Interruption-Sensitive Empty Result Feedback: Rethinking the Visual Query Feedback Paradigm for Semistructured Data

Sourav S Bhowmick
School of Computer Engg.
Nanyang Technological
University, Singapore
assourav@ntu.edu.sg

Curtis Dyreson
Department of Computer Sc.
Utah State University, USA
curtis.dyreson@usu.edu

Byron Choi
Department of Computer Sc.
Hong Kong Baptist University,
Hong Kong, China
choi@hkbu.edu.hk

Min-Hwee Ang
School of Computer Engg.
Nanyang Technological
University, Singapore
Y070005@e.ntu.edu.sg

ABSTRACT

The usability of visual querying schemes for tree and graph-structured data can be greatly enhanced by providing feedback during query construction, but feedback at inopportune times can hamper query construction. In this paper, we rethink the traditional way of providing feedback. We describe a novel vision of *interruption-sensitive* query feedback where relevant notifications are delivered *quickly but at an appropriate moment* when the mental workload of the user is low. Though we focus on one class of query feedback, namely *empty result detection*, where a user is notified when a partially constructed visual query yields an empty result, our new paradigm is applicable to other kinds of feedback. We present a framework called *i*SERF that bridges the classical database problem of empty-result detection with intelligent notification management from the domains of HCI and psychology. Instead of immediate notification, *i*SERF considers the structure of query formulation tasks and *breakpoints* when reasoning about when to notify the user. We present an HCI-inspired model to quantify the performance bounds that *i*SERF must abide by for checking for an empty result in order to ensure interruption-sensitive notification at *optimal* breakpoints. We implement this framework in the context of visual XML query formulation and highlight its effectiveness empirically.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query processing

General Terms

Algorithms, Experimentation, Human Factors, Performance

CIKM'15, October 19–23, 2015, Melbourne, Australia.

DOI: http://dx.doi.org/10.1145/2806416.2806432.

Keywords

Visual querying; query formulation; graphs; XML; query feedback; empty result detection; interruption; notifications; breakpoints

1. INTRODUCTION

Formulating queries over semistructured databases (*e.g.,* XML, JSON, graphs) using database query languages (*e.g.,* XQuery, SPARQL, Cypher) often demands considerable cognitive effort from users and requires "programming" skill that is at least comparable to SQL. Providing a visual query interface (GUI) is a popular approach to making query construction user-friendly [1, 6, 7, 10, 30]. Typically, these interfaces enable visual query formulation involving a sequence of tasks ranging from *primitive* operations such as pointing and clicking a mouse button to higher-level tasks such as selection of a menu item.

1.1 Interactive Visual Query Feedback

A user-friendly visual query system is expected to interactively guide users into constructing correct queries. A non-exhaustive list of the kinds of guidance such a system may employ is given below.

- *Syntactic and semantic help.* A visual query system can detect and notify users when there is a syntax or semantic problem in a query (*e.g.,* a missing quote on a string or a wrong type passed to a function). A system could even suggest better alternative syntax. These techniques can greatly enhance users' ability to formulate syntactically correct queries visually without resorting to memorizing various syntactic flavors of a query language.
- *Empty result feedback.* When a user specifies *"Leningrad"* instead of *"St. Petersburg"* (Leningrad is the former name of St. Petersburg) as a value predicate on an attribute `city` in a query on the *Mondial* dataset[1], the query returns an empty result. It is important to detect such scenarios during query construction and alert a user in a timely fashion so that she can undertake appropriate remedial action(s).
- *Long-running query feedback.* A recent study [29] shows that a graph query that runs fast may slow down significantly after a slight modification to the query's structure. Unexpected query behaviors like this can confuse and annoy users

[1] http://datahub.io/dataset/mondial.

of visual query systems, especially in the context of exploratory search. It would be better if the system could warn a user about a potentially long-running query fragment during query formulation.

- *Query fragments suggestion.* Given a partially-constructed visual XML or subgraph query, it is useful to suggest top-k possible query fragments that the user may potentially add to her query in the subsequent step. Such suggestions can enhance user experience by greatly reducing the query formulation time.

There are two common threads in the above scenarios. First, without the aid of a visual query feedback mechanism, users would be unaware of these cases, since each case requires comprehensive knowledge of the underlying data or query language. Feedback about various problems encountered during query construction as well as possible solutions is critical to enhancing the usability of a visual querying system. Second, as query conditions in a visual querying environment are typically constructed iteratively, it is often critical to detect and notify the aforementioned issues *opportunely*. It is ineffective to provide feedback at the *end* of query formulation. For instance, consider the empty result problem. It is ineffective to provide feedback *after* the query formulation as a user may have wasted her time and effort in formulating additional query conditions. Similarly, it is ineffective if the visual querying scheme fails to alert the user opportunely when a "long-running" query fragment is detected. Such opportune notification is also important in the context of query fragment suggestion as it is not beneficial if suggestions are made after the query has been visually constructed.

1.2 Importance of Intelligent Notifications

Feedback during visual query construction can be intuitively modeled as an alert or notification for a secondary task (*e.g.,* handling the empty result problem) when a user is working on a primary task (query formulation). It is desirable to build an *intelligent notification system* that can efficiently *detect* issues and *notify* users effectively. Such notification systems play a pivotal role in visual query systems, as well as in many software systems. Research in HCI demonstrates that there are numerous benefits of notifications such as quick availability of important and relevant information (*e.g.,* long-running query fragment, empty answer) as well as access to nearly instantaneous communication [18].

A notification can be *intrusive* or *non-intrusive* [19]. *Non-intrusive* notifications are usually less disruptive to user activities as they are often presented in the periphery of a user's attention. In general, non-urgent information can be presented to a user in a non-intrusive fashion. *Intrusive* notifications are for urgent messages. They demand immediate action from the user and hence they need to grab a user's attention. For instance, an empty result problem that arises during visual query formulation might generate an intrusive notification since an immediate response is needed by the user.

1.3 Interruption - The Other Side of the Coin

Notifications, however, come with a cost: they *interrupt* a primary task (*i.e.,* query formulation). This is because notifications divert attention [15, 25]. Intuitively, an *interruption* is a distraction that causes one to stop a scheduled task to respond to a stimulus. Many studies in the cognitive psychology and HCI communities have reported that interrupting users engaged in tasks by delivering notifications inopportunely can negatively impact task completion time, lead to more errors, and increase user frustration [3, 5, 9, 12, 15, 16, 21, 22]. In summary, three key findings are reported in these studies. First, interruptions slow and degrade performance on the primary task. In particular, interruptions lead to *resumption lag*, which is the additional time needed to resume the primary task after interruption [22]. Second, the timing of the interruptions can adversely impact performance. It has been observed that notifications occurring at points of higher mental workload are more disruptive in nature compared to those that occur during lower mental workload [3, 9, 15, 16]. The former may lead not only to larger resumption lags but also increase frustration. Third, interruptions with similar content with the main task could be quite disruptive even if they are very short [12].

For instance, suppose a user is notified intrusively (*e.g.,* invoking a pop-up dialog box, highlighting a condition) of an empty result (due to previously formulated condition) when she is dragging an attribute from one panel and preparing to drop it in to another panel to construct a new condition. This interruption may frustrate her as mental resources allocated for the current task are disrupted. Such inopportune, intrusive feedback adversely affects the usability of the system.

It is worth noting that recent research [28] in the HCI community has demonstrated that *acceptability* of a notification is essentially a tradeoff between the cost of interrupting the user's activities and the value of receiving the notification message. Specifically, acceptability of low-urgency and medium-urgency messages (*e.g.,* query fragment suggestions) improves when presented in a non-intrusive manner. This is because this type of notification is expected to be less disruptive of the primary task (*e.g.,* visual query formulation) in comparison to its intrusive counterpart. Furthermore, recent research has suggested that users typically delay processing of a non-intrusive interruption, when it is sent to them at points of higher mental workload, until they have reached a point of lower mental workload [26]. On the other hand, acceptability of high-urgency messages (*e.g.,* empty results, syntax error) is expected to be low when presented non-intrusively as immediate user response is expected for such notifications. In other words, *a user would typically like to see urgent messages in an immediate and intrusive way* (*e.g.,* modal boxes, highlighting query conditions).

