Interactive Query Formulation over Web Service-Accessed Sources

Michalis Petropoulos
CSE Department
SUNY Buffalo
mpetropo@cse.buffalo.edu

Alin Deutsch
CSE Department
UC San Diego
deutsch@cs.ucsd.edu

Yannis Papakonstantinou
CSE Department
UC San Diego
yannis@cs.ucsd.edu

ABSTRACT

Integration systems typically support only a restricted set of queries over the schema they export. The reason is that the participating information sources contribute limited content and limited access methods. In prior work, these limited access methods have often been specified using a set of parameterized views, with the understanding that the integration system accepts only queries which have an equivalent rewriting using the views. These queries are called *feasible*. Infeasible queries are rejected without an explanatory feedback. To help a developer, who is building an integration application, avoid a frustrating trial-and-error cycle, we introduce the CLIDE query formulation interface, which extends the QBE-like query builder of Microsoft's SQL Server with a coloring scheme that guides the user toward formulating feasible queries. We provide guarantees that the suggested query edit actions are complete (i.e. each feasible query can be built by following only suggestions), rapidly convergent (the suggestions are tuned to lead to the closest feasible completions of the query) and suitably summarized (at each interaction step, only a minimal number of actions needed to preserve completeness are suggested). We present the algorithms, implementation and performance evaluation showing that CLIDE is a viable on-line tool.

1. INTRODUCTION

Many information sources support only a limited set of queries over their schema, as a result of privacy constraints [17, 9] or a result of limited access methods [23, 7]. In both privacy and mediation-oriented systems, a source specifies a set of queries that can be answered directly using views over its schema. A mediator extends the set of directly supported queries with a set of indirectly supported ones by appropriately rewriting the latter so that they are answered by filtering and combining the results of directly supported queries. If a submitted query is not supported the user simply receives a rejection, being forced into a trial-and-error query development loop. We propose that the user should be guided toward feasible (i.e., supported) queries and we developed the CLIDE interactive system for this purpose.

The CLIDE (CLient guIDE) system is a graphical query formulation interface that mimics the visual paradigm of Microsoft's Query Builder, incorporated in MS Access and MS SQL Server [1]. Microsoft's Query Builder, in turn, is based on the Query-By-Example (QBE) [24] paradigm. CLIDE guides the user toward formulating feasible conjunctive queries and indicates any action that will lead toward a non-feasible conjunctive query. In particular, CLIDE provides compactly-presented guidance in the form of a color scheme, which in every step of the query formulation indicates which possible actions should, should not or may be taken in order to reach a feasible query. A flag indicates whether the current query is feasible or not. If it is, colors indicate how to reach another feasible query, which will be a syntactic extension of the current one. As usual, an action is the inclusion of a table in the `FROM` clause, the formulation of a selection condition in the `WHERE` clause or a projection of a column in the `SELECT` clause.

We illustrate the use of CLIDE and the color-driven interaction using an example from service-oriented architectures.

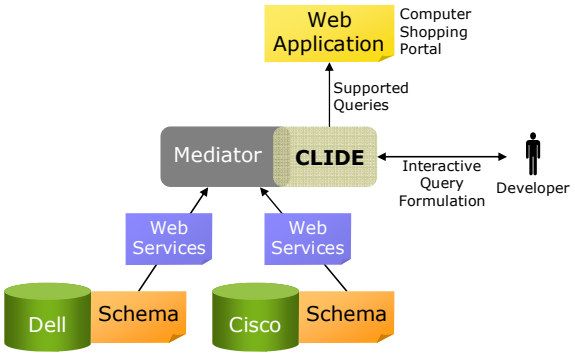


Figure 1: Service-Oriented Architecture

Service-Oriented Architectures Information systems offer limited access to their data by publishing views, web services or APIs. For example, Amazon's E-Commerce Service [2] provides a set of web services that allow one to query its catalog and product data, and Google's Web APIs [3] export web services for developers to issue search requests and receive results as structured data.

Service-oriented architectures [4] aggregate a collection of such services in order to provide more sophisticated web services and to support web applications. Figure 1 shows a simple instance of an architecture where the mediator enables a computer shopping portal, such as CNET.com, to have integrated query access to two sources. We assume that Dell and Cisco export a set of web services on their computer and router catalogs, respectively. Since we want to be able to issue (distributed) queries, we associate schemas with Dell and Cisco and model the web services as parameterized

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.

```
Computers(cid, cpu, ram, price)                    (Dell Schema)
NetCards(cid, rate, standard, interface)

ComByCpu(cpu) → (Computer)*
SELECT DISTINCT Com.*                                      (V1)
FROM Computers Com
WHERE Com.cpu=cpu

ComNetByCpuRate(cpu, rate) → (Computer, NetCard)*
SELECT DISTINCT Com.*, Net.*                               (V2)
FROM Computers Com, NetCards Net
WHERE Com.cid=Net.cid AND Com.cpu=cpu
AND Net.rate=rate


Routers(rate, standard, price, type)               (Cisco Schema)

RoutersWired() → (Router)*
SELECT DISTINCT Rou.*                                      (V3)
FROM Routers Rou
WHERE Rou.type='Wired'

RoutersWireless() → (Routers)*
SELECT DISTINCT Rou.*                                      (V4)
FROM Routers Rou
WHERE Rou.type='Wireless'


(S1.Computers.cid, S1.NetCards.cid)            (Column Associations)
(S1.NetCards.rate, S2.Routers.rate)
(S1.NetCards.standard, S2.Routers.standard)
```

Figure 2: Source Schemas and Web Services

views over those schemas [11][1]. Figure 2 illustrates part of their respective schema and the signatures of four web services they export.

The Dell schema describes computers that are characterized by their cid, CPU model (e.g., P4), RAM installed and price, and have a set of network cards installed. Each network card has the cid of the computer it is installed in, accommodates a specific data rate (e.g., 54Mbps), implements a standards (e.g., IEEE 802.11g) and communicates with a computer via a particular interface (e.g., USB). The web service *ComByCpu* returns the computers of a given *cpu*. (We assume there is a *Computer* type.) The service *ComNetByCpuRate* provides computers of a given *cpu* that have installed network cards of a given data *rate*. The Cisco source describes routers that also accommodate a specific data rate, implement standards, have their own price and are of a particular type. The *RoutersWired* and *RoutersWireless* services return routers that are of either wired or wireless type respectively.

In Figure 1, a user builds the computer shopping portal by formulating queries against the source schemas, and deploys a mediator in order to execute queries against the exported web services during run-time. The mediator can answer the query "return all P4 computers with a 54Mbps network card and the compatible wireless routers" by combining the answers of web service calls *CompNetByCpuRate* and *RoutersWired*. However, it cannot answer the query "return all computers with 1GB of RAM". The reader is pointed to Chapter 20.3 of [7] for similar examples. CLIDE appropriately guides the user toward the formulation of feasible queries by employing the following coloring scheme:

- *Red* color indicates actions that lead to unsupported queries, regardless of what is included next. For example, conditioning the `type` column of `Routers` with a constant other than 'Wired' and 'Wireless' leads to unsupported queries.
- *Yellow* color indicates actions that are necessary for the formulation of a feasible query. For example, conditioning the `cpu` of `Computers` will be yellow since all queries that the mediator can answer and involve the `Computers` table require a given cpu.
- *Blue* color indicates a set of actions where at least one of them is required to be taken in order to reach one of the next feasible queries. Notice that one can choose among many blue options. For example, after the `cpu` of `Computers` has been conditioned and a feasible query has been reached, one should condition either the `ram` or the `price` column (among other choices) in order to reach the next feasible query.
- *White* color indicates selection conditions, tables and projections whose participation in the query is optional.

CLIDE is based on a modular architecture consisting of the front-end and a back-end that enables the front-end's behavior by deciding the color of each action. The above coloring scheme is implemented by CLIDE's front-end and is independent of the specification that has been used to describe the set of supported queries. Multiple back-ends are possible, depending on the nature of the specification of the supported query set. For example, one could have a back-end for the P3P privacy-related supported query set specification of [9] or the specification of [23] that is related to queries supported as a result of wrapping web forms.

Parameterized Views One of the most common and, at the same time, most challenging back-ends relates to the case where the set of directly supported queries are described using parameterized views, a technique that has been used to describe content and access methods in the widely used Global-as-View (GaV) integration architectures [8], and also recently to describe privacy constraints in [17]. Going back to Figure 2, the parameterized view V_1 corresponds to the web service *ComByCpu*. Notice that the parameterized view not only indicates the input (cpu) and output ($Computer$) of the service, but also indicates how the input and output are semantically related with respect to the underlying database. Typically the sources considered by the mediator can be too many to individually browse in order to formulate a feasible query.

Deciding whether a given query is feasible or not is a query rewriting problem: The mediator is given a query q over a database D and a set of parameterized views $V_1, \ldots, V_n$ and it searches for a plan (if any) that combines the views and computes $q(D)$. The plan is typically in the form of a query $q'(V_1(D), \ldots, V_n(D))$ that runs on the views and often incorporates primitives that indicate the passing of information across sources and web services.

