Exporting and Interactively Querying Web Service-Accessed Sources: The CLIDE System

MICHALIS PETROPOULOS
University at Buffalo, State University of New York
and
ALIN DEUTSCH, YANNIS PAPAKONSTANTINOU, and YANNIS KATSIS
University of California, San Diego

The CLIDE System assists the owners of sources that participate in Web service-based data publishing systems to publish a restricted set of parameterized queries over the schema of their sources and package them as WSDL services. The sources may be relational databases, which naturally have a schema, or ad hoc information/application systems whereas the owner publishes a virtual schema. CLIDE allows information clients to pose queries over the published schema and utilizes prior work on answering queries using views to answer queries that can be processed by combining and processing the results of one or more Web service calls. These queries are called *feasible*. Contrary to prior work, where infeasible queries are rejected without an explanatory feedback, leading the user into a frustrating trial-and-error cycle, CLIDE features a query formulation interface, which extends the QBE-like query builder of Microsoft's SQL Server with a color scheme that guides the user toward formulating feasible queries. CLIDE 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.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing; relational databases*; H.3.3

M. Petropoulos was supported by NSF ITR 313384. A. Deutsch was supported by the Alfred P. Sloan Foundation and by the NSF under grant numbers IIS-0347968 (CAREER) and IIS-0415257. Y. Papakonstantinou was supported by NSF ITR 313384, the Gordon and Betty Moore Foundation, and the NSF Award # EAR-0225673 (GEON ITR).

Authors' addresses: M. Petropoulos, Department of Computer Science and Engineering, University at Buffalo, State University of New York, 201 Bell Hall, Box 602000, Buffalo, NY 14620-2000; email: mpetropo@cse.buffalo.edu; A. Deutsch and Y. Papakonstantinou, and Y. Katsis, Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Dr., Mail Code 0404, La Jolla, CA 92093-0404; email: A. Deutsch and Y. Papakonstantinou, {deutsch, yannis}@cs.ucsd.edu; Y. Katsis, ioannis@ucsd.edu.

[**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering; query formulation; selection process*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Data sharing; web-based services*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces (GUI); interaction styles (e.g., Commands, menus, forms, direct manipulation)*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Middleware, query rewriting, limited access patterns, Web services

ACM Reference Format:

Petropoulos, M., Deutsch, A., Papakonstantinou, Y., and Katsis, Y. 2007. Exporting and interactively querying web service-accessed sources: The CLIDE system. ACM Trans. Datab. Syst. 32, 4, Article 22 (November 2007), 56 pages. DOI = 10.1145/1292609.1292612 http://doi.acm.org/10.1145/1292609.1292612

1. INTRODUCTION

In the recent generation of Web service-based data publishing and integration systems [Carey 2006], information source owners need to export only a limited set of calls over their sources' schema, where these calls correspond to parameterized queries. A typical reason for providing limited access methods is privacy constraints [Rizvi et al. 2004; LeFevre et al. 2004; Fan et al. 2004]. In other cases the source owners need to publish a set of services that correspond to limited access methods because the underlying information system offers only such [Yerneni et al. 1999; Garcia-Molina et al. 2001]. For example, an SAP application offers a set of calls that access the underlying database but does not offer access to the database per se. Performance of database servers is also a major concern. Administrators want to protect their databases by controlling the query plans executed against them. Hence they export a limited set of parameterized queries instead of providing full query access to their systems.

The exporting functionality of the CLIDE System, called the *Relational Services Description Language (RSDL)* and presented in Section 11, assists information source owners to publish the relevant parts of the schema of the underlying source, along with a set of calls that correspond to parameterized queries. The same formalism can be used if the source is not a database, for example, if it is an SAP application. In this case, the source owner publishes a virtual schema and writes parameterized queries over this schema, whose results are equivalent to the results of the corresponding calls. As we will see, CLIDE allows developers to formulate queries, using SQL, against the schemas published by the sources. This shields the developers from having to browse a long list of Web services and having to manually construct a composition that answers their query. The challenge in providing this functionality stems from the fact that not all queries against published schemas are supported by composing the exported parameterized queries. An unassisted developer will be caught in a frustrating trial-and-error query formulation cycle. To avoid this problem, CLIDE features a graphical interface which interactively guides the developer toward formulating supported queries only.

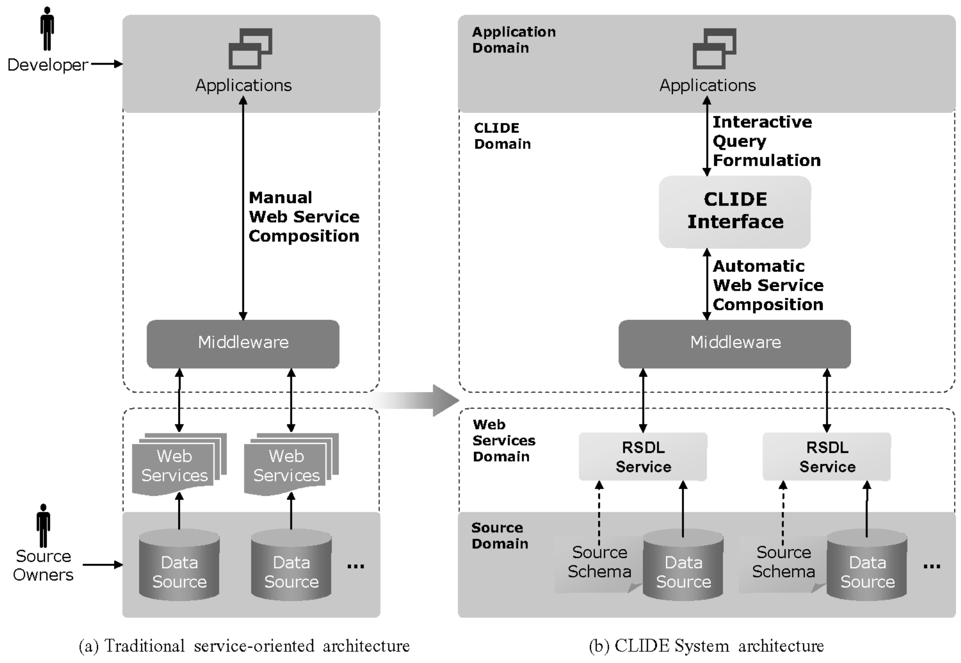


Fig. 1. Paradigm shift.

We propose a comprehensive solution for exporting semantically rich Web services on top of relational sources, in terms of both data and functionality, and interactively querying multiple such sources based on their exported services. The proposed solution constitutes a paradigm shift from traditional service-oriented architectures that target scenarios involving data sources, such as Web data publishing, portals, enterprise information integration applications, and workflow systems. Figure 1(a) shows a typical service-oriented architecture, where the two participating roles are the source owners (database administrators) exporting part of their data on the Web, and the developers building their applications by querying and composing data from multiple sources.

According to current practices,[1] source owners typically construct parameterized views over their data sources, encapsulate each one within a procedural language, and export them as individual Web services, where the input types describe the parameters expected by the query and the output type describes the structure of the results. Subsequently, a developer inspects the Web services, possibly exported by multiple data sources, decides which ones she will use based on the input and output types, manually composes the output of some with the input of others in order to retrieve the data needed, and then incorporates this Web service composition into her application, which will be executed by a middleware component during run-time.

Developers and source owners have different considerations when it comes to exporting and querying Web services on top of data sources. The task of the

[1]For a detailed discussion on current practices and approaches, please see Section 2.

developers becomes substantially easier if they

(D1) have a semantically rich description of the input and output types of Web services they inspect in order to judge if a Web service is useful to their application;
(D2) understand the relationship between the input and output of each Web service, so that they know which input to provide in order to receive the desired output;
(D3) know which Web services are exported on top of the same data source, so that they don't retrieve the same data twice and/or query the same data source twice thus impeding the performance of their application;
(D4) locate the Web services of interest quickly, without having to browse through a large number of them; and
(D5) can discover which Web service compositions can be executed by the middleware, without having to enumerate all of them, and then chose the one that serves their needs; note that all possible Web service compositions are potentially exponential in the number of Web services in consideration.

At the other end, the source owners are concerned with

(S1) control the number and granularity of the exported Web services so that their data servers do not overload, and
(S2) use an exporting mechanism that allows them to easily manage the Web services on top of their data servers.

The service-oriented architecture shown in Figure 1(a) utilizes the Web Service Definition Language (WSDL) [Christensen et al. 2001] and the XML Schema type system [Fallside and Walmsley 2004] to describe the input and output types of Web services, which serves points D1 and S1 well. The rest of the points, though, are not addressed. Point D2 is not satisfied because Web services are currently designed to encapsulate and export arbitrarily complex software artifacts and not just declarative database queries that allow for such relationships to be exported in a compact and informative way. Point D3 is not satisfied either, because Web services are supposed to be treated as black boxes by the developers, without having to expose their internal logic. Hence, there is no way of knowing if two Web services query the same underlying data or not. Developers rely on textual descriptions in order to choose which Web services to use. Regarding point D4, there is an academic proposal of how to search for Web services that could potentially be composed, based on the similarity of their input and output types [Dong et al. 2004]. Still, the developer has to inspect a potentially long list of results and, moreover, there is no guarantee that the located services can be composed in a semantically "meaningful" way. That is, the developer might not be able to formulate the query and retrieve the information that she has in mind.

The above observation brings us to point D5; the most time- and effort-consuming task in service-oriented application development. Currently, a developer has to discover and manually construct a web service composition that retrieves the needed data without having an indication which compositions are feasible (or executable) by the middleware. Moreover, if a given composition does not provide all the data columns they need, they can't be sure that no alternative compositions exist which actually retrieve all the necessary ones. The same argument applies when a composition is overrestricting or underrestricting a query result the developer has in mind. The task becomes harder when taking into consideration the capabilities of middleware components, such as mediators [Garcia-Molina et al. 2001] and distributed query processors [Borkar et al. 2006], to postprocess the results of Web service calls by filtering, joining, and combining them.

The CLIDE System addresses the considerations of developers and source owners by introducing to service-oriented architectures two key components shown in Figure 1(b); a publishing component for exporting semantically rich Web services on top of relational sources based on the proposed *Relational Services Description Language (RSDL)*, and the CLIDE *Interface* for interactively formulating queries on top of multiple RSDL-based Web services. We will use the (running) example below to exhibit the characteristics of the CLIDE System and justify the need for a new paradigm.

1.1 RSDL

Assume that Figure 1(b) shows a simple example instance of an architecture where the CLIDE System enables a computer shopping portal user to have integrated query access to two sources, namely, Dell and Cisco. The publishing task is accomplished by the Dell and Cisco source owners exporting a set of RSDL-based Web services on their computer and router catalogs, respectively. Figure 2 illustrates the signatures of four Web service operations [Christensen et al. 2001] the source owners export, the parameterized views that are encapsulated by them, and the parts of their respective schema that is accessed by the views.

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., 54 Mb/s), implements a standard (e.g., IEEE 802.11g), and communicates with a computer via a particular interface (e.g., USB). The Web service operation *ComByCpu* returns the computers with a given *cpu*.[2] The Web service operation *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* Web service operations return routers that are of either wired or wireless type, respectively.

WSDL [Christensen et al. 2001] only exposes the Web service operations. The novelty of RSDL is that it exports and ties together all the pieces of information shown in Figure 2 in a single service; the Web service operations, the parameterized views, and the relevant part of the source schema. For each operation,

[2]We assume there is a *Computer* type defined that has the same attributes as the corresponding Dell source relation. The same applies to the other Web service operations as well.

