are discussed in Section 2.14 and 2.15.

We use the following example to demonstrate how the extended *idIVM* handle nested data and view in SQL++.

Example 2.9.1. Consider the Yelp Open Dataset [33] that stores businesses, reviews and users. The **businesses** table stores data of businesses (restaurants). An example **businesses** tuple is as follows:

```
{
  business_id: 'tnhfDv5Il8EaGSXZGiuQGg',
  name: 'Garaje',
  neighborhood: 'SoMa',
  address: '475 3rd St',
  city: 'San Francisco',
  categories: [
    'Mexican',
    'Burgers',
    'Gastropub'
  ],
  // ...
}
```

The **reviews** table stores data of business reviews and references the **businesses** table and the **users** table by ID. An example **reviews** tuple is as follows:

```
{
  review_id: 'zdSx_SD6obEhz9VrW9uAWA',
  user_id: 'Ha3iJu77CxlrFm-vQRs_8g',
  business_id: 'tnhfDv5Il8EaGSXZGiuQGg',
  stars: 4,
  date: '2016-03-09',
  text: 'Great place to hang out after work...',

  // ...
}
```

The **users** table stores data of users and has nested data including **friends** for friends list and **elite** for elite years. An example **users** tuple is as follows:

```
{
  user_id: 'Ha3iJu77CxlrFm-vQRs_8g',
  name: 'Sebastien',
  fans: 1032,
  friends: {{
      'wqoXYLWmpkEH0YvTmHBsJQ',
      'KUXLLiJGrjtSsapmxmpvTA',
      '6e9rJKQC3n0RSKyHLViL-Q'
  }},
  elite: {{
      2012,
      2013
  }},

  // ...
}
```

Note that **friends** and **elite** were originally JSON arrays in Yelp Open Dataset. Since we restrict the scope of data model in this work to exclude arrays, we have changed arrays in the example to bags (e.g., $\{\{\ldots\}\}$). Formal description of the data model and scoping can be found in Section 2.10.

Consider the following view **V** returning list of businesses in 'San Francisco' and their reviews in a nested bag by using SQL++ GroupBy operator:

```
CREATE VIEW V AS
SELECT business_id, (
  SELECT *
  FROM reviews NATURAL JOIN users
  WHERE business_id=businesses.business_id
) AS full_reviews
FROM businesses
WHERE city = "San Francisco"
```

A base table diff Δ^b that modifies an elite year of a user

```
<users: {type: update,
  diff: {{
    #(uid=Ha3i) {
      type: update,
      diff: {
        elite: {
          type: update,
          diff: {{
            #(eid=9kjb) { type: update, post: 2014 }
          }}
        }
      }
```

```
    }
  }}
}>
```

is transformed to the following view i-diff Δ^V by *idIVM*

```
{{
  #() <
    type: update,
    diff: {
      full_reviews: {
        type: update,
        diff: {{
          #(uid=Ha3i) {
            type: update,
            diff: {
              elite: {
                type: update,
                diff: {{
                  #(eid=9kjb) { type: update, post: 2014 }
                }}
              }
            }
          }}
        }
      }
  >
}}
```

Notice that in the above i-diff, the provenance of the first level (i.e., the businesses) is left uncon-

strained as `#()`, which makes the i-diff applicable to multiple businesses and multiple reviews in **V** as the user can review a business multiple times. Such i-diffs can be applied efficiently because *idIVM* employs a smart provenance-based index, to be described in Section 2.15.1, which can target directly into the second level. Also notice that the diff to elite years is deeper than the view definition of **V** and therefore is unknown from any inferred schema of **V**, whereas *idIVM* can handle such i-diffs automatically.

The above view **V** has an equivalent view **V1** as follows:

```
CREATE VIEW V1 AS
SELECT business_id, full_reviews
FROM businesses NATURAL LEFT OUTER JOIN (
  reviews NATURAL JOIN
  users
)
WHERE city = "San Francisco"
GROUP BY business_id INTO full_reviews
```

Notice that **V** uses a sub-query to create the nested **full_reviews** table, whereas **V1** uses a combination of JOINs and GROUP BY. We will show in this work that *idIVM* can handle both views equivalently well by transforming base table diff Δ^B to view diff Δ^V with partial provenance.

2.10 SQL++ data model

In this section we describe SQL++ data model and query language, which *idIVM* is generalized to handle.

2.10.1 Data model

We adopt the SQL++ data model of [42], which is a superset of both SQL relational tables and JSON. Figure 2.13 shows the BNF grammar for SQL++ values. The provenance part

1	**value**	→	**provenance `null`**
2			| **provenance `missing`**
3			| **provenance scalar**
4			| **provenance complex**
5	**provenance**	→	**`#` `(` key `=` value (`,` key `=` value)* `)`**
6	**scalar**	→	**primitive**
7			| **enriched**
8	**primitive**	→	**`'` string `'`**
9			| **number**
10			| **`true`**
11			| **`false`**
12	**enriched**	→	*type* **`(` primitive (`,` primitive)* `)`**
13	**complex**	→	**tuple**
14			| **array**
15			| **bag**
16	**tuple**	→	`{` (**name `:` value (`,` name `:` value)***)? `}`
17	**array**		| **`[`** (**value (`,` value)***)? **`]`**
18	**bag**	→	`{{` (**value (`,` value)***)? `}}`

Figure 2.13. BNF Grammar for SQL++ Values

of the BNF is an extension of *idIVM* and is discussed in Section 2.11.1.

A SQL++ database generally contains one or more SQL++ top-level *named values*. A name is a string and is unique. A value is a scalar, complex, missing or null. A complex value is either a tuple or a collection. A tuple is a set of attribute name/value pairs, where each name is a unique string with the tuple (as in SQL).

A collection is either an array or a bag. Both arrays and bags may contain duplicate elements. An array is ordered (similar to a JSON array) and each element is accessible by its ordinal position. In contrast, a bag is unordered (similar to a SQL table) and its elements cannot be accessed by ordinal position.

A scalar value is either primitive or enriched. Primitive values are the scalar values of the JSON specification, i.e. strings, numbers or booleans. Enriched values are extensions over JSON, and are specified using a type constructor over primitives.

The elements of an array/bag can be any kind of value and can be heterogeneous. That is, there are no restrictions between the elements of an array/bag. Furthermore, unlike SQL where the values are tables that have homogeneous tuples that have scalars, SQL++ allows arbitrary composition of complex values.

Scope

In this work, we restrict the scope of data model to exclude arrays, `missing` and enriched values.

2.10.2 Algebra of query language

Table 2.5. SQL++ Operators

Novel Semi-Structured Operators		
1	ScanCollection	$\ggg^C$
2	ScanTuple	$\ggg^T$
3	Ground	
4	NavArray	[ ]
5	NavTuple	•
6	FunctionCall	λ
7	ReturnArray	
8	ReturnBag	
9	ReturnTuple	
10	ReturnSingle	
11	ApplyPlan	α
12	Assign	
13	InnerCorrelate	
14	LeftCorrelate	
15	ConstructArray	
16	ConstructBag	
17	ConstructTuple	

Extensions of Relational Operators		
18	Select	σ
19	Project	π
20	InnerJoin	⋈
21	LeftJoin	⟕
22	FullJoin	⟗
23	GroupBy	γ
24	Sort	
25	OffsetLimit	
26	Union	⋓
27	Intersect	⋒
28	Except	\
29	Distinct	δ
30	Exists	
31	SemiJoin	⋉
32	AntiJoin	▷

By first translating a SQL++ query into an algebraic plan comprising SQL++ operators, *idIVM* is able to take an algebraic approach to rewrite each operator in the view query into operators that comprise the i-diff query. Table 2.5 lists the algebraic operators of SQL++. Since SQL++ is a superset of SQL, SQL++ operators include extensions of relational operators (e.g., Select, Project, Join operators and Set operators) as well as novel operators for semi-structured query processing (e.g., Return operators, Construct operators, ApplyPlan and InnerCorrelate). These include a class of operators e.g., set operations, join operators, returns, constructs, correlate, apply etc. The semantics of these operators can be found in Appendix D. Although this paper

focuses on the algebraic level of operators, there is a corresponding query language with syntax that can be found in [42].

One important note is that unlike classical relational operators that require source tables to have fixed schemas, SQL++ operators impose no schema requirements on the source data. A SQL++ operator inputs/outputs a collection of binding tuples. A binding tuple is a tuple of various variables. All variables can be inferred exclusively from the SQL++ query without accessing the source data. There are two exceptions regarding operators: (a) the Return operators, which input a collection of binding tuples and output an array/bag/tuple value, and (b) the ground operator which is used to bootstrap the query evaluation process by outputting a single binding tuple without receiving anything in its input.

An environment in SQL++[42] is also a binding tuple and used to model the data that is used by the query. Suppose Γ is the environment of a plan P. Consider an operator within P that inputs a bag of binding tuples B. For each input binding tuple $b \in B$, the operator will be evaluated in the environment $b||\Gamma$. In case of a nested plan that is attached to an operator (e.g., ApplyPlan and InnerCorrelate), its environment is the concatenation of the outer plan's environment and the new context introduced by the operator (e.g., input binding tuples to ApplyPlan and InnerCorrelate). In *idIVM*, environment Γ is available to every query by having the Ground operator produce Γ as the single binding tuple in its output. Since Ground is the only allowed leaf node of an algebraic plan, in this way Γ becomes available throughout the query.

Each SQL++ operator can be specified with one or more *terms*, denoted as $\ddot{x}$, that are evaluated in the environment of the operator. For example, $\sigma_{\ddot{c}}$ is specified with a term $\ddot{c}$ that is a boolean condition. A term is a SQL++ expression that is restricted to:

- A value literal (i.e constant)
- A variable
- A boolean logic expression (i.e. $\wedge$, $\vee$, $\neg$) of terms

- An equality condition of two terms
- An inequality condition (i.e. $<$, $\leq$, $>$, $\geq$) of two terms

For ease of exposition, we also use $\ddot{x}$ to denote the result of evaluating $\ddot{x}$ within the environment of its operator, wherever the meaning is unambiguous.

Scope

For SQL++ operators, we restrict the scope to a subset of SQL++ denoted by Q_{SPJADU} that includes SPJ, Scan, Navigate, InnerCorrelate, ApplyPlan, ConstructTuple, GroupBy and Aggregate (as function calls), which are colored in brown color in Table 2.5.

2.11 Extension of IDs to Provenance

To efficiently maintain a view, *idIVM* needs to know how data in the view is computed by keeping around the provenance of the data. In earlier sections, we have shown that *idIVM* uses IDs to track provenance. Since SQL++ data model is schema-less and IDs are not available, the generalized *idIVM* needs to extend both the data model to add provenance to the data, and the algebraic operators so that they can propagate provenance from input to output. The following subsections describe extensions to data model and then to algebraic operators.

2.11.1 Provenance extension to data model

To guarantee efficient incremental maintenance of a view, *idIVM* needs to be able to identify view tuples based on their provenance. To this end, we extend the SQL++ data model by attaching to each (nested) SQL++ value some *provenance*, as defined below:

Provenance. The *provenance* $\#(k_1{=}u_1, \ldots, k_m{=}u_m)$ of a SQL++ value is a set of one or more key-value pairs, where each unique key k_i maps to value u_i. Within a tuple value $\{n_1 : v_1, \ldots n_o : v_o\}$ (resp. array value $[v_1, \ldots, v_o]$, bag value $\{\{v_1, \ldots, v_o\}\}$), each value v_i has a provenance which is distinct from the provenance of all other values in the tuple (resp., bag or

array). That is, provenance is locally unique within a tuple/array/bag. Note that values within a tuple (resp., bag or array) do not need to have the same provenance keys.

Figure 2.13 shows the resulting BNF grammar for SQL++ values extended with provenance.

2.11.2 Provenance extension to query operators

Since *idIVM* i-diffs identify view data to be modified by its provenance, the view and any subview has to have provenance computed. Each supported operator is extended with provenance propagation logic, which is described as provenance inference rules in Appendix D.3. Table 2.6(a) shows the main idea of the provenance inference rules of each operator for inferring the provenance of output binding tuples from the provenance of input data. Here $Prov(R)$ stands for the provenance of R data, $key(R)$ stands for the key attribute and values of R when R is a relational base table, and $Prov(R) \cup Prov(S)$ means concatenating the provenance of R and the provenance of S from the input to form the provenance of the output. For most operators, provenance of nested data is simply propagated from input to output. Some operators have specific provenance inference rules for nested data, which is shown in Table 2.6(b). Given an SQL++ query plan, the provenance extension to the operators allows provenance to be computed for each operator's output from the bottom up.

Provenance keys may have name conflicts when an output provenance is combined from multiple input provenance. The solution to such conflicts, which is not described in each individual rule, is to rename the keys based on where they come from.

2.11.3 Additional operators for i-diff queries

The i-diff queries for *idIVM* may need to join values on their provenance, which cannot be done using existing SQL++ operators. Therefore several new operators are required in addition to those in Section 2.10.2.

Table 2.6. Operator provenance inference rules

Operator	Provenance of output binding tuples
Ground	`#( )`
$\sigma_{\ddot{c}}(B)$	$Prov(B)$
$\pi_{x_1,\ldots,x_n}(B)$	$Prov(B)$
$\ggg^{C}_{\ddot{c}\mapsto(x,y)}(B)$	$Prov(B) \cup Prov(c)$ if c is a variable, or $Prov(B) \cup key(c)$ if c is a relational base table
$\bullet_{(\ddot{x},\ddot{y})\mapsto z}(B)$	$Prov(B)$
$\bowtie_{\ddot{c}}(B^l,B^r)$	$Prov(B^l) \cup Prov(B^r)$
$\bar{\bowtie}_{\ddot{c}}(B^l,B^r)$	$Prov(B^l) \cup Prov(B^r)$ Note: if there is no match from B^r, then $Prov(B^r)$ values will be `null`
$\gamma_{(\ddot{x}_1\mapsto y_1,\ldots,\ddot{x}_n\mapsto y_n),g}(B)$	`#(` $y_1 : v_1, \ldots, y_n : v_n$ `)` where v_i is the value of variable y_i
$\text{Assign}_{\ddot{x}\mapsto y}(B)$	$Prov(B)$
$\text{InnerCorrelate}_P(B^l)$	$Prov(B^l) \cup Prov(P)$
$\text{ConstructTuple}_{(\ddot{x}_1:\ddot{y}_1\ldots\ddot{x}_n:\ddot{y}_n)\mapsto z}(B)$	$Prov(B)$

(a) Provenance for binding tuples

Operator	Nested Data	Provenance
$\ggg^{C}_{\ddot{c}\mapsto(x,y)}(B)$	x	$Prov(\ddot{c})$
$\bullet_{(\ddot{x},\ddot{y})\mapsto z}(B)$	z	$Prov(\ddot{x}.\ddot{y})$
$\gamma_{(\ddot{x}_1\mapsto y_1,\ldots,\ddot{x}_n\mapsto y_n),g}(B)$	tuples of g	$Prov(B)$
$\text{Assign}_{\ddot{x}\mapsto y}(B)$	y	$Prov(\ddot{x})$

(b) Provenance for nested data

ProvenanceSemiJoin

The ProvenanceSemiJoin operator $\hat{\ltimes}(B^l,B^r)$ semijoins B^l and B^r using provenance attributes in them. A tuple $b^l \in B^l$ is kept in the output if and only if there exists a tuple $b^r \in B^r$ so that all provenance attributes of b^r exist in the provenance of b^l with the same values. An example usage of this operator is to semijoin potential i-diffs with their corresponding data.

ProvenanceJoin

The ProvenanceJoin operator $\hat{\bowtie}(B^l,B^r)$ is the InnerJoin version of ProvenanceSemiJoin. It joins B^l and B^r using provenance attributes in them. For each input binding tuple $b^l \in B^l, b^r \in B^r$, the operator outputs $b^l \| b^r$ if all provenance attributes of b^r exist in the provenance of b^l with the same value, or vice versa. An example usage of this operator is to join input i-diff with input data. In this case, the operator effectively retains i-diffs for future use while appending the post

state of the data to the i-diff tuple.

ProvenanceUnion

The ProvenanceUnion operator $\hat{\cup}(B^l, B^r)$ unions B^l and B^r and eliminates duplicates using provenance. For each input binding tuple $b \in B^l$ or $b \in B^r$, if the provenance of b is unique across B^l and B^r, the operator outputs b. If b has the same provenance as other input binding tuples, the operator picks only one of them and output it. An example usage of this operator is in the i-diff queries for aggregate functions including SUM and COUNT.

2.12 SQL++ i-diff format & semantics

idIVM represents view maintenance time and view definition time diff information using i-diff instances and i-diff signatures respectively. This section discusses their format and semantics. Notice that even though Section 2.11 has extended ID-based information to provenance for SQL++ setting, in the IVM context we will still use the name ID-based diff (i-diff) for continuity.

2.12.1 i-Diff instance format

An i-diff instance describes changes that will happen to a value x, and in general is of the following form:

```
#(p) {pre: ..., post: ..., diff: ..., type: ...}
```

where **p** matches the provenance of x, **pre** and **post** correspond to x's pre-state and post-state, **diff** corresponds to the nested i-diffs (i.e., i-diffs of nested data of x), and **type** can be one of insert, delete, update, which are explained later in this section. An i-diff instance may match multiple values and be applicable to them, as long as its provenance **p** matches the provenance of the values.

For certain i-diff types, the **pre** and/or **post** can be optional as they are not required to update the final view. The optionality will be specified later in each i-diff type's section. For

i-diffs of intermediate results, **post** can be required to compute i-diffs of certain operators like Select or FunctionCall. If they are required by a per-operator IVM rule but absent in the i-diffs, the *idIVM* framework will automatically add **post** by joining the i-diffs with the original data (i.e., the view query). For example, for an operator that takes B as input, if its i-diff query requires **post** state of variable a that is not included in the input i-diff Δ_B, the framework of *idIVM* will automatically replace Δ_B with the following query snippet:

$$\pi_V \,\text{ConstructTuple}_{(\texttt{post:apost},\texttt{diff:adiff})\mapsto a}(\\ \pi_{V\setminus\{a\}}(\pi_{a\mapsto\texttt{apost}}(B) \,\hat{\bowtie}\, \bullet_{(a,\texttt{diff})\mapsto\texttt{adiff}}(\Delta_B)))$$

where V stands for the set of all variables. In some other cases, using the **pre** and **post** in an i-diff query is not required, but can improve IVM efficiency by reducing overestimation. The per-operator IVM rules in Section 2.13 mark such optional uses of **pre** or **post** in blue color so that the framework can instantiate these parts in the i-diff query only when **pre** or **post** exist.

The nesting structure of i-diff instances in *idIVM* is designed to follow the nesting structure of the corresponding values. At a particular level, the i-diff describes how the corresponding level of the value is changed. Among the different types of i-diffs, only update i-diffs for bag and tuple values can have nested i-diffs. Insert and delete i-diffs do not have nested i-diffs because their corresponding values are inserted or deleted in their entirety. Update i-diffs of scalar values do not have nested i-diffs either since scalar values have no nested data.

We will next describe update i-diff instances of bag and tuple values where nested i-diffs can exist. The i-diff semantics is defined using SQL++ DML, which can be found in [44].

Update i-diff instance of bags and tuples

An update i-diff instance Δ^{u} of a bag value b describes that the content of b is updated, where **pre** and **post** are optional and correspond to the pre and post-state of b. **diff** is a nested bag of i-diffs, each describing modifications to some of the values inside bag b. The

semantics of applying the update i-diff instance has the same effect of applying the following DML statement:

Apply Δ^u (to a bag value b):
    FROM b as bc
    FROM Δ^u.diff as dc
    WHERE PROVENANCE_MATCH(bc, dc)
    Apply dc to bc

An update i-diff instance Δ^u of a tuple value t contains optional **`post`** and **`pre`**, similar to the update i-diff instance of a bag value. **`diff`** is a nested tuple of key-value pairs, where each value is a nested i-diff describing modifications to the corresponding attribute in the original tuple. The semantics of applying an update i-diff has the same effect of applying the following DML statement:

Apply Δ^u (to a tuple value t):
    Apply Δ^u.diff.a_1 to $t.a_1$
    ...
    Apply Δ^u.diff.a_n to $t.a_n$

When post-state exists in the i-diff, an alternative way to apply update i-diffs to either a bag or a tuple x is to apply the post-state to x directly as follows:

Apply Δ^u (to a tuple value t):
    update(x, Δ^u.post)

Here we assume that the i-diff is well-formed, meaning that the two ways of applying i-diffs will

always lead to the same result. In other words, applying post-state should not produce a different result than applying the nested data.

As can be seen from the above update i-diffs, the nested format makes the i-diffs structurally similar to the original values. Compared to non-nested formats, the nested one can lead to simpler i-diff queries being generated. In many cases, an i-diff query with nested i-diff format will have a symmetry to the original view query.

Another benefit of nested i-diffs is that the i-diff can have finer grain information describing what is changed in the nested data. This leads to smaller i-diffs to be computed and to be applied to the final view. Also for i-diff propagation, diff queries can work with a smaller amount of data.

The other types of i-diff instances do not have nested i-diffs and are explained next.

Insert i-diff instance

An insert i-diff instance Δ^{+} describes a newly inserted value x under its parent. It does not have **`pre`** because x dost not exists in pre-state. There is no nested **`diff`** either since the entire x is newly inserted. When x is a child of a bag b, applying the insert i-diff Δ^{+b} has the same effect as applying the following DML statement:

Apply Δ^{+b} (to a bag value b):

insertbag(b, Δ^{+b}.post)

When x is the value of an attribute a under a tuple t, applying the insert i-diff Δ_V^{+t} is equivalent to applying the following DML statement:

Apply Δ^{+t} (to a tuple value t):

inserttuple(t, a, Δ^{+t}.post)

Delete i-diff instance

A delete i-diff instance Δ^- describes that x is removed from its parent, where **`pre`** is optional and corresponds to the pre-state of (possibly part of) x. The semantics of applying the delete i-diff has the same effect of applying the following DML statement:

Apply Δ^-:
    delete(x)

In case x is an attribute a under a tuple t, then the corresponding attribute-value pair is deleted from the parent tuple.