Several rewriting algorithms have been published; the reader is referred to the survey [8]. However, these algorithms are not sufficient for CLIDE's back-end since whenever there is no plan they only declare that the query cannot be answered. Some algorithms return overestimate or underestimate approximations of the query result, thus addressing a different goal than the one in our setting where the developer needs to know the exact queries that can be issued and program accordingly. Nevertheless, there are important technical connections between those algorithms and our work that are discussed in later sections.

1.1 Contributions

Formal Guarantees on the Interaction Any good interface that guides the user toward some action must be comprehensive (complete) and, at the same time, avoid overloading the user with information at every step [13, 20]. CLIDE achieves both goals since it satisfies the following guarantees at every step of the interaction:

[1]Indeed, it is often the case that web services are based on parameterized queries over databases. However, for the purposes of mediation it is not necessary to assume that the Dell and Cisco schemas are known.

1. *Completeness*: Every feasible query can be built by following suggested actions only.
2. *Minimality*: The minimal set of actions that preserves completeness is suggested.
3. *Rapid convergence*: The shortest sequence of actions from a query to any feasible query consists of suggested actions.

Interaction sessions between the user and the CLIDE front-end are formalized using an *Interaction Graph*, which models the queries as nodes and the actions that the user performs as edges. Consequently, the color of each action is formally defined as a property of the set of paths that include the action and lead to feasible queries. Then the above guarantees are formally expressed as graph properties.

Back-End Algorithms The challenge facing the CLIDE back-end is that the coloring properties cannot be trivially turned into an algorithm since they require the enumeration of an infinite number of feasible queries. Note that the number of queries is infinite for two reasons. First, there is an infinite number of constants that may be used. We tackle this problem by considering parameterized queries (similar to JDBC's prepared statements) where each one stands for infinitely many queries. Still, the number of parameterized queries is infinite, because the size of the `FROM` clause is unlimited, which then leads to unlimited size `SELECT` and `WHERE` clauses.

We describe a set of algorithms that find a finite set of *closest feasible queries*, related to the current query, and determine the coloring by inspecting it. For our purpose, we leverage prior algorithms and implementations for finding exact and maximally-contained rewritings [14, 6, 16]. However, we needed to significantly optimize and extend current implementations in order to achieve online performance and to ensure that the produced maximally-contained queries are syntactic extensions of the current query, hence enabling the color algorithm. We provide a set of experiments that illustrate the class of queries and views CLIDE can handle, while maintaining on-line response.

CLIDE Demo We implemented the CLIDE front-end and the back-end algorithms which are available as an on-line demonstration at `http://www.clide.info`.

Paper Outline Section 2 provides definitions and notation conventions. Section 3 discusses query building interfaces, focusing on CLIDE-related issues, and introduces the interaction graph, which allows us to formally define their behavior. Section 4 discusses the aspects of CLIDE that pertain to interaction in the presence of a limited set of feasible queries. Section 5 describes the algorithms of the CLIDE back-end. Section 6 describes the implementation and optimizations, which are experimentally evaluated in Section 7. Section 8 presents related work and discusses CLIDE's applicability to other settings. Section 9 concludes the paper.

2. DEFINITIONS AND NOTATIONS

The CLIDE front-end formulates queries from the set of conjunctive SQL queries with equality predicates $\mathrm{CQ}^{=}$ under set semantics. The `FROM` clause consists of *table atoms* `R r`, where `R` is some table name and `r` an alias. The `SELECT` clause consists of the SQL keyword `DISTINCT` and *projection atoms* `r.x`, where `x` is a column of `r`. The `WHERE` clause is a conjunction of *selection atoms* and *join atoms*. *Constant* selection atoms are of the form `r.x=`*constant*, where `r` is some alias and `x` some column, while *parameterized* selection atoms are of the form `r.x=`*parameter*. Obviously, at most one selection atom for each alias-column pair can appear in the `WHERE` clause. Join atoms are of the form `r.x=s.y`. We define the *empty query* to have no table, join, selection or projection atom.

Column associations identify pairs of columns, within a source or across sources, whose join is meaningful. Figure 2 illustrates the association of the `cid` columns of `Computers` and `NetCards` and the `rate` and `standard` columns of `NetCards` and `Routers`[2]. The user can configure CLIDE to suggest either arbitrary joins or only joins between columns that are associated, in order to reduce the number of suggestions displayed to the user. In the latter case, the user still has the option to formulate joins between non-associated columns, but the CLIDE front-end will not suggest them. For the rest of the presentation, we assume the user has configured CLIDE to suggest joins between associated columns only. We denote this class of queries with $\mathrm{CQ}^{=CA}$.

The views that CLIDE takes as input are from the set of parameterized conjunctive SQL queries $\mathrm{CQ}^{=P}$, where *parameterized* selection atoms of the form `r.x=`*parameter* appear in the `WHERE` clause. We assume that all joins are between associated columns. $\mathrm{CQ}^{=CA}$ is a subset of $\mathrm{CQ}^{=P}$.

Two queries q_1 and q_2 are *syntactically isomorphic*, denoted by $q_1 \cong q_2$, if they are identical modulo table alias renaming. Syntactic isomorphism is important since the users of query writing tools typically do not have control (or do not care to control) the exact table alias names.

We denote the set of feasible queries by $FQ \subseteq \mathrm{CQ}^{=CA}$. As in [16], we define the feasible queries given a set of views $\mathcal{V} = V_1, \ldots, V_k \in \mathrm{CQ}^{=P}$ over a fixed schema D, to be the set of queries $q_{F1}, \ldots, q_{Fm} \in \mathrm{CQ}^{=CA}$ over D that have an equivalent $\mathrm{CQ}^{=}$ rewriting using $\mathcal{V}$. In the absence of parameters a rewriting is simply a query that refers exclusively to the views. In the presence of parameters we need to also ensure that there is a viable order of passing parameter bindings across the views of the rewriting [16, 18]. We capture this requirement as follows: First associate to each view a schema that includes both the columns that the view returns and the columns that participate in parameterized selections (even if they are not returned). Then we associate with each view schema a *binding pattern* that annotates every column that participates in a parameterized selections as *bound*, which is denoted by a '*b*' superscript, and every other column as *free*, denoted by an '*f*' superscript. For example, we associate the following schema and binding pattern to V_1 in Figure 2:

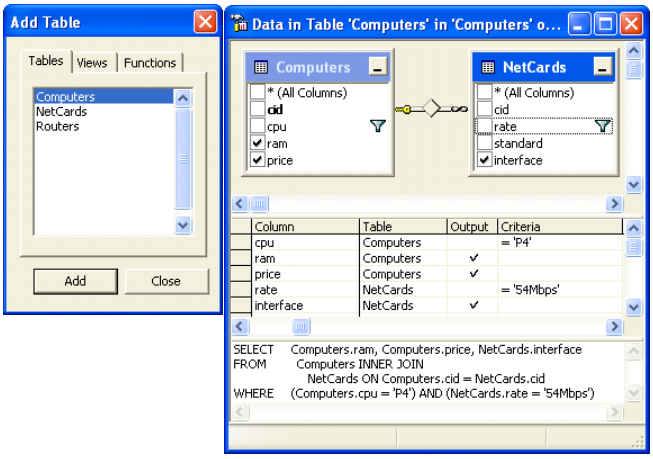
$$V_1(\texttt{cid}^f, \texttt{cpu}^b, \texttt{ram}^f, \texttt{price}^f)$$

A *valid* rewriting is a query that refers to the views only and there is an order $V_1, \ldots, V_n$ of the used views such that if a column `x` is bound in V_i then either there is a selection atom V_i`.x=`*constant* or a join atom V_i`.x=`V_j`.y` where $j < i$.

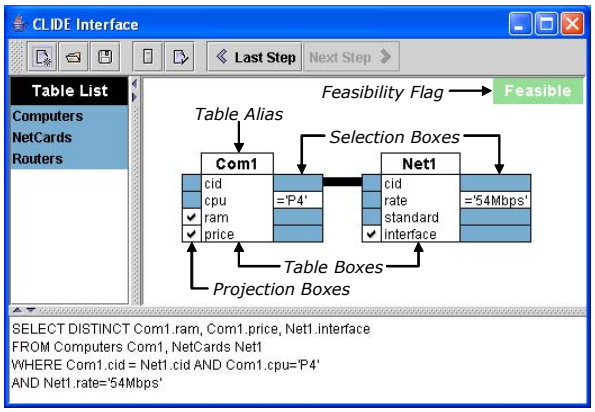
3. QUERY BUILDING INTERFACES

The CLIDE front-end is a QBE-like [24] graphical interface. It adopts Microsoft's Query Builder interface [1] as the basis for the interactive query formulation, since users are very familiar with it. Figure 3a shows a snapshot of Microsoft's Query Builder, where the user formulates a query over the schemas of Figure 2. The top pane displays the join of the `Computers` table with the `NetCards` table on `cid` and the projection of the `ram` and `price` columns of `Computers` and of the `interface` column of `NetCards`. The middle pane shows selections that set `cpu` equal to 'P4' and `rate` equal to '54Mbps', and the bottom pane displays the corresponding SQL expression. The user can add to the top pane tables from the

[2]Column associations can be explicitly declared by the mediator owner. They can also be derived from the pairs of type-compatible columns, from foreign-key constraints, the join atoms in the views, or any of the recently proposed schema matching techniques [15].



