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 isar 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, isar considers the structure of query formulation tasks and breakpoints when reasoning about when to notify the user. We present an nci-inspired model to quantify the performance bounds that isar must abide by for checking 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ACM’15, October 19–23, 2015, Melbourne, Australia.
© 2015 ACM. ISBN 978-1-4503-3794-6/15/10...
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 (vui) 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 opportunistically. 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 inopportune 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 inopportunistic, 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-sensitive 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 query feedback notifications. Specifically, we focus on one class of visual query feedback problem to illustrate this novel paradigm, namely empty result (ER) feedback\(^1\), 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 the following: (a) The ER problem re-\(^2\)

---

\(^1\)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.

\(^2\)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].
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 nci and cognitive psychology. We propose a novel psychology and nci-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 nci-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 iserf (Interruption-Sensitive Empty Results notification), grounded on well-founded principles from nci 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, iserf is not designed with the explicit aim to minimize query construction time, which is an orthogonal problem. Instead, the broad goal of iserf is to improve each user’s experience during visual query formulation by reducing the adverse effects of intrusive notifications (Section 5).
- We implement iserf 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 iserf (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.](image)

**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 if $R(C') = \emptyset \cap (R(Q) = \emptyset)$. 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 xml 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, isar sup-
ports interruption-sensitive notification which is absent from [17].

Research in non-interactive [8] and interactive [20, 23] query re-
laxation solutions proposes solutions to the empty-result problem. However, none of these efforts focus on opportune notification de-
ivery 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 nci has investigated intel-
ligent 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 nci community has shown that the negative effects of inter-
ruption (recall from Section 1.3) can be mitigated by deferring inter-
ruptions 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 mo-
moment of transition between two observable, meaningful units of task execution, and reflects a change in perception or action [24]. Re-
cent research in the nci 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 edit-
ing 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 fol-
lowing interesting results demonstrated by the nci community [5, 15]. First, the interruption cost is reduced and users are less frustr-
ated when intrusive notifications are scheduled to occur at break-
points rather than when delivered immediately. Second, users pre-
fer having notifications scheduled at breakpoints and they react faster to notifications that were scheduled at breakpoints. Third, ap-
lications 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, notifica-
tions of general interest should be delivered at Coarse breakpoints.

**Generic Breakpoints in Visual Query Formulation.** Recon-
consider the generic structure of the gui for tree or graph query con-
struction 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,

<table>
<thead>
<tr>
<th>Type</th>
<th>Tasks for Tree Queries</th>
<th>Tasks for Graph Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>CB1: User switch to another application (e.g., email); CB2: Minimize query window.</td>
<td>Same as tree queries.</td>
</tr>
<tr>
<td>Medium</td>
<td>MB1: Clicking a menu item to initiate creation of a new query; MB2: Switching to another query.</td>
<td>Same as tree queries.</td>
</tr>
<tr>
<td>Fine</td>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 1: Breakpoints for visual query formulation.

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 ek problem and then highlight how it can also be used to model other types of query feedback.

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 ek 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) ≠ ∅. Let Cnew 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 Cnew, our proposed framework takes two key steps. First, it checks whether R(C′ ∪ Cnew) = ∅. 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 condi-
tion Cnew (e.g., the next edge in a subgraph query fragment) is fin-
ished. 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 de-
levered at fine breakpoints FB1 and FB2 (Table 1). We refer to these breakpoints as optimal breakpoints. Observe that construction of Cnew is already completed after FB3.
Given the most recent constructed condition $C_{\text{new}}$ and a set of previously constructed conditions $C'$, the onset denoted as $T_{\text{onset}}$ refers to the amount of time available for checking if $R(C_{\text{new}} \cup C') = \emptyset$, so that notification can be delivered at optimal breakpoints. We now present an nci-inspired quantitative model to estimate this time. Given $C'$, adding a new condition $C_{\text{new}}$ 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 ($T_m$) and collection time ($T_s$), respectively. Then, $T_{\text{onset}}$ can be bounded by the following equation.

$$0 < T_{\text{onset}} < T_m + T_s$$

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

4.2 Estimating Movement and Selection Times

Estimating movement time $T_m$. Consider 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.