Update i-diff instance of scalars

An update i-diff instance Δ^u of a scalar value x describes that x is updated, where **`pre`** is optional and describes some pre-state of the updated values, **`post`** must exist and carry the post-state of the value. The semantics of applying the update i-diff has the same effect of applying the following DML statement:

Apply Δ^u:
    update(x, Δ^u.post)

Binding tuple update i-diffs

For a view query that outputs a bag v of binding tuples $\{\{t_1, \ldots, t_n\}\}$, its i-diff query outputs a bag d of binding tuples $\{\{d_1, \ldots, d_m\}\}$, where each d_i describes nested i-diffs to one or more binding tuples of t's. The semantics of applying the bag of binding tuple i-diffs has the same effect of applying the following DML statement:

```
FROM v as vc
FROM d as dc
WHERE PROVENANCE_MATCH(vc, dc)
#apply dc.a_1 to vc.a_1#
...
#apply dc.a_n to vc.a_n#
```

Effective i-diff instances

Given a set of i-diff instances $\bar{\Delta}$ for a view V, applying them on V leads in general to different results depending on the order of application. However, in this work we only look at sets of i-diffs where any order of applying them on V yields the same result. To this end, we define the notion of *effective* i-diff instances. Given the pre-state V^{pre} and post-state V^{post} of a view V, an i-diff instance Δ_V is said to be *effective* w.r.t. V^{pre} and V^{post} if for each value of V it reflects its final value. Formally, an i-diff instance is effective iff it satisfies the following properties:

- If an i-diff instance is an insert i-diff Δ^{+b} inserting a value under a parent value b: the value inserted by the i-diff exists in the post-state of b (i.e., $\Delta^{+b}.\text{post} \in b^{post}$).
- If an i-diff instance is a delete i-diff Δ^{-} deleting a value x from a parent value b: The value deleted by the i-diff does *not* exist in the post-state of b (i.e., $x \notin b^{post}$).
- If an i-diff instance is an update i-diff Δ^{u} updating a scalar value x: The value set by the i-diff is equal to the post-state of x (i.e., $\Delta^{u}.\text{post} = x^{post}$).
- If an i-diff instance is an update i-diff Δ^{u} updating a bag or a tuple x: First of all, the same properties of effectiveness shall apply recursively to the nested i-diffs Δ^{u}.diff. Also, if the i-diff contains the post-state part, then it is equal to the post-state of x (i.e., $\Delta^{u}.\text{post} = x^{post}$).

It can be shown that a set of effective i-diffs lead to the same result regardless of the order in which they are applied. In the following the i-diff instances we consider are assumed to

be effective. We will discuss in Sections 2.4 how *idIVM* makes sure that it always operates on effective i-diff instances.

2.12.2 i-Diff signatures

An i-diff signature is created at *idIVM* view definition time and statically describes what components (including any optional **`pre`** and **`post`**) an i-diff instance should have. Similar to an i-diff instance, an i-diff signature is recursively defined and can have nested i-diff signatures. An i-diff instance satisfies an i-diff signature when the former has all the components the latter declares, and every nested i-diff instance satisfies the corresponding nested i-diff signature recursively.

Example 2.12.1. Section 2.9 describes an i-diff instance Δ^V of the view **V** in our running example. One diff signature S_1 that Δ^V satisfies is as follows:

```
#() {
  type: update,
  diff: {
    #(uid) full_reviews: {
      type: update,
      diff: {
        user_id: { post }
      }
    }
  }
}
```

However, the following i-diff instance Δ_2^V will not satisfy the above signature since the diff type does not match:

```
{{
  #() <
```

```
    type: update,
    diff: {
      full_reviews: {
        type: update,
        diff: {{
          #(uid=U4n7) {
            type: delete
          }
        }}
      }
    }
  >
}}
```

Although i-diff instances in general need not have homogeneity, when they satisfy the same i-diff signature, they are homogeneous up to the level where the signature ends. In particular, when an update i-diff signature targeting a bag, its **`diff`**, if exists, is a single signature describing the nested i-diffs, which means that any i-diff instance that satisfies this signature should have its nested i-diff instances be homogeneous at the next level. This homogeneity requirement allows IVM i-diff queries to batch process i-diffs of the same type.

Example 2.12.2. Consider the following i-diff instance Δ_3^V which combines both update and delete i-diffs of **full_reviews**.

```
{{
  #() <
    type: update,
    diff: {
      full_reviews: {
        type: update,
```

```
        diff: {{
          #(uid=U4n7) {
            type: delete
          },
          #(uid=Ha3i) {
            type: update,
            diff: {
              elite: {
                type: update,
                diff: {{
                  #(eid=9kjb) { type: update, post: 2014 }
                }}
              }
            }
          }}
      }
    }
  >
}}
```

Notice that Δ_3^V is considered as a valid i-diff instance according to the semantics defined for i-diff instances. However, Δ_3^V does not satisfy the diff signature S_1 due to the homogeneity requirement imposed by the signature.

An i-diff signature may not have certain nested i-diff signature for i-diff instances that appear at view maintenance time, simply because information about such nested i-diffs cannot be inferred at view definition time. In this case, IVM i-diff queries will not assume any knowledge of the nested i-diffs, but propagate them as is to the view.

Example 2.12.3. Consider the i-diff instance Δ^V and i-diff signature S_1 defined in Example 2.12.2. Notice that S_1 does not have any information regarding elite years which Δ^V is about, since the existence of such data and hence its diffs may not be known at view definition time. Nonetheless, Δ^V still satisfies S_1 and can be applied to **V** by *idIVM* as valid instances of S_1.

Although structurally an i-diff signature tree from the root can bundle i-diffs of various values together, architecturally *idIVM* limits each i-diff signature from the root to be about a single bag. This means that each intermediate i-diff up to the bag has one nested i-diff which targets an ancestor of the bag, and the i-diff of the bag has no nested i-diffs for any further nested bags. The propagation of i-diff signatures is discussed in Section 2.3. The process of generating base table i-diff signatures is described in Section 2.14.1.

Example 2.12.4. The following i-diff signature, although valid by definition, will not be generated or handled by *idIVM*. Instead, *idIVM* would generate two separate i-diff signatures for **categories** and **full_reviews** in this case.

```
#() {
  type: update,
  diff: {
    $(cid) categories: {
      type: update,
      diff: {
        type: delete
      }
    },
    #(uid) full_reviews: {
      type: update,
      diff: {
        user_id: { post }
      }
```

```
    }
  }
}
```

2.12.3 i-Diffs deeper than view definition

Although the SQL++ data model may not have any schema-level information declared, from the view definition there is still information regarding the schema that can be statically inferred. The following pseudocode describes an *idIVM* algorithm that traverses the logical plan of the view definition query from the root and infers for each base table what are the attributes including nested ones that are accessed by the view definition query.

```
function markBaseTables(op, attrs):
  if op is a ScanCollection operator on a base table b:
    mark b as a base table referenced by the view;
    mark attrs as attributes in b;
  else
    let s_attrs  = attributes (a) from child operator(s) that correspond to any
      of attrs or (b) referenced in op;
    for each child operator c:
      let c_attrs = part of s_attrs that come from c;
      markBaseTables(c, c_attrs);
    endfor
  endif
end
```

Section 2.14.1 describes how to generate base table i-diff signatures using an extended version of the above algorithm.

In actual instances of the base tables, there may be values that are not covered by the inferred attributes from the view definition and hence do not affect computation of the view since

they are not referenced by the query. However, throughout the view computation, some of these values may be propagated along with their ancestor values and end up in the view.

Similarly for i-diffs, while i-diff signatures are statically inferred from the view definition, at runtime there could be i-diffs to values that are not covered by by the inferred attributes. We call such i-diff instances *i-diffs deeper than view definition*. As to be shown in Section 2.15, such i-diff instances need not be explicitly handled by i-diff queries, but can be applied to the view as part of their ancestor i-diff values. Furthermore, Section 2.16 discusses the cost of applying such i-diffs to the view.

2.13 i-Diff Propagation Rules for SQL++

This section describes the algorithm and examples of instantiating i-diff propagation rules for SQL++. Table E.1 to Table E.14 show the i-diff propagation rules for the SQL++ operators considered in this work.

The rule instantiation algorithm applies rules for all operators in a botoom-up fashion, and then composes operator-level instantiated rules into an i-diff query. This is very similar to relational *idIVM* as discussed in Section 2.4. However, the algorithm also needs to deal with nested plans which are unique to SQL++ algebra as follows. For an operator x that applies a nested plan in the context of each input binding tuple (e.g., ApplyPlan and InnerCorrelate), the algorithm invokes rule instantiation recursively into the nested plan with (a) the i-diffs of the input binding tuple as the triggering i-diff of the nested plan, and (b) the i-diffs of the base tables accessed by the nested plan. The i-diff queries of the nested plan are then incorporated in the i-diff query of x, according to the specific rules of x.

As described in Section 2.12, an i-diff instance has potentially three different parts: **`pre`**, **`post`** and **`diff`**. In case of binding tuple i-diffs, the three parts correspond to three variables under each binding tuple. As Table E.1 to Table E.14 show, an input i-diff signature can lead to one or more output i-diff signatures of the same type and/or different types. As a result, a typical

i-diff query operates on one particular part of the i-diff format that corresponds to the input i-diff type (i.e., **`post`** for insert, **`pre`** for delete, and **`diff`** for update as described in Section 2.12) and produces part of the i-diff format that corresponds to the output i-diff type. For example, for a π operator, since an input binding tuple update i-diff may lead to an output binding tuple update i-diff, an i-diff query that does this transformation will operate on the **`diff`** variable in the input which contains the input update i-diffs, and produce output update i-diffs under the **`diff`** variable in the output i-diffs. In addition, a binding tuple update i-diffs may also have the full pre-state and/or post-state of the binding tuple under the optional **`pre`** and **`post`** variables. In such case, the input **`pre`** and **`post`** can usually be transformed to the output **`pre`** and **`post`** respectively, similar to how binding tuple insert and delete i-diffs are transformed.

For simplicity of presentation, an IVM rule that only accesses one part of the i-diff (either **`pre`**, **`post`** or **`diff`**) uses a shortcut format assuming that each i-diff binding tuple directly carries the value of that particular part of i-diff, getting rid of one level of nesting in the original i-diff format. For an i-diff query to work on the actual i-diff format in practice, *idIVM* wraps the i-diff query with (a) a pre-transformation step that pulls the value of the variable of interest in the input i-diff binding tuple to the top level, and (b) a post-transformation step that wraps the result of the transformation as a tuple value that is assigned to the variable of interest in the output i-diff. The pre-transformation step can be achieved using a combination of $\bullet$ operators that navigates all attributes $a_1, \ldots, a_n$ of the variable u:

$$\bullet_{(u,a_1)\mapsto a_1} \cdots \bullet_{(u,a_n)\mapsto a_n}$$

whereas the post-transformation step can be achieved using the ConstructTuple operator that creates a variable w from variables $w_1, \ldots, w_n$ as follows:

$$\text{ConstructTuple}_{(w_1:w_1,\ldots,w_n:w_n)\mapsto w}$$

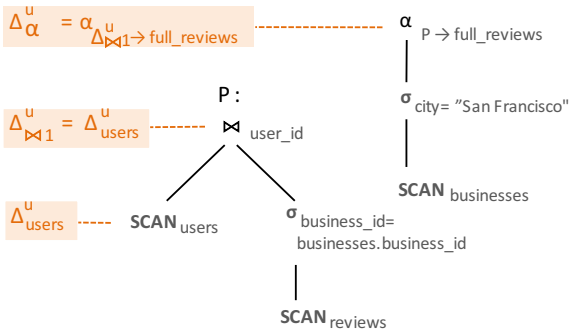


Figure 2.14. Algebraic plan of **V** (annotated by the Δ-script generator for Δ^u_{users})

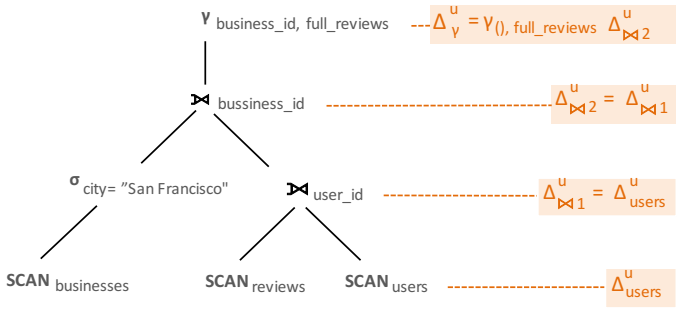


Figure 2.15. Algebraic plan of **V1** (annotated by the Δ-script generator for Δ^u_{users})

We use the above transformation to simplify the following examples. The first example shows how rules are instantiated for two equivalent plans.

Example 2.13.1. Consider an update i-diff signature Δ^u_{users} modeling updates on the fans attribute of table **users** of our running example. Figures 2.14 and 2.15 show the algebraic plans for **V** and **V1** respectively, and on the left the corresponding instantiated rules generated by the algorithm.

In Figure 2.14, $\Delta^u_{\bowtie 1}$ is the i-diff query for the nested plan P. Since $\Delta^u_{\bowtie 1}$ does not depend on the input of the outer ApplyPlan operator α, the i-diff query of the α operator is therefore $\alpha_{\Delta^u_{\bowtie 1} \rightarrow full\_reviews} Ground$ according to its rules specified in Table E.5. This i-diff query effectively runs the i-diff query of P as a nested query, and puts the i-diffs of the nested plan in a nested bag named $full\_reviews$. Since the ApplyPlan operator in the i-diff query is on top of a Ground operator, the top-level i-diff is an update with empty provenance.

In Figures 2.15, the two outer join operators propagate Δ^u_{users} as is according to rules in Table E.9. Therefore the input i-diffs to the GroupBy operator $\gamma_{business\_id, full\_reviews}$ does not

contain the grouping attribute *business_id*. Then according to the rules of GroupBy operator in Table E.11, the i-diff query for $\gamma_{business\_id, full\_reviews}$ is effectively grouping input i-diffs into a singleton group under nested bag $full\_reviews$ using empty grouping attribute. Since the rules of GroupBy operator uses overestimation, the resulting top-level i-diff is a singleton update with empty provenance, which applies to all businesses.

Based on the rule instantiation in 2.14 and 2.15, *idIVM* composes the rules up to the views **V** and **V1** respectively. The resulting i-diff queries, simplified for presentation purpose, are as follows:

$$V \quad : \quad \alpha_{\Delta^u_{users} \rightarrow full\_reviews}$$

$$V1 \quad : \quad \gamma_{(), full\_reviews} \Delta^u_{users}$$

The two i-diff queries are equivalent as they both return a singleton binding tuple that has a nested bag $full\_reviews$ that holds Δ^u_{users}. This demonstrates that *idIVM* is able to efficiently handle views that create nested data, no matter they are using ApplyPlan or GroupBy.

Example 2.13.2. To illustrate how i-diff propagation rules work for conditional updates and nested plans, consider an update i-diff signature $\Delta^u_{businesses}$ modeling updates of table **businesses** in our running example, including updates to the **business_id** and **city** attributes. Here we assume that the provenance of **businesses** table is independent of the **business_id** attribute, so that the latter is free to change without affecting provenance.

Figure 2.16 shows the algebraic plan for **V** and corresponding instantiated rules generated by the algorithm with respect to updates of **business_id**. First, the diff query $\Delta^u_{\sigma_1}$ for the select operator in the outer plan simply propagates $\Delta^u_{businesses}$ as is according to the rules in Table E.3 since $\Delta^u_{businesses}$ is considered a non-conditional update for the operator. Then at the ApplyPlan operator, the diff propagation algorithm is applied recursively to the nested plan P with the modification to **business_id** in the environment of the nested plan as the triggering i-diff. Since $\Delta^u_{business\_id}$ is a conditional i-diff to the nested select operator, the rules for the select operator

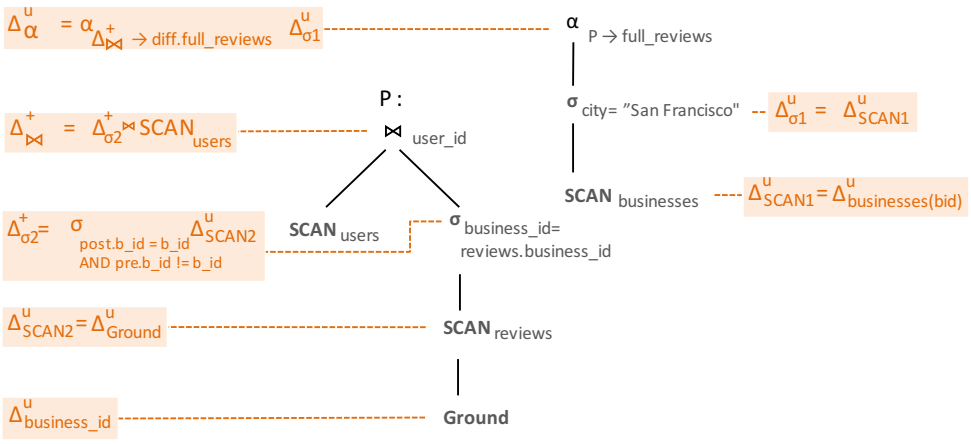


Figure 2.16. Algebraic plan of **V** (annotated by the Δ-script generator for $\Delta^u_{businesses}$ Part 1)

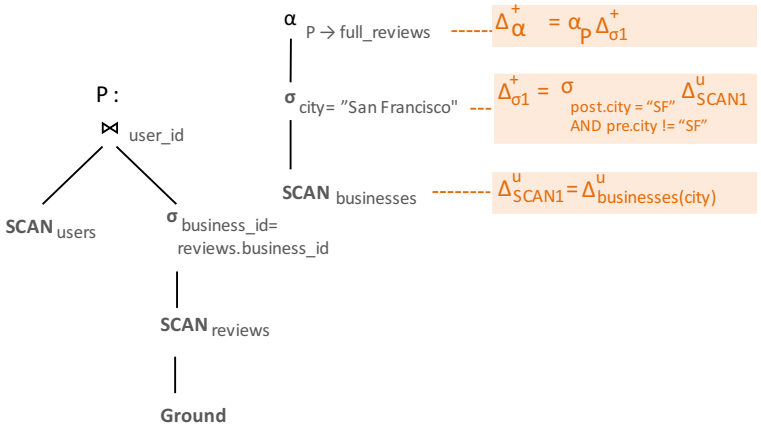


Figure 2.17. Algebraic plan of **V** (annotated by the Δ-script generator for $\Delta^u_{businesses}$ Part 2)

turns $\Delta^u_{business\_id}$ into insert, delete, and update output i-diffs. For simplicity of presentation, only $\Delta^+_{\sigma_2}$ is illustrated in Figure 2.16, while the handling for delete and update i-diffs are similar. Next, $\Delta^+_{\bowtie}$ joins the insert i-diffs $\Delta^+_{\sigma_2}$ with the **users** table to produce the output insert i-diffs of the InnerJoin operator, which also becomes a diff query for the nested plan. At the end, according to the rules of the ApplyPlan operator in Table E.5, the i-diff query for the ApplyPlan operator Δ^u_{α} takes $\Delta^u_{\sigma_1}$ as input and runs the nested i-diff query $\Delta^+_{\bowtie}$ as the nested plan.

Figure 2.17 shows the same algebraic plan and instantiated rules with respect to updates of **city**. Since the update i-diff is a conditional i-diff to the select operator in the outer plan, the rules of the select operator in Table E.3 turns $\Delta^u_{businesses}$ into insert, delete, and update output

i-diffs. For simplicity of presentation, only $\Delta^+_{\sigma_1}$ is illustrated in Figure 2.17, while the handling for delete and update i-diffs are similar. Then according to the rules of ApplyPlan operator in Table E.5, Δ^+_{α} takes $\Delta^+_{\sigma_1}$ as input and runs the same P as the nested plan to produce output insert i-diffs for the view.

2.14 Generation of SQL++ base table i-diffs

To generate base table i-diffs, *idIVM* needs to generate the i-diff signatures at view definition time, and generate i-diff instances that correspond to the i-diff signatures at view maintenance time. This section describes these two tasks in detail.

2.14.1 Generation of base table i-diff signatures

Given a view definition V, *idIVM* generates suitable base table i-diff signatures as follows. This section assumes that top-level base tables are bags of tuples (e.g., **Users**, **Reviews** tables from the running example). The i-diff signature generator analyzes the view query to infer attribute names (including nested ones) in each base table and its nested collections. This is explained by the pseudocode below. In the pseudocode, the i-diff signature generator recursively traverses the view query's logical plan from top down. At each operator it tracks the set of attributes (including nested attributes) from output to input. Among the attributes, some are inferred as nested collections from the query, such as being the target of a ScanCollection operator. Such attributes are referred to as the set of collection attributes. Some attributes are involved in conditions of some operators in the view's plan, such as Select and Join. For an operator op, attributes involved in its condition are referred to as the set of conditional attributes C_{op}. The set of attributes not included in any condition for any operator in the view's plan is referred to as the set of non-conditional attributes. Such non-conditional attributes can appear in operators including Project and NavTuple.

```
function markBaseTables(op, col_attrs, attrs,
cond_attrs):
  if op is a ScanCollection operator on a base
  table b:
    mark b as a base table referenced by the view;
    mark col_attrs as (nested) collections of b;
    mark cond_attrs as conditional attributes of b;
    let nc_attrs = attrs minus cond_attrs;
    mark nc_attrs as the non-conditional attributes of b;
  else
    let s_col_attrs = attributes from child operator(s) that correspond to any
      of col_attrs or referenced in op as a collection;
    let s_cond_attrs = attributes from child operator(s) that correspond to any
      of cond_attrs or involved in any conditions in op;
    let s_attrs  = attributes from child operator(s) that correspond to any
      of attrs or referenced in op;
    for each child operator c:
      let c_col_attrs = part of s_col_attrs that come from c;
      let c_attrs = part of s_attrs that come from c;
      let c_cond_attrs = part of s_cond_attrs that come from c;
      markBaseTables(c, c_attrs, c_cond_attrs);
    endfor
  endif
end

function generateBaseTableDiffSigs(view_query)
  let op = root operator of view_query;
  markBaseTables(op, emptySet, emptySet);
```

```
  for each marked base table b:
    for each marked collection c of b, including b itself:
      create an insert diff signature for c;
      create a delete diff signature for c;
      let cond_attrs be conditional attributes of b whose closest collection
        ancestor is c;
      let nc_attrs be non-conditional attributes of b whose closest collection
        ancestor is c;
      create an update diff signature for c including pre and post-states
        of cond_attrs;
      create an update diff signature for c including post-state of nc_attrs;
    endfor
  endfor
end
```

Notice that the top-down propagation of attributes between output and input of an operator may involve transformation. Possible transformation includes renaming (e.g. Project) and consolidation of multiple attributes in the output into one in the input due to construction/copying of values (e.g. NavTuple, ConstructTuple).