1.4 Interruption-sensitive Query Feedback

Classical visual query feedback strategies are "interruption-insensitive". These strategies have historically focused on speed and scalability, often devoting very little attention to the cognitive aspect of a solution. Hence, we need to make our visual query feedback mechanism "cognitive-aware" by devising models and techniques to deliver a notification (especially an intrusive one) *quickly but at an appropriate moment* when the mental workload of the user is minimal. Detecting such an opportune moment should be transparent to the user and must not seek explicit input from her.

In this paper, we rethink the classical interruption-insensitive query feedback paradigm and lay down the vision of *interruption-sensitive* visual query feedback by taking the first step to mitigate the problem of inopportune interruptions made by *intrusive*[2] query feedback notifications. Specifically, we focus on *one class of visual query feedback problem* to illustrate this novel paradigm, namely ***empty result (ER) feedback***[3], where a user is notified when a partially constructed visual query yields an empty result. Note that an empty result is not always undesirable to an end user especially in an exploratory querying environment. However, whenever a visual query fragment returns an empty result, it is particularly important to notify the user about[4] the following: (a) The ER problem re-

[2] We focus on intrusive notification in this paper as this type of notification delivers messages that are important to the user and requires immediate attention.

[3] We chose the ER feedback as a representative problem as it has consistently received attention from the database community over decades [8, 17, 20, 23].

[4] The notification can be a single message.

sulted from the constructed query fragment *opportunely* so that she can avoid wasting time and effort in continuing constructing new conditions; (b) Identify the constructed query condition(s) that is responsible for an empty result; and (c) Optionally, suggest a query modification to mitigate the ER problem.

Hence our proposed paradigm *bridges the classical visual query feedback problem with intelligent notification management from the domains of* HCI *and cognitive psychology*. We propose a novel psychology and HCI-inspired model for delivering intrusive notification of an empty result for visual queries, the theory based on this model, and a framework for detecting empty results efficiently and delivering notifications at opportune times. Although as we shall see later, our proposed paradigm can be realized on visual query interfaces for tree or graph-structured databases, to demonstrate its effectiveness we realize it on top of a visual XML query formulation framework. Specifically, we make the following contributions.

- We present a psychology-inspired notification scheme that exploits a *defer-to-breakpoint* strategy [14–16] for the effective delivery of intrusive notification to the user when a visual query fragment returns an empty result (Section 3).
- We present an HCI-inspired extensible quantitative model for estimating the *optimal notification time* available for empty-result detection so that the defer-to-breakpoint notification scheme can be effectively realized. The model takes into consideration the time to undertake various actions in order to construct a visual query condition (Section 4).
- We propose a novel and generic framework called *i*SERF (**I**nterruption-**S**ensitive **E**mpty **R**esults noti**F**ication), grounded on well-founded principles from HCI and cognitive psychology, to efficiently detect an empty result during visual query formulation and intrusively notify users along with the query condition(s) that is responsible for this problem. Note that here we do not focus on providing to the user suggestions on how to modify or relax the visual query to get a non-empty result as it is orthogonal to our framework (we shall elaborate on it further in Section 6). Additionally, *i*SERF is *not* designed with the explicit aim to minimize query construction time, which is an orthogonal problem. Instead, the broad goal of *i*SERF is to *improve each user's experience during visual query formulation by reducing the adverse effects of intrusive notifications* (Section 5).
- We implement *i*SERF on a visual XML querying environment to demonstrate its effectiveness in realizing the interruption-sensitive visual query feedback paradigm (Section 6).
- We conduct an empirical study on real XML data with real users to demonstrate the effectiveness of *i*SERF (Section 7).

2. BACKGROUND

We begin by laying out the generic structure of a visual query interface for tree or graph-structured data. Then, we formally define the *empty result feedback* problem. Lastly, we discuss related research in this arena.

2.1 Visual Query Interfaces for Trees or Graphs

Most visual query interfaces for tree or graph-structured data are comprised of *at least* three key panels: (a) a *Schema Panel* to display the structural summary (*e.g.,* XML schema or *DataGuide* [13]) or distinct set of node and edge labels of the underlying tree or graph data, respectively, (b) a *Query Panel* for iteratively constructing the query conditions (value and structural constraints) graphically, and (c) a *Results Panel* (Panel 4) that displays the query results.

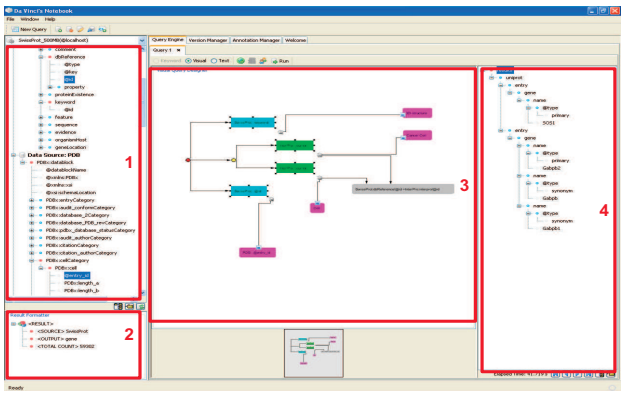


Figure 1: A visual interface for formulating XML queries.

EXAMPLE 1. Figure 1 depicts a screen dump of a visual interface for querying XML data [31]. Observe that Panels 1, 3, and 4 in this GUI correspond to the *Schema*, *Query*, and *Results* Panel, respectively. In addition it has an *Output Panel* (Panel 2) that displays the item(s) to be returned by the query. Note that this panel may not appear in other interfaces for querying XML data as different interfaces may visually specify the result nodes differently. A new XML query can be formulated using the interface as follows.

1. Specify the output item(s) by dragging selected item(s) from Panel 1 and dropping it to Panel 2.
2. Move the mouse pointer to Panel 1.
3. Scan and select an item in Panel 1.
4. Drag the item to Panel 3 and drop it. Each such action represents formulation of a non-join predicate (condition).
5. A *Dialog Box* will automatically appear for users to fill (optionally) in the value predicates, comparison operators, aggregate functions, etc.
6. Combine two or more conditions in Panel 3 using AND/OR connectives. Note that in this step one may also build join predicates if necessary.
7. Repeat Steps 2–6 for each new condition.
8. Execute the query by clicking on the Run icon. Panel 4 displays the query results.

Very similar visual query interfaces for graph queries are described elsewhere [6, 30]. Hence we treat tree and graph query interfaces as the same for the purposes of this paper. □

2.2 Empty Result (ER) Feedback Problem

Given a visual query Q (tree or graph query), we denote the result set of Q as $R(Q)$. A visual query fragment Q' of Q consists of a subset of conditions $C' \subseteq C$ of Q. Then the result set satisfying C' is denoted as $R(C')$. A visual query fragment Q' (query Q) returns an *empty result iff* $R(C') = \emptyset$ ($R(Q) = \emptyset$). In this paper, we present a framework for (a) *detecting* if $R(C') = \emptyset$ for a partially-constructed visual query Q'; (b) identifying the condition(s) in Q' responsible for an empty result, and (c) notifying the user intrusively at an *opportune time* so that the "interruption cost" is minimized. Observe that traditional ER feedback techniques typically focus on (a) and (b) but not (c).

2.3 Related Work

To the best of our knowledge, none of the state-of-the-art visual querying schemes [1, 6, 7, 10, 30] are interruption-sensitive. In [17], an empty-result detection method is proposed for SQL queries. The

key idea is to reuse the evaluation results from prior empty-result queries by leveraging *lowest-level* query fragments from the query plans that lead to an empty result. In our proposed paradigm, we operate in a visual querying environment where the entire query is available only at the end of query formulation. Hence, the aforementioned technique is inapplicable here. Importantly, *i*SERF supports interruption-sensitive notification which is absent from [17].

Research in non-interactive [8] and interactive [20, 23] query relaxation solutions propose solutions to the empty-result problem. However, none of these efforts focus on opportune notification delivery of the relaxation solutions. In fact, the work reported in this paper complements these efforts as it automatically determines the opportune time to notify users about relaxation suggestions.

Research in cognitive psychology and HCI has investigated intelligent notification management [5, 9, 15, 21, 24], focusing on visual tasks such as document editing, diagram editing, image manipulation, and programming. However, prior to our work, it has not been studied in the context of database query feedback.