(a) Microsoft's Query Builder



(b) CLIDE's Front-End expressing the query of Figure 3a

Figure 3: QBE-Like Query Building Interfaces

list shown on the left. The user can also formulate joins, like the one on `cid`.

Figure 3b provides a snapshot of CLIDE's front-end for the query of Figure 3a. Apart from the feasibility flag and the coloring, the correspondence with Microsoft's Query Builder is straightforward: CLIDE displays a *table box* for each table alias in the `FROM` clause. Selections on columns are displayed in *selection boxes*. Columns are projected using check boxes, called *projection boxes*. Joins are displayed as solid lines, called *join lines*, connecting the respective column names. The list of available tables is shown in a separate pane. Also shown is the SQL statement that the interface graphically represents. The "Last Step" and "Next Step" buttons allow the user to navigate into the history of queries formulated during the interaction.

The user builds $\mathrm{CQ}^{=CA}$ queries with the following visual actions:

1. *Table action*: Drag a table name from the table list and drop it in the main pane. The interface draws a new table box with a fresh table alias and adds a table atom to the `FROM` clause of the SQL statement.
2. *Selection action*: Typing a constant in a selection box results to adding a selection atom to the `WHERE` clause.
3. *Join action*: Dragging a column name and dropping it on another one results to a join line connecting the two column names and a new join atom in the `WHERE` clause.
4. *Projection action*: Checking a projection box adds a projection atom to the `SELECT` clause.

4. CLIDE INTERACTION IN THE PRESENCE OF LIMITED ACCESS METHODS

When not all $\mathrm{CQ}^{=CA}$ queries against a database schema are feasible, CLIDE guides the user toward formulating feasible queries by coloring the possible next actions in a way that indicates what has to be done, what may and what cannot be done. Table actions are suggested by coloring the background of table names in the table list. Selections and projections are suggested by coloring the background of their boxes. Joins are suggested by coloring join lines.

We illustrate the color scheme using the interaction session of Figure 4, which refers to the running example of Figure 2. The user wants to formulate a query that returns computers that meet various selection conditions, including conditions about network cards and routers - as long as those conditions are supported. Figure 4 shows snapshots of the interaction session, where CLIDE's color scheme suggests, at each interaction step, which actions lead to a feasible query.

Required and Optional Actions Consider the query that the user has formulated in Snapshot 1. The interface indicates that this query is infeasible (see flag at top right) and that every feasible query that extends it must have a selection on `cpu`. The latter indication is given by coloring *yellow* (light gray on a B/W printout) the `cpu` selection box. The rest of the selection boxes and projection boxes are *white* suggesting that these actions are *optional*, i.e., feasible queries can be formulated with or without these actions being performed.

So the user performs the yellow selection on `cpu` by typing a constant in the selection box. This leads to the feasible query of Snapshot 2. This query is feasible since the mediator can run view V_1 with the parameter instantiated to 'P4' and then project out the `cid` and `cpu` columns.

Required Choice among multiple Actions The user may terminate the interaction session and incorporate the query of Snapshot 2 in her application or may continue to extend the query. The interface indicates that, in order to reach a next feasible query, at least one of the `NetCards`, `Routers` or (an additional) `Computers` tables has to be included in the query, among other options. The indication is provided by coloring the corresponding names in the table list *blue* (medium gray). Each given blue atom, say `NetCards`, does not appear in all feasible queries that extend the current query. If it did appear in all, then it would be yellow (i.e., required).

Non-Obvious Feasible Queries Snapshot 3 presents a complex case, where the interface's color scheme informs the user about non-obvious feasible queries. After the user introduces a `NetCards` table, the interface suggests that one of the following extensions to the query is required: The join line between the `cid`'s of `Computers` and `NetCards` is suggested since it leads to the formulation of view V_2. It is blue since the user has more options: She can introduce a second copy of `Computers`, say `Com2`, which will lead toward the feasible query that joins `Networks` with `Com2`, selects on `rate` and takes a Cartesian product with `Com1`. If Cartesian product queries are of no interest to the user, she can set an option to have CLIDE ignore them. In such case the `cid` join would be a required (yellow) extension. For the remainder of the example, we assume that this option is set.

The user has another pair of options at Snapshot 3. She can perform the blue `rate` selection, which leads to the formulation of view V_2. Alternatively, she may introduce a `Routers` table and join the `rate` columns of `NetCards` and `Routers`, thus

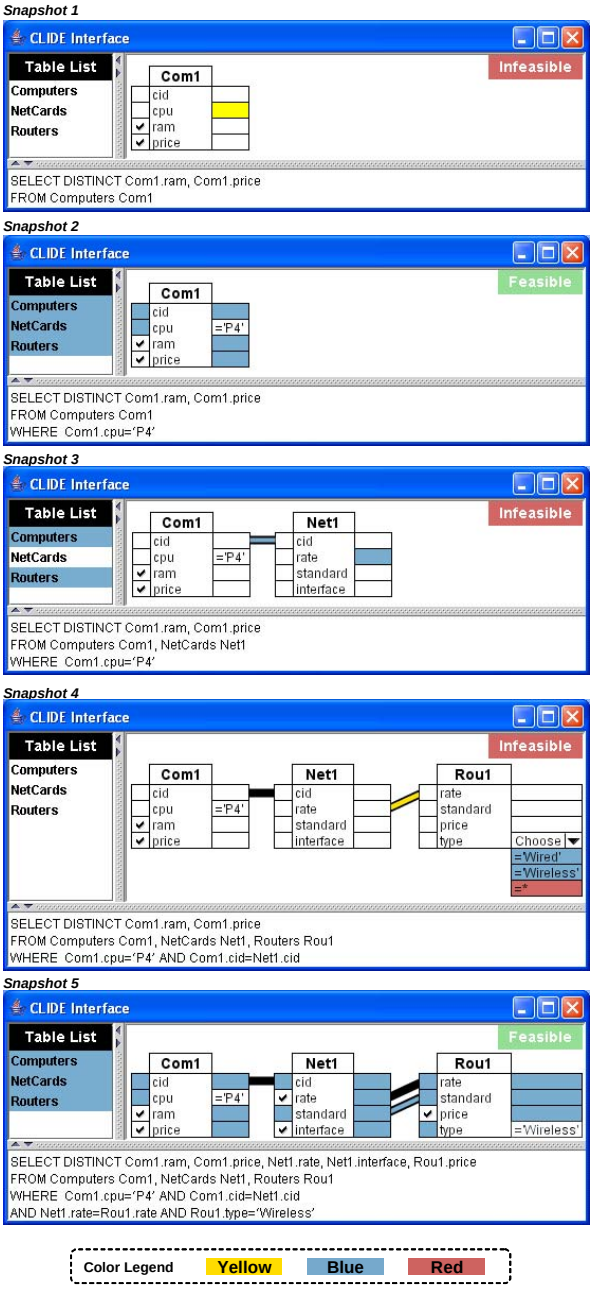


Figure 4: Snapshots of an Interaction Session

instantiating the `rate` parameter of V_2 with constants provided by another table.

Selection Options In Snapshot 4, the user has performed the suggested join and introduced a `Routers` table. Now the `Routers.type` column needs to be bounded and the interface presents to the user a drop-down list that explains which constants may be chosen. She can either choose 'Wired' or 'Wireless'. The symbol `*` denotes any other constant and is colored *red* (dark gray) to indicate that no feasible query can be formulated if she chooses this option. Note that the options of a drop-down list can have different colors. If there were only one constant that she could choose, then this option would be yellow. In the special case where any constant can be chosen, then no drop-down list is shown, as in the case of the `cpu` selection box in Snapshot 1.

The front-end can also be configured to hide all red actions, including columns with red selection and projection boxes. Note that a red selection box implies a red projection box and vice versa. So the front-end can remove the column from the corresponding table box altogether.

In the next steps, the user performs the suggested join, chooses the 'Wireless' constant and checks several projection boxes. Snapshot 5 shows the new query, which is feasible. The mediator plan that implements this query first accesses view V_4, then for each `rate` returned accesses view V_2 with its parameters instantiated to 'P4' and the given `rate`, and finally performs the necessary projections.

The CLIDE front-end displays only yellow and blue join lines. Red and white join lines are typically too many and are not displayed. If the user wants to perform a join other than the ones suggested, she has to follow a trial-and-error procedure.

Note that unchecked projection boxes can be either blue, white or red. A projection box cannot be yellow, because if there is a feasible query that has the corresponding projection atom in the `SELECT` clause, then the query formulated by removing this atom is also feasible.

Finally, if the user performs a red action, then all boxes, lines and items in the table list are colored red, indicating that the user has reached a dead end, i.e., no feasible query can be formulated by performing more actions and it is necessary to backtrack, i.e., undo actions.