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

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 $\eta$ in $[1/7, 1/5]$. Note that $\eta$ 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.

$$T_s = m + n \times \log_2(p + 1)$$

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 scenarios. Can our proposed model be easily extended to handle such scenario? We posit that it is indeed the case as additional times (e.g., $T_s$) 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 opportunistically 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 $C_{\text{new}}$ already constructed in the ncni, the long-running query feedback problem aims to detect if this query fragment will take a long time to run and notify the user opportunistically. Similarly, the query fragment suggestions problem aims to provide top $k$ suggestions based on conditions $C'$ and $C_{\text{new}}$ constructed by the user. Observe that these notifications must be delivered to the user before the construction of $C_{\text{onset}}$ for reasons identical to the ncni problem. For instance, it is ineffective to provide suggestions when the user has already constructed $C_{\text{onset}}$. Hence, similar to the ncni problem, notifications for these problems must be 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 iserf 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 $C_{\text{new}}$ does not have any match in the underlying database (i.e., $R(C_{\text{new}}) = \emptyset$). 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 $C_i \in C'$, $R(C_i) \neq \emptyset$). Observe that efficient implementation of this module depends on the type of database (graphs or xml).

Interuption-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 $\eta$ (Equations 2 and 3) are empirically determined. The hasCourseBreakpoint 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,

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

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 $\beta$. Then, as long as the direction of motion of the mouse is on the left of $A$ and $\beta$ is within $(\theta_1 + \theta_2)$, the cursor is moving towards the Schema Panel. That is, given the angle of movement $\beta$, 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 $D$ for a given query as the distance $T$ where $\tan(\theta_2) = 0$.

If the condition-results tree is because we intend to deliver two different notifications.
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 iserf 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 iserf 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 vectors\(^5\) satisfying the conditions are stored. Figure 3 depicts examples of cr-trees. Note that we only materialize the vertex identifiers of an XML document \( D \) 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) \( U \) is a 2-tuple \( U = (V_u, E_u) \), where \( V_u \) is a set of nodes in \( U \) and \( E_u \) is a set of edges. A leaf node \( v \in V_u \) is a 2-tuple \( v = (V_v, E_v) \), where \( V_v \) is a set of non-join conditions that are processed together and \( E_v \) is the vertex identifier set that stores the prefetched data satisfying \( C_v \). If \( v_i \in V_v \) is an internal node then label \( (v_i) \in \{\text{AND}, \text{OR}) \).\(^\Box\)

\(^5\)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].

**Algorithm 2: iserf on XBLEND**

**Input:** Actions on the Query Panel, set of vertex identifiers \( R_v \) 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 \( C = \emptyset \);
4. Set parameters a, b, m, n, \( \eta \);
5. \( A \leftarrow \text{gui}\left(\text{Algorithm}\right)\);
6. while \((A \text{ is } \text{"Run"}) \text{ or } (A \neq \text{"Abort"})\) do
   1. If \((A = \text{"Add"})\) then
      1. (\("\text{Thread 1}\""));
      2. \(C_{new} = \text{the new condition and } C_i = \text{the drop target}\
      3. \(R_{v_i} = \text{prefetch}(C_{new}, C_i, C_j, R_j)\)
      4. \(\text{XBLEND}()\);
      5. \(C_i = \text{insert}(C_{new})\);
      6. \(U\leftarrow\text{insert}(C_{new}, R_{v_i})\)
      7. \(U\leftarrow\text{insert}(\text{nullCond, allCond})\)
      8. Lines 1–21 in Algorithm 1 (\("\text{Thread 2}\"")
   2. if nullCond is true \(\lor\) allCond is true then
      1. return \(N\);
   3. \(N\leftarrow\text{Lines 22–26 in Algorithm 1}\);
7. \(A \leftarrow \text{gui}\left(\text{Algorithm}\right)\);

**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 \( M_1, M_2, \text{ and } M_3 \). Here, \( M_1 = R(C_1), M_2 = R(C_2), \text{ and } M_3 = R(C_3) \). Let \( C_1 \) be the most recent condition constructed by a user. Then, the query represented by this cr-tree returns empty result if any one of the following conditions is satisfied: (a) Scenario 1: \( M_1 = \emptyset \); (b) Scenario 2: \( M_1 \neq \emptyset, M_2 \neq \emptyset, \text{ and } M_3 = \emptyset \). 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 iserf 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 \( D \). \( R_v \) 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 \( A \) 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 \( C_i \) (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 \( R_v \) 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
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), the interrupting condition of Algorithm 1 is 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 $T_{on}$ satisfies the bounds in Equation 1. Hence, we take several optimization strategies towards this goal. We first check if $R_{on}$ in $U$ is empty (before traversing to other nodes) and it is not connected by the $\land$ operator with other existing conditions. If it is then Scenario 1 is satisfied. Consequently, $\text{nullCond}$ is set to true, $C_{on}$ is set to $C_{on}$, 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 $M_{on}$ and rest of the identifier sets are returned. Lastly, $\text{allCond}$ is set to true if Scenario 2 is satisfied (Lines 2-3). Additionally, the conditions $C_{on}$ 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 iserf 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 varieties 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 rich query 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 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 iserf framework is amenable to richer variety of queries. Third, the iserf framework can easily incorporate more advanced features related to the $\text{ER}$ 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