For each base table and each of its nested collections R inferred from the view, the i-diff signature generator creates an i-diff signature of insert type on the elements of R containing the post-state of the element, and a single i-diff signature of delete type on R's elements containing the pre-state of all attributes. To generate update i-diff signatures, for each base table and each of its nested collections R, the i-diff signature generator creates (a) an update i-diff signature containing non-conditional attributes of R and (b) for each set C_{op} an update i-diff containing the pre and post-state of conditional attributes of R.

Furthermore, as an optimization not shown in the pseudocode, the i-diff signature generator records a set of base-table collections, including top-level and nested ones that have a

wildcard projection in the view. This information can be used later to determine whether some base table i-diff instances shall trigger IVM or not, which is discussed in Section 2.14.2.

Example 2.14.1. Consider **V** in the running example. The above algorithm will take the view query of **V** as input, and generates one insert, one delete, and one update i-diff signature for each of the three base tables **businesses**, **reviews**, and **users**. The algorithm will also find the **city** and **business_id** attributes of **businesses** base table as conditional attributes, and that **users** and **reviews** table are accessed by a wildcard projection in the view. Also notice that there is no i-diff signatures generated for the nested **elite** collection in **users** since the existence of it cannot be inferred from the view definition. Applying the algorithm to the equivalent view **V1** will lead to the same results as above.

2.14.2 From modification logs to base table i-diff instances

At run time, a modification log records changes happening to base tables. An implementation of modification log should provide the following functionality: (a) listing values that have been inserted, deleted or updated between the pre and post-state of the base tables, and (b) optionally capturing the post-state of the affected value. Notice that (b) is optional since the post-state of a value can always be retrieved from the base table.

When IVM is invoked for a view, changes from modification logs are converted to base table i-diff instances. Note that since different views can be maintained at different interval and therefore have different base table i-diff instances, the algorithm in this section applies this conversion for each view independently. The following pseudocode shows how this conversion is done for a base table *b* and its modification log *m*:

```
function conversion(b, m):
  for each modification d recorded by m:
    let p = path of target value of d;
    if p is not referenced in any diff signature of b:
```

```
      if p is not projected in the view, either explicitly or through a wildcard:
        skip d as it does not affect the view;
      else:
        let x = the closest ancestor of p that has an update diff signature;
        let ds = non-conditional update diff signature of b;
        let di = diff instance of ds;
        put p into di;
      endif
    else if p is a conditional-attribute of b:
      let ds = conditional update diff signature of b that contains p;
      let di = diff instance of ds;
      put p into di;
    else:
      let ds = diff signature of b that corresponds to modification d;
      let di = diff instance of ds;
      put p into di;
    endif
  endfor
end
```

As shown in above pseudocode, at view maintenance time, there can be i-diff instances to part of a base table that is not known from the view definition. These include i-diffs to attributes and nested collections that are not mentioned by the view query. *idIVM* does not create new i-diff signatures for such i-diff instances. Instead, such i-diff instances are described at view maintenance time by i-diff signatures of the closest ancestor value that can be inferred from the view at view definition time. Also, these i-diffs are treated as i-diffs to non-conditional attributes since their target values are not mentioned by the view. Furthermore as an optimization, certain i-diff instances may not be relevant to the view and can be safely discarded. Such i-diff instances

could be targeting a base table value on a path across a collection attribute that is not projected in the view either explicitly or by a wildcard. This test can be done using the information collected by the i-diff signature generator as described in 2.14.1.

Example 2.14.2. Following the running example, consider view **V** and a modification to **businesses** base table that changes a business's **city** from "San Francisco" to "Palo Alto". The above algorithm will find that **city** is a conditional attribute for **V**, and convert this modification to the following update i-diff instance for **businesses** base table:

```
<businesses: {
  type: update,
  diff: {{
    #(bid=tnhf) {
      type: update,
      diff: {
        city: {
          type: update,
          pre: 'San Francisco',
          post: 'Palo Alto'
        }
      }
    }
  }}
}>
```

Consider another modification that inserts an elite year to a user. The above algorithm will find that the **elite** attribute is not referenced by any diff signature of the **users** base table for **V**. However, **users** base table is previously found to be projected with a wildcard in **V**, and therefore **elite** attribute will end up in **V**. As a result, the algorithm will convert this modification to the following i-diff instance, which belongs to the update i-diff signature of **users** table:

```
<users: {
  type: update,
  diff: {{
    #(uid=Ha3i) {
      type: update,
      diff: {
        elite: {
          type: update,
          diff: {{
            #(eid=9kjb) { type: insert, post: 2014 }
          }}
        }
      }
    }
  }}
}>
```

2.15 Application of SQL++ i-diffs to the view

In *idIVM*, Δ-script needs to be executed as efficiently as possible. To this end, we describe in this section what are the indexes that are beneficial for applying i-diffs to the view, which is in Section 2.15.1. Then Section 2.15.2 describes how to select indexes to build for this purpose, and Section 2.15.3 explains how to apply i-diffs using these indexes. Finally, Section 2.15.4 proves an upper bound on the number of selected indexes.

2.15.1 Global provenance index

In a SQL++ document, a value x can be identified by the provenance along the path from the root of the document to x, which includes x's provenance and all its ancestors' provenance.

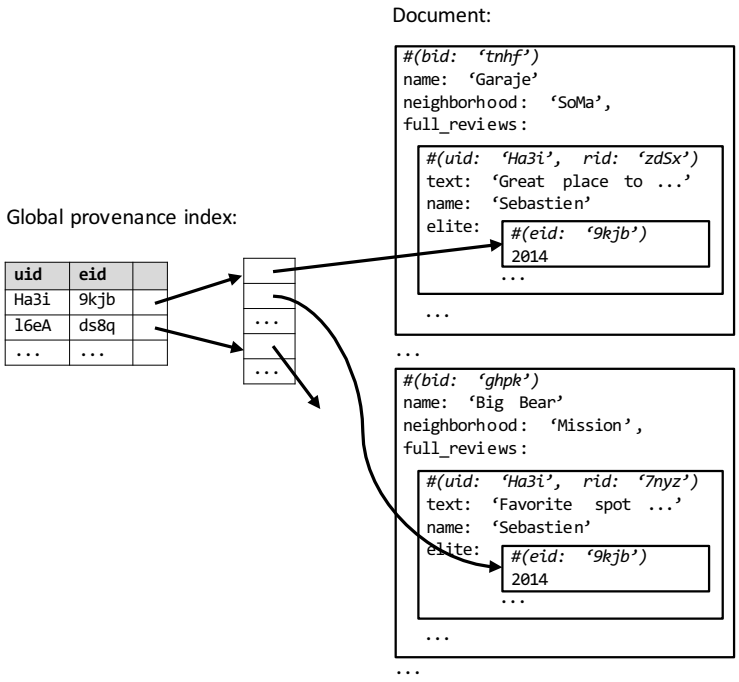


Figure 2.18. Global provenance index example

The *global provenance* of x is the concatenation of all provenance attributes of x and its ancestors. A *global provenance index* is an index of values on their global provenance attributes or a subset of them. A SQL++ value matches an index key if at each step its provenance attributes matches the key's provenance attributes that are present. Any provenance attributes not present in the index are considered wildcard and match unconditionally. Therefore, it is possible for a key in a global provenance index to point to multiple values in the document.

Example 2.15.1. Figure 2.18 shows the view instance of **V** (and equivalently **V1**) in our running example and a global provenance index built on *uid* and *eid* that point to nested elite values. The index can be used when applying the view i-diff Δ^V as described in Section 2.9 to the view, by looking up targeted elite values using the provenance attributes *uid* and *eid* that are present in Δ^V. Recall that *bid* and *rid* are not specified in Δ^V so that a concise i-diff about a single user and elite value can be applied to possibly many businesses and nested reviews in the view. Therefore, even though *bid* and *rid* are part of the global provenance of elite values, they are not included in

the global provenance index, which makes this index a most suitable data structure for applying i-diffs.

Recall that a local provenance value in general may not be globally unique, which can happen for derived data or due to other nature of data source. Therefore, global provenance indexes are designed to consider the path of provenance from document root as a value's global identity and index on it. Should the underlying query engine be extended to capture constraints that some local provenance can be globally unique, it is possible to further simplify and improve certain global provenance indexes, which we leave to future work.

Example 2.15.2. Notice that the running example makes no assumption that *eid* is globally unique (e.g., two elite values under different users may have identical *eid*). Therefore, the global provenance index in Figure 2.18 is built on provenance attributes *uid* and *eid*, instead of *eid* alone. This is necessary to make sure the index can work correctly for applying Δ^V to the view.

Global provenance indexes provide an efficient data structure for applying view i-diffs produced by *idIVM* to the view instance. The rest of this section describes how indexes are selected and used, whereas Section 2.16 discussed the IVM cost model based on the use of global provenance indexes.

2.15.2 Index selection algorithm

Given a view definition and base table i-diff signatures, the index selection algorithm determines which global provenance indexes to build. Its pseudocode is as follows.

```
function indexSelection(view):
  run IVM algorithm with base table diff signatures to get i-diff signatures
    to the view;
  for each i-diff signature d to the view:
    let c be the target collection of d;
    if there does NOT exist a global provenance index of the elements of c on
```

```
    d's provenance attributes:
      Build a global provenance index of the elements of c on d's provenance
      attributes;
    endif
  endfor
end
```

The index selection algorithm is designed to build indexes for elements of each collection in the view. For i-diffs to values that are not elements of a collection, navigation is needed to apply them to the view. An alternative approach which would save the navigation is to build indexes for all kinds of values in the view, but would lead to many more indexes being created and maintained.

At view maintenance time, there could be view i-diff instances to part of the view not known at view definition time. *idIVM* does not build indexes for them, but apply such i-diff instances without using indexes. The alternatives are: (a) build indexes after the view is populated by inspecting the view data; (b) build indexes on the fly at view maintenance time by inspecting the i-diff instances. Both alternatives can lead to as many indexes to be created as the combination of base table i-diffs. Therefore, we consider building indexes for i-diffs deeper than view definition as a future optimization.

Example 2.15.3. Consider **V** in the running example. Based on the inferred i-diff signatures per Section 2.14.1, the above index selection algorithm will build five global provenance indexes as follows: a global provenance index on the root collection of **V** with **bid** as provenance attribute, four global provenance indexes on the nested collection **full_reviews** with **uid**, **rid**, (**uid**, **rid**) and (**bid**, **uid**, **rid**) respectively as provenance attributes. Notice that no index is built for the nested **elite** collection which is deeper than the view definition.

2.15.3 i-Diff application using index

The semantics of i-diff instances has been defined in Section 2.12 as DML statements that apply i-diffs to the view. Here we describe how to apply i-diffs to the view using the global provenance indexes described earlier in this section.

Given an i-diff instance, *idIVM* traverses the tree and works on i-diff nodes that correspond to collection elements. For each such node d, *idIVM* tracks its global provenance during traversal, and matches it against the corresponding global provenance index. For each matched element value t in the view, *idIVM* then applies nested i-diffs under d that are not crossing a nested collection to t. The applied nested i-diffs include insert and delete i-diff instances, as well as update i-diff instances to primitive values. Notice that only leaf i-diffs are applied, whereas intermediate i-diffs are not.

During the traversal, *idIVM* may also come across i-diffs that are deeper than the view definition. Such i-diff nodes in an i-diff instance are not captured by any i-diff signature and therefore do not have corresponding indexes built. *idIVM* will start from the closest node to the i-diff that can be accessed through an index, and then use path navigation to apply the rest of the i-diff instance.

2.15.4 Upper bound on number of indexes

The number of indexes required for a view is upper-bounded by the number of provenance components from all view i-diffs, which is in turn upper-bounded by the number of provenance components from all i-diffs in the entire plan (including view i-diffs, intermediate result i-diffs, and base table i-diffs).

Notice that during i-diff propagation, if an input i-diff and an output i-diff have the same provenance component, then we need not count the output i-diff's provenance component as new. Therefore the problem becomes finding and counting all new provenance components that can be created during i-diff propagation. This number plus the number of provenance components

from base table i-diffs can serve as an upper bound on the number of view indexes.

From the rule tables, it turns out that for the majority of the operators, output i-diffs have the same provenance components as input i-diffs. The exceptions are (a) the select, join, inner-correlate operators, which may create output i-diffs that use the full provenance of the original output as the provenance component, and (b) the group-by operator, which may create output i-diffs that use the grouping attributes as the provenance component.

Therefore an upper bound to the number of indexes required for a view is:

```
number of provenance components from
base table i-diffs
  + number of selects
  + number of joins
  + number of inner-correlate
  + number of group-bys
```

Note that when all base tables are relational, "number of provenance components from base table i-diffs" can be replaced by the number of base tables, which leads to a static upper bound at view definition time.

2.16 SQL++ IVM Cost model

In this section we analyze the cost model of *idIVM* extended for SQL++ with several common queries. We focus on update i-diffs and uses the following assumptions: (a) data of a tuple (excluding nested collections) shall fit into one page, (b) if the data of a tuple including its nested collections is small enough to fit in one page, they are stored physically together inside a page, and (c) view tuples affected by the same i-diff are not clustered together, and caching has minimal effects on the overall cost, same as the assumptions made in Section 2.7. Note that in AsterixDB [3], data of a tuple including nested collections should always fit into one page. Therefore the assumptions (a) and (b) here cover the AsterixDB setting and are more

general. Later in this section, we will comment on how the cost model can be simplified under the AsterixDB setting.

2.16.1 Application of nested i-diffs to the view

We first consider the cost of applying nested i-diffs to the view where the i-diff signature is known at view definition time. In this case, *idIVM* builds global provenance indexes for the view i-diff signature as described in Section 2.15.1. Here we do not assume any particular view queries, which will be considered in later sections.

Let $R_1, \ldots R_n$ be the collections that have i-diffs. Notice that each R_i can have multiple instances in the i-diff or in the view. Let X_i be the total number of i-diff tuples for R_i. Let p_i be the compression factor for i-diff tuples of R_i. Notice that i-diffs of different collections can have different compression factor because they can have different set of wildcard in the provenance. Let q be the page size. Let m_i be the maximum size of the parent collection tuple of R_i, including nested data. When q is larger than m_i, applying i-diffs to an R_i tuple can be done during applying i-diffs to an ancestor of R_i (because of assumption (b)) and hence cause no extra page access.

Each i-diff tuple of R_i corresponds to an average of p_i tuples in the view. Looking up the p_i tuples in the index requires one page access to the index, followed by p_i accesses for each view tuple. Therefore, the tuple access cost is:

$$H = \sum_{i=1}^{n} (1 + p_i) c_i X_i$$

where

$$c_i = \begin{cases} 0 & \text{if } m_i \leq q \\ 1 & \text{otherwise} \end{cases}$$

2.16.2 Application of i-diffs deeper than view definition

Here we consider the cost of applying i-diffs that are deeper than the view definition and therefore not known at view definition time. Following the notation in Section 2.16.1, let R_k be one such collection that has nested i-diffs that are deeper than the view. Then for each of the X_k i-diff tuples of R_k, there are p_k matching tuples in the view that the i-diff should be applied to. Since the i-diffs deeper than the view are from native i-diffs of base tables, for each of the p_k matching tuples in the view, a nested i-diff value will match exactly one value under the view tuple. That is, the provenance of the i-diff that is deeper than the view will be full provenance without any wildcard. Assuming that a single R i-diff tuple including nested i-diffs can fit in memory, there are several ways to apply such i-diffs to the view, leading to different cost models.

One way to apply i-diffs is for each matching R_k tuples, to sequentially scan its nested data and apply i-diffs accordingly. Let the size of an R_k tuple be s_k (in terms of number of page accesses), then the cost becomes:

$$H = p_k X_k s_k$$

The above method serves as a bottom-line approach since it is not always necessary to scan the entire nested data of each matching R_k tuple. An improved way is to traverse the nested data of R_k using the provenance information in the i-diffs and thus scan only the nested data that is relevant to the i-diffs. For each R_k tuple, let $s_k^1, \ldots, s_k^{n_k}$ be the size of the nested collections that have to be scanned to apply the i-diffs, where each such collections in the view would correspond to a collection in the i-diff instance. Then the cost becomes:

$$H = p_k X_k \sum_j^{n_k} s_k^j$$

A different approach is to build indexes at view maintenance time when i-diff instances are observed, and then apply the i-diffs using the same way as in Section 2.16.1. However, the cost of this approach depends on the number of different indexes to be built, which is in turn

decided by the actual i-diff instances.

2.16.3 SPJ views

For an SPJ view, let D be the number of updated base table tuples (top-level) in the i-diff, p be the compression factor. Tuple access cost for maintaining the SPJ view is:

$$(1+p)D+H$$

Here $(1+p)D$ is the cost of top-level tuple lookup & access, and H stands for nested-level tuple lookup & access should the base table i-diffs have any nested i-diffs, as defined in Section 2.16.1 and 2.16.2.

Note that under the AsterixDB setting that nested data always fits in one page, the entire second term can be dropped and the cost becomes:

$$(1+p)D$$

This simplification also applies to the following types of view. Also notice that the above cost model is the same as the cost model for *idIVM* to maintain a relational SPJ view discussed in Section 2.7.

Example 2.16.1. Consider the SPJ part of the query in the running example **V1** as a new view **V2**:

```
CREATE VIEW V2 AS
SELECT business_id, reviews.*, users.*
FROM businesses NATURAL LEFT OUTER JOIN
  reviews NATURAL JOIN
  users
WHERE city = "San Francisco"
```

Consider update i-diffs instances to the **users** base table that update the **fans** attribute as well as inserting new **elite** years. The cost of IVM on **V2** depends on the characteristics of the data and has two main cases:

1. If the data of a **users** tuple including its nested collections can fit into one page (which is also the AsterixDB setting):

$$(1+p)D$$

   Notice that here applying nested diffs can be done when accessing the corresponding top-level tuple for applying **fans** diff, and hence does not incur additional cost.

2. If the **elite** data of each user tuple is stored in a separate page:

$$(1+p)D+pD=(1+2p)D$$

   Compared to the first case, this case requires $2pD$ additional page accesses because for each of the pD top-level tuples in the view that has update diffs, a separate page that stores its **elite** data has to be accessed to apply nested diffs.

3. If the **elite** data of each user tuple is stored in m separate pages:

$$(1+p)D+mpD=(1+p+mp)D$$

   This case serves as a generalization of the second case, though in practice it is unlikely that **elite** data will be so big to span multiple pages. Here for each of the pD top-level tuples in the view that has update i-diffs, since the **elite** diffs are deeper than the view, *idIVM* will scan the pages of **elite** data to apply its diffs due to lack of indexes, incurring mpD extra page accesses compared to case 1.

2.16.4 Group-by views

In SQL++ setting, GroupBy and Aggregate are two separate operators, each of which can exist on its own. For a view that has a GroupBy on top of SPJ (but without any aggregation), let D be the size of base table i-diffs (i.e., number of updated tuples), G be the number of groups created by the group-by, p be compression factor (due to join). Tuple access cost (with global provenance index) for maintaining this view is

$$(1+p)D+H$$

Notice that the above cost is identical to the cost for maintaining an SPJ view in Section 2.16.3. This is because the SQL++ GroupBy rearranges input tuples into groups (i.e., nested collections) whereas a global provenance index can directly locate the tuples after grouping without any extra cost. In contrast, the tuple access cost without using global provenance index would be:

$$(G+p)D+H$$

since for each i-diff tuple, all G groups need to be accessed to find possible matching tuples.

2.16.5 Aggregate views

Here we consider aggregate view queries that do not come with a GroupBy. Since there is no GroupBy, we assume the nested data that is being aggregated is from base tables. Also, we assume the aggregate function is incrementally maintainable (e.g., distributive functions such as sum and count). Let D be the number of updated nested tuples. Let g be grouping compression factor. Then gD is the number of affected groups, assuming $gD \ll G$. Tuple access cost for maintaining such views becomes:

$$2gD$$

Note that the multiplier 2 consists of a view index lookup and a view tuple access. Also, in the presence of aggregate, we do not consider the cost of maintaining and applying any nested i-diffs. Should there be any nested i-diffs that are propagated to the view, their cost can be modeled separately per Section 2.16.1 and 2.16.2.

2.16.6 GroupBy + Aggregate views

To maintain a GroupBy + Aggregate view, we first have to choose what to cache. Possible options are:

- To cache output of GroupBy. This is architecturally clean since GroupBy and Aggregate/FunctionCall are separate. For performance reasons, this would require global provenance indexes.
- To cache output of SPJ. In this way cost analysis is similar to relational counterpart.

For architecture cleanness, we proceed with caching the output of GroupBy. Let D be size of base table i-diffs (i.e., number of updated tuples), p be compression factor (because of join). Then pD is the number of updated nested tuples after group-by but before aggregate. Let g be grouping compression factor. Then gpD is the number of affected groups, assuming $gpD \ll G$. Tuple access cost for maintaining the view is:

$$(1+p)D + 2gpD$$

The first part $(1+p)D$ is for maintaining the cache, which is effectively the result of SPJ and Group-By. The second part $2gpD$ is for maintaining the final view, which is similar to the cost of maintaining an Aggregate-only view.

2.16.7 Comparison to relational IVM cost model

From Section 2.16.3 and Section 2.16.6, it can be seen that in *idIVM*, the cost for maintaining SPJ and GroupBy + Aggregate views in SQL++ is in line with the cost of maintaining

their relational counterparts. In particular, for a SPJ view in SQL++, when there is no nested i-diff, or when the nested data fits in one page, the view maintenance cost is identical to the ID-based IVM cost for a relational SPJ view in Table 2.3. The same also applies to maintaining Group-By + Aggregate views in SQL++ (Section 2.16.6) and in relational (Table 2.4). Furthermore, Section 2.16.4 shows that maintaining a GroupBy in isolation view, which only exists in SQL++, leads to no more cost than maintaining an SPJ view thanks to global provenance index. As a result, the performance analysis of relational ID-based approach versus tuple-based approach in Section 2.7 could be carried over to the SQL++ extension.