4.1 Specification of CLIDE's Color Scheme

Interaction sessions between the user and the CLIDE front-end are formalized by an *Interaction Graph*. The nodes of the interaction graph correspond to $\mathrm{CQ}^{=CA}$ queries and the edges to actions.

DEFINITION 4.1. *(**Interaction Graph**) Given a database schema D and a set of $\mathrm{CQ}^{=P}$ views $\mathcal{V}$ over D, an interaction graph is a rooted DAG $G_I = (N, s, E)$ with labeled nodes N and labeled edges E such that:*

- *For every query $q \in \mathrm{CQ}^{=CA}$ over D there is exactly one node $n \in N$ whose label $q(n)$ is syntactically isomorphic to q. We call n feasible if $q(n)$ is feasible.*
- *s is the root node and is labeled with the empty query.*
- *Every edge $e(n \xrightarrow{\mathtt{a}} n') \in E$ is labeled with an action `a` which yields a query that is isomorphic to $q(n')$ when applied to $q(n)$. `a` is either a table, a projection, a join, a* specific *selection of the form* `r.x=`constant, *or a* generic *selection of the form* `r.x=*`. *Here `*` denotes* any constant *other than the ones that appear in specific selections and label edges originating from n.*

◇

Figure 5 shows part of an interaction graph for the schemas in Figure 2, where nodes n_1 to n_5 correspond to the queries formulated in Snapshots 1 to 5 of Figure 4. Notice that there are multiple interaction graphs that correspond to a given schema, since each node n can be relabeled with any of the queries that are syntactically isomorphic to $q(n)$, i.e., with any query that uses other alias names. CLIDE considers a single interaction graph by controlling the generation of aliases. By convention, the generated aliases follow the lexical pattern `Ti` where `T` is the first three letters from the name of the table (for illustration purposes) and `i` is a number that is sequentially produced.

Figure 5 indicates feasible queries by green (shaded) nodes. The root s is indicated by a hollow node. The outgoing edges of a node n capture all possible actions that the user can perform on

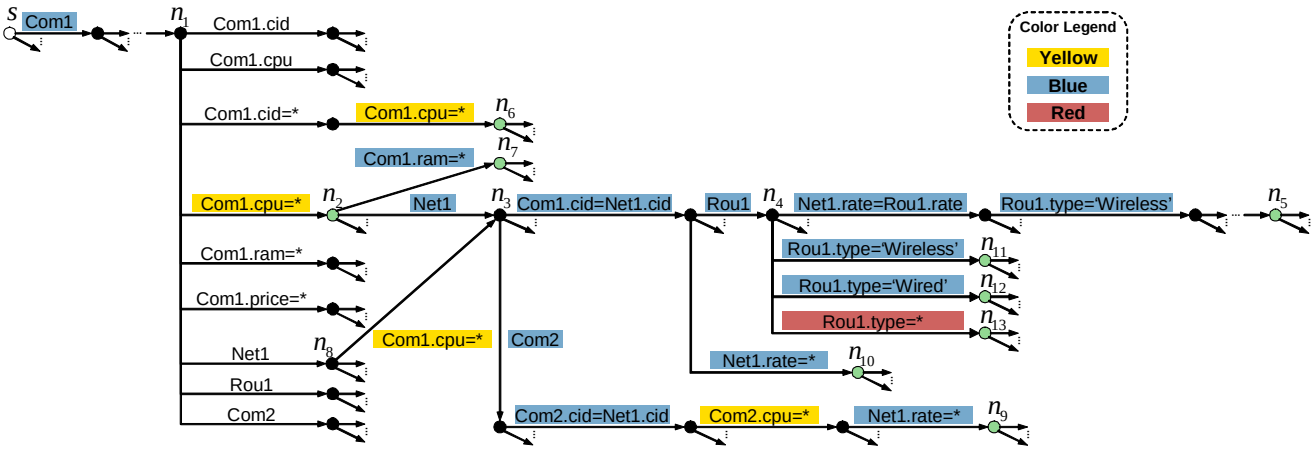


Figure 5: Part of an Interaction Graph

$q(n)$. These are the actions that the front-end colors and they are finitely many. Even though there are infinitely many constants that can potentially generate infinitely many selections for a given column, they are aggregated by the `*` symbol. In Figure 5, for example, the `*` in the selection `Com1.cpu=*` labeling an outgoing edge of n_1 aggregates all possible constants. The `*` in the selection `Rou1.type=*` labeling an outgoing edge of n_4 denotes all constants except 'Wired' and 'Wireless', because corresponding selections label adjacent edges.

For a query $q(n)$, the coloring rules are formally expressed as a coloring of the actions labeling outgoing edges of node n.

DEFINITION 4.2. *(**Colors**) Given an interaction graph $G_I = (N, s, E)$, a node $n \in N$ and an outgoing edge $e(n \xrightarrow{\mathtt{a}} m)$, the action* a *is colored:*

- Yellow *(Required) if every path p_i from n to a feasible node n_F contains an edge labeled with* a.
- Blue *(At Least One Required) if (i)* a *is not yellow, (ii) at least one path p_i from n to a feasible node n_F contains an edge labeled with* a, *and (iii) there is no path from n to n_F that contains a feasible node, excluding n and n_F.*
- Red *(Forbidden) if there is no path from n to a feasible node that contains an edge labeled with* a.
- White *(Optional) if not colored otherwise.*

◇

We say that actions colored yellow or blue are called *suggested*. The same action may have different color at various points in the interaction. For example, table action `NetCards Net1` is white when it labels an outgoing edge of n_1 and blue when it labels an outgoing edge of n_2.

CLIDE assigns colors according to Definition 4.2 and features the following characteristics of desirable behavior.

1. **Completeness of Suggestions** Every feasible query can be formulated by starting from the empty query and at every interaction step picking only among blue and yellow actions.
2. **Minimality of Suggestions** At every step, only a minimal number of actions, which are needed to preserve completeness, are suggested as required. Equivalently, for each blue or yellow action a, there is at least one feasible query toward which no progress can be made without picking a.
3. **Rapid Convergence by Following Suggestions** Assume that the user is at node n of the interaction graph and consequently follows a path p consisting of yellow and blue edges until she reaches feasible query $q(n')$. It is guaranteed that there is no path p' that is shorter than p and also leads from n to n'.

5. THE CLIDE BACK-END

The CLIDE back-end is invoked every time the interaction arrives at a node n in the interaction graph. It takes as input the query $q(n)$, the schemas and the views exported by the sources, and the set of column associations. The back-end partitions the set of possible actions, which label outgoing edges of n, into sets of blue, red, white and yellow suggested actions. It also decides if $q(n)$ is feasible or not.

The first challenge in determining the partition is that the color definitions make statements about all possible extensions of the current query, i.e., all feasible nodes reachable from n. These correspond to an infinite set of infinitely long paths in the interaction graph. Hence, the color definitions cannot be trivially translated into an algorithm.

We show that at each interaction step, it is sufficient to consider only a representative subgraph of the interaction graph to color the possible actions either blue or yellow. This subgraph consists of n, the feasible nodes that are closest to n, and the paths connecting n to these feasible nodes. The closest feasible nodes are labeled with queries in $FQ_C(n)$ which is defined below.

DEFINITION 5.1. *(**Closest Feasible Queries** FQ_C) Given an interaction graph $G_I = (N, s, E)$ and a node $n \in N$, the set of closest feasible queries $FQ_C(n)$ are the ones that label feasible nodes n_F reachable from n such that there is no path p from n to n_F that contains a feasible node, excluding the endpoints of p.* ◇

Section 5.1 presents the computation of $FQ_C(n)$ when parameterized selection atoms do not appear in the views. We show that $FQ_C(n)$ is finite and present optimizations for computing it, which proved crucial to CLIDE's usability. If parameterized selection atoms appear in the views, then $FQ_C(n)$ is infinite. Section 5.3 shows that CLIDE's back-end faces this additional challenge without compromising any of the formal guarantees by computing a finite representative set of *seed queries* $SQ(n)$.

The second challenge (Section 5.2) that the back-end faces is to efficiently color the possible actions given the set of closest feasible queries. Even though coloring an action yellow or blue is straightforward and inexpensive, coloring the remaining actions red

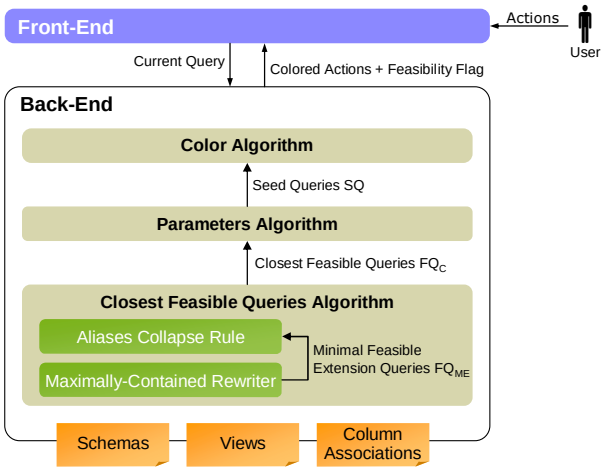


Figure 6: CLIDE Architecture

or white using a brute force algorithm leads to significant performance overhead.