3. DEFER-TO-BREAKPOINT-BASED INTRUSIVE NOTIFICATION

The HCI community has shown that the negative effects of interruption (recall from Section 1.3) can be mitigated by deferring interruptions until more opportune moments in a task sequence [5, 15, 16]. The argument being that when a user completes a task, mental resources allocated to perform the task are released, momentarily reducing workload before the cycle of allocation and deallocation occurs again in the next task. One particular approach for deferring interruptions is to schedule them at *breakpoints*.

Defer-to-breakpoint strategy. A *breakpoint* represents the moment of transition between two observable, meaningful units of task execution, and reflects a change in perception or action [24]. Recent research in the HCI community identified three granularities (types) of breakpoints during interactive tasks - *Coarse*, *Medium*, and *Fine* [14]. Coarse exists between the largest units while Fine exists between the smallest. For example, consider the task of editing documents. Fine may be switching paragraphs, Medium may be switching documents, and Coarse may be switching to an activity other than editing (*e.g.,* checking emails).

Iqbal and Bailey have recently shown that the best moment to interrupt a user is on breakpoints between tasks [15]. That is, to defer the notification to appear at the next breakpoint detected in the user's task-sequence. *We shall adopt this defer-to-breakpoint-based strategy for interrupting query formulation tasks.*

Our decision to choose the above strategy is bolstered by the following interesting results demonstrated by the HCI community [5, 15]. First, the *interruption cost* is reduced and users are less frustrated when intrusive notifications are scheduled to occur at breakpoints rather than when delivered immediately. Second, users prefer having notifications scheduled at breakpoints and they react faster to notifications that were scheduled at breakpoints. Third, applications that generate notifications that are relevant to the user's ongoing activity should request that they be delivered at Medium or Fine breakpoints. This ensures that notifications are delivered when they have most utility, and are least disruptive. In contrast, notifications of general interest should be delivered at Coarse breakpoints.

Generic Breakpoints in Visual Query Formulation. Reconsider the generic structure of the GUIs for tree or graph query construction tasks as described in Section 2.1. The different types of breakpoints for these tasks are shown in Table 1. Observe that for each of the tasks related to Medium and Fine breakpoints, mental resources allocated to perform the task are released at the end of the task, thus momentarily reducing the workload. For instance, consider the task of selecting an item or node label in the *Schema Panel*. The mental resource of a user is allocated to scanning it and selecting an item/label. Upon completion the resource is released which reduces the mental workload. The mental resource allocation cycle begins again when the subsequent task of dragging the selected item/label to the *Query Panel* begins.

Type	Tasks for Tree Queries	Tasks for Graph Queries
Coarse	**CB1:** User switch to another application (*e.g.,* email); **CB2:** Minimize query window.	Same as tree queries.
Medium	**MB1:** Clicking a menu item to initiate creation of a new query; **MB2:** Switching to another query.	Same as tree queries.
Fine	**FB1:** Move mouse to *Schema Panel* to search items; **FB2:** Selection of an item in the *Schema Panel*; **FB3:** Dragging an item from *Schema Panel* to *Query Panel*; **FB4:** Combining/joining query conditions; **FB5:** Clicking OK on any Dialog box related to predicates;**FB6:** Click on the Run icon.	**FB1:** Move mouse to *Schema Panel* to search node labels; **FB2:** Selection of a label in the *Schema Panel*; **FB3:** Dragging a label from *Schema Panel* to *Query Panel* for query node creation, **FB4:** Connecting two nodes with an edge; **FB5:** Clicking OK any Dialog box for node/edge predicates; **FB6:** Click on the Run icon.

Table 1: Breakpoints for visual query formulation.

EXAMPLE 2. Reconsider the steps for formulating XML queries using the GUI in Example 1. The Fine breakpoints according to Table 1 are at the end of Steps 2, 3, 4, 5, 6, and 8. In addition, there is another Fine breakpoint at the end of Step 1, which is specific to this interface. □

Observe that for a given query there is only one Medium breakpoint. Hence, in the subsequent sections we shall combine Medium and Fine breakpoints as a single type of breakpoint. Specifically, *our objective is to ensure delivery of intrusive notifications at Fine breakpoints*.

4. MODELING NOTIFICATION TIME

We now present a quantitative model for estimating the *optimal notification time* (ONT) so that the defer-to-breakpoint notification scheme can be effectively realized for the query feedback problem. We first discuss it w.r.t the ER problem and then highlight how it can also be used to model other types of query feedback.

4.1 Optimal Notification Time

Let C' be a non-empty set of currently constructed conditions representing a twig pattern or a subgraph pattern and $R(C') \neq \emptyset$. Let C_{new} be a new condition (*i.e.,* a new edge in a subgraph query fragment, a new XPath condition in an XQuery fragment) drawn by a user (drawn after the construction of C'). For each C_{new}, our proposed framework takes two key steps. First, it checks whether $R(C' \cup C_{new}) = \emptyset$. If so then the query formulated thus far does not produce any results and the next available breakpoint to deliver the empty-result notification is found. Clearly, efficient checking for an empty result is pivotal as breakpoint selection can only proceed after detection. Consequently, inefficient execution of the first step can result in inopportune notification delivery, which may increase user frustration. Ideally, we should be able to deliver empty-result notification *before* the construction of the succeeding condition C_{next} (*e.g.,* the next edge in a subgraph query fragment) is finished. In this situation, a user does not need to waste her time and effort in constructing additional conditions before realizing that the query produces an empty result. Hence, notifications should be delivered at fine breakpoints FB1 and FB2 (Table 1). We refer to these breakpoints as *optimal breakpoints*. Observe that construction of C_{next} is already completed after FB3.

Given the most recent constructed condition C_{new} and a set of previously constructed conditions C', the ONT, denoted as T_{ont}, refers to the amount of time available for checking if $R(C_{new} \cup C') = \emptyset$, so that notification can be delivered at optimal breakpoints. We now present an HCI-inspired quantitative model to estimate this time.

Given C', adding a new condition C_{new} in a tree or graph query involves tasks related to the breakpoints FB1–FB5 in Table 1. Let us now refer to the times taken to complete tasks that end with fine breakpoints FB1, FB2, and FB3 as *movement time* (denoted by T_m), *selection time* (T_s), and *drag time* (T_d), respectively. Then, T_{ont} can be bounded by the following equation.

$$0 < T_{ont} < T_m + T_s \tag{1}$$

That is, as long as the check for an empty result of $(C' \cup C_{new})$ can be finished before the breakpoint FB2 of C_{next}, notification can be easily delivered at an optimal breakpoint. Observe that we did not include T_d in the right-hand side of the above equation. When a user starts dragging a selected item or label, C_{next} is in the process of materialization. Hence, the notification related to the conditions $(C' \cup C_{new})$ will be displayed only when she drops C_{next} on the *Query Panel* (breakpoint FB3). This may confuse the user as the notification related to C_{new} appears only after the creation of C_{next}. Additionally, she may have wasted her time in constructing C_{new}. Hence, Equation 1 enforces a tighter bound on T_{ont}. Note that the values of T_m and T_s vary with end users. For a given GUI, how can we theoretically quantify $(T_m + T_s)$? We now address this question.

4.2 Estimating Movement and Selection Times

Estimating movement time T_m. Reconsider the task ending with breakpoint FB1. It involves acquisition of a target from the *Schema Panel* at a distance D from the mouse cursor which is in the *Query Panel*. Note that typically the *Schema Panel* is a rectangular two-dimensional target. Consequently, the item selection is constrained by both the width and height of the panel and the cursor must travel along a two-dimensional vector to it. Hence, we adopt the model proposed by Accot and Zhai [2] that focuses on acquiring targets having rectangular, square, or circular shapes. The movement time T_m is quantified as follows.

$$T_m = a + b \log_2 \left(\sqrt{\left(\frac{D}{W}\right)^2 + \eta \left(\frac{D}{H}\right)^2} + 1 \right) \tag{2}$$

where D is the *Schema Panel*'s distance to the cursor, H and W denote *Schema Panel*'s height and width, respectively. The parameter a varies approximately in the range of [-50, 200], b in [100, 170], and η in [1/7, 1/3]. Note that η allows the model to weight the effect of the height differently from the effect of the width.