2.17 Conclusions

We have shown how to exploit IDs and provenance towards an IVM algorithm that is more efficient than existing tuple-based approaches under common assumptions. We also have extended this approach to SQL++ data model and query language using a theoretical model of indexing and data structures. Finding the appropriate data model is a working progress by the industry, and corresponding application will be evaluated in an appropriate context.

Chapter 2 contains material from "Utilizing IDs to Accelerate Incremental View Maintenance" by Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao, ACM SIGMOD Conference, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Related Work

In this chapter we discuss existing tools and libraries for simplifying Ajax programming. We also discuss related database research work about web application and incremental view maintenance. Their contributions to simplifying Ajax programming are discussed.

3.1 Existing Ajax Frameworks

In this section we examine existing tools and frameworks that are related to Ajax programming. Section 3.1.1 describes JavaScript libraries. Section 3.1.2 discusses Ajax frameworks that allow web application to be written using object oriented languages without JavaScript code.

3.1.1 JavaScript Libraries

A JavaScript library is a pre-written collection of JavaScript subroutines and classes that componentize common JavaScript implementation for client-side web applications in a modular fashion. The main purpose of JavaScript libraries is to allow web developers to reuse the implementation provided by those libraries, so that they can write less code and concentrate more upon more distinctive applications of Ajax. This goal has become more important since Ajax techniques take JavaScript as a major option for client-side programming language. The common implementation and components provided by JavaScript libraries include UI widgets (from labels, text fields to maps and calendars), asynchronous calling using XMLHttpRequest, events handling, RSS parsing and DOM traversing. jQuery [37], Prototype [45], YUI [60], and

Dojo [52] are some of the currently most popular JavaScript libraries.

Another important advantage of using JavaScript libraries is the cross-browser ability provided by most of the libraries. Due to various interfaces and implementations of different browsers (as well as different versions of browsers), compatibility issues arise in some essential parts of client-side programming, including style sheets, XMLHttpRequest calling, and DOM manipulation. It is therefore tedious and error-prone for web developers to accommodate different possible user browsers. Many JavaScript libraries provide web developers with a single interface that contains different implementations to incorporate with various browsers, hence hiding the complexity from the developers.

JavaScript libraries made an important contribution to Ajax programming by providing reusable JavaScript implementations and componentizing them into modules. As a result, the web developers can write less JavaScript code for the client-side programming. However, JavaScript programming cannot be completely avoided by just using JavaScript libraries.

3.1.2 Ajax Frameworks

This category is represented by two frameworks: Google Web Toolkit (GWT) and Echo2, which are described next.

Google Web Toolkit (GWT)

GWT [25] is a set of tools that allows developers to build and maintain client-side JavaScript applications using only Java language. The key idea is to use the GWT Java-to-JavaScript compiler to translate Java code to JavaScript code, so that JavaScript programming is completely avoided. In order to achieve the translation, GWT implements in JavaScript some common classes in the Java standard class library, including most of the java.lang package classes and a subset of the java.util package classes. Also, GWT provides a library for web page widgets and the interfaces for creating customized ones. Furthermore, GWT offers a hosted-mode web browser which runs and execute GWT applications directly as Java programs in the JVM. This

allows the developer to debug web application in Java where more powerful and complete debugging tools are available.

GWT also tries to tackle the distributed programming challenge of Ajax by supporting a simple RPC mechanism for client-sever communication. Web developers can use the provided interface to transparently make calls between client and server side Java code, and let GWT take care of low-level details like object serialization.

Echo2 and similar frameworks

Echo2 [20] is an Ajax framework that provides to web developers the ease of programming in a single language (typically Java) and exclusively at the server. It allows developers to create web application like creating desktop application, in a model similar to Java Swing [34]. This is achieved by Echo2 automatically maintaining a complete and synchronized mirror of the client side page state on the server side, so that user actions are transferred to the server and translated to events which can be followed up by programs. Similarly, the server's modifications to the mirrored copy of the page are translated back to the client-side browser page by the Echo2 framework. In this way, web developers do not need to be aware of the client and server distinction, but can instead assume that users are interacting directly with the server-side mirror. Therefore Echo2 is able to solve the first challenge of Ajax, namely the requirement of one more language on client side and distributed programming. Similar frameworks like Echo2 include ASP.NET [55], ZK [61], Backbase [5] and ICEfaces [32].

Echo2's approach to maintaining a complete and synchronized mirror of the client page state on the server also creates problems. It is assumed that in this approach most of the computation is done on the server side, while the web developers are not given the power to customize the client-side computation. As a result, the power of client-side computation is limited and becomes captive to the framework's power. Moreover, maintaining the perfectly synchronized mirror requires frequent communication between the client-side web browser and the server-side framework. Sometimes when the mirror is not well modeled, the communication

cost can become significant. For example, styling properties and implementation details are modeled as part of the page state in Microsoft's ASP.NET, a pure server-side framework that provides mirroring of page state by always sending the page state from the browser to the server in a hidden form field. This would lead to a high memory footprint and slow mirroring.

Finally, the language for specifying the page initialization and update using frameworks in this category is still imperative rather than declarative. Therefore, Echo2 and similar frameworks cannot address the challenge of Ajax that requires web developers to write codes to update different parts of the page depending on different user actions and possibly actions from multiple users.

3.2 Related Database Research

This section discusses two lines of database research related to the FORWARD framework. Section 3.2.1 surveyed database-driven declarative frameworks for web applications. Section 3.2.2 gives a summary of incremental view maintenance that is leveraged by incremental page update of the FORWARD framework.

3.2.1 Declarative Web Application Specifications

The data management research community has influenced the FORWARD framework by database-driven frameworks for web site [21] and application [17, 14, 58] development. Three related projects are presented next in detail.

Specification of Data-Driven Web Services

Web services, viewed broadly as interactive web applications providing access to information as well as transactions, are typically powered by databases. They are governed by protocols of interaction with users or programs, ranging from the low level input-output signatures used in WSDL [54], to high-level workflow specifications like BPML[12]. [53, 17] models the structure of the web pages using relational schemas, and the content of a web page as the dynamic result

of querying the underlying database and the application state. This subsection in particular describes how the WAVE project [17] models web applications.

In the scenario considered in WAVE, a web service is provided by an interactive web site that posts data, takes input from the user, and responds to the input by posting more data and/or taking some actions. The web site can access an underlying database, as well as state information updated as the interaction progresses. The structure of the web page the user sees at any given point is described by a web page schema. The contents of a web page is determined dynamically by querying the underlying database as well as the state. The actions taken by the web site, and transitions from one web page to another, are determined by the input, state, and database. The main purpose of declaratively specifying web services and pages in a relational model in WAVE is to facilitate static analysis and verifications. For example, in a web service supporting an e-commerce application, it may be desirable to verify that no product is delivered before payment of the right amount is received.

Figure 3.1 shows the specification of an example page using WAVE. In this page a list of proposals are shown to the reviewer user who can recommend any of them to committee chairs and then be navigated to Recommended Proposals (`RP`) page. The `proposal` relation is part of the database. All the remaining relations in the specification are used by the framework of WAVE to manage user input, input options, application states, external actions, and target pages. The modification to the above-mentioned relations are through rules in first-order logic which can be translated to SQL queries and modification commands.

The `Posted Data` block specifies the content of the page as a result of a query, which in this example is the `proposal` relation. The `Input Rules` block specifies what options are available to each actions. Here any proposal can be the option of the `recommend` action as shown. Notice that clicking `logout` button is also a valid action, but since there is no option associated with it there is no rule needed either. The `State Rules` block describes how the input can trigger transition to the application state. In particular, a recommended proposal will be put into the `recommend_state` relation. The `Action Rules` block specifies what external actions

```
%[Proposals Page]

Posted Data:
    proposal(pid,title,plan)

Input Rules:
    Options@recommend(pid, title, plan) := proposal(pid,title,plan)

Input:
    recommend(pid, title, plan);
    clickbutton("logout")

State Rules:
    recommend_state(pid,title,plan) := recommend(pid, title, plan)

Action (side effect) Rules:
    clean_session_action():= clickbutton("Return")

Target Webpages: RP, HP

Target Rules:
    RP := recommend(pid, title, plan);
    HP := clickbutton("Return")
```

Figure 3.1. Example of declarative page specification in WAVE

or side effects would be fired. The actions are modeled as relations so that a record put into an action relation indicates that the associated action should be fired externally. Finally, the `Target Webpages` and `Target Rules` blocks decides which are the possible next pages to navigate to and how they are decided using data from other relations.

WAVE gives a general approach to declaratively specifying web services and pages using relational model. As we discussed earlier, declarative specifications lead to rapid programming, much fewer bugs and easy application maintenance and evolution. Section 3.2.2 shows that using relational model opens the gate to years of database research on incremental view maintenance that can significantly improve the performance of re-rendering a page specified using this model.

Web Modeling Language (WebML)

WebML [53, 14] considers the specification of web sites in more perspectives, including the logical layer and presentational layer, in addition to the data layer. Its main idea is to enable designers to express the core features of a web site at a high level, without committing to detailed architectural details. The specification of a web site in WebML is divided into the following four layers:

1. Structural Model: This is the data layer which expresses the data content of a web site, in terms of the relevant entities and relationships. It uses an E/R-like language and is expected to be specified by the data experts.

2. Hypertext Model: This is the logical layer of pages that is supposed to be specified by the application architects. It consists of two parts:

    - Composition Model: It specify how a page is composed of *units*. WebML offers six types of content units: data, multi-data, index, filter, scroller and direct units. Data units are used to publish the information of a single Object (e.g., a proposal), whereas the remaining types of units represent alternative ways to browser a set of object

(e.g., the set of reviews of a proposal). The units are defined on top of the entities or relationship in the data layer.

- Navigation Model: It expresses how pages and units are linked to form the hypertext. Links can be non-contextual, when they connect semantically independent pages (e.g., the page of a proposal to the home page of the site), or contextual, when the content of the destination unit of the link depends on the content of the source unit. For example, the page showing a proposal's data is linked by a contextual link to the page showing the information of the proposal's applicant. Contextual links are based on the relationships in the data layer.

Figure 3.2 shows a high-level graphical representation of the logical layer of two web pages in WebML. The solid boxes are units which include data, index and direct units, and edges show the connections between pairs of units.

3. Presentation Model: It expresses the layout and graphic appearance of pages, independently of the output device and of the rendition language, by means of an abstract XML syntax. Presentation specifications are either page-specific or generic, and they are designed by style architects.

4. Personalization Model: This layer considers user and personalization options, as well as business logics, and is orthogonal to the other layers. User and user groups are explicitly modeled in the data layer in the form of predefined entities. The features of these entities can be used for storing group-specific or individual content, like suggestions, favorites, and resources for graphic customization. Then, derived content can be added based on the profile data stored in the user and group entities. This personalization content can be used both in the composition of units or the definition of presentation specifications. Moreover, high level business rules can be defined for reacting to site-related events, like user clicks and content updates. This layer is supposed to be managed by site administrators.

WebML does not take Ajax into account since it is created before the Web 2.0 era. However, the idea of logical units that bind to data and describe the composition of web pages inspires the design of logical layer of the FORWARD framework.

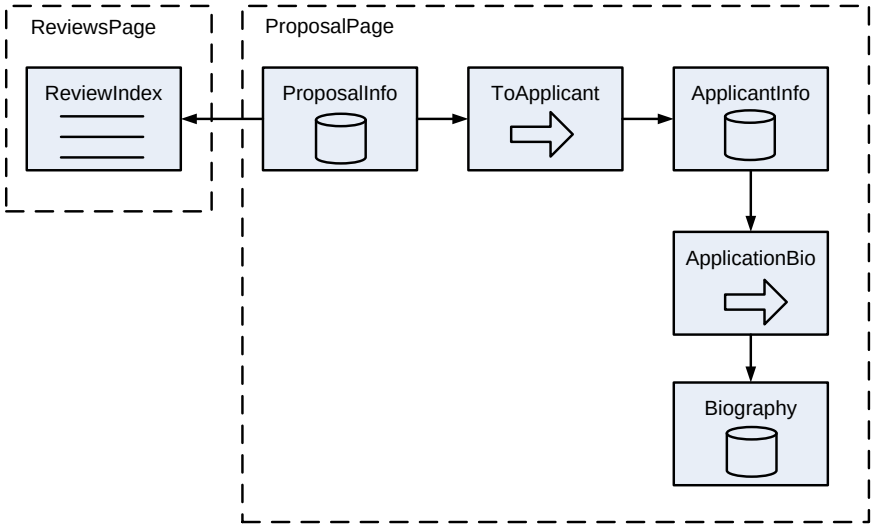


Figure 3.2. An example of unit logical layer in WebML

Hilda

The Hilda projected described in [58, 57] provides a domain-specific language for developing web applications, where all application logic is declaratively specified using SQL-like queries. All application state is captured exclusively in the relational model, while customizable code by users involving arbitrary computation are allowed.

The basic building block in Hilda is the application unit, or AUnit which is a essentially a single-entry single-exist programming construct. The AUnits specify both the logical layer of the page and the application logic. In many ways, an AUnit resembles a class in conventional object-oriented (OO) programming languages. There are primitive AUnits provided by the language framework, as well as user-defined AUnits. An Aunit can be instantiated; the life cycle of AUnit instances involve both instance creation and deletion. AUnits are arranged into a tree

structure so that an AUnit can have child AUnits. At run time, an AUnit instance can activate multiple child AUnit instances of the same type by using an activator (a rule) in terms of an SQL query, so that each tuple in the query result guides the creation of a child unit instance. Furthermore, Hilda also supports inheritance for extending the functionality of AUnits. The inheritance can be used to add new application logic and/or logical structure of the web site.

Unlike OO classes, however, AUnits store all state within relations. Each AUnit instance declares a local schema of relations, and data stored in the local schema is private to the instance. Since the database state is also stored as a schema of relations in the RDBMS, populating local schemas with data can occur using queries as an uniform mechanism, regardless whether the data comes from transient application state or persistent database state.

In keeping with the declarative model, Hilda does not have conventional OO methods. To effect state transitions, each AUnit instance has instead a set of handlers. Each handler has a condition and an action. The condition is a query that returns zero or more tuples; the action is an update to the local schemas of the AUnits. For each user action, the conditions of all handlers are evaluated, and an action is executed if its corresponding condition evaluates to true (i.e. at least one tuple). In Hilda, if more than one condition is true, only one action is arbitrarily executed. In many ways, this style of programming is reminiscent of trigger programming in databases.

In Hilda, an entire web application is specified by a single AUnit tree with a distinguished node being the root of the tree. For example, Figure 3.3 shows an example of simplified AUnit tree of our running example. This tree applies to all users of various roles. At run time, different user session would activate different instance of AUnit trees, which form the activation forest of the application. Hilda tries to automatically partition the computation between client and server by making a cut in an activation tree so that the side containing the root node instance belongs to the server and runs as Java servlets, while the other side belongs to the client and runs as Java applets. The cut is dynamic and can change at run time, and the Hilda framework offers an algorithm for deciding the cut.

Presentation is cleanly separated from application logic through presentation units

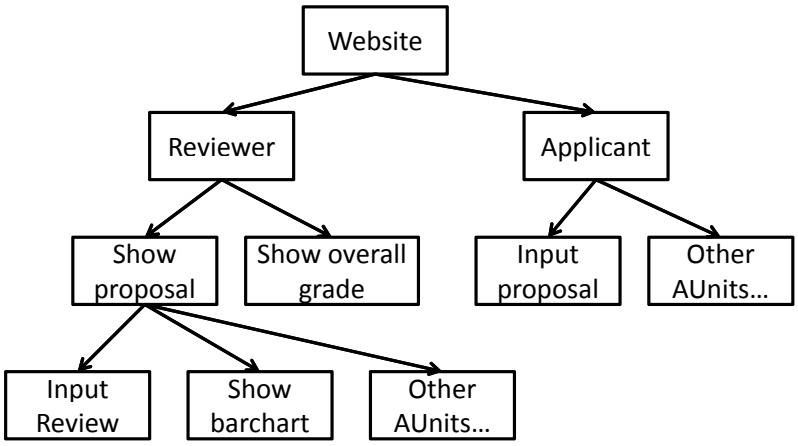


Figure 3.3. An example of an AUnit tree in Hilda

(PUnits). Each AUnit has a corresponding PUnit; each tree of AUnit instances therefore has a corresponding tree of PUnit instances. Each PUnit has embedded HTML code that renders the appropriate HTML fragment for its corresponding AUnit. Furthermore, each PUnit can recursively invoke its children PUnits to build up the entire HTML page.

The combined AUnit structure for both logical page structure and application logic does not have a clear concept model like MVC architecture, and creates a limit that is not friendly towards Ajax-style partial page update. In Hilda, if certain user action would affect part of a page, the AUnit node corresponding to the acted part needs to return and the AUnit node corresponding to the changed part needs to be reactivated. For example, in Figure 3.3, the `Input Review` node will cause the control flow return to its parent once the reviewer types in and submit a review. A return handler at the node `Show proposal` will process the data passed from the child node, and in this case reactivate all child units. In this way, the bar chart on the page can be properly refreshed to reflect the changed grade distribution. However, suppose there is an overall grade shown on the page that is represented by node `Show overall grade`. To refresh the overall grade, a web developer needs to specify rules on AUnit nodes all the way to the lowest common ancestor to guide the reactivation. At run time, the reactivation of the entire branch of

the activation tree also creates efficiency problem and possibly loss of state.

3.2.2 Incremental View Maintenance

IVM is a long studied problem with a lot of influential works [10, 9, 13, 46, 29]. *idIVM* falls under the category of IVM works that employ the *algebraic* [46, 26, 47, 39] approach. Due to the vast amount of related work in IVM, we focus next on approaches that are particularly related to the main aspects of our work, which are: (a) exploiting primary key information together with the associated (b) overestimation and (c) caching. Note that we cover all works in these areas, even if they do not follow the algebraic approach. For comprehensive surveys on IVM, the reader is referred to [28, 15].

Exploiting primary key constraints. The idea of exploiting primary key constraints to speed up IVM was first presented in [27, 48]. However, in contrast to our work, [27, 48] study only self-maintenance (potentially together with some auxiliary views) and not general view maintenance where some data from the base relations may be required to maintain the view. Furthermore, they are limited to maintenance of SPJ (including outer-join) views and their algorithms are not easily extensible to more general classes of queries as they operate by looking holistically at the view definition, in contrast to our modular algebraic approach. The first work that exploited primary keys in an extensible algebraic setting and introduced the notion of partial diffs, is [36]. However, the partial diffs of [36] always contain the entire primary key of the view. Thus, they are not true ID-based diffs, but instead (relaxed) tuple-based diffs, that may lack some of the (non-key) attributes of the view but will still incur the same number of accesses as tuple-based approaches. Finally, primary key information has also been used to optimize the rules for maintaining the output of particular operators (e.g., outer-join in [39]) within a tuple-based approach. However, these approaches do not look at exploiting the keys to avoid tuple-based diffs altogether, as done in this work.

Overestimation. Our definition of overestimation is similar to *safe overestimation* described in [7] and *ineffective updates* in [36]. While overestimation in these works appears

only because of selection conditions, *idIVM* exploits also overestimation that arises because of joins, which do not appear in the former, since they are both (relaxed) tuple-based approaches.

Caching. Several works looked at the problem of materializing additional results to speed up IVM. These can be classified into two broad categories. The first category includes approaches where the cached results are operator-specific. Examples of such works include the IVM of aggregation under the assumption that previous aggregation results are available [47, 43] and of top-k results by caching additional view tuples that are beyond the top k in order to reduce the frequency of accessing the base tables [59]. These caches correspond to our notion of operator caches and can thus be incorporated in our framework as part of an operator definition. The second category contains approaches where the cached results are not tied to a particular operator, but are additional views that are then exploited holistically during the IVM of the original view [49, 41, 48, 2]. In contrast to our work that uses only caches that correspond to subplans of the original plan, these works benefit by employing caches that may not be subplans of a single plan. This aggressive materialization allows more efficient IVM, though at the cost of maintaining an increased number of intermediate views. A prime example of such approaches is DBToaster [2], which is discussed extensively in Section 2.8.3. However, by not being ID-based such approaches always access at least one materialized view, in contrast to our approach, which in some cases can avoid accessing base tables or cached views altogether. Finally, a related area to caching in IVM is view selection, consisting of works that decide which views to materialize to speed up query evaluation [1, 30]. Such approaches can be used in the context of *idIVM* to decide which intermediate caches to materialize.

XML Views Maintaining XPath views has been studied in [50, 8]. [50] considers maintaining XPath views in an IVM system that is loosely coupled with the base data system where incremental maintenance is not always possible, and focuses on updates that are irrelevant to the view or self-maintainable. [8] proves the upper bounds on time and space for checking whether an XPath expression is satisfied or not after an update, where the update is inserting, deleting, or relabeling (renaming) a single node.

XQuery view maintenance has been studied in [18, 23, 11]. [18] studies the IVM of order-sensitive XQuery views using XAT algebra. It extends bag semantics with LexKeys (lexicographical order encoding) to support IVM with ordered data. Although LexKeys can serve as IDs, a LexKey always identifies a value (XML node) and there is no notion of partial provenance or IDs. Even though the paper takes an algebraic approach to handle each operator, it does not transform diffs using algebraic plan to form diff queries. Instead, the IVM rules are expressed in a calculus format and implemented in the query processor specific to in-memory storage.

In [23], views are expressed in the Galax tree algebra and a core set of Galax operators are considered for IVM. It takes an interesting approach of transforming update statements (also expressed in Galax algebra) from input to output of each operator in the set. Although the paper listed IVM rules for operators including concatenation, XPath navigation, conditional, ApplyPlan, it misses rules for some important operators including select and join/product. Without employing partial provenance/IDs, this approach is limited to the same performance characteristics as tuple-based IVM.