Figure 6 shows the architecture of the CLIDE system implementation. Currently, the system parses the schemas, view definitions and column associations from corresponding text files. The *Closest Feasible Queries Algorithm* computes the set $FQ_C(n)$ and implements the algorithm of Section 5.1. When parameterized selection atoms do not appear in the views, the *Color Algorithm* component inputs the set $FQ_C(n)$ and implements the algorithm of Section 5.2. When parameterized selection atoms appear in the views, the *Color Algorithm* component inputs a set of *seed queries* $SQ(n)$ produced by the *Parameters Algorithm* component described in Section 5.3.

5.1 Closest Feasible Queries Algorithm

The search for closest feasible queries faces an infinite search space, namely all possible extensions of the current query. We limit this space to a finite one, corresponding to nodes in the interaction graph that are within a bounded distance from n. Then, we present an efficient method for enumerating $FQ_C(n)$ without exploring the whole search space.

Maximally-Contained Feasible Queries Intuitively, as the user syntactically extends the current query with new tables, selections and joins, she creates queries which are contained in the initial one. It is therefore a natural starting point to search for the closest feasible queries among the contained and feasible ones. We can further focus on the maximally-contained [8] and feasible queries since they are the least constraining (semantically) and hence they have the least number of additional tables, selections and joins. As in [8], the set of maximally-contained feasible queries is formally defined as the set of queries such that

1. for each maximally-contained query q_1, q_1 is feasible and contained in $q(n)$ ($q_1 \sqsubseteq q(n)$),
2. for each maximally-contained query q_1 and any feasible query $q_1' \sqsubseteq q(n)$, if q_1' contains q_1, then q_1' is equivalent to q_1, and
3. for each feasible query $q_1 \sqsubseteq q(n)$ there exists a maximally-contained query q_2 such that $q_1 \sqsubseteq q_2$.

Among the maximally-contained feasible queries, we focus on the ones which are *minimal syntactic extensions* of $q(n)$, in the sense that dropping any table, selection or join compromises feasibility or containment in $q(n)$ or the property of syntactically extending $q(n)$. We denote this set as $FQ_{ME}(n)$. Section 6 describes how we extended one of the several maximally-contained rewriting algorithms proposed in the literature [8] to obtain $FQ_{ME}(n)$.

$FQ_{ME}(n)$ is known to be finite if we restrict $q(n)$ and the views to conjunctive queries with constant selection atoms [8].

LEMMA 5.1. *All minimal feasible extensions of $q(n)$ which are maximally-contained are also closest feasible queries ($FQ_{ME}(n) \subseteq FQ_C(n)$).* ◇

However, there are closest feasible queries that are not in $FQ_{ME}(n)$, as the next example shows, and we will have to find them.

EXAMPLE 5.1. *Assume that views V_1 and V_2 of Figure 2 are replaced by the following views V_1' and V_2', respectively, which contain constant selections only.*

```
SELECT DISTINCT Com.*                                          (V1')
FROM Computers Com
WHERE Com.cpu='P4'

SELECT DISTINCT Com.*, Net.*                                   (V2')
FROM Computers Com, Network Net
WHERE Com.cid=Net.cid AND Com.cpu='P4'
AND Net.rate='54Mbps'
```

If the current query is $q(n_3)$ in Figure 5 (Snapshot 3 in Figure 4), then the only query in $FQ_{ME}(n_3)$ is $q(n_9)$ given below.

```
SELECT DISTINCT Com1.ram, Com1.price                           q(n9)
FROM Computers Com1, Computers Com2, NetCards Net1
WHERE Com2.cid = Net1.cid AND Com1.cpu='P4'
AND Com2.cpu='P4' AND Net1.rate='54Mbps'
```

Note that $q(n_{10})$ is also a closest feasible query to $q(n_3)$, but it is not in $FQ_{ME}(n_3)$ since it is contained in $q(n_9)$.

```
SELECT DISTINCT Com1.ram, Com1.price                           q(n10)
FROM Computers Com1, NetCards Net1
WHERE Com1.cid = Net1.cid AND Com1.cpu='P4'
AND Net1.rate='54Mbps'
```

Intuitively, one can extend $q(n_9)$ with joins until the `Com2` *alias "collapses" into* `Com1`*, leading to a closer query, reachable from $q(n_3)$ and clearly contained in $q(n_9)$ due to the added joins.* ◇

Even though $FQ_{ME}(n)$ does not give us the set of closest feasible queries, we can use it to bound the search space for $FQ_C(n)$. Theorem 5.1 below states that all queries in $FQ_C(n)$ correspond to nodes located within a bounded distance from n.

THEOREM 5.1. *Given a node n in the interaction graph and the set $FQ_{ME}(n)$, if p_L is the longest path from n to a node labeled with a query in $FQ_{ME}(n)$, then all nodes labeled with queries in $FQ_C(n)$ are reachable from n via a path p, where $|p| \leq |p_L|$.* ◇

Theorem 5.1 enables a brute force algorithm for computing $FQ_C(n)$: (i) compute $FQ_{ME}(n)$, (ii) compute the bounded distance $|p_L|$ as the length of the longest path from n to some query in $FQ_{ME}(n)$, (iii) enumerate the set of queries $B(n, |p_L|)$ reachable from $q(n)$ by systematically applying up to $|p_L|$ actions in all possible ways, and (iv) return all feasible queries from $B(n, |p_L|)$.

This algorithm computes $FQ_C(n)$, but is highly inefficient. In the worst case, it enumerates all paths of length $|p_L|$. The following observations allow us to prune this search dramatically, by starting from $FQ_{ME}(n)$.

Alias Collapse Rule We can compute $FQ_C(n) \setminus FQ_{ME}(n)$ starting from the queries in $FQ_{ME}(n)$ and rewriting them using the *alias collapse rule*, which rewrites a query q into a query q' as follows: pick a pair of table atoms sharing the same relation name, say `R R1`, `R R2`, and rename `R2` with `R1` in q, to obtain q'.

EXAMPLE 5.2. *One can obtain the closest feasible query $q(n_{10})$ from query $q(n_9)$ by collapsing the aliases* `Com1` *and* `Com2`. ◇

Notice that indiscriminate application of the collapse rule can lead to unsatisfiable queries. To see this, assume that q contains the selection conditions `R1.x='5'` and `R2.x='3'`. After collapsing `R1` and `R2`, q' contains the inconsistent selection conditions `R1.x='5'` and `R1.x='3'`. We apply the alias collapse rule only if they lead to satisfiable queries.

LEMMA 5.2. *For any $q_1 \in FQ_C(n) \setminus FQ_{ME}(n)$, there exists $q_2 \in FQ_{ME}(n)$ such that q_1 is obtained from q_2 by repeatedly applying the alias collapse rule.* ◇

Lemmas 5.1 and 5.2 lead to the following algorithm for computing $FQ_C(n)$.

algorithm QuickFQ_C
Input: node n
Output: $FQ_C(n)$

begin
compute $M := FQ_{ME}(n)$ using an algorithm for finding maximally-contained rewritings, extended to produce minimal syntactic extensions of $q(n)$

// compute $FQ_C(n) \setminus FQ_{ME}(n)$ in AC:
let $AC :=$ the empty set $\emptyset$
for each $q_M \in M$ do
  for each pair of distinct aliases $\mathtt{r}_1$,$\mathtt{r}_2$ of some relation in q_M do
    let q := collapse $\mathtt{r}_1$ and $\mathtt{r}_2$ in q_M
    $AC := \mathbf{CollapseToFeasible}(q, AC)$
return $M \cup AC$
end

procedure CollapseToFeasible
Input: query q, query set AC
Output:all feasible queries obtainable from q by collapsing aliases
begin
if q is unsatisfiable return the empty set $\emptyset$

if q is feasible and not contained in any $q_i \in AC$
  $AC := AC \cup \{q\}$
for each pair of distinct aliases $\mathtt{r}_1$,$\mathtt{r}_2$ of some relation in q do
  let q' := collapse $\mathtt{r}_1$ and $\mathtt{r}_2$ in q
  $AC := \mathbf{CollapseToFeasible}(q', AC)$
return AC
end

THEOREM 5.2. **Quick**FQ_C *computes* $FQ_C(n)$. ◇

5.2 Color Algorithm

After computing the set of closest feasible queries $FQ_C(n)$, CLIDE decides if the current query is feasible or not, and then colors all possible actions that the user can perform next. The current query is feasible if it is a closest feasible one, i.e., $q(n) \in FQ_C(n)$, and infeasible otherwise. We first present the algorithm for finding the yellow and blue actions when the current query is infeasible. We deal with the white and red actions, as well as the feasible case, next.