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

$ComByCpu(cpu) \rightarrow (Computer)^*$ (WS Operation_1)
```
SELECT DISTINCT Com.*
FROM Computers Com
WHERE Com.cpu=cpu
```
(V_1)

$ComNetByCpuRate(cpu,\ rate) \rightarrow (Computer,\ NetCard)^*$ (WS Operation_2)
```
SELECT DISTINCT Com.*, Net.*
FROM Computers Com, NetCards Net
WHERE Com.cid=Net.cid AND Com.cpu=cpu
AND Net.rate=rate
```
(V_2)

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

$RoutersWired() \rightarrow (Router)^*$ (WS Operation_3)
```
SELECT DISTINCT Rou.*
FROM Routers Rou
WHERE Rou.type='Wired'
```
(V_3)

$RoutersWireless() \rightarrow (Routers)^*$ (WS Operation_4)
```
SELECT DISTINCT Rou.*
FROM Routers Rou
WHERE Rou.type='Wireless'
```
(V_4)

Fig. 2. RSDL-based Web services.

an RSDL-based Web service can provide the corresponding parameterized view that is executed when a call is made by a middleware or a user. Hence, point D2 is addressed since the relationship between the input and the output is expressed explicitly through a declarative parameterized view definition, and a user knows exactly how the input affects the output.

An RSDL service also exports the part of the source schema that is accessed by the parameterized views, and logically associates it with all the operations exported within the same service. Hence, point D3 is addressed since a developer can easily realize which services are exported on top of the same data source. Another advantage of exporting the underlying schema is that a developer can realize the relationship among the operations exported by an RSDL service. The source schema can carry additional semantic information, such as foreign keys, which is not available by the Web service operations or the parameterized views alone.

Even though the features of RSDL are required by the CLIDE Interface, as shown in the next section, we believe RSDL can be used as a general exporting mechanism of database functionality that can benefit source owners, middleware components, and developers alike. RSDL can facilitate the development of visual interfaces and administration consoles that will give the source owners a coherent picture of the Web services they export on top of their data sources and help them reduce the complexity of maintaining them, thus addressing point S2. Moreover, RSDL has the potential to render Web services on top of relational databases more usable for data integration and exchange tasks, such as schema matching [Rahm and Bernstein 2001] and Web services similarity search [Dong et al. 2004].

RSDL is not an alternative proposal to WSDL. The syntax of RSDL is a specialization of the WSDL syntax, which means that current WSDL clients are able to import and utilize RSDL-based Web services. Vice versa, existing WSDL-based Web services can be wrapped by RSDL services. In an alternate deployment of the architecture shown in Figure 1(b), Dell and Cisco already provide the Web service operations shown in Figure 2 using an exporting mechanism other than RSDL. In such case, the source owners or an independent developer can utilize RSDL in order to publish virtual source schemas for Dell and Cisco and the parameterized views shown in Figure 2 that explain the results of the Web services in terms of the virtual source schemas. From now on we will refer to *source schemas*, without making a distinction on whether they are the real ones or virtual ones.

The concrete syntax and semantics of RSDL are presented in Section 11.

1.2 The CLIDE Interface

The CLIDE Interface is the second component introduced by the proposed paradigm in Figure 1(b) and is positioned between the middleware and the developer trying to formulate a composition of Web services. This component is an *interactive query formulation graphical interface* the developer is using to formulate queries against the RSDL-based Web services known to the middleware. The CLIDE Interface effectively addresses points D4 and D5 by shielding the developer from having to manually discover the Web service composition she needs. Instead, the developer focuses on *what* information she wants to obtain and not *how* to obtain it by browsing a long list of Web service operations. The CLIDE Interface consists of a *front-end* and a *back-end* modules that achieve this goal.

The front-end initially presents to the developer the schemas exported by the RSDL-based Web services in consideration, which are much more concise than a long list of exported Web service operations, thus addressing point D4. The developer formulates a query that retrieves the data she needs in a step-by-step process by executing visual actions. The front-end of the CLIDE Interface is based on the well-known visual paradigm of the Microsoft Query Builder, incorporated in Microsoft's MS Access and MS SQL Server (go online to `http://www.microsoft.com/sql/`) (see Figure 4 and Section 4 for a detailed discussion). The Microsoft Query Builder, in turn, is based on the Query-By-Example (QBE) [Zloof 1975] paradigm. According to this paradigm, a visual action is the inclusion of a table in the `FROM` clause, the formulation of a selection/join condition in the `WHERE` clause, or a projection of a column in the `SELECT` clause of a SQL query.

Of course, not all queries against the source schemas are *feasible*, since the access to these schemas is restricted by the RSDL services. For a query to be feasible, there must be an equivalent Web service composition[3] that retrieves the same data and can be executed by the middleware. For this reason, and in every step, the back-end computes all feasible Web service compositions that are *syntactic extensions* of the currently formulated query, thus addressing point D5. Once computed, the possible feasible compositions are translated by the front-end into visual suggestions of what actions the developer can perform next in order to reach a feasible query. This process is repeated whenever the developer performs a visual action and until the desired and feasible composition is formulated. It is important to note that the CLIDE Interface is not trying to guess the query the developer is trying to formulate. All feasible compositions that can be formulated by extending the current query are considered when suggestions are made to the developer.

The computation of the feasible compositions by the back-end is possible only because RSDL-based Web services provide the parameterized views for all exported operations. For this computation, the back-end also takes into consideration the postprocessing capabilities of the middleware. These issues, as well as the reason we need to compute the feasible compositions whenever the developer performs a visual action, are discussed in detail in Section 6.

In every step of the query formulation, the CLIDE Interface guides the developer toward formulating feasible queries and indicates any action that will lead toward an unsupported one. In particular, the front-end suggests to the developer which possible actions should, should not or may be performed in order to reach a feasible query. These different types of suggested actions are indicated on the front-end by a color scheme. Also, a flag indicates whether the currently formulated query is feasible or not. If it is, colored actions suggest how to reach another feasible query, which will be a syntactic extension of the current one.

As an example of the CLIDE Interface and the color-driven interaction, consider the RSDL services shown in Figure 2. A developer uses the front-end to formulate the query "return all P4 computers with a 54-Mb/s network card and the compatible wireless routers" against the source schemas. The middleware can answer this query because there is an equivalent Web service composition that combines the answers of calls to the Web services*CompNetByCpuRate* and *RoutersWired*. However, it cannot answer the query "return all computers with 1 GB of RAM." The reader is pointed to Chapter 20.3 of Garcia-Molina et al. [2001] for similar examples. The CLIDE Interface appropriately guides the developer toward the formulation of feasible queries by employing the following color scheme:

—*Red* color indicates actions that lead to unsupported queries, regardless of what actions are performed 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

[3]Formally defined in Section 3.

all equivalent Web service compositions that the middleware can execute 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.

Any good interface that guides a user toward some action must be comprehensive (complete) and, at the same time, avoid overloading the user with information in every step [Nielsen 2000; Tufte 1997]. The CLIDE Interface achieves both goals since it guarantees the following important properties in every step of the interaction:

(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 developer and the CLIDE Interface front-end are formalized using an *Interaction Graph*, which models the queries as nodes and the actions that the developer 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.

The above properties present a challenge to the CLIDE Interface back-end because they cannot be trivially turned into an algorithm, since they conceptually 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 in selection conditions. We tackle this problem by considering parameterized queries (similar to Sum Microsystem's JDBC (JDBC API; go online to `http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/index.html`) 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 prove that they are sufficient to determine the colors of possible actions. For our purpose, we leverage prior algorithms and implementations for rewriting queries using parameterized views [Pottinger and Halevy 2001; Duschka et al. 2000; Rajaraman et al. 1995]. However, we needed to significantly optimize and extend current implementations in order to achieve online performance and to ensure that the above properties are preserved. We provide a set of experiments that illustrate the class of queries and views that the CLIDE Interface can handle, while maintaining online response.

1.3 Contributions

The CLIDE System is a comprehensive proposal for exporting semantically rich Web services on top of relational sources and interactively querying multiple such sources based on their exported services. More specifically, the CLIDE System makes the following novel contributions:

(1) The Relational Services Description Language (RSDL) exports database functionality on top of relational sources. Each RSDL-based Web service exports and semantically connects a set of operations with the corresponding parameterized views and the source schema.
(2) The CLIDE Interface front-end enables color-guided interactive query formulation, based on the well-known QBE paradigm, over sources that only allow a limited set of Web service calls over their data.
(3) The suggestions made by the CLIDE Interface front-end satisfy the completeness, minimality, and rapid convergence properties, which avoid overloading the developer with unnecessary suggestions.
(4) The CLIDE Interface formalizes the interaction sessions between the developer and the front-end using the *Interaction Graph*, which serves as the basis for defining the action colors and the above three properties.
(5) The CLIDE Interface back-end leverages, extends, and optimizes existing algorithms for rewriting queries using parameterized views in order to achieve online performance and to ensure that the above properties are preserved.

1.4 Demonstration

We implemented the front-end and all back-end algorithms of the CLIDE Interface which is available online at `http://www.clide.info`.

1.5 Outline

Section 2 presents previous work related to RSDL and the CLIDE Interface, and discusses the applicability of the CLIDE Interface to exporting mechanisms other than RSDL. Section 3 provides definitions and notation conventions. Section 4 discusses query formulation interfaces, focusing on issues related to the CLIDE Interface, and introduces the interaction graph, which allows us to formally define their behavior. Section 5 discusses the aspects of the CLIDE Interface that pertain to interaction in the presence of a limited set of feasible queries. Sections 6 and 7 describe the algorithms of the CLIDE Interface back-end, and Section 10 analyzes their complexity. Section 8 describes the implementation and optimizations, which are experimentally evaluated in Section 9. Section 11 presents the concrete syntax and semantics of RSDL and Section 12 concludes the article.

2. RELATED WORK

The CLIDE System relates to both industrial systems and academic proposals. First, we discuss the BEA AquaLogic Data Services Platform [Carey 2006], a sophisticated data integration commercial product that relates to the CLIDE System's paradigm as a whole. Subsequently, we present previous work in the following areas that relate to each component individually: (i) *Database Functionality Exporting Mechanisms*, which relate to RSDL, (ii) *Query Formulation Interfaces*, which relate to the CLIDE Interface front-end, and (iii) *Answering Queries using Views*, which relate to the algorithms employed by the CLIDE Interface back-end.

The BEA AquaLogic Data Services Platform (ALDSP) provides a mechanism, called *Data Services*, for exporting functionality on top of databases, among other types of data sources, which relates to RSDL. A Data Service is a collection of functions all of which have a common output schema. Each function accepts different sets of parameters and is implemented by an XQuery expression [Boag et al. 2007]. In a simplified example, a Data Service can export a set of functions returning `Customer` objects where one operation takes as input the customer's last name, another one her city and state, and so on. Data services are a concept internal to the ALDSP but can be deployed as WSDL-based Web services, in which case each function becomes an operation. Data services address points D2 and D3 as long as a developer stays within the ALDSP development environment. Once outside, the XQuery expressions implementing the Web service operations and the schema against which the expressions are executed are not accessible.

Data Services can be exported directly on top of data sources, in which case they loosely correspond to the RSDL services exported by source owners in Figure 2, or on top of other Data Services by composing them. In the latter case, the query expression implementing the Data Service can be thought of as the query the developer is formulating using the CLIDE Interface. In the ALDSP development environment, the formulation of these query expressions is assisted by a visual XQuery Editor. The developer is presented with the output schemas of the Data Services and is assisted in her effort to formulate a feasible composition by a mechanism called *Model Diagrams*, which captures relationships among Data Services. Considering the `Customer` Data Service from above and assuming another one that returns `Order` objects, a foreign key relationship between them captures the fact that every order has a customer. In Model Diagrams, such relationships are implemented as Data Services and allow a developer to retrieve a `Customer` object given an `Order` object by composing two Data Services. In effect, Model Diagrams capture "meaningful" Data Service compositions. This paradigm is inspired by the Functional Data Model [Shipman 1981] and the corresponding query language. Model Diagrams also serve well point S2.