[11] focuses on a subset and dialect of XQuery including for, let, return, string comparison and XPath$^{\{/,//,*,[]\}}$, and translates them to a decorrelated tree pattern algreba for IVM. Rather than handling each individual operator for IVM and constructing a diff query, it takes a holistic approach that handles the tree pattern query and focuses on pruning optimizations. The algorithm does not employ partial provenance or IDs and has to compute tuple-based diffs. It considers insert and delete diffs and is limited to monotonic scenarios, where insert and delete diffs can only lead to diffs of the same type.

Appendix A

FORWARD Mapping Framework Specification

Given two schemas S and T, a *mapping configuration* M from S to T expresses some query logic that turns any instance of S to an instance of T. In execution, M is translated to an equivalent query that inputs S and outputs T.

A.1 Motivation

FORWARD mapping framework can express common translations of data from a source schema to a target schema in FORWARD's SQL++ extension. It is used extensively in FORWARD page layer, where data needs to be transformed from one page schema to another page schema of the same page. It represents the translation as mapping lines from source type nodes to target type nodes, and turns a valid mapping into a SQL++ query which takes input in the source schema and produces output in the target schema. The main motivation for building the mapping framework (instead of writing queries directly) is that the computation needed for FORWARD to automatically transform and integrate data is more limited than the full power of SQL++ queries. In particular, this includes translation of data between different schemas of a page (see Section 1.2.4). Therefore, the core mapping framework which is designed to produce just the right amount of computation power greatly simplifies implementation and maintenance of the framework. Currently the mapping framework is used internally in FORWARD and not

exposed to application developers.

A.2 Syntax

The syntax of a mapping configuration is shown in Figure A.1. The basic building block of core mapping language is a *conjunctive mapping* that consists of a group of atomic mappings, each of which can be a scalar mapping, a tuple mapping, an expression mapping, or a union mapping.

A *scalar mapping* (or *tuple mapping*) maps a source scalar type (or tuple type) to a target scalar type (or tuple type). For a scalar mapping or tuple mapping $s \rightarrow t$, if there exists an s' that is the closest tuple ancestor of s such that there is a tuple mapping $s' \rightarrow t'$, then s' is called the *closest mapped tuple ancestor (cmta)* of s. If there is no such s' then the cmta is the root of the source.

MappingConfiguration	→	*source-schema*, *target-schema*, **ConjunctiveMapping**
ConjunctiveMapping	→	**AtomicMapping+** [**SelectionCondition**]
SelectionCondition	→	`Expression`(*SourceNodes*+)
AtomicMapping	→	**ScalarMapping**
	|	**TupleMapping**
	|	**UnionMapping**, *UnionRootMark*
	|	**ExpressionMapping**
ScalarMapping	→	*SourceScalarNode* → *TargetScalarNode*
TupleMapping	→	*SourceTupleNode* → *TargetTupleNode*
UnionMapping	→	**ConjunctiveMapping**+[2]
ExpressionMapping	→	`Expression`(*SourceNodes*+) → *TargetNode*

Figure A.1. The syntax of mapping tree

We allow a conjunctive mapping to have multiple scalar mappings pointing to a single target scalar type, where the source types in those scalar mappings have to be mutually exclusive in the sense that at run time at most one of them can be evaluated to be non-null. To generated the corresponding SQL++ query, `CASE WHEN ...THEN ...` statement is applied.

Union of source tuples is supported by *union mapping*. A union mapping consisted of

two or more conjunctive mappings, all of which must contain a tuple mapping towards a target tuple type called the *union root*. During the construction of target data tree, each conjunctive mapping of a union mapping is first evaluated independently, and then their generated tuples are unioned together to become the target output.

Expression mapping allows a target scalar type to be mapped by an expression of SQL++ functions. The expression can be composed from SQL++ functions in a tree structure, while the leaves can be either source types or constant literals.

A.3 Validity Check of Mapping Configuration

A mapping configuration needs to be internally compiled and checked for validity before it can be translated to a query. We do the following validity checks of a mapping configuration.

- Within a single conjunctive mapping, a tuple in the target schema is involved in at most one tuple mapping or union mapping.
- Every table tuple type and switch case tuple type in the target schema is mapped.
- A scalar type in the target schema can be mapped by zero, one, or multiple scalar mappings or expression mapping.
- Descendant and ancestor relationship must be preserved through tuple mappings. In particular, we check that for a scalar mapping or tuple mapping $s \to t$, the cmta of s should be mapped to an ancestor of t.
- If a mapped target tuple type is not a table tuple, then either
  - Its corresponding source tuple type and its cmta has no table type in between, or
  - The conjunctive mapping has a selection condition that is under the target tuple type.

A.4 Query Generation

The query generation algorithm is a recursive descent driven into the target tree and resulting into a query tree that populates the target. Due to page space limit, the pseudo-code is separated into three parts and shown in Figure A.2, Figure A.3, and Figure A.4.

```
function generateQuery(S, T, conjunctive mapping C_M) return Query begin
  // S is the source schema
  // T is a subtree of the target schema
  // C_M is the current conjunctive mapping node, not necessarily the
     root
  // the following procedures help in query writing
  // the procedure path(s_1, s_2) writes the query path from the tuple
     node s_1 to the node s_2
  // the procedure closestVar(s) writes the variable or data object name
     associated with the cmta of the node s in S
  // the procedure merge(e_1, ..., e_n) writes the CASE expression that
     returns the only one of the e_i expressions that is not null.
     Whenever used it is expected that at most one will be not null.
     If they are all null then return null.
  t is the root of T ;
  if t is a tuple marked as UnionRootMark by a union mapping in C_M then
     /* Merge of non-table tuple -- use CASE WHEN                    */
     U ← the union mapping that corresponds to t in C_M;
     foreach conjunctive mapping M_i, i = 1, ..., n of U do
        q_i ← generateQuery(S, T, M_i) ;
     return merge(q_1, ..., q_n) ;
  else if t is a tuple NOT marked by UnionRootMark in C_M then
     /* t can be mapped by one or zero tuple mapping.  Both are valid.
        */
     foreach attribute a_j, j = 1...m of t whose schema tree is c_T^j do
        q_j ← generateQuery(S, c_T^j, C_M) ;
     return TUPLE(q_1 AS a_1, ..., q_m AS a_m) ;
  ...;
```

Figure A.2. Query Generation from Mappings Configuration - Part 1

```
function generateQuery(S, T, conjunctive mapping C_M) returns Query begin
    ...;
    else if t is a scalar targeted by more than one mappings in C_M then
        foreach mapping s_i → t, i = 1...n in C_M that targets t do
            e_i ← closestVar(s_i).path(cmta(s_i), s_i) ;
        return merge(e_1, ..., e_n) ;
    else if t is a scalar targeted by exactly one scalar mapping in C_M then
        return closestVar(s).path(cmta(s), s) ;
    else if t is a scalar targeted by exactly one expression mapping in C_M then
        /* Construct the expression                                    */
    else if t is a scalar targeted by no mapping at all then
        return default value (constant) of type t
    ...;
```

Figure A.3. Query Generation from Mappings Configuration - Part 2

A.5 Selection Condition

We add selection condition to mapping configuration because we need the feature of pinpointing a particular table tuple in the source schema to fill a single (non-table) tuple in the target schema. This use case arises during transforming data from a page's page.complete schema to one of its page.context schemas.

An optional selection condition can be added to a conjunctive mapping. With this selection condition added, we can then allow a tuple mapping to "skip" an arbitrary number of table tuples in the source schema, leaving them unmapped. The selection condition is used to navigate across those tables. Figure A.5 gives the updated pseudocode of the mapping-to-query translation to handle selection condition. It becomes the mapping creator's responsibility to ensure that at run time, exactly one tuple value under each of those tables will meet the selection condition. In FORWARD's page compiler, this is enforced by design during the building of the page.complete schema, the page.context schema, and their mappings.

function `generateQuery(`*S, T, conjunctive mapping* C_M`)` return Query **begin**

...;

else if *t is a table and its tuple* t_c *is NOT a UnionRootMark* **then**

Find the tuple mapping (s, t_c);

foreach *Not registered table tuple type* s_i *where* $(i = 1, \cdots, n)$ *from root to s* **do**

produce a fresh variable name v_i ;

associate the variable v_i with s_i ;

T' is the subtree of T rooted at t_c ;

foreach *attribute* $a_j, j = 1 \ldots m$ *of t whose schema tree is* c_T^j **do**

$q_j \leftarrow$ `generateQuery`(S, c_T^j, C_M) ;

return `SELECT`q_1 `AS` $a_1, \ldots, q_m$ `AS` a_m `FROM`
$closestVar(s_1).path(cmta(s_1), s_1)$ `AS` $v_1, \cdots,$
$closestVar(s_n).path(cmta(s_n), s_n)$ `AS` v_n ;

else if *t is a table and its tuple* t_c *is marked as UnionRootMark* **then**

$U \leftarrow$ the union mapping that corresponds to t_c ;

foreach *conjunctive mapping* $M_i, i = 1, \ldots, n$ *of* U **do**

$q_i \leftarrow$ `generateQuery`(S, T, M_i) ;

return q_1 `UNION ... UNION` q_n ;

else if *t is a switch type* **then**

foreach *case* $a_j, j = 1 \cdots m$ *of t whose schema tree is* c_T^j **do**

$q_j \leftarrow$ `generateQuery`(S, c_T^j, C_M) ;

$d_j \leftarrow$ the condition that leads to a_j;

`/* Notice that the above condition can be disjunctive if case tuple` a_j `is the root of a union mapping. */`

return `SWITCH WHEN` d_1 `THEN` $q_1 \cdots$ `WHEN` d_m `THEN` q_m;

Figure A.4. Query Generation from Mappings Configuration - Part 3

function `generateQuery(`*S, T, conjunctive mapping* C_M`)` return Query **begin**

...;

else if *t is a table and its tuple* t_c *is NOT a UnionRootMark* **then**

Find the tuple mapping (s, t_c);

unmappedSrcColTuple $\leftarrow$ Not registered table tuple type s_i where $(i = 1, \cdots, n)$ from root to s;

requireSelection $\leftarrow$ *unmappedSrcColTuple* is not empty;

foreach s_i *in unmappedSrcColTuple* **do**

produce a fresh variable name v_i ;

associate the variable v_i with s_i ;

T' is the subtree of T rooted at t_c ;

foreach *attribute* $a_j, j = 1 \ldots m$ *of t whose schema tree is* c_T^j **do**

$q_j \leftarrow$ `generateQuery`(S, c_T^j, C_M) ;

if *requireSelection* **then**

exists $\leftarrow$ translate C_M's selection condition to an EXISTS clause;

return `SELECT`q_1 `AS` $a_1, \ldots, q_m$ `AS` a_m `FROM`
$closestVar(s_1).path(cmta(s_1), s_1)$ `AS` $v_1, \cdots,$
$closestVar(s_n).path(cmta(s_n), s_n)$ `AS` v_n WHERE *exists*;

else

return `SELECT`q_1 `AS` $a_1, \ldots, q_m$ `AS` a_m `FROM`
$closestVar(s_1).path(cmta(s_1), s_1)$ `AS` $v_1, \cdots,$
$closestVar(s_n).path(cmta(s_n), s_n)$ `AS` v_n ;

Figure A.5. Updated Query Generation from Mappings Configuration with Selection Condition

A.6 Mapping Provenance ID and Mapping Inversion

Mapping inversion is a feature that propagates changes to scalar values of a taget data tree back to its corresponding source data tree. To make it possible, we need to establish connections between source and target values. This is done by adding *mapping provenance IDs* to the target schema and adding corresponding computation logic as mapping lines to the mapping configuration. *Mapping provenance inferrer* is in charge of this process. It can work with any base mapping configuration to add provenance IDs. The challenge here is that we need to deal with union mappings, which means that a target type may be mapped by several source types. Consequently, at run time, we need to figure out given a particular target value, which union branch the value corresponds to, and hence which source value populated it.

A.6.1 Mapping Provenance Inferrer

We first present the algorithm of mapping provenance inferrer. Figure A.6 gives the pseudo-code of it. The algorithm traverses the mapping configuration from top down, and for each union branch, it keeps track of the branch index number. When a tuple mapping is visited, the algorithm creates an ID attribute (named "id") in the target tuple type. This ID attribute is then filled by an expression mapping of the form "$id\$(keys[1])\$(keys[2])\$\ldots\$(keys[x])\$$". This value captures both the current union branch index and the global primary key of the corresponding source table tuple. Later during mapping inversion, the union branch index is used to guide mapping inversion process, while the source global primary key serves as a signature for values.

A.6.2 Mapping Inversion Algorithm

At run time, when a target data tree originated from a source data tree through a mapping configuration gets updated, the mapping inversion engine can propagate the changes from the target data tree back to the source data tree. Figure A.7 and Figure A.8 show the pseudo-code of this process. It first builds a map from signatures of source scalar values to the values themselves.

```
function inferProvenanceMappings(S, T, M, id) begin
    input  : source schema S, target schema T, conjunctive mapping M, and branch id bid
    output : this algorithm updates T and M in place

    foreach union mapping UM in M do
        foreach i-th conjunctive mapping CM in UM do
            inferProvenanceMappings(S, T, CM, i)
    foreach tuple mapping tm in M do
        target_tuple ← target type of tm;
        Add a new string type attribute named "id" to target_tuple;
        if target_tuple is a table tuple then
            Declare "id" as the primary key attribute of target_tuple's table;
        source_coll ← lowest ancestor of source type of tm;
        keys ← global primary key attributes of source_coll;
        Create an expression mapping to "id" attribute in target schema of the form
         "id$(keys[1])$(keys[2])$...$(keys[x])$";
```

Figure A.6. Inferring mapping provenance IDs

Then it traverses the target data tree, and at the same time keeps track of the current conjunctive mapping by using the branch index number stored in ID attributes added by mapping provenance inferrer. When the traversal comes to a target scalar value, the algorithm manages to construct the signature of the source value that mapped to this target value. Therefore we can locate the source value using the signature map and hence update the source value using the new target value.

Optimization with list of target changes

The baseline solution shown in Figure A.7 and Figure A.8 works without the knowledge of which part of the target data tree has been updated. This means that mapping inversion has to traverse the entire target data tree to see which nodes need to be propagated back to the source data tree, which leads to a running time linear to the size of the data tree. In practice, when mapping inversion is used to propagate user input on a page's visual data object back to its page.complete data object, the number of changes on visual data object (i.e., the target) is usually small. In our full algorithm and implementation, we utilize the list of target changes (provided by the visual layer) to guide mapping inversion so that only the part of the target tree which

```
function mappingInversion(s,t,MC) begin
    input  : old source data tree s, new target data tree t, mapping configuration MC
    output : this algorithm updates s in place

    hash_map ← a hash map from a string to a source value ;
    foreach Source value sv do
        if sv is a scalar value or a null value then
            st ← path expression of type of sv;
            keys ← primary key attribute values of sv;
            s ← build a string of the form "st$(keys[1])$(keys[2])$···$(keys[x])$";
            Add (s → sv) to hash_map;
    MC ← root conjunctive mapping of M;
    recursivelyInverse(s,t,MC, "")
```

Figure A.7. Mapping Inversion - Part 1

```
function recursivelyInverse(t,M,sig,hash_map) begin
    input  : target value t, conjunctive mapping M, primary key part of signature sig,
             signature to source value map hash_map
    output : this algorithm updates s in place

    if t is a tuple then
        id ← the "id" attribute value of t;
        (bid, src_key) ← break id into branch index part and source key part;
        if t is marked as union root then
            CM ← the bid-th conjunctive mapping of the union mapping rooted at t;
            foreach child attribute c of t do
                mappingInversion(c,CM,src_key,hash_map)
        foreach child attribute c of t do
            mappingInversion(c,M,src_key,hash_map)
    else if t is a scalar value or null value then
        if t is mapped by a scalar mapping sm then
            st ← path expression of the source type of sm;
            full_sig ← st + sig;
            sv ← the source value that full_sig points to in hash_map;
            Update sv's value to t's value;
     else
        /* t is a table type or switch type.                                  */
        Call this function recursively with the children types of t;
```

Figure A.8. Mapping Inversion - Part 2

involves changes is traversed. This improves runtime performance of mapping inversion, since the running time now becomes linear to the size of the data diff.

Appendix B

Key Inference and Retouching Specification

The *key inference module* infers keys for the output of a logical plan. If a key cannot be found for certain table, it tries to change ("retouch") the logical plan (e.g., to include more attributes in the output) so that keys can be found.

In our design, an inferred key is guaranteed to be unique, but not necessarily minimal (i.e., it may not be a candidate key).

B.1 Motivation

Keys as one form of unique IDs serve as a crucial piece of information for achieving Ajax effects of partial page update. In typical Ajax web development, developers have to mark parts of page with HTML id attributes, so that Ajax actions can return responses of page updates addressing each part using their IDs. It follows that developers have to come up with unique values of the ID attribute for each part of the page. This can become tricky with dynamically generated pages of nested structure. If a developer made a mistake and ID's become not unique, incorrect behaviors may come up during runtime and it is hard to debug.

To relieve developers from this burden and its associated risk, we designed and built the key inference module as part of the FORWARD framework to automate key creation (and hence HTML ID creation) in FORWARD pages.

Besides the above main motivation, key inference (or its variants) also serve the pattern-based incremental view maintenance module, which uses keys to describe functional dependency in patterns (see Chapter 2), and SQL++ DML statements, which use keys to identify and change values in originating data sources.

B.2 Main Workflow

We do per-operator analysis and retouching to figure out output keys of an operator given the inferred keys of the inputs. The pseudocode of the main function is listed in Figure B.1.

```
function inferAndRetouch(op) begin
    Analyze op to decide which children need to have key information ;
    foreach child operator c of op required to have key information do
        inferAndRetouch (c) ;
    Refresh input schema of op since it may have changed ;
    Apply per operator rule to infer output keys and retouch op if needed ;
```

Figure B.1. Main flow of key inference and retouching

Notice that not every operator in a logical plan needs to have keys inferred. For instance, to infer the keys of a semijoin operator, we only need the keys of its left child, but not the right child.

B.2.1 Tables with Unknown Keys

Tables with unknown keys may come up during key inference for the following possible reasons:

- Primary key constraints are naturally missing for tables in certain data sources.
- Some black-box operator or UDF (user-defined function) do not provide key inference rules or custom routines to infer keys.
- There is a circular dependency that keys of the input table cannot be determined until the keys of the query are inferred. This scenario may happen in page compilation.

The appearance of tables of unknown keys does not mean that key inference for the entire query plan should fail right away. It is possible that such tables do not appear in the query output or affect the key inference of tables that do appear in the query output. For instance, we have mentioned that for semijoin operator, the key information for the right child of the operator is never needed. Indeed, according to the above pseudo-code, the entire sub-tree rooted at the right child will not be visited for key inference at all. However, sometimes we cannot decide easily whether a sub-tree of query plan needs to be analyzed for key inference based on per-operator rules.

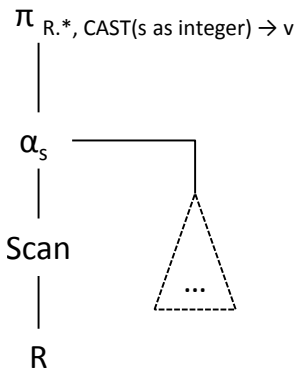


Figure B.2. An example of key inference that can tolerate intermediate tables with unknown keys.

Example B.2.1. Consider a query logical plan illustrated in Figure B.2. The apply plan create a nested table *s*, which is intended to be a singleton table that contains a single scalar attribute. With this intent, the apply plan is followed by a projection that transforms the nested table to a scalar attribute *v*. In reality, this plan can appear when translated from a query AST where the attribute *v* is specified by a sub-query in select clause[1]. Now the nested table *s* may fail to have keys inferred. But it is acceptable since it does not affect the key inference for the final output of the query. □

To handle the use cases represented by the above example, we design rules to accept unknown keys of intermediate result, only failing when any tables in the eventual query output

[1] supported by FORWARD SQL++

has unknown keys. This logic is contained in each per-operator rule, and therefore not showing in Figure B.1. We have considered the alternative design that tracks from top down which tables requires key information and only infer when it is necessary. However, this design implies passing around extra information among key inference rules, which we do not want because of higher implementation and maintenance complexity.

B.3 Key Information and Annotation

In the output schema of an operator, every table (either the root or any nested one) can be associated with either of the following kinds of key information:

- Singleton flag "SINGLETON". For instance, the output of a Ground operator is a singleton (and empty) table.
- Key attributes. In the simplest version, it is a set of attributes that form a composite key. E.g., tuple/alias1/did, tuple/alias2/eid. As a standard and convention in FORWARD, we use path expression to represent attributes since there might be intermediate tuples.

We can arrange the inferred result as a map, called *key annotation*, from a table to either SINGLETON flag of key attributes, called *key information*.

Example B.3.1. Consider the simple logical plan accessing an Applicants table that has a nested Schools table, as shown in Figure B.3. Here the scan operator reads tuples from the input table, and for each input tuple x, it attach it to an output tuple under an alias. In this example, the alias is simply "a".

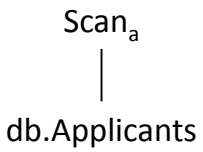


Figure B.3. Example of a single scan operator

The key annotation for the output is as follows:

{Root: {tuple/a/aid},

tuple/a/Schools: {tuple/sid}}

B.3.1 Equivalent Key Attribute Groups

Sometimes Project and Navigate[2] operators create copies of a key attribute, and in the output of the operator, either the original copy or a new copy can serve as the key attribute. We want to keep track of equivalent copies of key attributes because later on in the query plan, one copy may be projected out, but we can still use a remaining copy to serve as the key attribute.

Notice that an alternative approach is to keep track of not all equivalent copies, but a single copy only. This would require that the designated copy has to be preserved. It would possibly lead to more retouching and bigger output, which we consider non-preferable.

With *equivalent groups*, the key attributes of a table become a set of key attribute groups.

Example B.3.2. Consider a logical plan shown in Figure B.4. The following table shows key annotations for each of its operators.