Blue and Yellow Instead of working with the infinite interaction graph, we can restrict our attention to the finite *close subgraph* consisting of n, all closest feasible nodes labeled with the closest feasible queries in $FQ_C(n)$ and the paths between them. Then we have:

LEMMA 5.3. *For an infeasible current query $q(n)$, and for every action* `a` *applicable to $q(n)$,* `a` *is colored yellow (blue) with respect to the interaction graph if and only if* `a` *is colored yellow (blue) with respect to the close subgraph of n.* ◇

At this point, it is easy and more efficient to color the actions without actually materializing the close subgraph. We color a join `a` yellow if it appears in all closest feasible queries, and blue if it appears in some. In the case of a table action `T`, we color it yellow (resp. blue) if in all (resp. some) closest feasible queries there exists a table atom `T Tj`, such that `T Ti` and `T Tj` do not necessarily refer to the same alias, and `T Tj` does not appear in the current query.

Specific selections, i.e., selections of the form `r.x=`*constant*, are colored either yellow or blue the same way joins are colored. The front-end displays these actions in the corresponding selection box as options of a drop-down list. *Generic selections* of the form `r.x=*` and projections cannot be colored blue or yellow when the current query is infeasible, because for each feasible query they participate in, there is another feasible query that can be formulated without performing them. Conversely, when the current query is infeasible, performing a projection or a generic selection that does not appear in the views will not yield a feasible query[3].

White and Red Any remaining actions are either white or red. For each such action `a`, a brute force approach would add `a` to the current query, thus yielding query $q(n')$, and then test if $FQ_C(n')$ is empty. If so, `a` is colored red, otherwise white. This approach, although simple, requires the non-emptiness test of $FQ_C(n')$, which is an expensive operation, as the experiments of Section 7 demonstrate. Hence, we need to devise more efficient techniques for coloring red and white actions.

In the case of table actions we color red the ones that are not used in any view, and white the remaining ones, since a feasible query q_F can lead to another feasible query that takes the Cartesian product of q_F and the view that contains the table in question.

For the case of projections and selections, we attach a *maximum projection list* to every closest feasible query $q_F \in FQ_C(n)$. A maximum projection list consists of all projections that can be added to q_F, in addition to the ones already in the current query, without compromising feasibility. For example, if we add all possible projections to $q(n_9)$ of Example 5.1, while preserving feasibility, then we formulate the following query $q'(n_9)$:

```
SELECT DISTINCT                                          q'(n9)
       Com1.cid, Com1.cpu, Com1.ram, Com1.price
       Com2.cid, Com2.cpu, Com2.ram, Com2.price
       Net1.cid, Net1.rate, Net1.standard, Net1.interface
FROM Computers Com1, Computers Com2, NetCards Net1
WHERE Com2.cid = Net1.cid AND Com1.cpu='P4'
AND Com2.cpu='P4' AND Net1.rate='54Mbps'
```

Hence, the maximum projection list of $q(n_9)$ consists of all projections in $q'(n_9)$ except `Com1.ram` and `Com1.price` which appear in $q(n_9)$. In Section 6 we show how we extended a maximally-contained rewriting algorithm to generate these lists in linear time.

Once we compute the maximum projection lists, we color a projection red if it does not appear in any list. Generic selections are colored red if the projection `r.x` is red. These selections are also shown as options of the corresponding drop-down lists. In the special case where no specific selections exist, then no drop-down list is displayed and the selection box is colored according to the color of the generic selection.

Any remaining actions are colored white. Note that specific selections can never be colored white or red. The CLIDE front-end does not display white and red joins, so they are not a consideration.

Feasible Current Query If the current query is feasible, we use the same algorithm, but we color all non-red actions blue, as each one leads to a new feasible query, not obtainable via other actions.

[3]Note that generic selections can be colored yellow or blue when parameterized selections appear in the views. Please see Section 5.3 for details.

5.3 Parameters

When parameterized selection atoms appear in the views, the algorithms in Sections 5.1 and 5.2 need to be extended, because the set of closest feasible queries becomes infinite. The following example illustrates this point.

EXAMPLE 5.3. *Assume the following employees and managers source schema. The exported parameterized view V_5 returns the* `mid` *of an employee's manager, given the employee's* `eid`*. V_6 returns the* `salary` *of a manager, given the manager's* `mid`*. Note that the source schema is recursive, i.e., an employee has a manager, but a manager is also an employee, who has a manager. One of the column associations we consider witnesses this recursion.*

```
Empls(eid, mid)                                        (Schema)
Mngrs(mid, salary)

EmplsMngrs(eid) → (Employee)*
SELECT DISTINCT E1.*                                   (V5)
FROM Empls E1
WHERE E1.eid=eid

MngrsSalary(mid) → (Manager)*
SELECT DISTINCT M1.*                                   (V6)
FROM Mngrs M1
WHERE M1.Mid=mid

(S1.Empls.eid, S1.Empls.mid)                           (Column Associations)
(S1.Empls.mid, S1.Mngrs.mid)
```

The user wants to find out the salaries of an employee's managers and has currently formulated query q_1:

```
SELECT DISTINCT M1.salary                              q1
FROM Mnrgs M1, Empls E1
WHERE M1.mid=E1.mid
```

At this point, `E1.eid` *has to be provided to reach a feasible query. Therefore, the front-end makes two suggestions: (i) perform a selection on* `E1.eid`*, or (ii) introduce a second* `Empls E2` *table, so that parameters can be passed from* `E2.mid` *to* `E1.eid` *(based on the first column association). The suggested actions are both blue.*

Option (i) will formulate the feasible query q_{2F} which returns the salaries of `E1.eid` *employee's immediate managers.*

```
SELECT DISTINCT M1.salary                              q2F
FROM Mnrgs M1, Empls E1
WHERE M1.mid=E1.mid
AND E1.eid=''A123''
```

Option (ii) leads toward a query that returns the salaries of managers that are two levels above an employee. More specifically, if the user introduces table a second table `Empls E2`*, then the front-end colors the join* `E1.eid=E2.mid` *yellow, which formulates q_3:*

```
SELECT DISTINCT M1.salary                              q3
FROM Mnrgs M1, Empls E1, Empls E2
WHERE M1.mid=E1.mid AND E1.eid=E2.mid
```

For q_3, the front-end makes the same kind of suggestions to the user as for q_1, since `E2.eid` *has to be provided now. A selection on* `E2.eid` *formulates the feasible query q_{4F} which returns the salaries of managers that are two levels above that employee.*

```
SELECT DISTINCT M1.salary                              q4F
FROM Mnrgs M1, Empls E1, Empls E2
WHERE M1.mid=E1.mid AND E1.eid=E2.mid
AND E2.eid=''A123''
```

It becomes evident that the user can build chains of `Empls` *aliases of an unbounded length, where each alias joins its* `eid` *with the next one's* `mid`*, before performing a constant selection on the* `eid` *of the last* `Empls` *alias. These queries are infinitely many and are all closest feasible. For example, q_{2F} and q_{4F} are two such queries, and there is no sequence of actions that applied on q_{2F} formulate q_{4F}.* ◇

Searching for closest feasible queries starting from the maximally-contained ones becomes problematic as it is known that the latter set is infinite in the presence of binding patterns [8]. Moreover, the coloring of actions cannot be done by enumerating all closest feasible queries.

Instead, CLIDE identifies a finite set of parameterized *seed queries* $SQ(n)$, where $q(n)$ is the current query. These are not necessarily feasible, but have the property that each path toward a closest feasible query must pass through some seed query first. In Example 5.3, q_{2F} is a feasible seed query of q_1, while q_3 is an infeasible one, which however must be constructed on the way to q_{4F}. The algorithm suggests to the user a finite set of actions leading from $q(n)$ toward the seed queries $SQ(n)$. This can be done by simply calling the color algorithm of Section 5.2 on $SQ(n)$ instead of $FQ_C(n)$. This approach does not compromise the guarantees of completeness, minimality of suggestions and rapid convergence.

It is a priori non-obvious that the finite set $SQ(n)$ even exists. However, it turns out that this is indeed the case, and moreover that $SQ(n)$ can be computed as follows. Start by ignoring the binding patterns of the views and computing the maximally-contained rewritings of $q(n)$ in terms of the views. Under the original binding patterns, not all obtained rewritings are valid, and the values of their parameters must be provided. In each such rewriting, parameter values may be provided by (i) selections with a constant, or (ii) via a parameter-passing join with a view alias from within the rewriting or (iii) via a parameter-passing join with a new view alias. The considered parameter-passing joins must be compatible with the column associations. Notice that there are only finitely many considered selections and parameter-passing joins. We obtain $SQ(n)$ by systematically extending the rewritings according to possibilities (i), (ii) and (iii), and unfolding the view definitions in all extended rewritings.

6. IMPLEMENTATION

The current implementation of CLIDE consists of the components that compute the closest feasible queries and color the actions, shown in Figure 6. The component that handles parameterized selections in the views is under development.

CLIDE uses MiniCon [14] as the core of its maximally-contained rewriting component. Even though an initial implementation was provided to us, we had to significantly optimize and extend it in order to enable CLIDE's color algorithm and achieve on-line performance. Figure 7 illustrates the anatomy of the maximally-contained rewriting component from Figure 6.