Compared to the CLIDE Interface, ALDSP uses the output schemas of the Data Services as a basis for query formulation and not the schemas of the data sources on top of which they are exported, which are generally much more concise. This fact also forces the exposure of complex Data Service compositions to the developer, who focuses on how to formulate a semantically "meaningful" and feasible one, and not just on what information she needs to retrieve. The CLIDE Interface hides the compositions from the developer, computes all feasible ones in the back-end, and uses the color scheme to guide her toward a feasible one, while guaranteeing important properties. On the other hand, the ALDSP paradigm scales well for integration scenarios, where Data Services can be constructed on top of other Data Services and form arbitrarily deep hierarchies. Also, ALDSP is based on the XML data model and the XQuery language, both more powerful than the relational model and the class of queries considered by the CLIDE System (see Section 3). Finally, ALDSP includes a powerful distributed query processor, which can be used as middleware in deployments such as the one shown in Figure 1(b).

Database Functionality Exporting Mechanisms are incorporated by most vendors in commercial database products. Representative examples are the IBM Document Access Definition Extension (DADX) (Db2 XML extender; go online to `www.306.ibm.com/software/data/db2/extenders/xmlext/`) technology and the Native XML Web Services for Microsoft SQL Server 2005 [Microsoft, Inc. 2005].

DADX is part of the IBM DB2 XML Extender (see publication data in previous paragraph), an XML/relational mapping layer, and facilitates the development of Web services on top of relational databases that can, among other things, execute SQL queries and retrieve relational data as XML. The developer incorporates into a DADX file a set of parameterized SQL queries. Subsequently, the IBM DB2 XML Extender parses the DADX file and automatically publishes a Web service operation for each query. The resulting services do not provide access to the parameterized queries. Schema information is provided by a special Web service, called *dynamic query service*. The same service, though, exports an operation that takes as input an arbitrary SQL expression to be executed against the underlying database.

The Native XML Web Services for Microsoft SQL Server 2005 [Microsoft, Inc. 2005] provide similar functionality through HTTP Endpoints, but no schema information is provided.

Other mechanisms have been proposed by previous academic work for compact descriptions of infinite sets of views, such as binding patterns [Duschka et al. 2000; Li and Chang 2001; Rajaraman et al. 1995; Yerneni et al. 1999] and parameterized views described by the infinite unfoldings of recursive Datalog programs [Levy et al. 1996; Vassalos and Papakonstantinou 1997]. RSDL services are based on parameterized views; a common technique that has been used to describe content and access methods in the widely used Global-as-View (GaV) integration architectures [Halevy 2001], and also recently to describe privacy constraints in Rizvi et al. [2004].

Query Formulation Interfaces relate to the CLIDE Interface front-end. To the best of our knowledge, there is no other query formulation interface for the relational model that is used as widely as the Microsoft Query Builder (go online to `http://www.microsoft.com/sql/`) and QBE [Zloof 1975]. Query formulation interfaces for other models include XQBE [Braga et al. 2005] and the XQuery Editor of the BEA AquaLogic Data Services Platform [Carey 2006], for the XML data model, and PESTO [Carey et al. 1996], for the object-oriented data model. In the area of data exchange, several interfaces have been proposed to assist a developer formulate a query that translates data from one schema to another. Representative examples are the IBM Clio [Popa et al. 2002] and the Microsoft BizTalk Mapper [Microsoft, Inc. 2004].

Answering Queries using Views Algorithms [Halevy 2001] relate to the algorithmic problem that the CLIDE Interface back-end is facing. Deciding whether a given query is feasible or not is a query rewriting problem. These algorithms take as input a query q over the source databases D, and a set of parameterized views $V_1, \dots, V_n$, and consider the postprocessing capabilities of a middleware component to filter and combine results obtained from the views. The output is a composition, if any, that accesses the views and computes $q(D)$. The composition is typically in the form of a query $q'(V_1(D), \dots, V_n(D))$ that runs on the views and often incorporates primitives that indicate the passing of information across views.

Several rewriting algorithms have been published [Halevy 2001]. The algorithms differ in their assumptions on the data model (relational or XML), expressive power of the input queries (e.g., conjunctive, union of conjunctive, etc), expressive power of views, and power of the composition that uses the views. However, these algorithms are not sufficient for the CLIDE Interface back-end. To the best of our knowledge, in all rewriting algorithms, if there is no composition for the input query, the developer simply receives a rejection without any explanation of what actions will lead to one. This is a key obstacle to the practical utilization of current query rewriting algorithms for interactive query formulation, forcing the developer into a trial-and-error query formulation loop. Instead, the algorithms implemented by the CLIDE Interface back-end enable the front-end to guide the developer toward queries for which a feasible composition exists.

Other rewriting algorithms [Nash and Ludaescher 2004] automatically formulate an overestimate or underestimate of the input query, thus addressing a different goal than the one in our setting. We believe that in many applications the developer needs full control and understanding of what she can ask and which precise query is being answered. Nevertheless, there are important technical connections between those algorithms and our work that are discussed in Section 6.

3. DEFINITIONS AND NOTATIONS

The CLIDE Interface front-end formulates queries from the set of conjunctive SQL queries with equality predicates $\text{CQ}^=$ 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

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

Fig. 3. Column associations.

`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. For the schemas in Figure 2, Figure 3 illustrates the column association of the `cid` columns of `Computers` and `NetCards` and the `rate` and `standard` columns of `NetCards` and `Routers`. Column associations can be explicitly declared by the middleware 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 [Rahm and Bernstein 2001]. The user can configure the CLIDE Interface 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 nonassociated columns, but the CLIDE Interface front-end will not suggest them. For the rest of the presentation, we assume the user has configured the front-end to suggest joins between associated columns only. We denote this class of queries with $\text{CQ}^{=CA}$.

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

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 over (or do not care to control) the exact table alias names.

We denote the set of *feasible* queries by $FQ \subseteq \text{CQ}^{=CA}$. As in Rajaraman et al. [1995], we define the feasible queries given a set of views $\mathcal{V} = V_1, \dots, V_k \in \text{CQ}^{=P}$ over a fixed schema D, to be the set of queries $q_{F1}, \dots, q_{Fm} \in \text{CQ}^{=CA}$ over D that have an equivalent $\text{CQ}^{=}$ rewriting using $\mathcal{V}$. In the absence of parameters, a rewriting of a query Q against D is simply a query R that refers exclusively to the views, such that, for any database db of schema D and database m of schema $\mathcal{V}$ computed by materializing the view results over db, the result of Q on db coincides with the result of R on m. 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 [Rajaraman et al. 1995; Roth and Schwarz 1997]. 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:

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

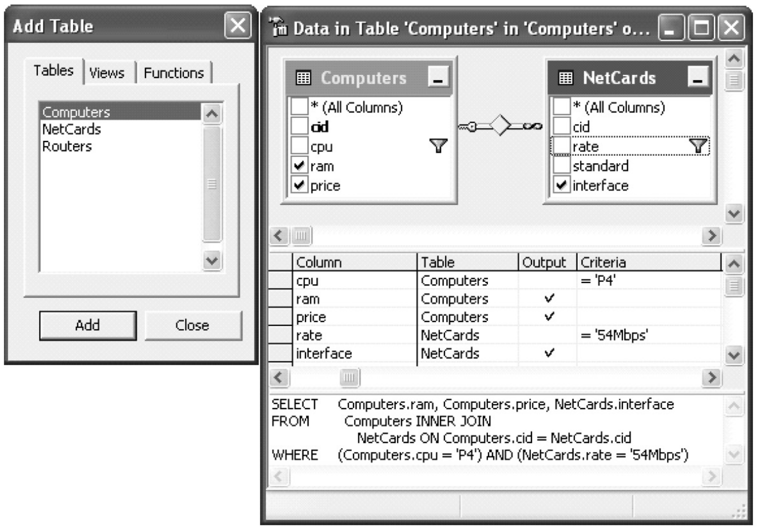
A rewriting R is *valid* only if there is an order $V_1, \ldots, V_n$ of the view aliases used in R's FROM clause such that if a column x is bound in V_i then R's WHERE clause includes either a selection atom V_i.x=*constant* or a join atom V_i.x=V_j.y where $j < i$. Note that valid rewritings have at least one viable evaluation plan, in which the values of the bound parameters for an alias V_i are provided by computing the plan induced by the aliases $V_1, \ldots, V_{i-1}$ (with respect to the above order).

In regard to extending the queries and views with more expressive predicates, the CLIDE Interface can be seamlessly extended to support queries and views with inequalities and arbitrary user-defined predicates for both selections and joins by treating them as uninterpreted predicate symbols. That means modeling the predicates as additional relations. This will result only in mappings between query and view (see Sections 6 and 8) which match predicate calls against calls to the same predicate if the arguments match. The net effect is that of loosing the guarantee of completeness of the front-end's suggestions, because by ignoring the semantics of the predicates existing rewritings may be missed. This is unavoidable since sufficiently expressive user-defined predicates lead to undecidability of the rewriting problem. On the other hand, for some built-in predicates, such as arithmetic comparisons ($<, \leq, >, \geq$), completeness of rewritings is preserved [Halevy 2001], which is sufficient to ensure completeness of the CLIDE Interface suggestions (see Section 6).

4. QUERY BUILDING INTERFACES

The CLIDE Interface front-end is a QBE-like [Zloof 1975] graphical interface. It adopts the Microsoft Query Builder interface (Microsoft SQL Server; go online to http://www.microsoft.com/sql/) as the basis for the interactive query formulation, since users are very familiar with it. Figure 4(a) shows a snapshot of the Microsoft 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 "54 Mb/s," and the bottom pane displays the corresponding SQL expression. The user can add to the top pane tables from the list shown on the left. The user can also formulate joins, like the one on cid.

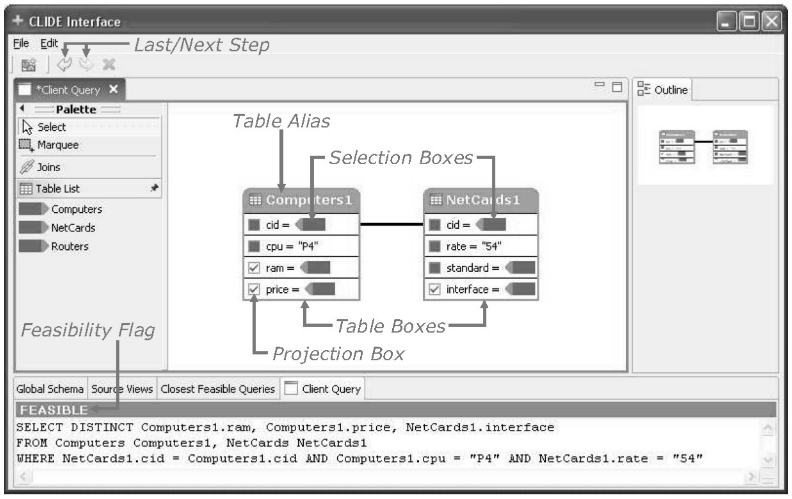
Figure 4(b) provides a snapshot of the CLIDE Interface front-end for the query of Figure 4(a). Apart from the feasibility flag and the coloring, the correspondence with the Microsoft Query Builder is straightforward: the front-end displays a *table box* for each table alias in the FROM clause. Selections 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



Column	Table	Output	Criteria
cpu	Computers		= 'P4'
ram	Computers	✓	
price	Computers	✓	
rate	NetCards		= '54Mbps'
interface	NetCards	✓	

```
SELECT    Computers.ram, Computers.price, NetCards.interface
FROM        Computers INNER JOIN
               NetCards ON Computers.cid = NetCards.cid
WHERE    (Computers.cpu = 'P4') AND (NetCards.rate = '54Mbps')
```

(a) Microsoft Query Builder



```
SELECT DISTINCT Computers1.ram, Computers1.price, NetCards1.interface
FROM Computers Computers1, NetCards NetCards1
WHERE NetCards1.cid = Computers1.cid AND Computers1.cpu = "P4" AND NetCards1.rate = "54"
```

(b) The CLIDE Interface front-end expressing the query of Figure 4(a)

Fig. 4. QBE-like query building interfaces.

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 in adding a selection atom to the `WHERE` clause.

(3) *Join action*. Dragging a column name and dropping it on another one results in 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.