Estimating selection time T_s. The above model for computing T_m can only be applied if the mouse movement is one-directional and involves a single target which is rectangular, square, or circular in shape. Consequently, selection of a label or item cannot be modeled using it. Observe that searching for a label involves moving the cursor over multiple targets to select an item. In fact, the *Schema Panel* is similar to a hierarchical menu and one needs to select an item during query formulation by navigating the cursor through the hierarchy using predominately vertical movements to select the desired label.

Note that we assume the labels or items are organized vertically and hence ignore horizontal movements in this panel as the horizontal width is negligible here. Furthermore, we assume that they are organized in a specific order (*e.g.,* lexicographically ordered). Hence, a user can move to the direction of the target item rapidly using an "open loop" movement. Consequently, we adopt the following *logarithmic* model proposed by Ahlström [4] for modeling selection time of an item, which integrates both the time to find the item and the time to move to the target.

$$T_s = m + n \times (\log_2(p + 1)) \tag{3}$$

where p is the *position number* of the target item, and m and n are empirically-determined constants. For XML or graph data, p can be computed as the total number of items below or above the item selected in the preceding condition.

Remark. Observe that Equation 1 demands empty result check should consume less than $(T_m + T_s)$ time. Although, as we shall see in Section 7, it is possible to realize this for a variety of queries, certain framework may possibly take more than $(T_m + T_s)$ time for certain scenario. Can our proposed model be easily extended to handle such scenario? We posit that it is indeed the case as additional times (*e.g.,* T_d) to complete tasks related to breakpoints FB3, FB4, etc. can easily be added to the right hand side of Equation 1 and notifications can then be delivered during corresponding breakpoints. Although such delivery is not at optimal breakpoints, it is still delivered opportunely when the mental workload is low.

4.3 Applicability to Other Feedback Problems

Equation 1 can also be used to model other kinds of query feedback problems (discussed in Section 1.1). We answer to this question affirmatively for the following reason. Consider the long-running query feedback or the query fragment suggestions problem. Given C' and C_{new} already constructed in the GUI, the long-running query feedback problem aims to detect if this query fragment will take a long time to run and notify the user opportunely. Similarly, the query fragment suggestions problem aims to provide top-k suggestions based on conditions C' and C_{new} constructed by the user. Observe that these notifications must be delivered to the user before the construction of C_{next} for reasons identical to the ER problem. For instance, it is ineffective to provide suggestions when the user has already constructed C_{next}. Hence, similar to the ER problem, notifications for these problems must also be delivered at breakpoints FB1 and FB2. That is, Equation 1 is applicable for these scenarios as well. This is also the case for the query syntax feedback problem. Hence, *our proposed model is applicable for a variety of interruption-sensitive query feedback problems.*

5. THE ISERF FRAMEWORK

We now present the *i*SERF framework to efficiently detect the empty-result (ER) phenomenon during query construction and to effectively deliver appropriate notification by realizing our defer-to-breakpoint-based notification scheme. It is comprised of two key modules, which are described below.

Empty results detection module. The goal of this module is to efficiently detect if the partially constructed query fragment returns an empty result. If it does, then the *interruption-sensitive notification* module is invoked to deliver intrusive notification at an opportune time. Specifically, there are two scenarios for empty result. *Scenario 1:* a newly constructed condition C_{new} does not have any match in the underlying database (*i.e.,* $R(C_{new}) = \emptyset$). *Scenario 2:* the constructed conditions are connected by AND connectives and each has a non-empty result match but there does not exist any data instance that satisfies *all* constructed conditions (*i.e.,* $R(C') = \emptyset$ and $\forall C_i \in C', R(C_i) \neq \emptyset$). Observe that efficient implementation of this module depends on the type of database (graphs or XML).

Interruption-sensitive notification module. Algorithm 1 outlines a generic implementation framework of this module. It takes as input parameters *nullCond* and *allCond* (output from the above module representing Scenarios 1 and 2 of empty-result queries, respectively) and are set to `true` if the query returns an empty result.

The reason we use two separate parameters to represent the two scenarios is because we intend to deliver two different notifications with different content to explain these scenarios. The values of the remaining input parameters a, b, m, n, and η (Equations 2 and 3) are empirically determined. The *hasCoarseBreakpoint* procedure in Line 3 detects Coarse breakpoints (Table 1). Lines 4 - 21 are executed if there are no Coarse breakpoints. The *getCursorDirection* procedure checks if the user is moving the cursor towards the *Schema Panel* (Panel 1). One approach to determine this movement is to use the following heuristic. Consider Figure 2(a). Let the cursor be at point A in the *Query Panel* and the rectangular box represents the *Schema Panel* with height H. Let the perpendicular distance from the cursor to the *Schema Panel* be Z. Then,

$$\theta_1 + \theta_2 \quad = \quad \arctan\left(\frac{\tan(\theta_1) + \tan(\theta_2)}{1 - \tan(\theta_1)\tan(\theta_2)}\right) \tag{4}$$

where $\tan(\theta_1) = \frac{H_1}{Z}$, $\tan(\theta_2) = \frac{H_2}{Z}$, and $H_1 + H_2 = H$.

Let the angle of motion of the mouse be β. Then, as long as the direction of motion of the mouse is on the left of A and β is within $(\theta_1 + \theta_2)$, the cursor is moving towards the *Schema Panel*. That is, given the angle of movement β, we can determine whether the cursor is moving towards the *Schema Panel* using $(\theta_1 + \theta_2)$.

If the mouse pointer is moving towards the *Schema Panel*, the movement time T_m is computed using Equation 2 (Line 6) and the notification delivery is suspended by T_m time (Line 7) to allow the cursor to move to the *Schema Panel*. Note that T_m is not constant for a given query as the distance D varies with each constructed condition. A keen reader may observe that we estimate T_m to determine the waiting time instead of waiting until the cursor reaches the *Schema Panel*. This is due to two key reasons. First, the theoretical estimate of T_m has been proven to have high real-world accuracy [2] and as a result it simulates the time to reach the *Schema Panel* with high degree of accuracy. Second, the mouse movement may be disrupted or stalled half way as a user may be distracted with other non-query activities (*e.g.,* phone call, discussion with a colleague). Consequently, it will prevent the notification to be delivered to the user even when she has stalled query formulation temporarily.

Lines 10 - 17 capture the case when the mouse pointer is already in the *Schema Panel* searching for an item. The selection time T_s is computed in Line 13. The notification delivery is suspended by T_s time (Line 14) to allow the item to be selected (for reasons justified above). Finally, Lines 22 - 26 display appropriate notification message identifying condition(s) $C_\varnothing$ responsible for empty result.

Observe that the above strategy delivers interruption-sensitive notifications by considering only optimal breakpoints (`FB1` or `FB2`). However, as remarked in Section 4.2, it can be easily augmented to deliver such notifications when empty result checking time exceeds $(T_m + T_s)$ by simply adding the wait times associated with tasks related to `FB3`, `FB4`, etc.

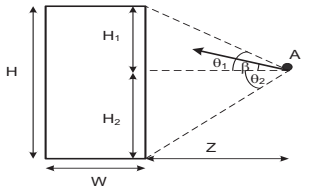
Generality of the framework. Observe the two notable features of the *i*SERF framework. First, it is orthogonal to the expressiveness of the GUI as well as the underlying query processor. Hence, *i*SERF can easily be built on top of any XML or graph query processor. Furthermore, it does not impact query evaluation time as it is only invoked during query formulation. Second, the interruption-sensitive notification module is not tightly coupled to the ER detection module. Hence, it can be easily incorporated to support other types of interruption-sensitive query feedback problem.