Views Expansion The first challenge we faced was that MiniCon does not produce maximally-contained rewritings that are syntactic extensions of the current one. MiniCon initially produces a set of rewritings expressed using the views. Once these rewritings are expanded so that they are expressed in terms of the source schemas, they are not syntactic extensions of the current query, because fresh aliases are introduced. For example, if the current query is $q(n_3)$ (Snapshot 3 in Figure 4), MiniCon produces the following rewriting query q_R that combines V_1' and V_2':

```
SELECT DISTINCT V1'.ram, V1'.price                     qR
FROM V1', V2'
```

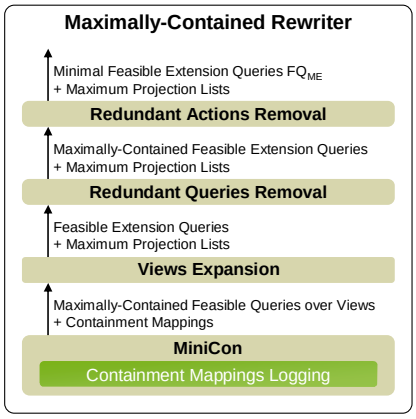


Figure 7: MiniCon Optimizations and Extensions

After expanding the views of q_R, we obtain the following query q_{RE}, which is expressed in terms of the source schemas.

```
SELECT DISTINCT ComA.ram, ComA.price                          q_RE
FROM Computers ComA, Computers ComB, NetCards Net
WHERE ComB.cid = Net.cid AND ComA.cpu='P4'
AND ComB.cpu='P4' AND Net.rate='54Mbps'
```

Query q_{RE} is syntactically isomorphic to the closest feasible query $q(n_9)$, but it is not a syntactic extension of $q(n_3)$, since $q(n_3)$ contains a table `Computers Com1`, while q_{RE} contains the tables `Computers ComA` and `Computers ComB`. It is not straightforward if `Com1` corresponds to `ComA` or `ComB`.

We could find the correspondences between the tables of $q(n_3)$ and the tables of q_{RE} by computing the containment mapping [5] from $q(n_3)$ into q_{RE}. The containment mapping considers all atoms of the two queries in order to find the correct correspondences. For example, `Com1` of $q(n_3)$ cannot be mapped into `ComB` of q_{RE}, because the `ComB.ram` and `ComB.price` projections do not appear in the `SELECT` clause, as is the case in $q(n_3)$. Once we compute the containment mappings, we can turn the MiniCon rewriting queries into syntactic extensions of the current query by renaming the aliases of the former.

We managed to avoid computing the containment mappings on top of MiniCon. We observed that while MiniCon searches for maximally-contained rewritings, it builds the containment mappings from the current query to the maximally-contained ones. So we extended MiniCon to log this information and output it along with the set of maximally-contained rewriting queries over the views, as shown at the bottom of Figure 7.

Subsequently, we wrote a *Views Expansion* component, which uses the logged containment mappings to expand the views in every MiniCon maximally-contained rewriting so that the resulting queries are syntactic extensions of the current one.

The *Views Expansion* component also generates the maximum projection lists used in the color algorithm of Section 5.2. In Section 5.2, we defined a maximum projection list to be the list of all possible projections that can be added to a query without compromising feasibility. It turns out that for each expanded query, the maximum projection list corresponds to all projections in the views that appear in the initial MiniCon rewriting. For example, the initial rewriting of $q(n_9)$ is q_R. We can safely add to q_R all projections in views V_1' and V_2', without compromising feasibility, and obtain the following query q_R':

```
SELECT DISTINCT                                               q'_R
       V'1.cid, V'1.cpu, V'1.ram, V'1.price
       V'2.cid, V'2.cpu, V'2.ram, V'2.price
       V'2.cid, V'2.rate, V'2.standard, V'2.interface
FROM V'1, V'2
```

Hence, the maximum projection list of $q(n_9)$ consists of all projections in q_R' except V_1'`.ram` and V_1'`.price` which are mapped into from $q(n_9)$. The containment mappings are used here as well, so that the aliases in the maximum projection lists refer to aliases that appear in the current query. These lists are constructed in linear time.

Redundant Queries Removal The *Views Expansion* component inputs maximally-contained queries, but not all syntactic extension queries it outputs are necessarily maximally-contained. It turns out that views expansion introduces redundancy across queries, i.e., expanded queries might contain one another. For example, if the current query is $q(n_1)$ in Figure 5 (Snapshot 1 in Figure 4), then MiniCon outputs two maximally-contained rewritings q_{R1} and q_{R2} over the views V_1' and V_2' which do not contain one another:

```
SELECT DISTINCT V'1.ram, V'1.price                            q_R1
FROM V'1

SELECT DISTINCT V'2.ram, V'2.price                            q_R2
FROM V'2
```

The expansion of q_{R1} though contains the expansion of q_{R2}, according to the definition of the views V_1' and V_2' in Example 5.1.

In order to preserve the rapid convergence and minimality guarantees of CLIDE (see Section 4.1), we have to eliminate contained queries. This additional work is performed by the *Redundant Queries Removal* component, which we built from scratch and tests if one query is contained in another. The query containment test amounts to finding containment mappings between queries and is in general NP-complete in the query size. In practice, the constructed queries are small, and this test is very efficiently implemented [22]. We compute the containment mappings from query q_1 into query q_2 by constructing a canonical database [5] for q_2, $canDB(q_2)$ and running q_1 over $canDB(q_2)$. To efficiently evaluate q_1, we employ standard algebraic optimization techniques: we construct an algebraic operator tree for q_1 (left deep join tree), in which selections and projections are pushed and joins are implemented as hash joins.

The efficient implementation of the *Views Expansion* component proved crucial to the on-line response of CLIDE, since query containment tests are the bottleneck for CLIDE's performance, as Section 7 demonstrates.

Redundant Actions Removal The output of the *Redundant Queries Removal* component is still not the set of minimal feasible extension queries FQ_{ME} that we are looking for, because they are not necessarily minimal extensions of the current query. For example, if q is the current query shown below, then q_E is the only feasible expansion query we get from MiniCon. q_E is not a minimal expansion query though. Query q_{ME} requires one action less than q_E to reach an equivalent query that minimally extends the current one.

```
SELECT DISTINCT Com1.ram, Com1.price                              q
FROM Computers Com1, Computers Com2

SELECT DISTINCT Com1.ram, Com1.price                              q_E
FROM Computers Com1, Computers Com2
WHERE Com1.cpu='P4' AND Com2.cpu='P4'

SELECT DISTINCT Com1.ram, Com1.price                              q_ME
FROM Computers Com1, Computers Com2
WHERE Com1.cpu='P4'
```

The *Redundant Actions Removal* component finds FQ_{ME} by systematically detecting two identical constants that refer to identical columns of two tables with identical names but distinct aliases, dropping one of them at a time, and testing for equivalence with the initial query. The same rule is applied on self-joins.

7. EXPERIMENTAL EVALUATION

Our experimental evaluation shows that CLIDE is a viable online tool. The MiniCon algorithm was evaluated via extensive experiments in [14] to measure the time to find the maximally-contained rewritings of queries using views. The goal of our experiments was to show that the rest of the CLIDE components do not add a prohibitive cost, and that the algorithms of Sections 5.1 and 5.2, as well as our extensions and optimizations (efficient implementation of containment test, logging MiniCon's containment mappings) are crucial in obtaining quick response times.

The Experimental Configuration To study how CLIDE scales with increasing complexity of the constructed query and with the number of views in the system, we used a synthetic experimental configuration, whose scaling parameters are K, L, M, as described below.

The schema. In the literature, synthetic queries are usually generated in one of two extreme shapes: chain queries and star queries. For a more realistic setting, we chose a schema which allowed us to build queries of a chain-of-stars shape, and in which joins follow foreign-key constraints (the most common reason for joins). To this end, we picked a schema comprised of a relation $A(ka, a)$ playing the role of a star center, which is linked (via foreign key constraints) to K relations $\{B_i(kb, fb, b)\}_{0\leq i\leq K}$ (the star corners). Each B_i is in turn the center of another star whose L corners are given by the relations $\{C_{i,j}(kc, fc, c)\}_{0\leq j\leq L}$. ka, kb, kc are respectively the key columns for A, the B_i's and the $C_{i,j}$'s. In each B_i, fb is a foreign key referencing ka from A. In each $C_{i,j}$, fc is a foreign key referencing kb from B_i.

The Views. The MiniCon experiments in [14] consider two extremes for view shapes, one very favorable, the other one leading to long rewriting time. The views in our configuration fall in the middle of this spectrum, and are more realistic. Each view we picked covers one of the foreign-key-based joins suggested by the schema. Moreover, we introduced selections with constants in these views, to force the interface to propose not only tables and joins, but also selections. For each i, we introduced M views $\{V_i^n\}_{0\leq n\leq M}$ joining A with B_i and imposing a selection comparing the b column with some constant c_n. For each i, j we introduced M views $\{V_{i,j}^n\}_{0\leq n\leq M}$ joining B_i with $C_{i,j}$ and comparing the c column to the constant c_n.