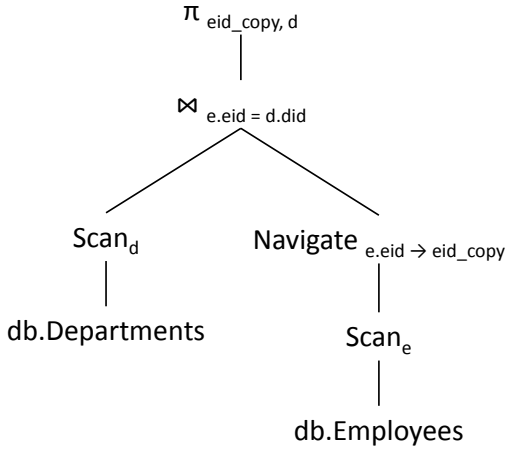


Figure B.4. Example of a query plan that has equivalent key attributes

[2] A project-like operator in SQL++

Table B.1. Example query plan with inferred keys

Project eid_copy, d	Root: [tuple/d/did], [tuple/eid_copy]
Join (e.did = d.did)	Root: [tuple/d/did], [tuple/e/eid, tuple/eid_copy]
Scan db.Departments as d	Root: [tuple/d/did]
Ground	Root: SINGLETON
Navigate e.eid as eid_copy	Root: [tuple/e/eid, tuple/eid_copy]
Scan db.Employees as e	Root: [tuple/e/eid]
Ground	Root: SINGLETON

B.4 Per Operator Rules

B.4.1 Ground

The only empty root table returned by the ground operator is marked as SINGLETON.

B.4.2 Scan (Access Path)

The child operator is analyzed first for key annotation. Then we infer the keys of scan operator in two steps. First, we get the key annotation for the scanned data. Second, we merge the key annotation of the scanned data with the key annotation of the child operator to get the key annotation of the output of scan.

Considering the scanned data, there are two scenarios:

1. If the scanned term is an absolute variable, then we construct the key annotation of the term by using the local primary key constraint of each table under the scanned term.

2. If the scanned term is a relative variable, then we get the key annotation of the term from the input key annotation.

When we merge the two key annotations together:

1. We preserve everything from the input key annotation, except for the key information for the root table.

2. We preserve everything from the key annotation of the scanned term, except for the key information for the root of the term (if the root of the term is a table and therefore has key information; notice that a scanned term can be a tuple or even a scalar value).

3. We merge the key information for the input root and the key information for the scanned term's root to get the key information for the output root. In detail:

   (a) The scanned term's root may be a tuple, in which case we consider it as SINGLETON.

   (b) If both input root and scanned term's root are SINGLETON, then the output root is SINGLETON.

   (c) Otherwise, the output root has the key information as the combination of the two roots' key information.

B.4.3 Operators that Preserve Key Annotation as Is

A class of operators that preserves input schema to the output has a very simple rule for key inference. That is, key annotation of the input is used as key annotation for the output without changes. Therefore, we only needs to analyze its child operator for key inference and propagate the input keys as is to the output. No retouching is needed.

Such operator includes Select, Sort, and Exists[3]. Notice that Exists does have an output schema that is slightly different from the input schema as the output has an additional boolean attribute representing whether the table to be tested is empty. Since this additional new attribute is not a table, Exists operator can therefore falls in this class.

B.4.4 Navigate

First, the child operator is required to be inferred with keys. The output key annotation is basically the input key annotation.

[3]Unlike the SQL Exists function, the SQL++ Exists logical operator takes a sub-plan and outputs a new boolean attribute (in addition to the input attributes) deciding whether the sub-plan evaluates to empty result.

There is only one additional handling regarding the root table. When the navigated term is a key attribute (of the root table), we add the new attribute created by the navigate to the equivalent group where the navigated term is in.

B.4.5 Project

The handling of project is based on that of navigate. Please refer to the handling of navigate first. Similar to Navigate, the child of Project needs to be inferred with keys.

Since project cannot change anything across a nested table, so the key information of any nested table will be preserved as is, if the nested table is in the projection list.

The main difference between project and navigate is about root table, since project may project out some attributes that happen to be key attributes of the root table. When this happens, it is still fine if a projected out key attribute has at least one equivalent copy that is preserved by the project. If an equivalent group becomes empty because every copy is projected out. Then we need to retouch.

Retouching

When an equivalent group of the root table becomes empty, we add an extra project item that preserves one input attribute in the equivalent group to the output. We make the new attribute a top level one (i.e., not under any intermediate tuple) and assign it a globally unique name.

Example B.4.1. Consider the following logical plan that needs retouching.

```
Project d.dname as dname
  Scan db.Departments as d
    Ground
```

After retouching, the logical plan becomes

```
Project d.dname as dname, d.did as __prov_102
  Scan db.Departments as d
```

Ground

And the key annotation is {Root: {[__prov_102]}}.

B.4.6 Product (Inner Join and Outer Join)

Product, Inner Join and Outer Join operators require all child operators be analyzed for key inference first. After that, the handling is simple. For non-root (i.e., nested) tables, we preserve the key information from its corresponding input key annotation to the output key annotation.

For the root table, if both children's root tables are SINGLETON, then the output root table is SINGLETON too. Otherwise, we combine the lists of key attributes (groups) of both input root tables to form the key information of the output root table.

No retouching is needed.

B.4.7 Semijoin and Anti-Semijoin

For these two operators, only the left child needs to be inferred with keys, since the schema of the output is the schema of the left child. After that, the handling is simple: Input key annotation of the left child becomes the output key annotation.

No retouching is needed.

B.4.8 ApplyPlan

We need to infer keys with the child operator. Also we need to run the inference and retouch procedure for the nested plan. And before we start call key inference routine with the nested plan, we need to have the key annotation sorted out for the ApplyPlan's input because the nested plan may access (through parameter) some outer context provided by the operator's input.

The output key annotation is basically the input key annotation, except for the sub-tree rooted at the new nested table created by ApplyPlan. We copy from the nested plan's output key annotation for the new nested table and any tables that are further nested under it.

No retouching is needed.

B.4.9 Distinct (Any Set Operator with DISTINCT Quantifier)

We assume there is a Distinct operator. If not, what is discussed here can be applied to every set operator that has a DISTINCT quantifier.

We also assume that input of distinct (and hence the output of it) does not have any nested table.

For the distinct operator, we do not need to visit its children for key inference. In fact, key retouching should be disallowed for children, because it may change the input schema and affect the correctness of distinct. Also since there is no nested table, we do not need to do key inference either with the children. The output key information (of the root table) is the entire list of attributes.

B.4.10 Union All and Outer Union All

Here we discuss Union and Outer Union with bag semantics. First, child operators need to be visited first for key inference and retouching. If there is any nested table in both input branches (if it appears in only one input branch it is trivial to handle), its type and inferred key information has to be the same from both input branches. Otherwise we will fail.

We describe the key inference rule with retouching steps, as they turn out to be very related.

Retouching

Retouching may be needed to infer key information of the root table. The steps are as follows.

Changing union to outer union

After retouching that happens to a union's two child branches, the type of the two branches may no longer match. Therefore we may need to transform a union to outer union[4].

Adding branch index attribute

The output key is basically the combination of the input keys from both branches. In addition, we add a *branch index attribute* to both input branches (using an extra project) before they are unioned or outer unioned. This branch index attribute will be filled with constant 1 for first branch and constant 2 for the second branch. This attribute will be added to output root table's key information. This implies that the output root table will never be SINGLETON.

It is possible to not require this branch index attribute. But it would require non-trivial analysis of the input. For instance, even if the two input branches have different key information for root table, we may still need the branch index attribute.

Merging key groups for root table

Key groups are tricky concept and probably does not survive across unions or set operators. Therefore we grab the first element from each group and combine those singleton groups, as opposed to potentially merge groups.

NULL value for key attribute

The NULL value we use to fill the key attributes of one branch in another branch of outer union has different equality semantics than the normal relational NULL value. We call it NULL* and two NULL* values are considered equal. Notice that this is similar to how SQL handles NULL value in grouping attributes of Group By.

Example B.4.2. Consider the following logical plan that needs retouching.

Union ALL

Project ename as name

[4]Outer union is a FORWARD extension in SQL++.

db.employees

Project cname as name

db.contractors

After retouching, the logical plan becomes

Outer-Union ALL

Project ename as name, eid as __prov_100, 1 as __union_branch

db.employees

Project cname as name, cid as __prov_101, 2 as __union_branch

db.contractors

And the key annotation for the output is {Root: {[__prov_100], [__prov_101], [__union_branch]}}.

B.4.11 Group-By

Group-by operator can always have its top-level key inferred as exactly the list of group-by attributes. However, we may do better if the group-by list contains all input key attributes. In that case, the input key attributes serve as a smaller, and better, choice of output key.

No retouching is needed since keys can always be inferred.

Group-By with Nest

In order to have keys inferred with the nested table created by NEST aggregate, the NEST aggregate has to include all input key attributes that are not included in the group-by attributes. The included key attributes minus the group-by attributes can serve as the inferred key of the nested table.

For nested tables that are preserved from input to output, either as part of the outer table or the NEST aggregate, their keys are carried over as is.

Appendix C

i-diff Propagation Rules

Table C.1. Rules for $\times$

For $\Delta^{+}_{Input_l}(\bar{I},\bar{A}''_{post})$
$\Delta^{+}_{V} = \Delta^{+}_{Input_l} \times Input_r^{post}$
For $\Delta^{+}_{Input_r}(\bar{I},\bar{A}''_{post})$
$\Delta^{+}_{V} = Input_l^{post} \times \Delta^{+}_{Input_r}$
For $\Delta^{-}_{Input_l}(\bar{I},\bar{A}'_{pre})$ ($\Delta^{-}_{Input_r}$ is symmetric)
$\Delta^{-}_{V} = \Delta^{-}_{Input_l}$
For $\Delta^{u}_{Input_l}(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$ ($\Delta^{u}_{int\,pu_r}$ is symmetric)
$\Delta^{u}_{V} = \Delta^{u}_{Input_l}$

Table C.2. Rules for $\cup$

For $\Delta^{+}_{Input_l}(\bar{I},\bar{A}_{post})$ (For $\Delta^{+}_{Input_r}$ replace 0 by 1)
$\Delta^{+}_{V} = \pi_{*,b\to 0}\Delta^{+}_{Input_l}$
For $\Delta^{-}_{Input_l}(\bar{I},\bar{A}'_{pre})$ (For $\Delta^{-}_{Input_r}$ replace 0 by 1)
$\Delta^{-}_{V} = \pi_{*,b\to 0}\Delta^{-}_{Input_l}$
For $\Delta^{u}_{Input_l}(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$ (For $\Delta^{u}_{Input_r}$ replace 0 by 1)
$\Delta^{-}_{V} = \pi_{*,b\to 0}\Delta^{u}_{Input_l}$

Table C.3. Rules for $\sigma_{\phi(\bar{X})}$

For $\Delta^{+}_{Input}(\bar{I}, \bar{A}_{post})$
$\Delta^{+}_{V} = \sigma_{\phi(\bar{X})}\Delta^{+}_{Input}$
For $\Delta^{-}_{Input}(\bar{I}, \bar{A}'_{pre})$
$\Delta^{-}_{V} = \sigma_{\phi(\bar{X}_{pre})}\Delta^{-}_{Input}$
For $\Delta^{u}_{Input}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$

if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{u}_{V} = \sigma_{\phi(\bar{X}_{pre})}\sigma_{\phi(\bar{X})}\Delta^{u}_{Input}$
else
$\quad \Delta^{u}_{V} = \Delta^{-}_{Input}$

if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{+}_{V} =$ not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{+}_{V} = Input^{post} \ltimes_{\bar{I}} \sigma_{\neg\phi(\bar{X}_{pre})}\sigma_{\phi(\bar{X})}\Delta^{u}_{Input}$
else
$\quad \Delta^{+}_{V} = \sigma_{\neg\phi(\bar{X}_{pre})}\sigma_{\phi(\bar{X})}(Input^{post} \ltimes \Delta^{u}_{Input})$

if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{-}_{V} =$ not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{-}_{V} = \pi_{\bar{I},\bar{A}'_{pre}}\sigma_{\phi(\bar{X}_{pre})}\sigma_{\neg\phi(\bar{X})}\Delta^{u}_{Input}$
else
$\quad \Delta^{-}_{V} = \pi_{\bar{I},\bar{A}'_{pre}}\sigma_{\phi(\bar{X}_{pre})}\sigma_{\neg\phi(\bar{X})}Input^{post} \ltimes \Delta^{u}_{Input}$

Blue portion applies when pre-state attributes present.

Table C.4. Rules for $\gamma_{\bar{G},f(\bar{X})\to c}$

For $\Delta^{-}_{Input}(\bar{I}, \bar{A}'_{pre})$ where $\bar{I} \subseteq \bar{G}$
$\Delta^{-}_{V} = \Delta^{-}_{Input}$
For any $\Delta^{t}_{Input}(\bar{I}, \bar{A}')$ and f, we can recompute groups

if $\bar{G} \subseteq (\bar{I} \cup \bar{A}')$ then
$\quad \Delta^{u}_{V} = \gamma_{\bar{G},f(\bar{X})\to c}(\Delta^{t}_{Input} \rtimes_{\bar{G}} Input^{post})$
else
$\quad \Delta^{u}_{V} = \gamma_{\bar{G},f(\bar{X})\to c}(\Delta^{t}_{Input} \rtimes_{\bar{I}} Input^{post} \rtimes_{\bar{G}} Input^{post})$

(Do not handle group creation/deletion)

Table C.5. Rules for $\pi_{\bar{D},f(\bar{X})\to c}$

For $\Delta^{+}_{Input}(\bar{I},\bar{A}_{post})$
$\Delta^{+}_{V} = \pi_{\bar{D},f(\bar{X})\to c,\bar{I}}\Delta^{+}_{Input}$
For $\Delta^{-}_{Input}(\bar{I},\bar{A}'_{pre})$

if $\bar{X} \subseteq \bar{I} \cup \bar{A}'_{pre}$ then
$\quad \Delta^{-}_{V} = \pi_{(\bar{D}\cap(\bar{I}\cup\bar{A}'_{pre}))\cup f(\bar{X}_{pre})\to c_{pre},\bar{I}}\Delta^{-}_{Input}$
else
$\quad \Delta^{-}_{V} = \pi_{\bar{D}\cap(\bar{I}\cup\bar{A}'_{pre}),\bar{I}}\Delta^{-}_{Input}$

For $\Delta^{u}_{Input}(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$

if $(\bar{I}\cup\bar{A}''_{post})\cap\bar{X} = \emptyset$ then
$\quad \Delta^{u}_{V} = \sigma_{isupd}\pi_{\bar{D}',\bar{I}}\Delta^{u}_{Input}$
else if $\bar{X} \subseteq \bar{I}\cup\bar{A}''_{post}$ then
$\quad \Delta^{u}_{V} = \sigma_{isupd}\pi_{\bar{D}',f(\bar{X})\to c_{post},f(\bar{X}_{pre})\to c_{pre},\bar{I}}\Delta^{u}_{Input}$
else
$\quad \Delta^{u}_{V} = \sigma_{isupd}\pi_{\bar{D}',f(\bar{X})\to c_{post},f(\bar{X}_{pre})\to c_{pre},\bar{I}}$
$\qquad (Input^{post} \ltimes_{\bar{I}} \Delta^{u}_{Input})$
where σ_{isupd} selects tuples corresponding to actual updates (i.e., where $c_{pre} \neq c_{post}$ or $a_{pre} \neq a_{post}$ for some attribute a), $\bar{D}' = (\bar{D}\cap(\bar{I}\cup\bar{A}''_{post}))\cup(\bar{D}_{pre}\cap\bar{A}'_{pre})$, and $\bar{D}_{pre}$ is the pre-state counterpart of $\bar{D}$.

Blue portion applies when pre-state attributes present.

Table C.6. Rules for $\gamma_{\bar{G},sum(\bar{X})\to c}$

For $\Delta^{u}_{Input}(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$, and $\bar{G}\cap\bar{A}''_{post} = \emptyset$
$\Delta^{i}_{1} = \pi_{\bar{I},x_{post}-x_{in}\to x_{\Delta}}(\Delta^{u}_{Input} \bowtie \pi_{x\to x_{in}}Input^{pre})$
For $\Delta^{-}_{Input}(\bar{I},\bar{A}'_{pre})$
$\Delta^{j}_{2} = \pi_{\bar{I},0-x_{in}\to x_{\Delta}}(\Delta^{-}_{Input} \bowtie \pi_{x\to x_{in}}Input^{pre})$
For $\Delta^{+}_{Input}(\bar{I},\bar{A}''_{post})$
$\Delta^{k}_{3} = \pi_{\bar{I},x\to x_{\Delta}}(\Delta^{+}_{Input} \triangleright Input^{pre})$
For converting Δ to output update i-diffs
Happens after all $\Delta^{i}_{1},\Delta^{j}_{2},\Delta^{k}_{3}$ are computed. $\Delta^{u}_{V} = \pi_{\bar{G},c\to c_{pre},c+c_{\Delta}\to c_{post}}(Output \bowtie$ $\quad \gamma_{\bar{G},sum(x_{\Delta})\to c_{\Delta}}(\Delta^{i}_{1}\cup\Delta^{j}_{2}\cup\Delta^{k}_{3}))$

(Do not handle group creation/deletion)

Table C.7. Rules for $\bowtie_{\phi(\bar{X})}$

For $\Delta^{+}_{Input_l}(\bar{I}, \bar{A}_{post})$
$\Delta^{+}_{V} = \Delta^{+}_{Input_l} \bowtie_{\phi(\bar{X})} Input_r^{post}$
For $\Delta^{+}_{Input_r}(\bar{I}, \bar{A}_{post})$
$\Delta^{+}_{V} = Input_l^{post} \bowtie_{\phi(\bar{X})} \Delta^{+}_{Input_r}$
For $\Delta^{-}_{Input_l}(\bar{I}, \bar{A}'_{pre})$ ($\Delta^{-}_{Input_r}$ is symmetric)
$\Delta^{-}_{V} = \sigma_{\phi(\bar{X}_{pre})} \Delta^{-}_{Input_l}$
For $\Delta^{u}_{Input_l}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$ ($\Delta^{u}_{Input_r}$ is symmetric)

if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{u}_{V} = \sigma_{\phi(\bar{X}_{pre})} \sigma_{\phi(\bar{X})} \Delta^{u}_{Input_l}$
else
$\quad \Delta^{u}_{V} = \Delta^{u}_{Input_l}$

if $\bar{I} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{+}_{V} =$ not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{+}_{V} = (Input_l^{post} \ltimes_{\bar{I}} \sigma_{\neg\phi(\bar{X}_{pre})} \sigma_{\phi(\bar{X}_{post})} \Delta^{u}_{Input_l})$
$\qquad \bowtie_{\phi(\bar{X})} Input_r^{post}$
else
$\quad \Delta^{+}_{V} = \pi_{Input_l, Input_r} \sigma_{\neg\phi(\bar{X}_{pre})} \sigma_{\phi(\bar{X}_{post})}$
$\qquad (Input_l^{post} \bowtie \Delta^{u}_{Input_l} \bowtie_{\phi(\bar{X})} Input_r^{post})$

if $\bar{I} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{-}_{V} =$ not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{-}_{V} = \pi_{\bar{I}, \bar{A}'_{pre}} \sigma_{\phi(\bar{X}_{pre})} \sigma_{\neg\phi(\bar{X}_{post})} \Delta^{u}_{Input_l}$
else
$\quad \Delta^{-}_{V} = \pi_{\bar{I}, \bar{A}'_{pre}} \sigma_{\phi(\bar{X}_{pre})} \sigma_{\neg\phi(\bar{X}_{post})} Input_l^{post}$
$\qquad \bowtie \Delta^{u}_{Input_l} \bowtie_{\phi(\bar{X})} Input_r^{post}$

Blue portion applies when pre-state attributes present.