6. ISERF FOR XML QUERY FORMULATION

In the preceding section we have presented the generic implementation of *i*SERF on tree or graph-structured data. To demonstrate its effectiveness, we have implemented it on top of XBLEND [27,31],

Algorithm 1: INTERRUPTION-SENSITIVE NOTIFICATION MODULE

```
   Input: nullCond, allCond, C∅, a, b, m, n, η.
   Output: Notification N
1  Initialize canInterrupt = false;
2  while canInterrupt is false do
3      if !hasCourseBreakpoint() then
4          moveDir ← getCursorDirection() ;
5          if moveDir is true then
6              T_m ← movementTime(D, W, H, a, b, η) /* Equation 2 */;
7              wait(T_m) ;
8              canInterrupt = true;
9          else
10             if cursor is in Schema Panel then
11                 if an item is not dragged then
12                     p ← getPosition();
13                     T_s ← selectionTime(m, n, p) /*Equation 3 */;
14                     wait(T_s) ;
15                     canInterrupt = true;
16                 else
17                     canInterrupt = false;
18         if cursor is in Query Panel and selected item is dropped then
19             canInterrupt = true;
20     else
21         canInterrupt = false;
22 if NullCond is true then
23     Display N = "The condition C∅ will return an empty result.";
24 else
25     if NullCond is false ∧ AllCond is true then
26         Display N = "The conditions C∅ will return an empty result.";
```



(a) β

Source	Size	No. of files	No. of Attributes	No. of Elements
UniProt	1.5GB	1	38,380,645	20,836,316
Interpro	69MB	1	1,427,234	988,079
PDB	287MB	30	692,583	5,578,498

(b) Datasets.

Figure 2: (a) Computing movement angle; (b) Datasets.

a visual XML query processor built on top of a relational framework. XBLEND interleaves visual query construction and query processing to prune false results and prefetch partial query results by exploiting the latency offered by the GUI-based query formulation. Note that we chose XBLEND as it supports materialization of intermediate *results synopsis* during visual query construction, which we leverage for realizing the *i*SERF framework. However, the techniques we develop in the rest of the paper are largely independent of this choice, and we can replace XBLEND with another visual querying system that supports such a query processing paradigm.

Next, we introduce the visual XML query model that we use to demonstrate the *i*SERF framework. Then, we briefly elaborate on the XBLEND GUI for formulating visual queries. We propose a data structure called a *condition-results tree* (CR-tree) to support efficient detection of the ER phenomenon in XBLEND. Lastly, we present the algorithm for realizing *i*SERF.

6.1 A Visual XML Query Model

For ease of presentation, we primarily focus on a special type of visual XML query prevalent in many applications. Specifically, it can be textually represented by an XQuery query $Q = (\mathcal{F}, \mathcal{W}, \mathcal{R})$ where $\mathcal{F}$ is a set of `for` clause items, $\mathcal{W}$ is a set of predicates logically connected by AND or OR operators in the `where` clause, and $\mathcal{R}$ contains the *output expression* specified in the `return` clause. The predicates in $\mathcal{W}$ can be categorized into two types, namely *join expressions* and *non-join expressions*. A *join expression* captures value-based join between elements (attributes) of single or multiple

data sources. On the other hand, a *non-join expression* expresses a non-join filtering condition on a single data source. In the sequel, we refer to each expression in the `where` clause as a *condition*. We denote a set of conditions as C. Finally, the `return` clause has a single *output expression r* representing the *output node*.

Remark. The above query model can easily be extended to support a wider variety of features such as different location steps[5] and qualifiers (*e.g.,* position predicate, not-predicate) as long as the underlying XML database engine can support their evaluation. For instance, if a user visually specifies a condition c involving a path expression containing `descendant` and `preceding` axis at a particular formulation step, then this visual action will be translated to a corresponding textual query and forwarded to the underlying query engine for execution. The result set of c is then used by *i*SERF to detect whether the partially-constructed query yields an empty result. Having said this, it is paramount to balance expressiveness and usability in a visual querying environment as a wide variety of XML queries (*e.g.,* queries with nested `for`s) are not easy to formulate even visually as it requires a deep understanding of the language which many end-users do not possess [7]. Nevertheless, as we shall see later, *our strategy to realize the i*SERF *framework on an* XML *querying system is orthogonal to the expressiveness of its visual querying environment.*

6.2 Visual Query Interface of XBLEND

A visual XML query interface can be classified into two types, namely, *node-based* and *path-based*. In a *node-based* interface an XML query is constructed by taking a node-at-a-time approach. On the other hand, in a *path-based* interface, a query is formulated by taking a path-at-a-time strategy. The visual interface of XBLEND belongs to the latter type. Figure 1 depicts a screen dump of the XBLEND visual interface. Specifically, the DataGuide [13] is adopted to construct the structural summary of an XML document in Panel 1. When a user drags a vertex from Panel 1, the path expression corresponding to this vertex is automatically built. To formulate a query, a user takes the steps described in Section 2.1.

6.3 Condition-Results Tree (CR-Tree)

A *condition-results tree* (CR-tree), denoted as U, describes the set of currently constructed conditions that are connected by AND or OR connectives and corresponding sets of intermediate results that satisfy these conditions. Specifically, each internal node of U represents an AND or OR connective. Each leaf node of U contains a set of non-join conditions that are processed together[6] and the *vertex identifier set M* in which the prefetched vertices[7] satisfying the conditions are stored. Figure 3 depicts examples of CR-trees. Note that we only materialize the vertex identifiers of an XML document $\mathbb{D}$ in M instead of entire content of XML subtrees because it is more space-efficient. Furthermore, the identifier scheme is not tightly coupled to any specific system as any numbering scheme that can uniquely identify vertices in an XML tree can be deployed.

DEFINITION 1. *The condition-results tree (CR-tree) is a 2-tuple* $U = (\mathcal{V}_u, \mathcal{E}_u)$*, where* $\mathcal{V}_u$ *is a set of nodes in U and* $\mathcal{E}_u$ *is a set of edges. A leaf node* $v \in \mathcal{V}_u$ *is a 2-tuple* $v = (C_{nj}, M)$*, where* C_{nj} *is a set of non-join conditions that are processed together and M is the vertex identifier set that stores the prefetched data satisfying* C_{nj}*. If* $v_i \in \mathcal{V}_u$ *is an internal node then* $label(v_i) \in \{AND, OR\}$. □

[5] We consider `parent-child(/)` and `attribute(/@)` location steps due to the availability of structural summaries of underlying XML data in the GUI (Panel 1 in Figure 1).

[6] The set of conditions that need to be processed together is determined by XBLEND [31].

[7] For clarity, we distinguish between a node in a CR-tree and a node in an XML document or structural summary by using the terms "node" and "vertex", respectively.

Algorithm 2: *i*SERF on XBLEND

Input: Actions on the *Query Panel*, set of vertex identifiers R_o satisfying the output expression r.
Output: Notification N

```
1  Initialize nullCond and allCond to false ;
2  Initialize CR-tree U;
3  Initialize set of conditions C = ∅;
4  Set parameters a, b, m, n, η ;
5  A ← getGUIAction();
6  while (A ≠ "Run" or A ≠ "Abort") do
7      if (A == "Add") then
8          /* Thread 1 */;
9          C_new is the new condition and C_t is the drop target;
10         R_new ← prefetch(C_new, C_t, C, R_o) //XBLEND ;
11         C.insert(C_new);
12         U.insert(C_new, R_new);
13         (nullCond, allCond, C_∅) ← checkEmpty(U, nullCond, allCond);
14         Lines 1–21 in Algorithm 1 /* Thread 2 */;
15         if nullCond is true ∨ allCond is true then
16             N ← Lines 22– 26 in Algorithm 1 ;
17             return N;
18     A ← getGUIAction();
```

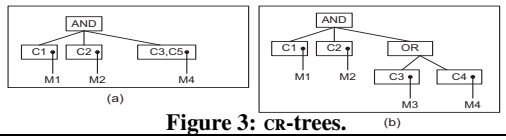


Figure 3: CR-trees.

Note that for practical cases the number of leaf nodes in a CR-tree is small as users typically do not formulate a large number of conditions using a visual interface. Furthermore, each leaf node is associated with an identifier set and not a set of XML subtrees. Hence, a CR-tree can easily fit in the main memory of a modern commodity desktop machine.

Observe that the CR-tree can be leveraged to efficiently detect the two scenarios of empty results (recall from Section 5). Consider the CR-tree with an AND node in Figure 3(a) and the vertex identifier sets M_1, M_2, and M_4. Here, $M_1 = R(C_1)$, $M_2 = R(C_2)$, and $M_4 = R(C_3, C_5)$. Let C_1 be the most recent condition constructed by a user. Then, the query represented by this CT-tree returns empty result if any one of the following conditions is satisfied: (a) *Scenario 1*: $M_1 = \emptyset$; (b) *Scenario 2*: $M_1 \neq \emptyset$, $M_2 \neq \emptyset$, $M_3 \neq \emptyset$, and $M_1 \cap M_2 \cap M_4 = \emptyset$. On the other hand, when the subtree root is an OR node, all vertex identifier sets associated with its leaf nodes have to be empty to qualify as an empty result. These conditions for AND and OR nodes of a CR-tree can be combined to determine whether a query fragment returns an empty result.