V_i^n: `SELECT` x.a, y.kb, y.b
`FROM` A x, B_i y
`WHERE` x.ka=y.fb AND y.b=c_n

$V_{i,j}^n$: `SELECT` y.kb, y.b, z.c
`FROM` B_i y, $C_{i,j}$ z
`WHERE` y.kb=z.fc AND z.c=c_n

There are $K \times M + K \times L \times M$ views in the configuration. For an intuitive interpretation of our abstract configuration, let the B_i tables stand for computer accessories, such as network cards, storage, keyboard, etc. For instance if B_1 plays the role of the `NetCards` table in Figure 2 and A that of `Computers`, then the view V_1^3 provides the computers compatible with a network card satisfying a selection condition with constant c_3.

The Queries. We scripted a family of interactions in which the simulated user starts by performing an A table action and then follows only blue and yellow suggestions, continuing even after reaching feasible queries.

After the initial A table action, CLIDE suggests joins with the B_i's. If any of these suggestions are taken (say by picking B_p), CLIDE suggests the corresponding selections on B_p's column b, as a list of options $c_1, \ldots, c_M$. It also suggests table actions leading to the join of A with some other B_j or of B_p with some $C_{p,o}$. When the simulated user picks a selection with c_n it reaches a feasible query having a rewriting using V_p^n. When this feasible query is extended to join B_p with some $C_{p,o}$, CLIDE suggests (among others) selections comparing $C_{p,o}$'s column c to some constant. Picking one of these, say c_r, generates another feasible query, which has a rewriting that joins V_p^n with $V_{p,o}^r$.

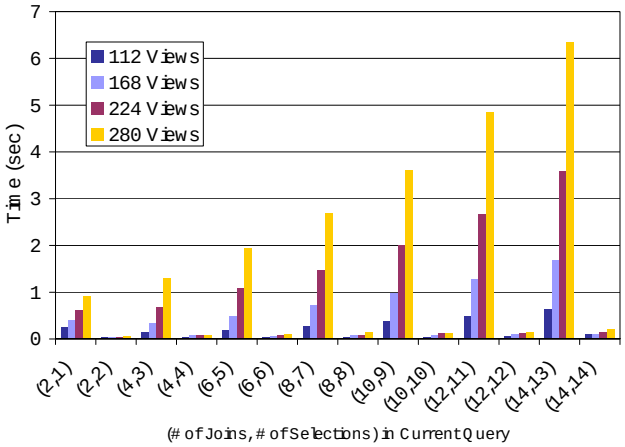


Figure 8: CLIDE's response time

The Measurements The measurements were conducted on a dedicated workstation (Pentium 4 3.2GHz, MS Windows XP Pro, 1GB RAM) using Sun's JRE-1.5.0. All measurements are elapsed times.

We generated four configurations by fixing $K = 7$, $L = 3$ and varying $M = 4, 6, 8, 10$, yielding respectively $112, 168, 224$ and 280 views. Figure 8 reports the time CLIDE took to come up with the suggestions at each current query. Query (n, m) is a query reached after performing n table actions and joins, and m selections. On the horizontal axis, all odd-position queries are infeasible, while even-position queries are all feasible, being obtained by adding a relevant selection to their predecessor. For instance, feasible query (2,2) is obtained from infeasible query (2,1) by adding a selection action.

Notice that, while CLIDE's response is good overall, scaling to large number of views, it is much better for feasible queries. This is an expected result, since CLIDE needs to consider a single closest feasible query, i.e., the one that the user has reached, as opposed to the number of closest feasible queries when the current query is infeasible. The bottleneck in CLIDE's performance turns out to be the containment tests, which are a consequence of the views expansion. For instance, for query $(14, 13)$, there are 700 expanded queries of which only 10 are non-redundant. These queries are quite sophisticated, joining up to 15 views. To identify them, CLIDE runs pairwise containment tests over the 700 redundant queries, then it minimizes the 10 queries invoking more containment tests. This work dominates the response time. 8311 containment tests need 6 seconds out of the 6.4 seconds of the elapsed time. The reason CLIDE scales to these query and view sizes is the efficient implementation of the containment test.

Note that when parameterized selections do not appear in the views, we could invoke MiniCon only when the user reaches a feasible query. We could exploit the fact that one interaction step along an edge $n \xrightarrow{a} n'$ changes $q(n)$ only incrementally. If a was a yellow or blue action, $FQ_C(n')$ would be contained in $FQ_C(n)$ and we would not need to call MiniCon to compute $FQ_C(n')$. Instead, we could inspect the containment mappings from $q(n)$ into $FQ_C(n)$ and we could compute $FQ_C(n')$ by pruning those mappings that would not be consistent with action a and dropping from $FQ_C(n)$ all queries into which there would be no more containment mappings. This optimization would be in effect as long as the user would perform yellow and blue actions and for the periods of the interaction between feasible queries.

8. DISCUSSION AND RELATED WORK

Alternative query formulation paradigms have been proposed in the literature [19], but the QBE paradigm is the one that users are mostly familiar with today. As an alternative to a visual query builder, one could try to exploit existing formalisms for compact descriptions of infinite sets of supported queries. These focus mainly on sets of binding patterns [6, 11, 16, 23] and sets of parameterized queries described by the infinite unfoldings of recursive Datalog programs [10, 21]. However, these representations are meant for consumption by rewriting algorithms and not by humans: checking whether a given query is supported requires non-obvious rewriting algorithms, especially when the set of indirectly supported queries is enhanced via additional processing inside a mediator. This is a key obstacle to the practical utilization of current query rewriting algorithms for interactive query development, forcing the query writer into a trial-and-error loop.

There are many scenarios which would benefit from CLIDE's approach to query building. One example is the setting of [23], which is a special case of a service-oriented architecture with parameterized views restricted to identity views over individual tables. Their algorithm infers binding patterns for queries against these views, and could conceptually be used by the user to reach a feasible query by providing appropriate bindings. However, the user queries may be adorned with exponentially many binding patterns, turning the visual inspection by the user into a cumbersome process. Another obvious CLIDE application is in data privacy enforcement. [17, 9] allow data owners to identify the non-sensitive data they are willing to export by means of parameterized, virtual views against the proprietary data. Data consumers formulate their queries against the proprietary database as well, but their queries are rejected [17] or return null values [9] if they are not feasible according to the virtual views.

The implementation of the CLIDE back-end described here requires as one building block an algorithm for finding maximally-contained rewritings. We picked MiniCon [14] because we had access to the code, but we could have swapped it with any other one [8]. Their applicability to our problem comes as a pleasant surprise, as the original goal of these algorithms is different: to provide an underestimate approximation of the query answer when the query is not feasible. Other systems [12] automatically formulate an overestimate or underestimate of the submitted query. We believe that in many applications the user needs full control and understanding of what she can ask and which precise query is being answered.

9. CONCLUSIONS

We presented the CLIDE interactive system and its color scheme that leads the user toward feasible queries in a setting where the content and access methods of the sources are described by parameterized conjunctive views. We have provided guarantees of completeness, minimality of suggestions and rapid convergence. We formalized the interaction with the front-end using an interaction graph and reduced coloring properties to interaction graph properties that the back-end has to decide upon. We developed the front-end and the back-end for the case where only constant selections appear in the views. We implemented effective optimizations that enable on-line use of CLIDE for a wide class of queries and views. A CLIDE demonstration is available at `http://www.clide.info`.

10. ACKNOWLEDGMENTS

The authors would like to thank Yannis Katsis, Alan Nash and Michael J. Carey for their valuable contribution. Alin Deutsch was supported by NSF CAREER award IIS-0347968. Yannis Papakonstantinou was supported by NSF ITR 313384, the Gordon and Betty Moore Foundation, and the NSF Award # EAR-0225673 (GEON ITR). Michalis Petropoulos was supported by NSF ITR 313384.

11. REFERENCES

[1] Microsoft SQL Server. http://www.microsoft.com/sql/.
[2] Amazon E-Commerce Service. http://www.amazon.com/gp/aws/sdk/.
[3] Google Web APIs. http://www.google.com/apis/.
[4] Web Services and Service-Oriented Architectures. http://www.service-architecture.com/.
[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[6] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1), 2000.
[7] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.
[8] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.
[9] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. J. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, pages 108–119, 2004.
[10] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external processors. In *PODS*, pages 227–237, 1996.
[11] Chen Li and Edward Y. Chang. Answering queries with useful bindings. *ACM Transactions on Database Systems (TODS)*, 26(3), 2001.
[12] A. Nash and B. Ludaescher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.
[13] Jakob Nielsen. *Designing Web Usability*. New Riders Publishing, 2000.
[14] R. Pottinger and A. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3), 2000.
[15] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
[16] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112, 1995.
[17] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562, 2004.
[18] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *VLDB*, pages 266–275, 1997.
[19] L. A. Rowe. "fill-in-the-form" programming. In *VLDB*, 1985.
[20] Edward R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, 1997.
[21] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, 1997.
[22] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.
[23] R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman. Computing capabilities of mediators. In *SIGMOD*, pages 443–454, 1999.
[24] M. Zloof. Query by example. *AFIPS NCC*, 44:431–438, 1975.