5. INTERACTION IN THE PRESENCE OF LIMITED ACCESS METHODS

When not all $CQ^{=CA}$ queries against a database schema are feasible, the CLIDE Interface guides the user toward formulating feasible queries by coloring the possible next actions in a way that indicates what has to be done and 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.

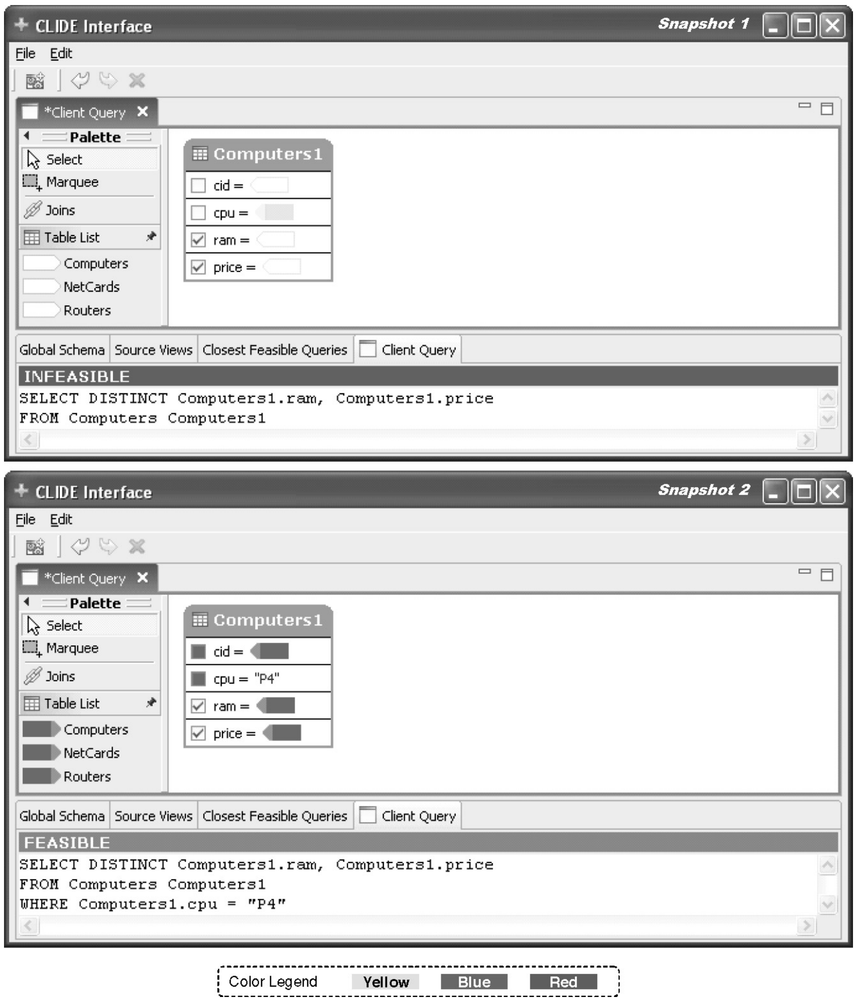
We illustrate the color scheme using the interaction session shown in Figures 5 and 6, which refers to the running example of Figure 2. The user wants to formulate a query that returns computers that meet various selection conditions, including conditions about network cards and routers—as long as those conditions are supported. Figures 5 and 6 show snapshots of the interaction session, where the color scheme of the CLIDE Interface front-end 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 (see Figure 5). The interface indicates that this query is infeasible (see feasibility flag at the bottom) 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 black and white printout) the `cpu` selection box. The rest of the selection boxes and projection boxes are *white*, suggesting that these actions are *optional*, that is, 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 (see Figure 2). This query is feasible since the middleware 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` table 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, then it would be yellow (i.e., required).

—*Nonobvious feasible queries.* Snapshot 3 in Figure 6 presents a complex case, where the interface's color scheme informs the user about nonobvious feasible



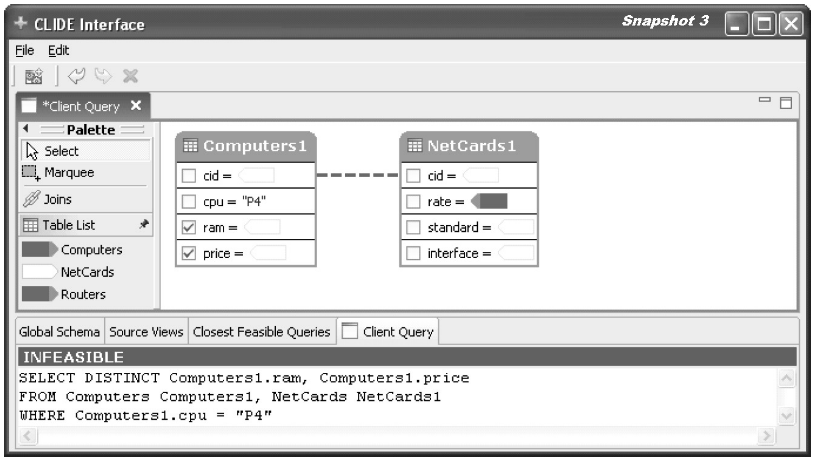
```
INFEASIBLE
SELECT DISTINCT Computers1.ram, Computers1.price
FROM Computers Computers1
```

```
FEASIBLE
SELECT DISTINCT Computers1.ram, Computers1.price
FROM Computers Computers1
WHERE Computers1.cpu = "P4"
```

Fig. 5. Snapshots of an interaction session (Snap shots 1 and 2).

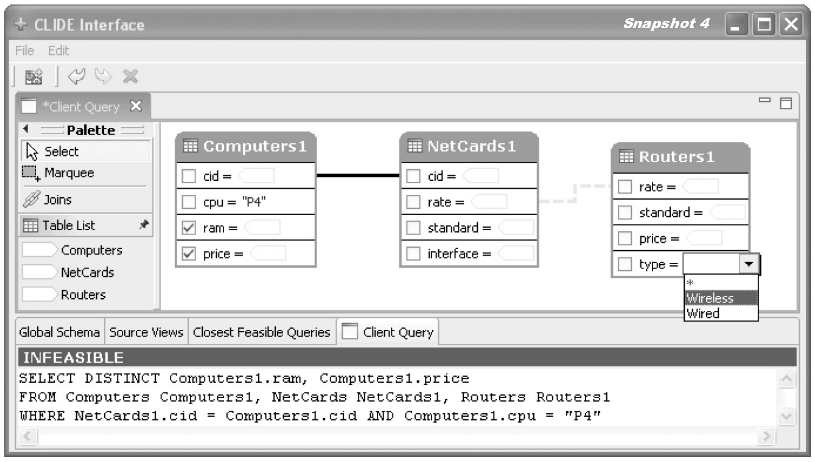
queries. After the user introduces a `NetCards` table, the interface suggests that one of the following extensions to the query is required: the join line between the `cid`'s of `Computers` and `NetCards` is suggested since it leads to the formulation of view V_2. It is blue since the user has more options: 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 the CLIDE Interface ignore them. In such a 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`



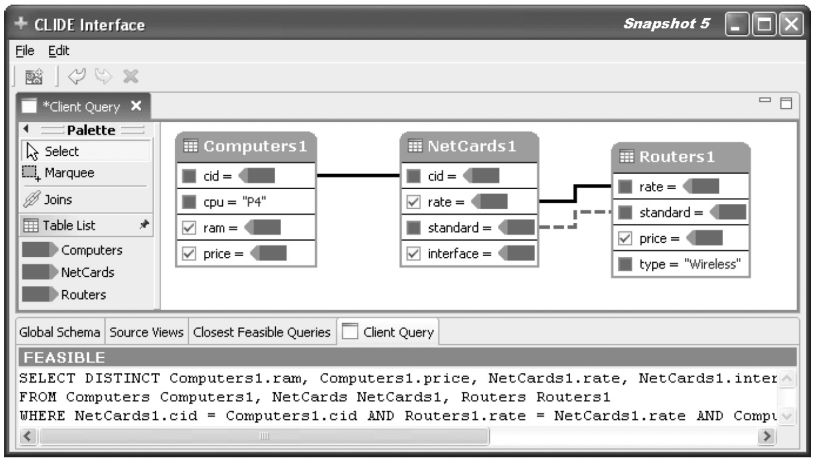
INFEASIBLE

```
SELECT DISTINCT Computers1.ram, Computers1.price
FROM Computers Computers1, NetCards NetCards1
WHERE Computers1.cpu = "P4"
```



INFEASIBLE

```
SELECT DISTINCT Computers1.ram, Computers1.price
FROM Computers Computers1, NetCards NetCards1, Routers Routers1
WHERE NetCards1.cid = Computers1.cid AND Computers1.cpu = "P4"
```



FEASIBLE

```
SELECT DISTINCT Computers1.ram, Computers1.price, NetCards1.rate, NetCards1.inter
FROM Computers Computers1, NetCards NetCards1, Routers Routers1
WHERE NetCards1.cid = Computers1.cid AND Routers1.rate = NetCards1.rate AND Compu
```

Color Legend Yellow Blue Red

Fig. 6. Snapshots of an interaction session (*continued*) (Snapshots 3, 4, 5).

and `Routers`, thus instantiating the `rate` parameter of V_2 with constants provided by another table.

—*Selection options*. In Snapshot 4 (Figure 6), 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 (Figure 5) shows the new query, which is feasible. The middleware composition 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 Interface 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, that is, no feasible query can be formulated by performing more actions and it is necessary to backtrack, that is, to undo actions.

5.1 Specification of the CLIDE Interface Color Scheme

Interaction sessions between the user and the CLIDE Interface 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 5.1 (*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 $e(n \xrightarrow{\texttt{a}} n') \in E$ 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 7 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 Figures 5 and 6. 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 $q(n)$, that is, with any query that uses other alias names. The CLIDE Interface 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 presentation purposes) and `i` is a number that is sequentially produced.[4]

Figure 7 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 $q(n)$. 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 7, 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 5.2 (*Colors*). Given an interaction graph $G_I = (N, s, E)$, a node $n \in N$, and an outgoing edge $e(n \xrightarrow{\texttt{a}} m)$, 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.

[4]Note that the aliases actually generated by the front-end on Figures 4, 5, and 6 use the whole table name. We used the three-letter abbreviation in the text for presentation purposes.

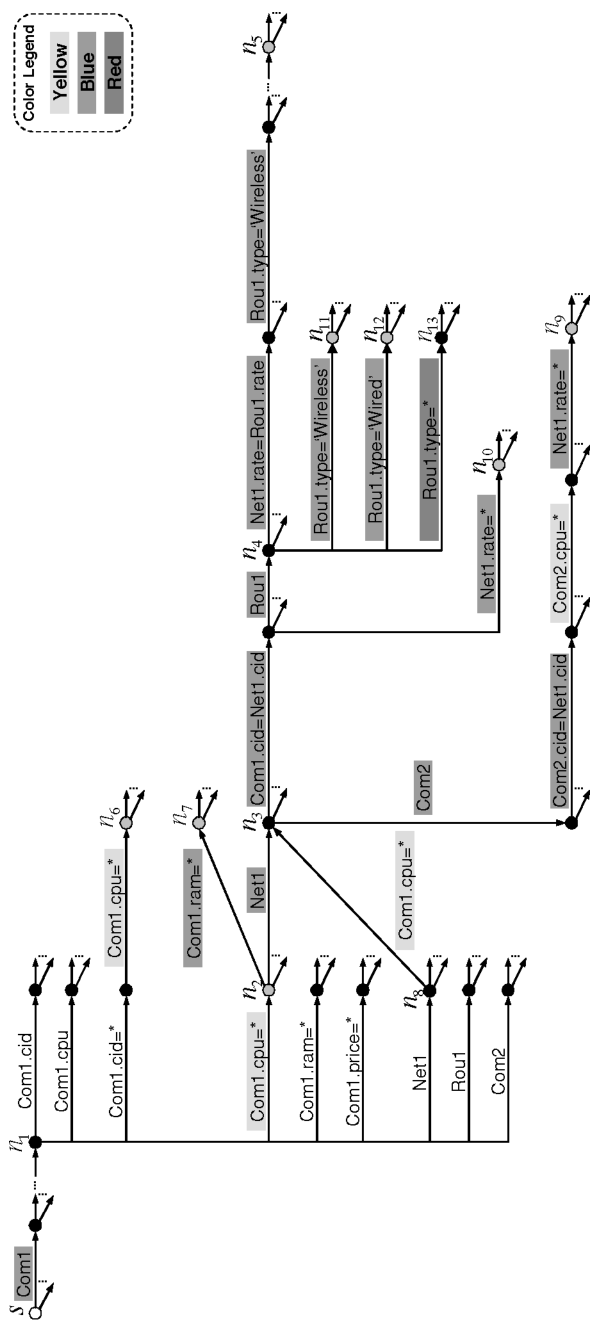


Fig. 7. Part of an interaction graph.