6.4 iSERF on XBLEND

Algorithm 2 outlines the implementation of *i*SERF on top of XBLEND. Since XBLEND uses a path-based visual interface that leverages structural summaries, we assume that the path expression in each formulated condition has at least one match in the XML document $\mathbb{D}$. R_o represents the set of vertex identifiers matching the output expression (generated by XBLEND) which is specified in Step 1 during query formulation (Section 2.1).

The variable $\mathcal{A}$ represents the query condition formulation action in the *Query Panel*. When a user drags a new query condition C_{new} and drop it on an existing condition C_t (Line 7), Lines 7–14 are executed in two concurrent threads. In the first thread, the algorithm invokes the prefetching technique of XBLEND [27, 31] (Line 10) to materialize the vertex identifiers in R_o that satisfy C_{new} in a temporary relation T_{new}. Details of this technique, which is orthogonal to this work, can be found in [27]. Next, it adds C_{new} and the vertex identifiers in T_{new} into the CR-tree U (Line 12). Then, the algorithm invokes the *checkEmpty* procedure which detects if

Algorithm 3: CHECKEMPTY

Input: CR-tree U, boolean flags *allCond*, *nullCond*.
Output: *nullCond*, *allCond*, Condition set $C_{\varnothing}$.

```
1 (IdSet, nullCond) ← U.TRAVERSE(C_new, nullCond) ;
2 if IdSet = ∅ then
3     allCond = true;
4     Find conditions C_∅ in U responsible for unsatisfiability;
5 return allCond, nullCond, C_∅
```

Id	Queries	Result Size
Q1	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro, $cellCategory in doc('PDB.BIOXML')/PDBx:datablock/PDBx:cellCategory where $interpro/pub_list/publication/year > "1950" {C2} and $entry/keyword = "3D-structure"{C1} and $interpro/@id = $entry/dbReference/@id {J2} and $PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference/@id {J3} return $entry/name;	8
Q2	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro, $publication in $entry/pub_list/publication where $publication/journal = "Structure" {C3} and $publication/year = "2002" {C4} and $entry/@created[contains(., "2001")] {C1} and $entry/organism/name = "Human" {C2} and $interpro/@id = $entry/dbReference/@id {J3} return $entry/name;	23
Q3	$interpro in doc('INTERPRO.BIOXML')/interprodb where $interpro/name[contains(., "hydrolase")] {C1} and $interpro/name[contains(., "subfamily")] {C2} return $interpro/pub_list/publication/journal	35
Q4	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro where $interpro/abstract/reaction[contains(., "H2O")] {C2} and $entry/keyword[contains(., "3D-structure")] {C1} and $interpro/@id = $entry/dbReference/@id {J2} return $entry/gene	5
Q5	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro, $classification in $interpro/class_list/classification where $classification/category/category = "Molecular Function" {C2} and $entry/keyword[contains (., "3D-structure")] {C1} and $classification/description[contains(., "binding")] {C3} $interpro/@id = $entry/dbReference/@id {J2} return $entry/gene;	2776

Figure 4: Query set.

the query fragment returns an empty result. Concurrently, in the second thread Lines 1–21 of Algorithm 1 are executed to determine the breakpoint for notifications. If the query result is empty (Line 16), then Lines 22–26 of Algorithm 1 are executed to deliver interruption-sensitive notification at an opportune time.

checkEmpty procedure. This procedure (Algorithm 3) encapsulates the checking of Scenarios 1 and 2. Note that our goal is to check both these scenarios efficiently so that for practical cases T_{ont} satisfies the bounds in Equation 1. Hence, we take several optimization strategies towards this goal. We first check if R_{new} in U is empty (before traversing to other nodes) and it is not connected by the OR operator with other existing conditions. If it is then Scenario 1 is satisfied. Consequently, *nullCond* is set to `true`, $C_{\varnothing}$ is set to C_{new}, and the algorithm terminates early (all these steps are encapsulated in Line 1). Otherwise, an empty result is only possible due to Scenario 2. Hence, it traverses U in a depth-first fashion. If an internal node is an "`AND`" node, then the vertex identifier sets associated with its child nodes are intersected to determine if some vertices are shared by all, indicating a non-empty result for the constructed query fragment. Otherwise, the vertex identifiers are combined (union). We use the adaptive set-intersection algorithm in [11] which aims to use a number of comparisons as close as possible to the minimum number of comparisons ideally required to establish the intersection. After traversing U, the set of vertex identifiers *idSet* that are shared by M_{new} and rest of the identifier sets is returned. Lastly, *allCond* is set to `true` if Scenario 2 is satisfied (Lines 2-3). Additionally, the conditions $C_{\varnothing}$ which are responsible for an empty result are identified and returned as well.

6.5 Extensibility

A keen reader may observe the three notable features of the *i*SERF framework on XML. First, the interruption-sensitive empty-result detection and notification scheme is orthogonal to the underlying visual XML query processor. Hence, *it can easily be built on top of any* XML *query processor that supports such incremental query processing paradigm*. Second, the solution is not limited to the visual XML query model in Section 6.1, but can be applied to richer variety of XML queries. To elaborate further, reconsider the steps for visual query formulation in Section 2.1. Now suppose that the visual interface enables us to formulate richer variety of queries by enabling specification of various XPath axis, qualifiers, `order-by` clause, aggregation functions, etc. Then the visual construction of these features in a query condition(s) will often take place during Steps 4 or 5 (*after* the selection of an attribute or element). For instance, if a user wishes to specify a `descendant` axis then she may do it on the dropped condition after Step 4 or in the *Dialog Box* that appears in Step 5. Once formulated, *the time to detect an empty result for such a query condition is still bounded by Equation 1*. Hence, as long as the underlying XML query processor is efficient to evaluate these complex conditions, the *i*SERF framework is amenable to richer variety of queries. Third, the *i*SERF framework *can easily incorporate more advanced features related to the* ER *problem* such as top-k query modification or relaxation suggestions [20,23] as the latter is orthogonal to the framework. Note that such suggestions also need to be delivered to the user (along with the empty results problem) using intrusive notifications at opportune times.

7. PERFORMANCE STUDY

*i*SERF is implemented in Java JDK 1.7. In this section, we investigate its performance in the context of visual XML query formulation using XBLEND (Section 6). The experiments were conducted on an Intel Core 2 Quad 2.66GHz processor and 3GB RAM. The operating system was Windows XP Professional SP3. A Logitech Mouse was used and its acceleration was set to its default settings (acceleration: 2/1). We compare *i*SERF (denoted by ISF) against XBLEND (denoted by XB), which is interruption-insensitive.

Specifically, we investigate the following issues. (a) Can the empty-result check be efficiently realized in a visual querying framework like XBLEND? (b) Is the defer-to-breakpoint notification scheme the most effective strategy for notification in *i*SERF? (c) Can *i*SERF deliver notification at optimal breakpoints (Equation 1) in most practical cases? (d) What is the impact of interruption-sensitive notifications on the *query formulation time*?

7.1 Experimental Setup

Data and Query Sets. We use the XML representations of UNIPROT, PDB, and INTERPRO downloaded from their official websites. The features of these datasets are given in Figure 2(b). We chose the representative queries in Figure 4 that join up to three data sources. Although our approach can support conditions which are connected by AND/OR connectives, here we chose queries with AND connectives as they are more likely to generate empty answers. The numbers with labels in curly braces in the `where` clause represent the default sequence of steps for formulation of conditions. Note that if a join (denoted by J_i) and a non-join condition (denoted by C_k) have same subscript then it means that the join condition is formulated immediately after its non-join counterpart[8]. For instance, the join condition $J2$ in $Q1$ and the non-join condition $C2$ share same subscript. That is, $J2$ is specified immediately after the formulation of $C2$. Unless mentioned otherwise, we shall be using the default query formulation sequence.

Observe that we did not choose XQuery queries that are too complex as it is observed by Augurusa *et al.* that a visual XQuery interface is useful when it serves the needs of the majority of the users in expressing their queries, which are typically simple [1].