**iserf** 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 **iserf** (denoted by is) 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 iserf? (c) Can iserf 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 unprof, 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 $\land$ or $\lor$ connectives, here we chose queries with $\land$ 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 $J$) and a non-join condition (denoted by $C_{on}$) have same script then it means that the join condition is formulated immediately after its non-join counterpart. For instance, the join condition $J2$ in $Q1$ and the non-join condition $C2$ share same script. 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 Angurusa 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.
to generate 11 empty-result queries. Each original query is modified one condition-at-a-time and is identified by \( Q_i[C] \) indicating that it is created by modifying condition \( C_j \) in \( Q_i \) (e.g., \( Q_1[C_1] \) means that the condition \( C_1 \) of \( Q_1 \) is modified to make \( Q_1 \) 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 \( Q_2[C_1] \) instead of \( C_1 \) while formulating \( Q_2 \). Note that when he/she formulates a condition that results in empty result (e.g., \( Q_i[C_i] \)), the \( \text{xml} \) problem is notified and suggested 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 \( \eta \). In order to empirically determine the parameter values of Equations 2 and 3, the participants were tasked to formulate three queries (\( Q_1, Q_3, Q_4 \)) 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 \( T_m = 1071.2ms \) with \( \sigma = 7.17 \). Hence, we use \( T_m \) as movement time for the XBlendo 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 \( T_s = 1802ms \) with \( \sigma = 52.12 \). Note that high \( \sigma \) 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 \( T_s (T_s = 1846ms) \).

### 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 XBlendo. Specifically, we compare the execution cost of Lines 10-11 (\( \text{xml} \) [27]) and Lines 10-13 (\( \text{isr} \)) 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., XBlendo). 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.

Figure 6 shows the execution times for times for prefetching (\( \text{xml} \)) and (additionally) checking an empty result (\( \text{isr} \)). Each column labeled \( \text{Steps} \) represents the running time associated with corresponding query condition \( C_i \) in a query. The values in parentheses represent the size of the materialized relations (vertex identifier set). Note that here we do not report times taken by \( \text{xml} \) 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 \( Q_1 \) and \( Q_5 \)) as \( \text{isr} \) only retrieves vertex identifiers. In other words, the checkEmpty procedure is efficiently realized by \( \text{isr} \).

**Justification for Defer-to-Breakpoint Notification Scheme.** Next, we design a user study to justify the defer-to-breakpoint notification scheme realized by \( \text{isr} \) and its impact on users. All participants were asked 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 (oنت).** 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 oنت 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, \( 0 < \text{oنت} < 2907.8 \).

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 \( C_i \) in the nonempty-result query \( Q_i \) (denoted by \( \text{sr} \)) and the modified condition in the empty-result query \( Q_i[C_i] \). 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 \( \text{xml} \).

Figure 7(c) reports the performance due to Scenario 2 for queries \( Q_3[C_1] \) and \( Q_5[C_1] \). Note that in both these modified queries each condition has non-empty result matches (\( R(C) \neq \emptyset \)) but together they return an empty result. For each modified query, we plot the
time for checking for an empty result after formulation of each condition (denoted by isr-ci). For QS[C1], we create two variants; one with the original query formulation sequence C1–C2–C3 (denoted by QS[C1–D]) and the other following the sequence: C1–C3–C2 (denoted by QS[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 $\text{xh}$ (Line 10) and check for an empty result (Lines 11–13) is significantly less than ($T_s + T_d$). 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 $T_{\text{soc}} < 0.5(T_s + T_d)$ 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 $0.4(T_s + T_d)$, 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 $xh$) and its modified versions (denoted by isr-ci for queries generated by modifying $C$). 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 resumption lag (recall from Section 1.3). However, the increase is very modest primarily due to the deferred notification scheme and resumption lag (recall from Section 1.3). Moreover, notifications can be delivered only at breakpoints, the participants react faster than others. Observe that $qft$ 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 $\text{ns}$ to quantitatively model the time available to isr-ci 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 isr-ci 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-ACRF Tier-1 Grant RG24/12. Byron Choi is partially supported by the Hong Kong RGC GRF12201315.

### 9. REFERENCES