The CLIDE Interface assigns colors according to Definition 5.2 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.* In 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'.

6. THE CLIDE INTERFACE BACK-END

The CLIDE Interface 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.

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

We first observe that, in coloring, we can limit the inspection of feasible queries to those closest to n in the sense defined below:

Definition 6.1 (*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.

THEOREM 6.1. *The colors defined by Definition 5.2 with respect to all feasible nodes $FQ(n)$ coincide with the ones obtained with respect to only $FQ_C(n)$ (i.e., by substituting "feasible node" with "*closest *feasible node" in the definition).*

PROOF. Follows from the fact that all feasible nodes are reachable from some closest feasible node in $FQ_C(n)$, and that $FQ_C(n) \subseteq FQ(n)$.

For instance, if an action a is yellow with respect to $FQ(n)$, then it appears on all paths from n to feasible nodes, and in particular on all paths from n to nodes in $FQ_C(n)$. Therefore it is yellow also with respect to $FQ_C(n)$. Conversely, if a is yellow with respect to $FQ_C(n)$, it appears on all paths from n to closest feasible nodes, but then it also appears on all paths from n to feasible nodes, as such paths pass through closest feasible nodes.

Similarly, if a is blue with respect to $FQ(n)$, then there is some feasible node n_F such that the path $p = n \to n_F$ contains the label a and no path from n to n_F has any internal feasible node. Call n_F a witness for a being blue. But then $n_F \in FQ_C(n)$ and thus witnesses the fact that a should be blue with respect to $FQ_C(n)$. Conversely, if a is blue with respect to $FQ_C(n)$, then the closest feasible node n_F witnessing this is also a feasible node witness. Indeed, if a path from n to n_F has no internal node in $FQ_C(n)$, then it has none in $FQ(n)$, either.

The case for red actions is very similar. The coincidence of the white actions follows from the fact that they are simply the complement of the union of yellow, blue or red actions. □

By restricting our focus to closest feasible queries, we avoid the inspection of infinitely many queries, as there are infinitely many feasible queries which are not closest. For instance, consider the infinite set of all queries obtained as a multiway join of k copies of a feasible query, for every integer $k > 1$.

We do not yet have an algorithm, however, as $FQ_C(n)$ is known to be itself infinite when the views allow parameterized selections [Duschka et al. 2000]. Furthermore, the question on the finiteness of $FQ_C(n)$ in the absence of view parameters remains open at this time.

We enable an algorithmic coloring by showing that, at each interaction step, it is sufficient to consider only a representative subgraph of the interaction graph to color the actions. This subgraph constitutes a neighborhood of bounded, finite radius centered around n and can be therefore materialized. The representative subgraph approximates the set of feasible queries which are closest to n in the following sense:

Definition 6.2 (*Cover of a Node Set*). Let n be a node in the interaction graph.

(1) Given node m reachable from n, we say that node c *covers* m with respect to n if and only if it lies on a path from n to m and is distinct from n.[5]
(2) Given set S of nodes reachable from n, we say that set C *covers* S with respect to n iff each node in S is covered by some node in C.
(3) We call a cover C of set S with respect to node n *minimal* if removing any node from C leaves at least one node in S uncovered.

We omit specifying node n whenever it is clear from the context, thus talking about a cover C of S. Notice that the set $FQ_C(n)$ covers the set of feasible queries reachable from n. Also notice that a set S may have several minimal covers, depending on where on the path from n to a node $m \in S$ we pick the covering node.

In the remainder of this section, we focus on the case when parameterized selection atoms do not appear in views, postponing the treatment of parameters to Section 7.

We first prove in Section 6.1 the existence of a *finite* cover $FQ_R(n)$ of $FQ_C(n)$ which leads to coloring that satisfies the desired properties of completeness,

[5]Notice that m covers itself.

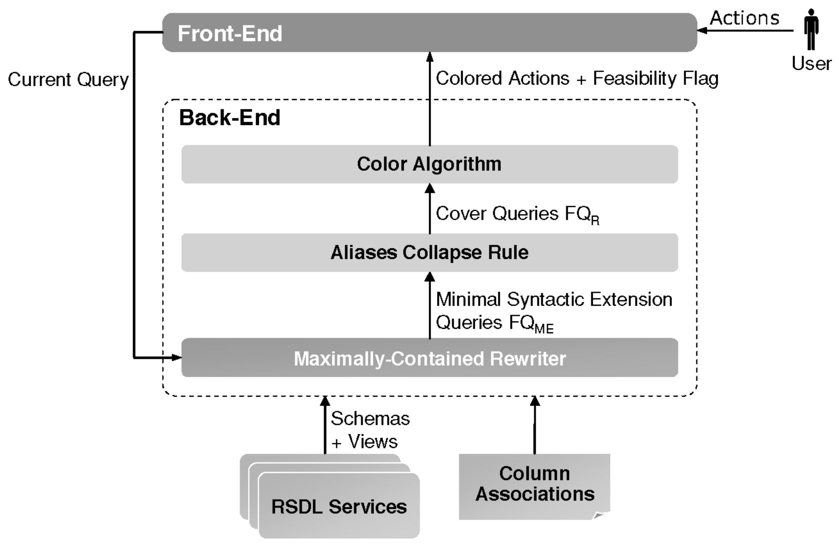


Fig. 8. The CLIDE System architecture for nonparameterized views.

minimality, and rapid convergence of the suggestions. We present optimizations for the computation of $FQ_R(n)$ which proved crucial to the CLIDE Interface's usability. We next address in Section 6.2 the challenge of efficiently coloring the possible actions given this finite cover. Our solution avoids the brute-force construction of the subgraph induced by node n and $FQ_R(n)$.

Figure 8 shows the architecture of the CLIDE System implementation, when parameterized selection atoms do not appear in the views. The *Maximally Contained Rewriter* and *Aliases Collapse Rule* components compute a finite cover $FQ_R(n)$ of the set $\boldsymbol{FQ}_C(n)$ and implement the algorithm given in Section 6.1. The *Color Algorithm* component inputs the set $FQ_R(n)$ and implements the algorithm given in Section 6.2. When parameterized selection atoms appear in the views, the *Color Algorithm* component inputs another finite cover, called a set of *seed queries* $\boldsymbol{SQ}(n)$, computed by the algorithm described in Section 7.

6.1 Finite Cover of the Closest Feasible Queries

6.1.1 *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, as they have additional conditions. It is therefore a natural starting point to focus on the maximally-contained [Halevy 2001] and feasible queries since they are the least constraining (semantically) and hence they have the least number of additional tables, selections, and joins, which makes them good candidates to be closest feasible queries. We adapt from Halevy [2001] the definition of the set of maximally-contained feasible queries 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 the set of maximally-contained feasible queries which are minimal syntactic extensions of $q(n)$ with $FQ_{ME}(n)$.

$FQ_{ME}(n)$ is known to be finite if we restrict $q(n)$ and the views to conjunctive queries with constant selection atoms (no parameters) [Duschka et al. 2000]. Section 8 describes how we extended the MiniCon algorithm [Pottinger and Halevy 2001] for finding maximally-contained rewritings to obtain $FQ_{ME}(n)$.

LEMMA 6.1. *All minimal feasible extensions of $q(n)$ which are maximally-contained in $q(n)$ are also closest feasible queries of $q(n)$ ($FQ_{ME}(n) \subseteq FQ_C(n)$).*

PROOF. By definition of the notion of syntactic extension, for any q_1 which is a syntactic extension of query $q(n)$, there exists a node n_1 reachable from n in the interaction graph, such that $q(n_1) = q_1$. Moreover, for the containment $q_1 \sqsubseteq q(n)$ to be meaningful, the two queries must have the same output schema and hence the path $n \rightarrow n_1$ must not contain any projection actions. In this case, the following claim is easily verified:

Claim 1. Let n_1 be a node reachable from node n in the interaction graph along a path containing no projection actions. Then for any node n_2 of the path $n \rightarrow n_1$, we have $q(n_1) \sqsubseteq q(n_2) \sqsubseteq q(n)$.

Proof of Claim 1. The claim is proven by observing that the only allowed actions on the path $n \rightarrow n_1$ guarantee the existence of a containment mapping from every node to its successors.

Claim 1 implies that no internal node n_2 of the path $n \rightarrow n_1$ is feasible. Indeed, assume toward a contradiction that $q(n_2)$ is feasible for some internal node n_2 on the path $n \rightarrow n_1$. Then if $q_1 \sqsubset q(n_2)$, this contradicts the maximal containment of q_1 in $q(n)$. If q_1 is equivalent to $q(n_2)$, this contradicts the extension-minimality of q_1 since dropping from q_1 all tables, joins, and selections introduced on the path $n_2 \rightarrow n_1$ results in the feasible, contained extension $q(n_2)$ of $q(n)$.

Since n_1 is reachable from n through no feasible query, $n_1 \in FQ_C(n)$. □

Lemma 6.1 provides a tool for finding some of the closest feasible queries. However, not all closest feasible queries can be obtained in this fashion as some of them are not in $FQ_{ME}(n)$. This point is illustrated by the next example.

Example 6.1. Assume that views V_1 and V_2 shown in Figure 2 are replaced by the following views V'_1 and V'_2, respectively, which contain constant selections only.

```
SELECT DISTINCT Com.*                                            (V'_1)
FROM Computers Com
WHERE Com.cpu='P4'
```

```
SELECT DISTINCT Com.*, Net.*                                     (V'_2)
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 7, then the only query in $FQ_{ME}(n_3)$ is $q(n_9)$ given below.

```
SELECT DISTINCT Com1.ram, Com1.price                             q(n_9)
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(n_10)
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 entire set of closest feasible queries, it turns out that we can use it as a starting point to search for queries which cover the ones in $FQ_C(n) \setminus FQ_{ME}(n)$.

6.1.2 *Alias Collapse Rule.*

The search starts from the queries in $FQ_{ME}(n)$ and rewrites them using the *alias collapse rule*. This rule 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`, substitute `R2` for `R1` in q, and drop the duplicate `R R1` atom.

Example 6.2. One can obtain the query $q(n_{10})$ from query $q(n_9)$ by collapsing the aliases `Com1` and `Com2`. $q(n_{10})$ covers $FQ_C(n_3) \setminus FQ_{ME}(n_3) = \{q(n_{10}\}$ as $q(n_{10})$ trivially covers itself.

In the above example, we obtained a covering query $q(n_{10})$ which happened to be feasible. There are examples showing that in general, the covering queries may not be feasible themselves, having extensions to feasible queries instead.

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 6.2. *For any $n_1 \in FQ_C(n) \setminus FQ_{ME}(n)$, there exists $n_2 \in FQ_{ME}(n)$ and n_3 such that $q(n_3)$ is obtained from $q(n_2)$ by repeatedly applying the alias collapse rule, and n_3 covers n_1 with respect to n.*

PROOF. Since n_1 is a closest feasible query, we claim that $q(n_1) \sqsubseteq q(n)$. To prove this, we use the following claim:

Claim 2. If $q(n_1) \in FQ_C(n)$, then every path $n \to n_1$ contains no projection actions.