Furthermore, observe that the above queries return a non-empty result. Hence, we create *modified* versions of these *original* queries

[8]They are evaluated together by XBLEND to create a single temporary relation. XBLEND can automatically detect the need for a join condition across multiple data sources during query formulation.

Id	Modifications	Id	Modifications
Q1[C1]	$entry/keyword = "3-D structure"	Q3[C2]	$interpro/name[contains(., "sub-family")]
Q1[C2]	$interpro/pub_list/publication/year >2050	Q4[C1]	$entry/keyword[contains(., "3-Dim-structure")]
Q2[C1]	$entry/@created[contains(., "2050")]	Q4[C2]	$interpro/abstract/reaction[contains(., "H3O")]
Q2[C2]	$entry/organism/name = "Human being"	Q5[C1]	$entry/keyword[contains(., "luminiscence")]
Q2[C3]	$publication/journal = "Chemistry"	Q5[C2]	$classification/category/category = "Mol. Function"
Q3[C1]	$interpro/name[contains(., "lipoprotein")]		

Figure 5: Modifications for creating empty-result queries.

to generate 11 empty-result queries. Each original query is modified one condition-at-a-time and is identified by $Qi[Cj]$ indicating that it is created by modifying condition Cj in Qi (*e.g.,* $Q1[C1]$ means that the condition $C1$ of $Q1$ is modified to make $Q1$ return an empty result). The set of actual modifications to the conditions is given in Figure 5.

Participants Profile. Ten unpaid volunteers with varying degrees of familiarity with XML participated in the experiments to formulate visual queries. At the start, participants were trained to use the GUI. For every query the participants were given some time to determine the steps that are needed to formulate it visually. This is to ensure that the effect of thinking time is minimized during query formulation. Note that if a user quickly formulates a query, less time is available for delivering notification at optimal breakpoints. The participants were given one query at a time. If an error was committed by a participant then that particular formulation effort is ignored and he had to start afresh.

Query formulation by participants. The participants were asked to formulate two categories of queries. (a) *Empty-result queries:* Each participant first formulates the modified queries (Figure 5) without being aware of the fact that these queries return empty result. For example, a participant constructs the modified condition $Q_1[C_1]$ instead of C_1 while formulating Q_1. Note that when he/she formulates a condition that results in empty result (*e.g.,* $Q_1[C_1]$), the ER problem is notified and suggestion to the participant is provided to modify the query by incorporating the original condition (*e.g.,* the condition C_1 in Figure 4). The participant then modifies the query and continues constructing remaining conditions in the query until completion. (b) *Nonempty-result queries:* After formulating all the modified queries, each participant formulates the original queries in Figure 4. Each query was formulated five times by each participant.

Values of Parameters a, b, m, n, and η. In order to empirically determine the parameter values of Equations 2 and 3, the participants were tasked to formulate three queries ($Q1$, $Q3$, $Q4$) according to the above setup. The movement and selection times (T_m and T_s) were recorded for all participants and for all trials. Then, the average movement time $\overline{T}_m = 1071.2ms$ with $\sigma = 7.17$. Hence, we use $\overline{T}_m$ as movement time for the XBLEND GUI. We set $\eta = 0.33$ according to [2]. Then, we compute a and b that best fits this movement time using Equation 2. It turns out that for $a = -30$ and $b = 106$, $T_m = 1061.8ms$. Similarly, the average selection time $\overline{T}_s = 1802ms$ with $\sigma = 52.12$. Note that high σ is expected as some users may be unfamiliar with the structural summaries and might expand and collapse the subtrees several times before selecting a vertex. Using these values and Equation 3, $m = 586$ and $n = 140$ give us selection time that is closest to $\overline{T}_s$ ($T_s = 1846ms$).

7.2 Experimental Results

Effect of Empty Result Check. Before we investigate the impact of interruption-sensitive notification during query formulation, we would like to investigate the cost of accommodating a query feedback technique (in our case, empty-result check) in a visual querying framework like XBLEND. Specifically, we compare the execution cost of Lines 10-11 (XB [27]) and Lines 10-13 (ISF) in Algorithm 2. Note that Line 13 is not executed if the visual querying framework does not support the check for an empty-result (*e.g.,* XBLEND). Also, in this set of experiments Lines 14 and 16 are not executed as we use the original queries in Figure 4. We shall study the impact of the notification service later.

Query	System	Step 1	Step 2	Step 3	Step 4
Q1	XB	0.26 (6123)	1.39 (156172)	0.38 (9)	
	ISF	0.36	1.47	0.46	
Q2	XB	0.32 (5850)	0.42 (9595)	7.21 (30294)	1.14 (4317)
	ISF	0.39	0.5	7.29	1.23
Q3	XB	0.08 (710)	0.11 (35)		
	ISF	0.15	0.19		
Q4	XB	0.22 (5601)	0.66 (166)		
	ISF	0.31	0.76		
Q5	XB	0.15 (5601)	1.21(130318)	1.01 (70327)	
	ISF	0.22	1.29	1.09	

Figure 6: Effect of an empty-result check (in sec.).

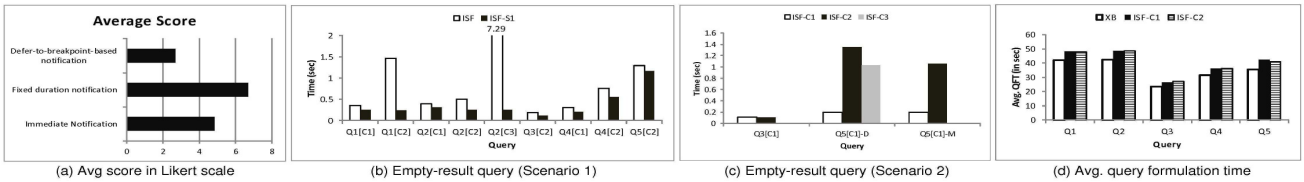
Figure 6 shows the execution times for prefetching (XB) and (additionally) checking an empty result (ISF). Each column labeled Step$i$ represents the running time associated with corresponding query condition C_i in a query. The values in parenthesis represent the size of the materialized relations (vertex identifier set). Note that here we do not report times taken by XB to retrieve the final query results as it is orthogonal to the problem. It is evident that the cost of accommodating an empty result check is sufficiently low as execution time of each step is not significantly increased for all queries. Also, it is robust to large size of intermediate results (*e.g.,* Step 2 of $Q1$ and $Q5$) as ISF only retrieves vertex identifiers. In other words, the *checkEmpty* procedure is efficiently realized by ISF.

Justification for Defer-to-Breakpoint Notification Scheme. Next, we design a user study to justify the defer-to-breakpoint notification scheme realized by ISF and its impact on users. All participants were tasked to formulate any four empty-result queries (Figure 5). Measurements were taken on frustration level of the participants at the timing of notification (interruption). The rating was made using a 7-point Likert scale, ranging from 1 being very pleasing to 7 being very frustrating. Three types of notification schemes were presented to participants to experience, namely, *"Immediate notification"*, *"Fixed duration notification"* and our proposed *"Defer-to-breakpoint-based notification"*. Note that the *"Immediate notification"* scheme delivers notification as soon as a query is detected to generate an empty result. In the *"Fixed duration notification"* scheme, the notification is triggered after a fixed 2 seconds delay. Figure 7(a) reports the results. Clearly, participants find our proposed approach most effective. On the other hand, they found the *"Fixed duration notification"* scheme most annoying.

Optimal Notification Time (ONT). Given the superiority of defer-to-breakpoint notification strategy, we now investigate whether notifications can be delivered at optimal breakpoints (Equation 1). Specifically, we study whether ONT satisfies the upper bound in Equation 1 in practice. In our experiments, $T_m = 1061.8ms$ and $T_s = 1846ms$ (see Section 7.1). Hence, $0 < T_{ont} < 2907.8$.

Figure 7(b) reports the execution times for checking for an empty result for Scenario 1 cases. We plot the execution times (Lines 10-13 in Algorithm 2) of the original condition Ci in the nonempty-result query Qi (denoted by ISF) and the modified condition in the empty-result query $Qi[Ci]$. Observe that the *time taken to execute these steps is less when the condition does not have a match* in the dataset. This is beneficial as necessity to deliver notification at optimal breakpoints only arises when the query returns empty answer. Note that the execution times *include* the prefetching times for partial matches in XB.