Table C.8. Rules for $\gamma_{\bar{G},count(\bar{X})\to c}$

For $\Delta^u_R(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$, and $\bar{G}\cap\bar{A}''_{post}=\emptyset$
$\Delta^i_1=\emptyset$
For $\Delta^-_R(\bar{I},\bar{A}'_{pre})$
$\Delta^j_2=\pi_{\bar{I},-1\to x_\Delta}(\Delta^-_R \bowtie \pi_{x\to x_{in}} Input)$
For $\Delta^+_R(\bar{I},\bar{A}''_{post})$
$\Delta^k_3=\pi_{\bar{I},1\to x_\Delta}(\Delta^+_R \triangleright Input)$
For converting Δ to output update i-diffs
Happens after all $\Delta^i_1,\Delta^j_2,\Delta^k_3$ are computed. $\Delta^u_V=\pi_{\bar{G},c\to c_{pre},c+c_\Delta\to c_{post}}(Output \bowtie$ $\quad \gamma_{\bar{G},sum(x_\Delta)\to c_\Delta}(\Delta^i_1\cup\Delta^j_2\cup\Delta^k_3))$

(Do not handle group creation/deletion)

Table C.9. Rules for $\gamma_{\bar{G},avg(\bar{X})\to c}$

Operator cache schemas: $Cache_{sum}(\bar{G},c_{sum})$, $Cache_{count}(\bar{G},c_{count})$
Cache maintenance rules: For $\Delta^u_{Cache_{sum}}$: Use rules of $\gamma_{\bar{G},sum(\bar{X})\to c}$ (Table C.6) For $\Delta^u_{Cache_{count}}$: Use rules of $\gamma_{\bar{G},count(\bar{X})\to c}$ (Table C.8)
i-diff propagation rules: $\Delta^u_V=\pi_{\bar{G},c^{sum}_{pre}/c^{count}_{pre}\to c_{pre},c^{sum}_{post}/c^{count}_{post}\to c_{post}}($ $\quad \Delta^u_{Cache_{count}} \bowtie_{\bar{G}} \Delta^u_{Cache_{sum}})$

Table C.10. Rules for $\triangleright_{\phi(Input_l.\bar{X},Input_r.\bar{Y})}$ first part

For $\Delta^{+}_{Input_l}(\bar{I},\bar{A}''_{post})$
$\Delta^{+}_{V} = \Delta^{+}_{Input_l} \triangleright_{\phi} Input^{post}_{r}$
For $\Delta^{-}_{Input_l}(\bar{I},\bar{A}'_{pre})$
$\Delta^{-}_{V} = \Delta^{-}_{Input_l}$
For $\Delta^{u}_{Input_l}(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$
$\Delta^{u}_{V} = \Delta^{u}_{Input_l}$

if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{+}_{V} =$ not triggered
if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{+}_{V} = Input^{post}_{l} \ltimes_{\bar{I}_{Input_l}} (\Delta^{u}_{Input_l} \triangleright_{\phi(\bar{X}_{post},\bar{Y})} Input^{post}_{r})$
else
$\quad \Delta^{+}_{V} = (Input^{post}_{l} \ltimes_{\bar{I}_{Input_l}} \Delta^{u}_{Input_l}) \triangleright_{\phi(\bar{X}_{post},\bar{Y})} Input^{post}_{r}$

if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{-}_{V} =$ not triggered
if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\quad \Delta^{-}_{V} = \pi_{\bar{I}}(\Delta^{u}_{Input_l} \ltimes_{\phi(\bar{X}_{post},\bar{Y})} Input^{post}_{r})$
else
$\quad \Delta^{-}_{V} = \pi_{\bar{I}}((Input^{post}_{l} \ltimes_{\bar{I}_{Input_l}} \Delta^{u}_{Input_l})$
$\qquad \ltimes_{\phi(\bar{X}_{post},\bar{Y})} Input^{post}_{r})$

For $\Delta^{+}_{Input_r}(\bar{I},\bar{A}''_{post})$
$\Delta^{-}_{V} = \pi_{\bar{I}}(Input^{post}_{l} \ltimes_{\phi(\bar{X}_{post},\bar{Y})} \Delta^{+}_{Input_r})$
For $\Delta^{-}_{Input_r}(\bar{I},\bar{A}'_{pre})$

if $\bar{Y} \subseteq \bar{I} \cup \bar{A}'_{pre}$ then
$\quad \Delta^{+}_{V} = (Input^{post}_{l} \ltimes_{\phi(\bar{X}_{pre},\bar{Y})} \Delta^{-}_{Input_r})$
$\qquad \triangleright_{\phi(\bar{X}_{post},\bar{Y})} Input^{post}_{r}$
else
$\quad \Delta^{+}_{V} = (Input^{post}_{l} \ltimes_{\phi(\bar{X}_{pre},\bar{Y})} (Input^{pre}_{r} \ltimes \Delta^{-}_{Input_r}))$
$\qquad \triangleright_{\phi(\bar{X}_{post},\bar{Y})} Input^{post}_{r}$

Table C.11. Rules for $\triangleright_{\phi(Input_l.\bar{X},Input_r.\bar{Y})}$ second part

For $\Delta^{u}_{Input_r}(\bar{I},\bar{A}'_{pre},\bar{A}''_{post})$
Treat input update as combination of insert and delete

if $\bar{Y} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{-}_{V} =$ not triggered
else
$\quad \Delta^{-}_{V} = \pi_{\bar{I}}(Input_l^{post} \ltimes_{\phi(\bar{X}_{post},\bar{Y})} (Input_r^{post} \ltimes \Delta^{u}_{Input_r}))$

if $\bar{Y} \cap \bar{A}''_{post} = \emptyset$ then
$\quad \Delta^{+}_{V} =$ not triggered
else if $\bar{Y} \subseteq \bar{I} \cup \bar{A}'_{pre}$ then
$\quad \Delta^{+}_{V} = (Input_l^{post} \ltimes_{\phi(\bar{X}_{pre},\bar{Y})} \Delta^{-}_{Input_r})$
$\qquad \triangleright_{\phi(\bar{X}_{post},\bar{Y})} Input_r^{post}$
else
$\quad \Delta^{+}_{V} = (Input_l^{post} \ltimes_{\phi(\bar{X}_{pre},\bar{Y})} (Input_r^{pre} \ltimes \Delta^{-}_{Input_r}))$
$\qquad \triangleright_{\phi(\bar{X}_{post},\bar{Y})} Input_r^{post}$

Appendix D

SQL++ Algebra Semantics

Most SQL++ operators are the direct correspondence of the SQL++ core semantics. The remaining operators support optimizations of special cases. For example, since `INNER JOIN` is a special case of the core `INNER CORRELATE`, SQL++ also provides the classical $\bowtie$ operator, which is amenable to join re-ordering due to its commutativity and associativity.

D.1 Novel Semi-Structured Operators

1. The ScanCollection operator $\ggg^{C}_{\ddot{c}\mapsto(x,y)}(B)$ is the algebraic counterpart of `FROM` $\ddot{c}$ `AS` x `AT` y. It inputs a bag of binding tuples B, and outputs a bag of binding tuples. For each input binding tuple $b \in B$, for each element $v \in \ddot{c}$, the operator outputs a binding tuple $\langle x : v,\ y : p\rangle$. As specified by SQL++ semantics, p is the ordinal position of v when $\ddot{c}$ is an array, or the value specified by `@from.bag_order` when $\ddot{c}$ is a bag. The cases when $\ddot{c}$ is not a collection are also as specified by config options. Since config options apply to SQL++ operators identically as they apply to SQL++ language semantics, we omit config options from subsequent definitions for ease of exposition.

2. The ScanTuple operator $\ggg^{T}_{\ddot{t}\mapsto(x,y)}(B)$ is the algebraic counterpart of `FROM` $\ddot{t}$ `AS` $\{x\!:\!y\}$. It inputs a bag of binding tuples B, and outputs a bag of binding tuples. For each input binding tuple $b \in B$, wherein $\ddot{t}$ evaluates to $\{a_1\!:\!v_1, \ldots, a_n\!:\!v_n\}$, the operator outputs binding tuples $\langle x : a_i,\ y : v_i\rangle$, for $i = 1,\ldots,n$.

3. The Ground operator is the only accepted leaf node of an algebraic plan. It has no input, and always outputs a bag comprising a single empty binding tuple $\langle\,\rangle$, so that its parent operator (such as $\ggg^C$and $\ggg^T$) has exactly one binding tuple to iterate over. In *idIVM* we use Ground to implement the environment tuple Γ such that instead of outputting an empty binding tuple, Ground will output Γ as a single binding tuple. Due to its trivial nature, Ground is omitted from example plans and further discussions.

4. The NavArray operator $[\,]_{(\ddot{x},\ddot{y})\mapsto z}(B)$ is the algebraic counterpart of $\ddot{x}[\ddot{y}]$. It inputs a bag of binding tuples B, and outputs a bag of binding tuples. For each input binding tuple $b \in B$, the operator outputs a binding tuple $b \| \langle z : v \rangle$, where v is the result of navigating into array $\ddot{x}$ by ordinal position $\ddot{y}$.

5. The NavTuple operator $\bullet_{(\ddot{x},\ddot{y})\mapsto z}(B)$ is the algebraic counterpart of $\ddot{x}.\ddot{y}$. It inputs a bag of binding tuples B, and outputs a bag of binding tuples. For each input binding tuple $b \in B$, the operator outputs a binding tuple $b \| \langle z : v \rangle$, where v is the result of navigating into tuple $\ddot{x}$ by attribute name $\ddot{y}$.

6. The FunctionCall operator $\lambda_{(f,\,\ddot{x}_1 \ldots \ddot{x}_n)\mapsto y}(C)$ is the algebraic counterpart of invoking function $f(\ddot{x}_1 \ldots \ddot{x}_n)$. It inputs an array (resp. bag) of binding tuples C, and outputs an array (resp. bag) of binding tuples. For each input binding tuple $b \in B$, the operator outputs a binding tuple $b \| \langle y : v \rangle$, where v is the result of evaluating function f with arguments $\ddot{x}_1 \ldots \ddot{x}_n$.

7. The ReturnArray$_{\ddot{x}}(A)$ operator is the algebraic counterpart of `SELECT ELEMENT` $\ddot{x}$, when the SFW query has an `ORDER BY` clause. It inputs an array of binding tuples A, and outputs an array value. For each input binding tuple $b \in A$, the operator outputs an array element $\ddot{x}$.

8. The ReturnBag$_{\ddot{x}}(B)$ operator is the algebraic counterpart of `SELECT ELEMENT` $\ddot{x}$, when the SFW query does not have an `ORDER BY` clause. It inputs a bag of binding tuples B,

and outputs a bag value. For each input binding tuple $b \in B$, the operator outputs a bag element $\ddot{x}$.

9. The $\text{ReturnTuple}_{\ddot{x},\ddot{y}}(B)$ operator is the algebraic counterpart of `SELECT ATTRIBUTE` $\ddot{x}:\ddot{y}$. It inputs a bag of binding tuples B, and outputs a tuple value. For each input binding tuple $b \in B$, the operator outputs a tuple attribute $\ddot{x}:\ddot{y}$.

10. The $\text{ReturnSingle}_{\ddot{x}}(B)$ operator is the algebraic counterpart of a restricted expression $\ddot{x}$. It inputs a bag B that has exactly one binding tuple, and outputs the value $\ddot{x}$.

11. The ApplyPlan operator $\alpha_{P \mapsto x}(B)$ is the algebraic counterpart of subqueries, i.e. SFW queries enclosed within `(...)`. Since a subquery can appear anywhere within a SQL++ query, an α operator can appear anywhere within a plan. The operator inputs a bag of binding tuples B, and outputs a bag of binding tuples. Suppose the operator's environment is Γ. For each input binding tuple $b \in B$, the operator evaluates plan P in an augmented environment $\Gamma \| b$ to a value v, and outputs the binding tuple $b \| \langle x : v \rangle$.

12. The $\text{Assign}_{\ddot{x} \mapsto y}(B)$ operator inputs an array (resp. bag) of binding tuples C, and outputs an array (resp. bag) of binding tuples. For each input binding tuple $b \in C$, the operator outputs $b \| \langle y : \ddot{x} \rangle$.

13. The $\text{InnerCorrelate}_P(B^l)$ operator is the algebraic counterpart of `FROM` l `INNER CORRELATE` r, where B^l is the bag of binding tuples output by the algebraic counterpart of query l, and plan P is the algebraic counterpart of query r. The operator inputs a bag of binding tuples B^l, and outputs a bag of binding tuples. Suppose the operator's environment is Γ. For each input binding tuple $b_i^l \in B^l$, the operator evaluates plan P in an augmented environment $\Gamma \| b$ to a bag of binding tuples B_i^r. The operator outputs all binding tuples $b_i^l \| b_{i,j}^r$ for $b_i^l \in B^l, b_{i,j}^r \in B_i^r$.

14. The $\text{LeftCorrelate}_P(B^l)$ operator is the algebraic counterpart of `FROM` l `LEFT CORRELATE` r, where B^l is the bag of binding tuples output by the

algebraic counterpart of query l, and plan P is the algebraic counterpart of query r. The operator inputs a bag of binding tuples B^l, and outputs a bag of binding tuples. Suppose the operator's environment is Γ. For each input binding tuple $b_i^l \in B^l$, the operator evaluates plan P in an augmented environment $\Gamma \| b$ to a bag B_i^r of binding tuples $\langle x_1 : v_1, \ldots, x_n : v_n \rangle$. The operator outputs all binding tuples $b_i^l \| b_{i,j}^r$ for $b_i^l \in B^l, b_{i,j}^r \in B_i^r$. In addition, for each $b_i^l \in B^l$ such that B_i^r is an empty bag, the operator also outputs a binding tuple $b_i^l \| \langle x_1 : \texttt{@from.no\_match}, \ldots, x_n : \texttt{@from.no\_match} \rangle$.

15. The ConstructArray$_{(\ddot{x}_1 \ldots \ddot{x}_n) \mapsto y}(C)$ operator is the algebraic counterpart of the array constructor `[`$\ddot{x}_1$`, ..., `$\ddot{x}_n$`]`. It inputs an array (resp. bag) of binding tuples C, and outputs an array (resp. bag) of binding tuples. For each input binding tuple $b \in C$, the operator outputs $b \| \langle y : v \rangle$, where v is an array value `[`$\ddot{x}_1$`, ..., `$\ddot{x}_n$`]`.

16. The ConstructBag$_{(\ddot{x}_1 \ldots \ddot{x}_n) \mapsto y}(C)$ operator is the algebraic counterpart of the bag constructor $\{\{\ddot{x}_1, \ldots, \ddot{x}_n\}\}$. It inputs an array (resp. bag) of binding tuples C, and outputs an array (resp. bag) of binding tuples. For each input binding tuple $b \in C$, the operator outputs $b \| \langle y : v \rangle$, where v is a bag value $\{\{\ddot{x}_1, \ldots, \ddot{x}_n\}\}$.

17. The ConstructTuple$_{(\ddot{x}_1 : \ddot{y}_1 \ldots \ddot{x}_n : \ddot{y}_n) \mapsto z}(C)$ operator is the algebraic counterpart of the tuple constructor

    $\{\ddot{x}_1 : \ddot{y}_1, \ldots, \ddot{x}_n : \ddot{y}_n\}$. It inputs an array (resp. bag) of binding tuples C, and outputs an array (resp. bag) of binding tuples. For each input binding tuple $b \in C$, the operator outputs $b \| \langle z : v \rangle$, where v is a tuple value $\{\ddot{x}_1 : \ddot{y}_1, \ldots, \ddot{x}_n : \ddot{y}_n\}$

D.2 Extensions of Relational Operators

Except for Sort and OffsetLimit, all SQL++ extensions of relational operators input and output a bag of binding tuples.

18. The Select operator $\sigma_{\ddot{c}}(B)$ is the algebraic counterpart of `WHERE` $\ddot{c}$. For each input binding tuple $b \in B$, the operator outputs b if $\ddot{c}$ is `true`.

19. The Project operator $\pi_{x_1 \ldots x_n}(B)$ retains only the specified variables. For each input binding tuple $b \in B$, the operator outputs a binding tuple b' that retains variables $x_1 \ldots x_n$ of b and input variables that belong to the environment, and omits all other variables.

20. The InnerJoin operator $\bowtie_{\ddot{c}} (B^l, B^r)$ is the algebraic counterpart of `FROM` l `INNER JOIN` r `ON` c, where B^l (resp. B^r) is the bag of binding tuples output by the algebraic counterpart of query l (resp. r). For each input binding tuple $b^l \in B^l, b^r \in B^r$, the operator outputs $b^l \| b^r$ if $\ddot{c}$ is `true`.

21. The LeftJoin operator $\bar{\bowtie}_{\ddot{c}}(B^l, B^r)$ is the algebraic counterpart of `FROM` l `LEFT JOIN` r `ON` c, where B^l (resp. B^r) is the bag of binding tuples output by the algebraic counterpart of query l (resp. r). For each input binding tuple $b^l \in B^l, b^r \in B^r$, the operator outputs $b^l \| b^r$ if $\ddot{c}$ is `true`. In addition, for each b^l such that there is no b^r for which $\ddot{c}$ is `true`, the operator also outputs a binding tuple $b^l \| \langle x_1 : \texttt{@from.no\_match}, \ldots, x_n : \texttt{@from.no\_match} \rangle$, where $x_1 \ldots x_n$ are the variables defined by query r.

22. The FullJoin operator $\bar{\bowtie}\bar{}_{\ddot{c}}(B^l, B^r)$ is the algebraic counterpart of `FROM` l `FULL JOIN` r `ON` c, where B^l (resp. B^r) is the bag of binding tuples output by the algebraic counterpart of query l (resp. r). For each input binding tuple $b^l \in B^l, b^r \in B^r$, the operator outputs $b^l \| b^r$ if $\ddot{c}$ is `true`. In addition, for each b^l such that there is no b^r for which $\ddot{c}$ is `true`, the operator also outputs a binding tuple $b^l \| \langle x_1 : \texttt{@from.no\_match}, \ldots, x_n : \texttt{@from.no\_match} \rangle$, where $x_1 \ldots x_n$ are the variables defined by query r. Conversely, for each b^r such that there is no b^l for which $\ddot{c}$ is `true`, the operator also outputs a binding tuple $\langle y_1 : \texttt{@from.no\_match}, \ldots, y_n : \texttt{@from.no\_match} \rangle \| b_r$, where $y_1 \ldots y_n$ are the variables defined by query l.

23. The GroupBy operator $\gamma_{(\ddot{x}_1 \mapsto y_1, \ldots, \ddot{x}_n \mapsto y_n), g}(B)$ is the algebraic counterpart of:

    `GROUP BY` $\ddot{x}_1$ `AS` y_1`, ...,` $\ddot{x}_n$ `AS` y_n `INTO` g. It inputs a bag of binding tuples B, and outputs a bag of binding tuples. The input binding tuples of B are partitioned into the minimal number of binding bags $B_1 \ldots B_m$, such that any two binding tuples $b, b' \in B$ are in the same binding bag B_j $(1 \leq j \leq m)$ if and only if $\ddot{x}_1 \ldots \ddot{x}_n$ evaluate to identical values $v_1 \ldots v_n$ for both b and b'. Also note that before an input binding tuple b is put into a binding bag, its variables that belong to the input environment Γ are removed. Then for each binding bag B_j, the operator outputs a binding tuple $\langle y_1 : v_1,\ \ldots,\ y_n : v_n,\ g : \text{bag}(B_j) \rangle \| \Gamma$. The function $\text{bag}(B_j)$ inputs a bag of binding tuples, and outputs the equivalent bag of tuples: each binding tuple $\langle a_1 : u_1, \ldots, a_p : u_p \rangle$ has equivalent tuple $\{a_1 : u_1, \ldots, a_p : u_p\}$.

D.3 Operator provenance inference rules

This section describes the provenance inference rules for selected operators in detail. First of all, some operators simply propagates input provenance to the output, and their provenance inference rules are trivial. Such operators include Select and Project.

Ground The output single binding tuple of a Ground operator has empty provenance.

ScanCollection For a ScanCollection operator $\ggg^{C}_{\ddot{c} \mapsto (x,y)}(B)$, for each input binding tuple $b \in B$, for each element $v \in \ddot{c}$, the provenance of the output binding tuple is the combined provenance from b and v. The provenance of variables in b stays the same from input to output. The provenance of the x variable which corresponds to $v \in \ddot{c}$ and its descendants stay the same from $\ddot{c}$ to the output. When $\ddot{c}$ is a relational base table, it is considered part of the environment Γ of the plan, and the provenance of v in the plan is inferred from its primary key in the base table. The tuples of $\ddot{c}$ and every attribute of the tuple have the same provenance in the form of $\#(pk_1 = v_1, \ldots, pk_n = v_n)$ where pk_i are the primary key attributes of $\ddot{c}$.

NavTuple For a NavTuple operator $\bullet_{(\ddot{x}, \ddot{y}) \mapsto z}(B)$ where both x and y are constants, each output binding tuple and every variable that comes from the input binding tuple have the same

provenance as their input counterparts. The provenance of z and its descendants are the same as the provenance of $\ddot{x}.\ddot{y}$ and its descendants.

InnerJoin For an InnerJoin operator $\bowtie_{\ddot{c}}(B^l, B^r)$, for each pair of input binding tuple $b^l \in B^l, b^r \in B^r$ where $\ddot{c}$ is `true`, the output binding tuple has the combined provenance from b^l and b^r. The provenance of variables from b^l and b^r stay the same from input to output.

LeftJoin For a LeftJoin operator $⟕_{\ddot{c}}(B^l, B^r)$, for each pair of input binding tuple $b^l \in B^l, b^r \in B^r$ where $\ddot{c}$ is `true`, the corresponding output binding tuple has the same handling of provenance as that of an InnerJoin. For each b^l such that there is no b^r for which $\ddot{c}$ is `true`, the provenance of the corresponding output binding tuple and its variables from b^l is the same as the input provenance from b^l.

GroupBy For a GroupBy operator $\gamma_{(\ddot{x}_1 \mapsto y_1, \ldots, \ddot{x}_n \mapsto y_n), g}(B)$, each output binding tuple (i.e., group) $\langle y_1 : v_1,\ \ldots,\ y_n : v_n,\ g : v_g \rangle$ has the provenance as $\#(y_1 : v_1, \ldots, y_n : v_n)$. Each tuple in variable g has the same provenance as its counterpart binding tuple from the input. Same for its descendants.

Assign The new variable y introduced by the $\text{Assign}_{\ddot{x} \mapsto y}(B)$ operator has the same provenance as $\ddot{x}$. The provenance for the rest of the output is the same as its input counterpart.

InnerCorrelate For an $\text{InnerCorrelate}_P(B^l)$ operator, for each output binding tuple that corresponds to $b^l_i \| b^r_{i,j}$, where b^l_i is from B^l and $b^r_{i,j}$ is from the evaluation of plan P, its provenance is the concatenated provenance of b^l_i and $b^r_{i,j}$. The provenance of variables in the output is the same as the provenance of its input counterpart from b^l_i and $b^r_{i,j}$.

ApplyPlan For an ApplyPlan operator $\alpha_{P \mapsto x}(B)$, each output binding tuple and every variable that comes from the input binding tuple have the same provenance as their input counterparts. For each input binding tuple $b \in B$, the operator evaluates plan P to a value v, infers the provenance of v recursively when evaluating P, and outputs the binding tuple $b \| \langle x : v \rangle$. The provenance of variable x and its descendants are the same as the provenance of v and its descendants.

Appendix E

SQL++ i-diff Propagation Rules

Table E.1. Rules for $V = \ggg^{C}_{\ddot{c}\mapsto(x,y)}(B)$

For Δ^{u}_{B}
$\Delta^{\mathrm{u}}_{V} = \Delta^{\mathrm{u}}_{B}$
$\Delta_{V} = \pi_{x,y} \ggg^{C}_{\ddot{z}\mapsto(x,y)} \bullet_{(c,diff)\mapsto z} \bullet_{(diff,\ddot{c})\mapsto c} (\Delta^{\mathrm{u}}_{B})$ Note: Top-level output i-diff type is same as the i-diff type of the element of $\ddot{c}$.