Proof of Claim 2. Assume toward a contradiction that the path $n \to n_1$ does contain projection actions. Then there is a query $q(n_2)$ for a node n_2 such that

—n_2 is reachable from n (along a path obtained by dropping the projection actions from path $p = n \to n_1$);
—n_1 is reachable from n_2 (along a path consisting solely of the projection actions dropped from path p); and
—$q(n_2)$ is feasible, since dropping projections does not affect feasibility of a query.

But then $q(n_2)$ witnesses that q_1 is not in $FQ_C(n)$, as it is reachable from feasible query $q(n_2)$. □

By Claim 2 and Claim 1 from the proof of Lemma 6.1, it follows that $q(n_1) \sqsubseteq q(n)$.

Now, since $n_1 \notin FQ_{ME}(n)$, $q(n_1)$ is not maximally contained in $q(n)$ and there exists $n_2 \in FQ_{ME}(n)$ such that $q(n_1) \sqsubset q(n_2)$. Hence there exists a containment mapping h from $q(n_2)$ into $q(n_1)$. We call a pair of table atoms $R\ r_1$, $R\ r_2$ in $q(n_2)$ which have the same image under h a *collision*.

The lemma now follows immediately from the following claim:

Claim 3. We prove that whenever there is a containment mapping h from $q(n_2)$ into $q(n_1)$ with k collisions, then a subquery $q(n_3)$ of $q(n_1)$ can be obtained from $q(n_2)$ by applying k alias collapse steps.

Proof of Claim 3. The proof is by induction on k. The base case $k = 0$ is trivial, as the image of $q(n_2)$ under h gives the subquery $q(n_3)$. For the step, assume $k + 1$ collisions under h. Pick a collision pair of table atoms Rr_1, Rr_2 in $q(n_2)$. These have the same image under h. Substitute r_1 for r_2 in $q(n_2)$ and drop the duplicate table atom to obtain a query q'. Notice that h remains a containment mapping from q' into $q(n_1)$, exhibiting only k collisions. By induction hypothesis, $q(n_3)$ is obtained by k alias collapse steps from q', therefore by $k + 1$ collapse steps from $q(n_2)$. □

Lemmas 6.1 and 6.2 lead to the following algorithm for computing a minimal cover of $FQ_C(n)$.

```
algorithm Cover$FQ_C$
Input: node $n$
Output: a minimal cover of $FQ_C(n)$

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

// compute a cover of $FQ_C(n) \backslash FQ_{ME}(n)$ in $AC$:
let $AC :=$ the empty set $\emptyset$
for each $q_M \in M$ do
    for each pair of distinct aliases $r_1$,$r_2$ of some relation in $q_M$ do
        let $q' :=$ collapse $r_1$ and $r_2$ in $q_M$
        $AC :=$ AliasCollapse($q(n), q', AC$)
return $M \cup AC$
end

procedure AliasCollapse
Input: query $q_0$, query $q$, query set $AC$
Output: all syntactic extensions of $q_0$ which cover at least one query in $FQ_C(n)$
        and are obtainable from $q$ by collapsing aliases
        begin
        if $q$ is unsatisfiable or $q$ is not an extension of $q_0$, return the empty set $\emptyset$

        if $q$ has some feasible extension (checkable by testing $FQ_{ME}(q) \neq \emptyset$)
            $AC := AC \cup \{q\}$
            for each pair of distinct aliases $r_1$,$r_2$ of some relation in $q$ do
                let $q' :=$ collapse $r_1$ and $r_2$ in $q$
                $AC :=$ AliasCollapse($q', AC$)
            return $AC$
            end
```

We denote with $FQ_R(n)$ the result of running algorithm **CoverFQ_C** at node n. The following shows that coloring actions with respect to $FQ_R(n)$ instead of $FQ(n)$ suffices to guarantee the desired properties of the interaction.

THEOREM 6.2. *Coloring actions with respect to $FQ_R(n)$ results in an interaction which exhibits completeness, minimality of suggestions and rapid convergence.*

Theorem 6.2 follows from the following lemmas.

LEMMA 6.3. *$FQ_R(n)$ is a finite minimal cover of $FQ_C(n)$.*

PROOF. The fact that $FQ_R(n)$ is a cover of $FQ_C(n)$ follows immediately from Lemmas 6.1 and 6.2. Finiteness follows from the fact that $FQ_{ME}(n)$ is known to be finite [Duschka et al. 2000] and the fact that every query admits only a finite number of alias collapse steps. Minimality follows from the observation that for any nodes n_1, n_2 such that $q(n_2)$ is obtained from $q(n_1)$ via alias collapse steps, n_1 is not reachable from n_2 in the interaction graph and conversely, n_2 is not reachable from n_1. □

LEMMA 6.4. *Let n be the current query node.*

(1) For every action a *applicable to $q(n)$,* a *is colored yellow with respect to $\boldsymbol{FQ}(n)$ if and only if it is colored yellow with respect to $FQ_R(n)$.*

(2) If action a *is colored blue with respect to $FQ_R(n)$, it is also colored blue with respect to $\boldsymbol{FQ}(n)$.*

(3) For every node $n_F \in \boldsymbol{FQ}(n)$, coloring the actions applicable at node n with respect to $FQ_R(n)$ results in at least one yellow or blue action at n that enables progress toward n_F.

PROOF. (1). If action a is colored yellow with respect to $\boldsymbol{FQ}(n)$, then it appears on the paths to all feasible nodes and thus to all nodes in $\boldsymbol{FQ}_{ME}(n)$ since $\boldsymbol{FQ}_{ME}(n) \subseteq \boldsymbol{FQ}_C(N) \subseteq \boldsymbol{FQ}(n)$. But, as shown in the proof of Lemma 6.2, for all nodes $n_1 \in FQ_R(n) \setminus \boldsymbol{FQ}_{ME}(n)$ there is some node $n_2 \in \boldsymbol{FQ}_{ME}(n)$ and a containment mapping from $q(n_2)$ into $q(n_1)$. Hence the table, selection or join introduced by action a into $q(n_1)$ also appears in $q(n_2)$ as the image of the containment mapping. This shows that a appears on the paths to all nodes in $FQ_R(n)$ and is hence colored yellow with respect to $FQ_R(n)$.

(2). If a is colored blue with respect to $FQ_R(n)$, there is some witness $n_c \in FQ_R(n)$ such that the paths from n to n_c contain a and there is no other node from $FQ_R(n)$ on these paths. Since by Lemma 6.3 $FQ_R(n)$ is a minimal cover of $\boldsymbol{FQ}_C(n)$, there exists some node $n_F \in \boldsymbol{FQ}_C(n)$ reachable from n_c, which is a witness of a being colored blue with respect to $\boldsymbol{FQ}_C(n)$. By Theorem 6.1, a is also colored blue with respect to $\boldsymbol{FQ}(n)$.

(3). Follows from the fact that $FQ_R(n)$ is a cover of $\boldsymbol{FQ}_C(n)$: any action enabling progress toward a feasible node $n_F \in \boldsymbol{FQ}(n)$ must enable progress toward the node $n_c \in FQ_R(n)$ covering n_F. □

PROOF OF THEOREM 6.2. Completeness follows from Lemma 6.4(3). Minimality follows from the fact that $FQ_R(n)$ is a minimal cover of $\boldsymbol{FQ}(n)$ (implied by Lemma 6.3). Rapid convergence follows from the fact that all paths from a node n to a node n' have the same length, as they must eventually perform the same set of syntactic extensions. □

6.2 Color Algorithm

After computing a minimal cover $FQ_R(n)$ of the set of closest feasible queries $\boldsymbol{FQ}_C(n)$, the CLIDE Interface 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 and only if $q(n) \in \boldsymbol{FQ}_{ME}(n)$, and infeasible otherwise.

Theorem 6.2 shows that, instead of working with the infinite interaction graph, we can restrict our attention to the finite *close subgraph* consisting of n, all nodes in $FQ_R(n)$ and the paths between them. In this section we show how to color the actions without actually materializing the close subgraph.

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.

6.2.1 *Blue and Yellow.* We color a join a yellow if it appears in all nodes of $FQ_R(n)$, and blue if it appears in some. In the case of a table action T, we color it yellow (respectively blue) if in all (respectively some) query in $FQ_R(n)$ 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, that is, 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.[6]

6.2.2 *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 $\boldsymbol{FQ_{ME}}(n')$ is empty (i.e., if it has no feasible extensions). If so, a is colored red, otherwise white. This approach, although simple, requires the nonemptiness test of $\boldsymbol{FQ_{ME}}(n')$, which is an expensive operation, as the experiments of Section 9 demonstrate. Hence, we need to devise more efficient techniques for coloring red and white actions.

Table actions are colored red if not used in any view, and white otherwise, 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 query q_F labeling a node in $FQ_R(n)$. 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 6.1, 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 8 we

[6]Note that generic selections can be colored yellow or blue when parameterized selections appear in the views. Please see Section 7 for details.

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 Interface front-end does not display white and red joins, so they are not a consideration.

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

7. PARAMETERS

When parameterized selection atoms appear in the views, the algorithms in Sections 6.1 and 6.2 need to be extended to deal with a new challenge: the set $FQ_{ME}(n)$ of maximally contained queries becomes infinite [Duschka et al. 2000] and therefore searching for a (cover of) the set of closest feasible queries starting from $FQ_{ME}(n)$ becomes problematic. The following example illustrates this point.

Example 7.1. 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, that is, 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 Mngrs 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 Mngrs 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 a second table atom `Empls E2`, then the front-end colors the join `E1.eid=E2.mid` yellow, which formulates q_3:

```
SELECT DISTINCT M1.salary                                          q3
FROM Mngrs 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 Mngrs 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, indeed they are maximally contained in q_1. For example, q_{2F} and q_{4F} are two such queries, there is no containment between them, nor any sequence of actions that applied on q_{2F} formulate q_{4F}.

Recall from Section 6 that the algorithms for finding the cover of $FQ_C(n)$ as well as the coloring algorithms relied on enumerating the set $FQ_{ME}(n)$, which according to Example 7.1 becomes impossible.

Instead, the CLIDE Interface back-end identifies a finite set of *seed queries* $SQ(n)$, possibly parameterized, where $q(n)$ is the current query. These are not necessarily feasible, but have the property that $SQ(n)$ is a cover of $FQ(n)$. In Example 7.1, q_{2F} is a feasible seed query of q_1, while q_3 is an infeasible one, which however covers 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 6.2 on $SQ(n)$ instead of $FQ_R(n)$. This approach does not compromise the guarantees of completeness, minimality of suggestions and rapid convergence.

It is a priori nonobvious 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.

The detailed algorithm computing $SQ(n)$ is given below and proceeds in seven steps. The first six compute a set of feasible syntactic extensions of $q(n)$, while Step 7 performs the alias collapsing as in Algorithm $\mathbf{Cover}FQ_C$.

Step 1. Parameterized selections are removed from the input views and columns in the projection lists are annotated with either a '*b*' (bound) or '*f*' (free) binding pattern (see Section 3). This step adapts the input to the one expected by standard algorithms for computing the maximally-contained rewritings in the absence of parameters.

Step 2. Maximally-contained rewritings are computed in terms of the views obtained in Step 1, without taking into account the binding patterns. The resulting rewritings are expressed in terms of the views, each of which has retained the binding patterns computed in Step 1.

Step 3. '*b*' binding patterns of columns in the rewritings are eliminated if during the computation of the rewritings the algorithm added constant selections.

Step 4. More '*b*' binding patterns of columns in the rewritings are eliminated if the rewriting algorithm added joins with '*f*' binding patterns. The elimination at steps 3 and 4 is meant to detect the parameters whose values still need to be provided by the user.

Step 5. The rewritings computed in the previous step might still have '*b*' binding patterns, signaling that the corresponding parameter values need to be provided by the user. This is always doable via selections with constants. In addition, parameter values can be provided via parameter-passing joins. This step extends the rewritings to rewritings which eliminate '*b*' binding patterns via such parameter-passing joins, either using views already in the rewritings or with newly added views.

Step 6. Views are expanded so that the rewritings can be used by the coloring algorithm. The '*b*' binding patterns are translated into parameterized selections. The obtained set of queries are syntactic extensions of $q(n)$ which can be made feasible by providing values for the parameterized selections.

Step 7. The queries obtained at Step 6 are now postprocessed through alias collapse steps, similarly to the last stage of the algorithm in Section 6.1.