Figure 7(c) reports the performance due to Scenario 2 for queries $Q3[C1]$ and $Q5[C1]$. Note that in both these modified queries each condition has non-empty result matches ($R(C_i) \neq \emptyset$) but together they return an empty result. For each modified query, we plot the



(a) Avg score in Likert scale (b) Empty-result query (Scenario 1) (c) Empty-result query (Scenario 2) (d) Avg. query formulation time

Figure 7: Performance results.

time for checking for an empty result after formulation of each condition (denoted by ISF-C*i*). For $Q5[C1]$, we create two variants; one with the original query formulation sequence `C1-C2-C3` (denoted by $Q5[C1] - D$) and the other following the sequence: `C1-C3-C2` (denoted by $Q5[C1] - M$). Note that the former becomes empty after formulation of the three conditions whereas the latter becomes empty after formulating the second condition $C3$. In all cases, the empty-result check can be completed within a second or less.

Importantly, *the time taken to prefetch partial results in* XB *(Line 10) and check for an empty result (Lines 11-13) is significantly less than* $(T_m + T_s)$. This is highly desirable as *notifications can be delivered at optimal breakpoints (FB1 and FB2 in Table 1)*. Note that the movement and selection times may vary with different users, *i.e.,* users familiar with the GUI and structural summary may move their mouse faster than others. Observe that $T_{ont} < 0.5(T_m + T_s)$ for all modified conditions. That is, even if a user moves twice as fast as normal users, the notification will still be delivered at optimal breakpoints. Only, when the speed of movement is less than $0.4(T_m + T_s)$, the notification has to be suspended until the construction of the next condition. However, such speed is highly unlikely from typical end users as seen in Section 7.1.

Effect on Query Formulation Times. Lastly, we study the impact of interruption-sensitive notifications on the *query formulation time* (QFT), which is the time taken to formulate each query. Each query is formulated by all participants in its original and modified forms. Specifically, the QFT of a modified query *includes time to acknowledge the notification and continue formulation of all remaining conditions* (if any). Figure 7(d) plots the average QFT of the original query (denoted by XB) and its modified versions (denoted by ISF-C*i* for queries generated by modifying *Ci*). Obviously, modified queries require higher QFTs than the original ones due to resumption lag (recall from Section 1.3). However, the increase is very modest primarily due to the deferred notification scheme and the time users take to acknowledge notifications. Since the notifications are delivered only at breakpoints, the participants react faster to notifications as reported in several HCI studies such as [15].

8. CONCLUSIONS

There is increasing interest in enhancing the usability of database systems. This paper improves usability by contributing a novel paradigm of interruption-sensitive visual query feedback. Specifically, we present a framework called *i*SERF for detecting and scheduling notifications of empty-result queries at breakpoints in a visual environment. A key feature of our framework is its multidisciplinary flavor, integrating principles from HCI and cognitive psychology with data management. First, we drew upon the literature in cognitive psychology to establish that *i*SERF needs to consider the structure of visual query formulation tasks and breakpoints when reasoning about when to notify the user. Second, we leveraged work in HCI to quantitatively model the time available to *i*SERF to detect empty-result queries in order to ensure notification delivery at optimal breakpoints that lower the interruption cost. Lastly, we showed the usefulness and effectiveness of *i*SERF in efficiently checking for an empty result and delivering notification at optimal breakpoints in a visual XML querying environment.

Acknowledgement: Sourav S Bhowmick is supported by the Singapore-MOE AcRF Tier-1 Grant RG24/12. Byron Choi is partially supported by the Hong Kong RGC GRF12201315.

9. REFERENCES

[1] E. Augurusa, D. Braga, A. Campi, S. Ceri. Design and Implementation of a Graphical Interface to XQuery. In *ACM SAC*, 2003.
[2] J. Accot, S. Zhai. Refining Fitts' Law Models for Bivariate Pointing. *In ACM SIGCHI*, 2003.
[3] P. D. Adamczyk, B. P. Bailey. If not now, when? The effects of interruptions at different moments within task execution. In *CHI*, 2004.
[4] D. Ahlstrom. Modeling and Improving Selection in Cascading Pull-Down Menus Using Fitt's Law, the Steering Law, and Force Fields. *In CHI*, 2005.
[5] B. P. Bailey, J. A. Konstan. On the Need for Attention Aware Systems: Measuring Effects of Interruption on Task Performance, Error Rate, and Affective State. *In Journal of Computers in Human Behavior*, 22(4), 2006.
[6] S. S. Bhowmick, B. Choi, S. Zhou. VOGUE: Towards a Visual Interaction-aware Graph Query Processing Framework. In *CIDR*, 2013.
[7] D. Braga, A. Campi, S. Ceri. XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. In *TODS*, 30(2), 2005.
[8] S. Chaudhuri. Generalization and a framework for query modification. *In ICDE*, 1990.
[9] E. Cutrell, et al. Notification, Disruption, and Memory: Effects of Messaging Interruptions on Memory and Performance. *In IFIP Int Conf on HCI*, 2001.
[10] S. Comai, E. Damiani, P. Fraternali. Computing Graphical Queries Over XML Data. *In ACM TOIS*, 19(4): 371–430, 2001.
[11] E. D. Demaine, et al. Experiments on Adaptive Set Intersections for Text Retrieval Systems. *In ALENEX*, 2001.
[12] T. Gillie, D. Broadbent. What makes Interruption Disruptive? A Study of Length, Similarity and Complexity. *Psychological Research*, 50(4), 1989.
[13] R. Goldman, J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
[14] S. T. Iqbal, B. P. Bailey. Understanding and Developing Models for Detecting and Differentiating Breakpoints during Interactive Tasks. *In CHI*, 2007.
[15] S. T. Iqbal, B. P. Bailey. Effects of Intelligent Notification Management on Users and Their Tasks. *In CHI*, 2008.
[16] S. T. Iqbal, B. P. Bailey. Investigating the effectiveness of mental workload as a predictor of opportune moments for interruption. *In CHI*, 2005.
[17] G. Luo. Automatic Detection of Empty-result Queries. *In VLDB*, 2006.
[18] D. S. McCrickard, et al. Introduction: design and evaluation of notification user interfaces. *Int. J. Human-Computer Studies*, 58, 2003.
[19] D. S. McCrickard, C. M. Chewar. Attuning notification design to user goals and attention costs. *CACM*, 46(3), 2003.
[20] C. Mishra, N. Koudas. Interactive query refinement. *In EDBT*, 2009.
[21] C. A. Monk, D. A. Boehm-Davis, J. G. Trafton. The Attentional Costs of Interrupting Task Performance at Various Stages. *In Proc of the Human Factors and Ergonomics Society*, 2002.
[22] C. A. Monk, J. G. Trafton, D. A. Boehm-Davis. The effect of interruption duration and demand on resuming suspended goals. *J. of Experimental Psychology: Applied*, 14, 2008.
[23] D. Mottin, et al. A Probabilistic Optimization Framework for the Empty-Answer Problem. *In PVLDB*, 6(14), 2013.
[24] D. Newtson. Attribution and the Unit of Perception of Ongoing Behavior. *In J. of Personality and Social Psychology*, 28(1), 1973.
[25] P. Palanque, M. Winckler, et al. A Formal Approach Supporting the Comparative Predictive Assessment of the Interruption-Tolerance of Interactive Systems. *In ACM EICS*, 2009.
[26] D. D. Salvucci, P. Bogunovich. Multitasking and Monotasking: The Effects of Mental Workload on Deferred Task Interruptions. *In CHI*, 2010.
[27] B. Q. Truong, et al. MUSTBLEND: Blending Visual Multi-Source Twig Query Formulation and Query Processing RDBMS. *In DASFAA*, 2013.
[28] M. H. Vastenburg, D. V. Keyson, H. de Ridder. Considerate home notification systems: A user study of acceptability of notifications in a living-room laboratory. *In Int. J of Human Computer Studies*, 67, 2009.
[29] X. Xie, et al. PIGEON: Progress Indicator for Subgraph Queries. *ICDE*, 2015.
[30] S. Yang, et al. SLQ: A User-friendly Graph Querying System. *In SIGMOD*, 2014.
[31] Y. Zhou, et al. XBLEND: Visual XML Query Formulation Meets Query Processing. *In ICDE*, 2009.