Table E.2. Rules for $V = \bullet_{(\ddot{x},\ddot{y})\mapsto z}(B)$

For Δ^{+}_{B}
$\Delta^{+}_{V} = \bullet_{(\ddot{x},\ddot{y})\mapsto z}(\Delta^{+}_{B})$
For Δ^{-}_{B}
if $\Delta^{-}_{B}.\ddot{x}$ exists $\quad \Delta^{-}_{V} = \bullet_{(\ddot{x},\ddot{y})\mapsto z}(\Delta^{-}_{B})$ else $\quad \Delta^{-}_{V} = \Delta^{-}_{B}$
For Δ^{u}_{B}
if $\Delta^{\mathrm{u}}_{B}.\ddot{x}.\mathtt{diff}.\ddot{y}$ exists $\quad \Delta^{\mathrm{u}}_{V} = \pi_{V} \bullet_{(\mathtt{diff},y)\mapsto y} \bullet_{(\ddot{x},\mathtt{diff})\mapsto \mathtt{diff}} (\Delta^{\mathrm{u}}_{B})$ else $\quad \Delta^{\mathrm{u}}_{V} = \Delta^{\mathrm{u}}_{B}$

Table E.3. Rules for $V = \sigma_{\ddot{c}}(B)$

For Δ_B^+
$\Delta_V^+ = \sigma_{\ddot{c}}(\Delta_B^+)$
For Δ_B^-
if $\{c_1, \ldots, c_n\} \subseteq \Delta_B^-$ $\quad \Delta_V^- = \sigma_{\ddot{c}}(\Delta_B^-)$ else $\quad \Delta_V^- = \Delta_B^-$
For Δ_B^u
$\Delta_V^u = \pi_B \sigma_{c^{pre}} \sigma_{c^{post}}$ $\quad \bullet_{(\ddot{c}_1,\texttt{pre})\mapsto\texttt{c\_1\_pre}} \cdots \bullet_{(\ddot{c}_n,\texttt{pre})\mapsto\texttt{c\_n\_pre}}$ $\quad \bullet_{(\ddot{c}_1,\texttt{post})\mapsto\texttt{c\_1\_post}} \cdots \bullet_{(\ddot{c}_n,\texttt{post})\mapsto\texttt{c\_n\_post}} (\Delta_B^u)$
For Δ_B^u if any of $c_1, \ldots, c_n$ has diff in Δ_B^u
$\Delta_V^+ = \pi_{\texttt{c\_1\_post}\mapsto\ddot{c}_1,\ldots,\texttt{c\_n\_post}\mapsto\ddot{c}_n}$ $\quad \sigma_{\neg c^{pre}} \sigma_{c^{post}}$ $\quad \bullet_{(\ddot{c}_1,\texttt{pre})\mapsto\texttt{c\_1\_pre}} \cdots \bullet_{(\ddot{c}_n,\texttt{pre})\mapsto\texttt{c\_n\_pre}}$ $\quad \bullet_{(\ddot{c}_1,\texttt{post})\mapsto\texttt{c\_1\_post}} \cdots \bullet_{(\ddot{c}_n,\texttt{post})\mapsto\texttt{c\_n\_post}} (\Delta_B^u))$
$\Delta_V^- = \pi_{\texttt{c\_1\_pre}\mapsto\ddot{c}_1,\ldots,\texttt{c\_n\_pre}\mapsto\ddot{c}_n}$ $\quad \sigma_{c^{pre}} \sigma_{\neg c^{post}}$ $\quad \bullet_{(\ddot{c}_1,\texttt{pre})\mapsto\texttt{c\_1\_pre}} \cdots \bullet_{(\ddot{c}_n,\texttt{pre})\mapsto\texttt{c\_n\_pre}}$ $\quad \bullet_{(\ddot{c}_1,\texttt{post})\mapsto\texttt{c\_1\_post}} \cdots \bullet_{(\ddot{c}_n,\texttt{post})\mapsto\texttt{c\_n\_post}} (\Delta_B^u))$
Note: Let $c_1, \ldots, c_n$ be variables mentioned in $\ddot{c}$. c^{pre} is c with $c_1, \ldots, c_n$ replaced by `c_1_pre`,...,`c_n_pre`. c^{post} is c with $c_1, \ldots, c_n$ replaced by `c_1_post`,...,`c_n_post`. Blue portions are skipped if the pre-state values are not available.

Table E.4. Rules for $V = \text{InnerCorrelate}_P(B)$

For Δ_B^+
$\Delta_V^+ = \text{InnerCorrelate}_P(\Delta_B^+)$
For Δ_B^-
$\Delta_V^- = \Delta_B^-$
For Δ_B^u
$\Delta_V^u = \Delta_B^u$
For each i-diff query $P_{\texttt{diff}}$ of P triggered by Δ_B^u if $P_{\texttt{diff}}$ produces update i-diffs $\quad \Delta_V^u = \pi_P \text{InnerCorrelate}_{P_{\texttt{diff}}}(\Delta_B^u)$ else if $P_{\texttt{diff}}$ produces delete i-diffs $\quad \Delta_V^- = \pi_P \text{InnerCorrelate}_{P_{\texttt{diff}}}(\Delta_B^u)$ else if $P_{\texttt{diff}}$ produces insert i-diffs $\quad \Delta_V^+ = \pi_V \text{InnerCorrelate}_{P_{\texttt{diff}}}(\Delta_B^u)$
Independent of Δ_B
For each i-diff query $P_{\texttt{diff}}$ of P if $P_{\texttt{diff}}$ produces update i-diffs $\quad \Delta_V^u = \pi_P \text{InnerCorrelate}_{P_{\texttt{diff}}}(B)$ else if $P_{\texttt{diff}}$ produces delete i-diffs $\quad \Delta_V^- = \pi_P \text{InnerCorrelate}_{P_{\texttt{diff}}}(B)$ else if $P_{\texttt{diff}}$ produces insert i-diffs $\quad \Delta_V^+ = \pi_V \text{InnerCorrelate}_{P_{\texttt{diff}}}(B)$
Note: π_P keeps only variables output by P. π_V keeps all variables in V.

Table E.5. Rules for $V = \alpha_{P \mapsto x}(B)$

For Δ_B^+
$\Delta_V^+ = \alpha_{P \mapsto x}(\Delta_B^+)$
For Δ_B^-
$\Delta_V^- = \Delta_B^-$
For Δ_B^u
$\Delta_V^u = \Delta_B^u$
For each i-diff query $P_{\texttt{diff}}$ of P triggered by Δ_B^u $\quad \Delta_V = \text{ConstructTuple}_{(\text{`diff'}:\texttt{t}) \mapsto x} \alpha_{P_{\texttt{diff}} \mapsto \texttt{t}}(\Delta_B^u)$
Independent of Δ_B
For each i-diff query $P_{\texttt{diff}}$ of P if $P_{\texttt{diff}}$ depends on data from B $\quad \Delta_V = \text{ConstructTuple}_{(\text{`diff'}:\texttt{t}) \mapsto x} \alpha_{P_{\texttt{diff}} \mapsto \texttt{t}}(B)$ else $\quad \Delta_V = \text{ConstructTuple}_{(\text{`diff'}:\texttt{t}) \mapsto x} \alpha_{P_{\texttt{diff}} \mapsto \texttt{t}}(\mathit{Ground})$

Table E.6. Rules for $V = \lambda_{(f,\, \ddot{x}_1 \ldots \ddot{x}_n) \mapsto y}(B)$

For Δ_B^+
$\Delta_V^+ = \lambda_{(f,\, \ddot{x}_1 \ldots \ddot{x}_n) \mapsto y}(\Delta_B^+)$
For Δ_B^-
$\Delta_V^- = \Delta_B^-$
For Δ_B^u
if none of $x_1, \ldots, x_n$ has diff in Δ_B^u $\quad \Delta_V^u = \Delta_B^u$ else $\quad \Delta_V^u = \pi_V\, \text{ConstructTuple}_{(\text{'pre'}:\text{pre},\, \text{'post'}:\text{post}) \mapsto y}$ $\quad\quad \lambda_{(f,\, \text{x\_1\_pre} \ldots \text{x\_n\_pre}) \mapsto \text{pre}} \lambda_{(f,\, \text{x\_1\_post} \ldots \text{x\_n\_post}) \mapsto \text{post}}$ $\quad\quad\quad \bullet_{(\ddot{x}_1, \text{pre}) \mapsto \text{x\_1\_pre}} \cdots \bullet_{(\ddot{x}_n, \text{pre}) \mapsto \text{x\_m\_pre}}$ $\quad\quad\quad \bullet_{(\ddot{x}_1, \text{post}) \mapsto \text{x\_1\_post}} \cdots \bullet_{(\ddot{x}_n, \text{post}) \mapsto \text{x\_m\_post}} (\Delta_B^u)$

Table E.7. Rules for $V = \pi_{x_1 \ldots x_n}(B)$

For Δ_B^+
$\Delta_V^+ = \pi_{x_1 \ldots x_n}(\Delta_B^+)$
For Δ_B^-
$\Delta_V^- = \pi_{\{x_1 \ldots x_n\} \cap \Delta_B^-}(\Delta_B^-)$
For Δ_B^u
$\Delta_V^u = \pi_{\{x_1 \ldots x_n\} \cap \Delta_B^u}(\Delta_B^u)$

Table E.8. Rules for $V = \times(B^l, B^r)$

For $\Delta_{B^l}^+$
$\Delta_V^+ = \times(\Delta_{B^l}^+, B^r)$
For $\Delta_{B^l}^-$
$\Delta_V^- = \Delta_{B^l}^-$
For $\Delta_{B^l}^u$
$\Delta_V^u = \Delta_{B^l}^u$
Note: Δ_{B^r} is symmetric.

Table E.9. Incomplete rules for $V = B^l \bar{\bowtie}_p B^r$

For $\Delta^+_{B^l}$
$\Delta^+_V = \Delta^+_{B^l} \bar{\bowtie}_p B^r$
For $\Delta^-_{B^l}$
$\Delta^-_V = \Delta^-_{B^l}$
For Δ^u_B
$\Delta^u_V = \Delta^u_B$
Note: The above rules are incomplete because insertion and deletion of V tuples that have no match from B^r are outside the scope of this work. Note: Rules for $\bar{\bowtie}$ are symmetric.

Table E.10. Rules for $V = \text{ConstructTuple}_{(x_1:\ddot{y}_1 \ldots x_n:\ddot{y}_n)\mapsto z}(B)$

For Δ^+_B
$\Delta^+_V = \text{ConstructTuple}_{(x_1:\ddot{y}_1 \ldots x_n:\ddot{y}_n)\mapsto z}(\Delta^+_B)$
For Δ^-_B
if $\ddot{y}_1, \ldots, \ddot{y}_n$ exists in Δ^-_B $\quad \Delta^-_V = \text{ConstructTuple}_{(x_1:\ddot{y}_1 \ldots x_n:\ddot{y}_n)\mapsto z}(\Delta^-_B)$ else $\quad \Delta^-_V = \Delta^-_B$
For Δ^u_B
Let $x_{w_1} \ldots x_{w_m}$ be the subset of $x_1 \ldots x_n$ that are in Δ^u_B $\Delta^u_V = \pi_V$ $\quad \text{ConstructTuple}_{(\text{`pre`}:\texttt{pre}, \text{`post`}:\texttt{post}, \text{`diff`}:\texttt{diff})\mapsto z}$ $\quad \text{ConstructTuple}_{(x_{w_1}:\ddot{y}_{w_1} \ldots x_{w_m}:\ddot{y}_{w_m})\mapsto \texttt{diff}}$ $\quad \text{ConstructTuple}_{(x_1:y_1^{pre} \ldots x_n:y_n^{pre})\mapsto \texttt{pre}}$ $\quad \text{ConstructTuple}_{(x_1:y_1^{post} \ldots x_n:y_n^{post})\mapsto \texttt{post}}$ $\quad \bullet_{(y_1,\texttt{pre})\mapsto y_1^{pre}} \cdots \bullet_{(y_n,\texttt{pre})\mapsto y_n^{pre}}$ $\quad \bullet_{(y_1,\texttt{post})\mapsto y_1^{post}} \cdots \bullet_{(y_n,\texttt{post})\mapsto y_n^{post}} (\Delta^u_B)$
Note: Assume all the $\ddot{y}$-s are variables. Note: Both pre- and post-states are optional in the output i-diff.

Table E.11. Rules for $V = \gamma_{(x_1 \mapsto y_1, \ldots, x_n \mapsto y_n), g}(B)$

For Δ_B of any type
$\Delta_V^{\mathrm{u}} = \pi_V \, \mathrm{ConstructTuple}_{(\texttt{diff:z}) \mapsto g}$ $\quad \mathrm{ConstructTuple}_{(\texttt{post:}y_1^{post}) \mapsto y_1}$ $\quad \ldots$ $\quad \mathrm{ConstructTuple}_{(\texttt{post:}y_n^{post}) \mapsto y_n}$ $\quad \gamma_{(x_1^{post} \mapsto y_1^{post}, \ldots, x_n^{post} \mapsto y_n^{post}), z}$ $\quad \bullet_{(x_1, \texttt{post}) \mapsto x_1^{post}} \ldots \bullet_{(x_n, \texttt{post}) \mapsto x_n^{post}} (\Delta_B)$ Note: The type of nested i-diff g is the same as the top-level i-diff Δ_B.
Note: Does not handle group creation/deletion. Note: If an x does not appear in Δ_B it is skipped in the group-by diff query (i.e., a group in the i-diff can match multiple groups in data).

Table E.12. Rules for $V = \lambda_{(sum, \ddot{x}) \mapsto y}(B)$

For Δ_B^{u} that updates elements of $\ddot{x}$
$\Delta_1 = \alpha_{P_1 \mapsto x_\Delta^1} \Delta_B^{\mathrm{u}}$ $P_1 : \mathrm{ReturnSingle}_w \, \lambda_{(-, post, pre) \mapsto w}$ $\quad \bullet_{(w, post) \mapsto post} \bullet_{(w, pre) \mapsto pre}$ $\quad\quad \ggg^{C}_{\ddot{x} \mapsto w} \mathit{Ground}$
For Δ_B^{u} that deletes elements of $\ddot{x}$
$\Delta_2 = \alpha_{P_2 \mapsto x_\Delta^2} \Delta_B^{\mathrm{u}}$ $P_2 : \mathrm{ReturnSingle}_w \, \lambda_{(-, 0, pre) \mapsto w}$ $\quad \bullet_{(w, pre) \mapsto pre} \ggg^{C}_{\ddot{x} \mapsto w} \mathit{Ground}$
For Δ_B^{u} that inserts elements into $\ddot{x}$
$\Delta_3 = \alpha_{P_3 \mapsto x_\Delta^3} \Delta_B^{\mathrm{u}}$ $P_3 : \mathrm{ReturnSingle}_{post}$ $\quad \bullet_{(w, post) \mapsto post} \ggg^{C}_{\ddot{x} \mapsto w} \mathit{Ground}$
For converting Δ to output update i-diffs
Happens after all $\Delta_1, \Delta_2, \Delta_3$ are computed. $\Delta_V^{\mathrm{u}} = \mathrm{ConstructTuple}_{(\texttt{post:}post, \texttt{pre:}y) \mapsto y}$ $\quad \lambda_{(+, y, y_\Delta) \mapsto post} \lambda_{(sum, x_\Delta) \mapsto y_\Delta} \alpha_{P_u \mapsto x_\Delta}$ $\quad\quad \mathrm{ProvJoin}(B, \mathrm{ProvOuterJoin}(\Delta_1, \Delta_2, \Delta_3))$ $P_u : x_\Delta^1 \hat{\cup} x_\Delta^2 \hat{\cup} x_\Delta^3$

(Do not handle group creation/deletion)

Table E.13. Rules for $V = \lambda_{(count,\ddot{x})\mapsto y}(B)$

For Δ_B^{u} that deletes elements of $\ddot{x}$
$\Delta_2 = \alpha_{P_2\mapsto x_\Delta^2}\Delta_B^{\mathrm{u}}$ P_2 : ReturnSingle$_c\,\pi_{-1\mapsto c} \ggg^{C}_{\ddot{x}\mapsto w}$ *Ground*
For Δ_B^{u} that inserts elements into $\ddot{x}$
$\Delta_3 = \alpha_{P_3\mapsto x_\Delta^3}\Delta_B^{\mathrm{u}}$ P_3 : ReturnSingle$_c\,\pi_{1\mapsto c} \ggg^{C}_{\ddot{x}\mapsto w}$ *Ground*
For converting Δ to output update i-diffs
Happens after all $\Delta_1,\Delta_2,\Delta_3$ are computed. $\Delta_V^{\mathrm{u}} =$ ConstructTuple$_{(\mathtt{post}:post,\mathtt{pre}:y)\mapsto y}$ $\lambda_{(+,y,y_\Delta)\mapsto post}\lambda_{(sum,x_\Delta)\mapsto y_\Delta}\alpha_{P_u\mapsto x_\Delta}$ ProvJoin$(B,$ProvOuterJoin$(\Delta_1,\Delta_2,\Delta_3))$ $P_u : x_\Delta^1 \hat{\cup} x_\Delta^2 \hat{\cup} x_\Delta^3$

(Do not handle group creation/deletion)

Table E.14. Rules for $V = \lambda_{(avg,\ddot{x})\mapsto y}(B)$

Operator caches: $Cache_{sum}$, $Cache_{count}$
Cache maintenance rules: For $\Delta^u_{Cache_{sum}}$: Use rules of $\lambda_{(sum,\ddot{x})\mapsto y}(B)$ (Table E.12) For $\Delta^u_{Cache_{count}}$: Use rules of $\lambda_{(count,\ddot{x})\mapsto y}(B)$ (Table E.13)
i-diff propagation rules: $\Delta_V^u =$ ConstructTuple$_{(\mathtt{post}:y_{post},\mathtt{pre}:y_{pre})\mapsto y}$ $\lambda_{(/,y^{sum}_{pre},y^{count}_{pre})\mapsto y_{pre}}\lambda_{(/,y^{sum}_{post},y^{count}_{post})\mapsto y_{post}}($ $\bullet_{(y,post)\mapsto y^{count}_{post}}\bullet_{(y,pre)\mapsto y^{count}_{pre}}\Delta^u_{Cache_{count}}\ \hat{\bowtie}$ $\bullet_{(y,post)\mapsto y^{sum}_{post}}\bullet_{(y,pre)\mapsto y^{sum}_{pre}}\Delta^u_{Cache_{sum}})$

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[4] Asp.net, 2009. http://www.asp.net/.

[5] Backbase enterprise ajax framework, 2009. http://www.backbase.com/products/enterprise-ajax/.

[6] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized views selection in a multidimensional database. In *VLDB*, 1997.

[7] Andreas Behrend and Thomas Jörg. Optimized incremental etl jobs for maintaining data warehouses. In *IDEAS*, pages 216–224, 2010.

[8] Henrik Björklund, Wouter Gelade, and Wim Martens. Incremental xpath evaluation. *ACM Trans. Database Syst.*, 35(4):29, 2010.

[9] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[10] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[11] Angela Bonifati, Martin Hugh Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. *ACM Trans. Database Syst.*, 38(3):14, 2013.

[12] Business process modeling language, 2009. http://www.bpmi.org.

[13] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[14] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[15] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[16] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[17] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. In *PODS*, pages 71–82, 2004.

[18] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *ER*, 2003.

[19] Echo web framework, 2009. http://echo.nextapp.com/site/.

[20] Echo web framework, 2009. http://echo.nextapp.com/site/.

[21] Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative specification of web sites with strudel. *VLDB J.*, 9(1):38–55, 2000.

[22] Forward web application framework, 2012. http://forward.ucsd.edu.

[23] J. Nathan Foster, Ravi Konuru, Jérôme Siméon, and Lionel Villard. An algebraic approach to view maintenance for xquery. In *PLAN-X*, 2008.

[24] Yupeng Fu, Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao, and Yannis Papakonstantinou. Ajax-based report pages as incrementally rendered views. In *SIGMOD Conference*, 2010.

[25] Google web toolkit, 2009. http://code.google.com/webtoolkit/.

[26] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[27] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.

[28] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 1995.

[29] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166. ACM Press, 1993.

[30] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[31] Hao He, Junyi Xie, Jun Yang, and Hai Yu. Asymmetric batch incremental view maintenance. In *ICDE*, pages 106–117, 2005.

[32] Icefaces, 2009. http://www.icefaces.org/main/home/.

[33] Yelp Inc. Yelp open dataset. https://www.yelp.com/dataset.

[34] Java swing, 2009. http://java.sun.com/javase/6/docs/technotes/guides/swing/.

[35] Javaserver pages standard tag library, 2010. http://java.sun.com/products/jsp/jstl/.

[36] Thomas Jörg and Stefan Deßloch. View maintenance using partial deltas. In *Datenbanksysteme für Business, Technologie und Web*, 2011.

[37] jquery javascript library, 2009. http://jquery.com/.

[38] Harumi A. Kuno and Goetz Graefe. Deferred maintenance of indexes and of materialized views. In *DNIS*, pages 312–323, 2011.

[39] Per-Åke Larson and Jingren Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, pages 56–65, 2007.

[40] Samek Miro. Who moved my state? *Dr. Dobb's*, 2003.

[41] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[42] Kian Win Ong and Yannis Papakonstantinou. The SQL++ query language: Configurable, unifying and semi-structured, 2015. https://arxiv.org/abs/1405.3631.

[43] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, pages 802–813, 2002.

[44] Yannis Papakonstantinou, Kian Win Ong, and Romain Vernoux. SQL++: Semi-structured data models and query capabilities in the nosql and newsql era, 2015.

[45] Prototype javascript framework, 2009. http://prototypejs.org/.

[46] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[47] Dallan Quass. Maintenance expressions for views with aggregation. In *VIEWS*, pages 110–118, 1996.

[48] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *4th International Conference on Parallel and Distributed Information Systems*, 1996.

[49] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pages 447–458, 1996.

[50] Arsany Sawires, Jun'ichi Tatemura, Oliver Po, Divyakant Agrawal, Amr El Abbadi, and K. Selçuk Candan. Maintaining xpath views in loosely coupled systems. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 583–594, 2006.

[51] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, 1998.

[52] The dojo toolkit, 2009. http://www.dojotoolkit.org/.

[53] The web modeling language, 2009. http://www.webml.org/.

[54] Web services description language (wsdl) 1.1, 2009. http://www.w3.org/TR/wsdl.

[55] Wikipedia. Asp.net, 2009. Accessed Nov 04 2009. http://en.wikipedia.org/w/index.php?title=ASP.NET&oldid=323456166.

[56] Fan Xia, Ye Li, Chengcheng Yu, Haixin Ma, and Weining Qian. BSMA: A benchmark for analytical queries over social media data. *PVLDB*, 7(13), 2014.

[57] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan J. Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW*, pages 341–350, 2007.

[58] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, page 32, 2006.

[59] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[60] Yui library, 2009. http://developer.yahoo.com/yui/.

[61] Zk direct ria, 2009. http://www.zkoss.org/.