```
**algorithm Compute**$SQ$
**Input:** node $n$, view set $\mathcal{V}$
**Output:** A set of seed queries $SQ(n)$ which covers $FQ_C(n)$

**begin**
// Step 1: remove parameters and attach a binding pattern to the views
let $\mathcal{V}_{BP}$ := the empty set $\emptyset$
for each view $V_i$ in $\mathcal{V}$
    for each column $c_i$ in the projection list of $V_i$
        annotate $c_j$ with binding pattern '$f$'
    for each column $c_i$ participating in a parameterized selection
        add $c_i$ to the projection list of $V_i$, if not already present
        annotate $c_i$ with binding pattern '$b$'
        remove from $V_i$ the parameterized selection which $c_i$ participates in
    $\mathcal{V}_{BP} := \mathcal{V}_{BP} \cup \{V_i\}$

// Step 2: compute maximally-contained rewritings
Compute $A$, the set of maximally-contained conjunctive query rewritings
              in terms of the views in $\mathcal{V}_{BP}$, ignoring the binding patterns

// Step 3: eliminate '$b$' binding patterns based on selections
for each $q_A \in A$
    for each view $V_i$ in the FROM clause of $q_A$
        for each column $c_i$ in the projection list of $V_i$ with binding pattern '$b$'
            if $c_i$ participates in a selection with a constant
                annotate $c_i$ with binding pattern '$f$'

// Step 4: eliminate '$b$' binding patterns based on joins
let $M$ := the empty set $\emptyset$
for each $q_A \in A$
```

```
    for each possible order of the views in the FROM clause of $q_A$
    containing a binding pattern ‘$b$’
        create query $q_A^i$ with the given order of the views in the FROM clause
        for each view $V_i$ in the FROM clause of $q_A^i$ containing a binding pattern ‘$b$’
            for each column $c_i$ in the projection list of $V_i$ with binding pattern ‘$b$’
                if $c_i$ participates in a join with a column $c_j$ with binding pattern ‘$f$’
                coming from a view $V_j$, where $j < i$
                   Annotate $c_i$ with binding pattern ‘$f$’
        $M := M \cup q_A^i$

// Step 5: eliminate ‘$b$’ binding patterns by adding parameters-passing joins
for each $q \in M$
    for each view $V_i$ in the FROM clause of $q$ containing a binding pattern ‘$b$’
        for each column $c_i$ in the projection list of $V_i$ with binding pattern ‘$b$’
            if there is a column association between $c_i$ and a column $c_j$
            with binding pattern ‘$f$’ that appears in the projection list
            of a view $V_j$, where $j < i$
               create a copy $q'$ of $q$
               add a join between $c_i$ and $c_j$ in $q'$
               annotate $c_i$ in $q'$ with binding pattern ‘$f$’
               $M := M \cup q'$
            if there is a view $V_j$ in $\mathcal{V}_{BP}$, such that there is a column association
            between $c_i$ and a column $c_j$ that appears in the projection list
            of $V_j$ with binding pattern ‘$f$’
               create a copy $q'$ of $q$
               add a new alias of $V_j$ in $q'$
               add a join between $c_i$ and $c_j$ in $q'$
               annotate $c_i$ in $q'$ with binding pattern ‘$f$’
               $M := M \cup q'$

// Step 6: expand the views
for each $q \in M$
    expand the views in $q$ such that parameterized selections are added for columns
    in the projection lists of the views with binding pattern ‘$b$’
    and such that the resulting query is a minimal syntactic extensions of $q(n)$


// Step 7: collapse aliases
run the code of Algorithm **Cover**$FQ_C$ from Section 6.1 starting at line 2 (let $AC := \emptyset$)
this code operates on the set $M$ of feasible syntactic extensions constructed in steps 1
through 6.
**end**
```

In regard to *coloring*, once the set $SQ(n)$ of seed queries is returned by Algorithm **Compute**SQ, the coloring is performed similarly to Section 6.2, using $SQ(n)$ instead of $FQ_R(n)$.

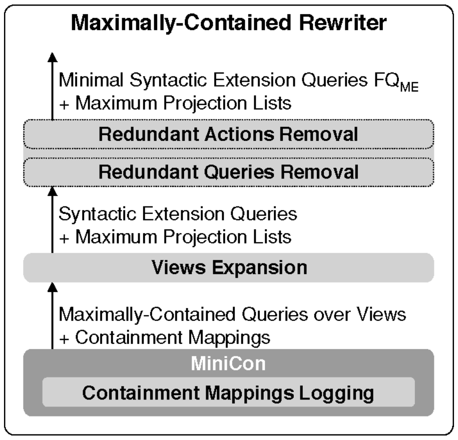

Maximally-Contained Rewriter
Minimal Syntactic Extension Queries FQ_ME + Maximum Projection Lists
Redundant Actions Removal
Redundant Queries Removal
Syntactic Extension Queries + Maximum Projection Lists
Views Expansion
Maximally-Contained Queries over Views + Containment Mappings
MiniCon
Containment Mappings Logging

Fig. 9. MiniCon optimizations and extensions.

8. IMPLEMENTATION

The current implementation of the CLIDE Interface consists of the components shown in Figure 8, where the views may be parameterized.

The CLIDE Interface back-end uses MiniCon [Pottinger and Halevy 2001] 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 back-end's color algorithm and achieve on-line performance. Figure 9 illustrates the anatomy of the maximally-contained rewriting component from Figure 8.

8.1 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 6), MiniCon produces the following rewriting query q_R that combines V_1' and V_2':

```
SELECT DISTINCT V'_1.ram, V'_1.price                                  q_R
FROM V'_1, V'_2
```

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 cover 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 [Abiteboul et al. 1995] 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 9.

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 6.2. In Section 6.2, we defined a maximum projection list to be the list of all possible projections that can be added to a query without compromising feasibility. Note that this definition suggests a direct yet expensive brute-force algorithm which adds each projection action and checks feasibility by calling MiniCon. A much more efficient, linear-time procedure is possible due to the fact 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.

8.2 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, that is, expanded queries might contain one another. For example, if the current query is $q(n_1)$ in Figure 7 (Snapshot 1 in Figure 5), 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 6.1.

In order to preserve the rapid convergence and minimality guarantees of the CLIDE Interface (see Section 5.1), 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 [Yannakakis 1981]. We compute the containment mappings from query q_1 into query q_2 by constructing a canonical database [Abiteboul et al. 1995] 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 the CLIDE Interface, since query containment tests are the bottleneck for the performance of the back-end, as Section 9 demonstrates.

8.3 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.

8.4 Skipping MiniCon Calls

We conclude this section mentioning an optimization which we are currently implementing into the CLIDE Interface back-end. It concerns the case when parameterized selections do not appear in the views and is based on the intuition that each interaction step along an edge $n \xrightarrow{a} n'$ changes $q(n)$ only incrementally. Therefore it is natural to investigate whether it is possible to compute $FQ_R(n')$ incrementally from $FQ_R(n)$ instead of doing so from scratch by an expensive call to MiniCon.

Indeed, if a is a yellow or blue action, the cover set $FQ_R(n')$ is contained in $FQ_R(n)$ and we do not need to call MiniCon to compute $FQ_R(n')$. Instead, we can inspect the containment mappings from $q(n)$ into $FQ_R(n)$ and we can compute $FQ_R(n')$ by pruning those mappings that are inconsistent with action a and dropping from $FQ_R(n)$ all queries left with no more more containment mapping. This optimization saves calls to MiniCon during those stretches of the interaction in which the user performs only yellow and blue actions, progressing toward but not yet reaching any query in $FQ_R(n)$.

9. EXPERIMENTAL EVALUATION

Our experimental evaluation shows that the CLIDE Interface is a viable online tool. The MiniCon algorithm was evaluated via extensive experiments in Pottinger and Halevy [2001] 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 Interface back-end components do not add a prohibitive cost, and that the algorithms of Sections 6.1, and 6.2, as well as our extension to parameters (Section 7) and optimizations (efficient implementation of containment test, logging MiniCon's containment mappings, see Section 8) are crucial in obtaining quick response times.

9.1 The Experimental Configuration

To study how the CLIDE Interface 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.

9.1.1 *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 $\{C_{i,j}(kc, fc, c, c')\}_{0\leq j\leq L}$. 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.

9.1.2 *The Views.* The MiniCon experiments in Pottinger and Halevy [2001] 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, 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.

9.1.3 *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, the CLIDE Interface front-end suggests joins with the B_i's. If any of these suggestions are taken (say by picking B_p), the front-end 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 $C_{p,o}$, the front-end 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 $V_{p,o}^r$.

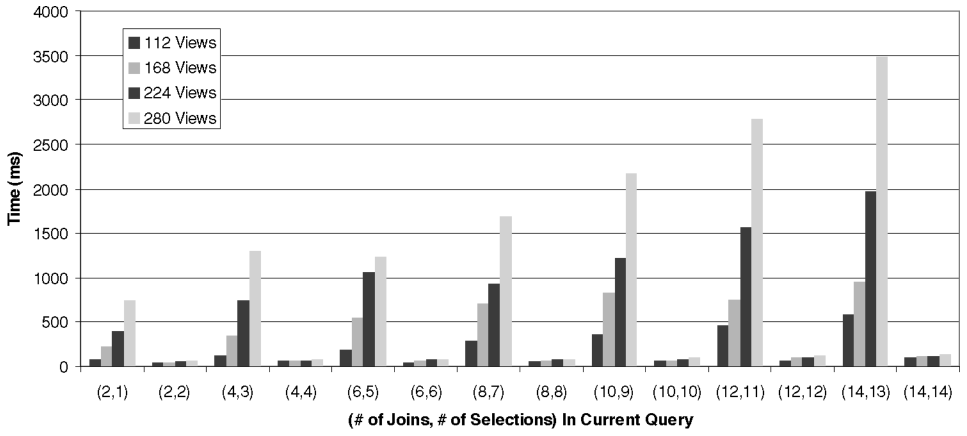


Fig. 10. The CLIDE Interface response time (no parameters).

9.2 The Measurements

The measurements were conducted on a dedicated workstation (Intel Core2 Duo 2.0 GHz, MS Windows XP Pro, 2-GB RAM) using Sun's JRE-1.6.0. 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 10 reports the time CLIDE Interface 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 the CLIDE Interface response is good overall, scaling to large number of views, it is much better for feasible queries. This is an expected result, since the back-end needs to consider a single cover query, that is, the one that the user has reached, as opposed to the number of cover queries when the current query is infeasible.

9.3 The Effect of Parameterized Views

We studied the impact of parameters on the running time of the algorithm of Section 7 by extending the experimental configuration described above as follows.

The schema and the systematic way of generating user interactions are the same.

The family of available views is extended to include a mix of unparameterized and parameterized views: for each view V_i^n (defined above) we add a view VP_i that performs a selection using a parameter p rather than the constant c_n, and similarly we add view $VP_{i,j}$ which turns the selection with constant c_n in view

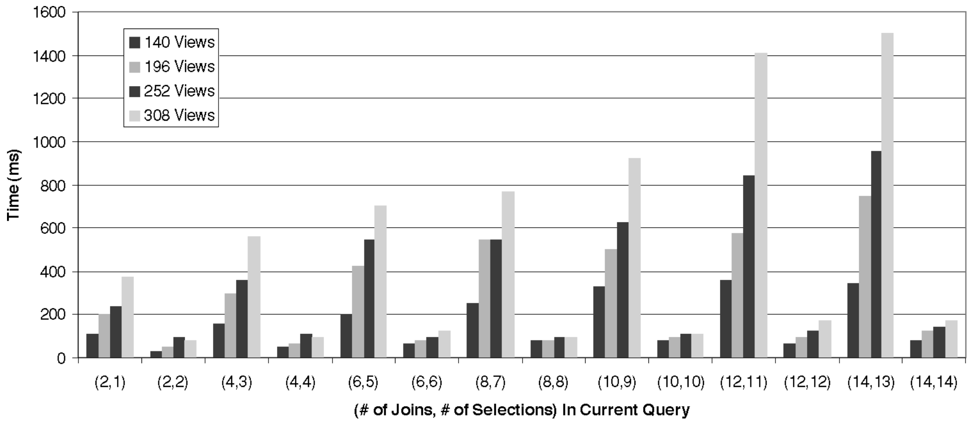


Fig. 11. The CLIDE Interface response time (with parameters).

$V^n_{i,j}$ into a selection by a parameter p:

$VP_i(p) \rightarrow (a, kb, b)*$
```
    SELECT x.a, y.kb, y.b
    FROM A x, B_i y
    WHERE x.ka=y.fb AND y.b=p
```

$VP_{i,j}(p) \rightarrow (kb, b, c)*$
```
    SELECT y.kb, y.b, z.c, z.c'
    FROM B_i y, C_{i,j} z
    WHERE y.kb=z.fc AND z.c=p
```

In the experiments, we simultaneously used the parameterized and unparameterized view families $\{V^n_i, V^n_{i,j}\}_{0\le i\le K, 0\le j\le L, 0\le n\le M} \cup \{VP_i, VP_{i,j}\}_{0\le i\le K, 0\le j\le L}$.

We also declared *column associations* between the b attribute of each B_i table and the c' attribute of each $C_{i,j}$ table: $\{(B_i.b, C_{i,j}.c')\}_{0\le i\le K, 0\le j\le L}$. These associations are exploited by CLIDE to propose parameter-passing joins leading to feasibility due to the parameterized views. In this setting, the joins can provide the parameter required by a view VP_i from the c' output of a $VP_{i,j}$ or of a $V^n_{i,j}$ view. The interface lists both options to the user.

The running times are reported in Figure 11, where the configurations are generated using the same instantiation of K, L, M as in the nonparameterized experiments: $K = 7$, $L = 3$, and $M = 4, 6, 8, 10$, yielding 140, 196, 252, 308 views, respectively. We note that, for up to 252 views, all response times are in the subsecond range even for large queries (up to 14 joins and 14 selections).

Note that the running times are even better than for the nonparameterized case. This is because each rewriting using unparameterized views V^n_i and $V^n_{i,j}$ is contained in a rewriting achieved by replacing the views with VP_i and $VP_{i,j}$, respectively, and by instantiating the parameters with the appropriate constants. The parameterized rewritings obtained after running MiniCon thus lead to the elimination of some of the unparameterized ones during the redundant query removal step (described in Section 8). The eliminated rewritings are no longer compared to other rewritings, thus yielding significant savings in the number of calls to the containment test.

9.4 Profiling

In addition to measuring the overall response time of the interface, we also broke down the time measurements by stages of the algorithm (details are given in Section 8, Figure 12). It turns out that, both in the absence and presence of parameters, the bottleneck in the performance of the CLIDE Interface lies in the containment tests, which are a consequence of the views expansion. In particular, the bottleneck is caused by both the number of containment tests that need to be performed and by their complexity, which is analyzed next.

10. COMPLEXITY ANALYSIS OF THE CLIDE INTERFACE BACK-END

Since the algorithms implemented by the CLIDE Interface back-end are run at every interaction step, their complexity impacts the time delay experienced by the user. We analyze the complexity of these algorithms, pointing out the steps where our optimization techniques achieved significant time savings. We introduce the following notation:

S_Q	Size of current query Q (number of table aliases in FROM clause)
N_V	Number of views
S_V	Maximum view size
N_R	Number of maximally-contained rewritings returned by MiniCon for current query Q
S_R	Maximum size of a rewriting (among the N_R)
S_{ER}	Maximum size of a rewriting's expansion

10.1 Redundant Query Removal

At every step, MiniCon returns N_R maximally-contained rewritings of current query Q. To guarantee minimality of suggestions, the CLIDE Interface removes redundant queries as explained in Section 8. This step involves computing pairwise containment mappings between the expansions of all rewritings, and it turns out to dominate the back-end's reaction time since

- there are many rewritings to pairwise test for containment ($O(N_R^2)$); and
- finding a containment mapping is expensive if not implemented carefully, as the problem is NP-hard in the size of the containing query [Abiteboul et al. 1995; Chandra and Merlin 1977]; here the queries are expansions of the rewritings using the views, so a direct implementation would require $O(S_{ER}^{S_{ER}})$ time.

In detail, we recall that each rewriting is known [Pottinger and Halevy 2001] to have size upper-bounded by that of the query ($S_R \leq S_Q$), which yields a number of rewritings upper-bounded by $N_R \leq N_V^{S_Q}$ and a maximum size of an expanded rewriting bounded by $S_{ER} \leq S_Q \times S_V$. Assuming a straightforward implementation of the containment mappings search, the overall effort for redundant query removal is therefore $O(N_R^2 \times S_{ER}^{S_{ER}})$, which is upper-bounded by $O(N_V^{2S_Q} \times (S_Q \times S_V)^{S_Q \times S_V})$. Note that, while the exponent S_Q is relatively

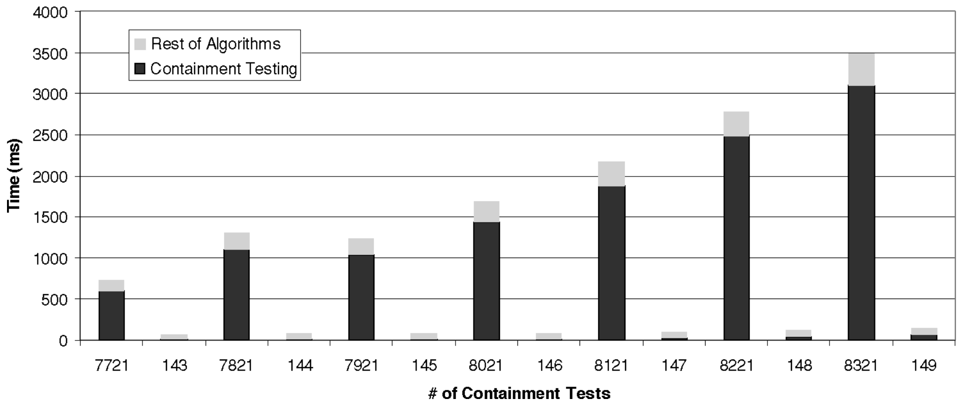


Fig. 12. The CLIDE Interface response time analysis for 280 views.

small in practice (user queries tend to join few tables), it occurs significantly amplified by the view size.

A major optimization effort we expended went into computing the containment mappings more efficiently, effectively reducing the exponent $S_Q \times S_V$. To this end, we used the well-known observation that computing a containment mapping from query q_1 into query q_2 amounts to running q_1 over a symbolic database instance containing one tuple for each alias in q_2's FROM clause, such that the attributes satisfy q_2's WHERE clause. This database is commonly known as the *canonical database* of q_2 [Abiteboul et al. 1995]. This prompted us to compile q_1 into a query plan using standard relational algebra operators, and evaluate the plan over the canonical database, applying the well-known optimization techniques of pushing selections and projections into the joins [Garcia-Molina et al. 2001]. This yields an improvement in worst-case complexity (and even more spectacular improvements in practice, as shown by the experiments below) analyzed next. After every join operator j, we aggressively project away all columns that are not required for the evaluation of the remaining plan, that is, we only keep columns in the projection list and columns involved in joins with columns not constructed by the plan rooted at j. We call the number of columns we need to keep the *width* of operator j and call the *width of the plan* the maximum width over all join operators in the plan. It immediately follows that the evaluation of the plan runs in time exponential only in the plan width, and the running time for the redundant query removal phase becomes

$$O\left(N_V^{2S_Q} \times (S_Q \times S_V)^w\right),$$

where w denotes the maximum plan width over all rewriting expansions. This is a significant improvement, as plan width is in practice quite close to the number of columns in the projection list and significantly smaller than the size of the rewriting expansion $S_Q \times S_V$.

Figure 12 analyzes the response time of the CLIDE Interface when the input is 280 views and the same set of user queries as in Figure 10. The horizontal axis shows the number of containment tests that were performed for each user query. According to the experimental configuration of Section 9, all odd-position queries are infeasible, while all even-position queries are feasible. The vertical axis shows the portions of the response time attributed to containment tests and the rest of the back-end algorithmic components including MiniCon, respectively. For all feasible queries shown in Figure 12, the initial number of rewritings received by MiniCon was 70, while the final number of rewritings after the containment tests was understandably 1, the one that is equivalent to the user query. For all infeasible queries shown in Figure 12, the initial number of rewritings received by MiniCon was 700, a much higher number than expected. The final number of rewritings after the containment tests, which serve as input to the Color Algorithm, was just 10.

The MiniCon implementation we received was including a naive implementation of a containment test component, which was developed for testing and verification purposes only and not for performance experiments. Using this version of the containment test, none of our experiments actually terminated.

10.2 Redundant Actions Removal

This phase (detailed in Section 8) consists in searching in the rewriting expansion for pairs of identical constants referring to identical tables with distinct aliases, attempting to drop one constant, and testing equivalence with the initial query. Similarly for self-joins. Therefore, it requires quadratically many equivalence checks in the size of the expansion ($O(S_{ER}^2)$). Each check is done using the same optimization as for redundant queries removal, requiring $O(S_{ER}^w)$ time, for a total of $O(S_{ER}^2 \times S_{ER}^w)$, upper bounded by $O(S_Q \times S_V)^{w+2})$. This additionally stresses the importance of implementing the containment check efficiently.

11. RELATIONAL SERVICE DESCRIPTION LANGUAGE (RSDL)

In service-oriented architectures, such as the one in Figure 1, Web services are deployed on top of relational databases publishing limited access methods in the form of parameterized queries. Each published query is represented by a *function signature* (also called *operation* in WSDL terminology [Christensen et al. 2001]) with typed input and output. Middleware, such as distributed query processors [Borkar et al. 2006] and workflow engines [Jordan and Evdemon 2006], utilize these services by instantiating their parameters and retrieving and postprocessing their results, without knowing the query connecting the input to the output. The semantic connection across Web services is also not known; two Web services might implement two alternative access methods to the same underlying data or they might be published on top of two different databases. The CLIDE Interface is another example of a middleware component requiring additional capabilities from such Web services, that is, the retrieval of schema information and the published query expressions that connect the schema with the output types of the published function signatures.

In light of the shortcomings exhibited by general-purpose Web services, we propose a specialization of WSDL [Christensen et al. 2001] for Web services published on top of relational databases, called *Relational Service Description Language (RSDL)*. We present the design of RSDL, which the CLIDE Interface uses to systematically communicate with relational databases exporting limited access methods. Moreover, we argue that RSDL renders Web services on top of relational databases more usable for data integration and exchange tasks, such as schema matching [Rahm and Bernstein 2001] and Web services similarity search [Dong et al. 2004], while maintaining the ease of use observed in general-purpose Web services.

In terms of *access restrictions*, schema information and query expressions are often sensitive information that cannot be exposed in an often publicly accessible WSDL file. RSDL should provide means to expose this information as output of certain operations and apply access control on the caller of the service.

In regard to *interconnection*, RSDL should provide a mechanism that connects the exported query operations, described in a WSDL file, with the schema information, query expressions, and valid instantiations, such that the Web service can be parsed and used programmatically by agents, such as the CLIDE Interface, without the need of human intervention. At the same time, the exported information should be readable by a human inspecting the Web service's description and responses.

11.1 RSDL Design

In this section, we present the design of RSDL Web services given an underlying database and a set of query expressions to be exported. RSDL interconnects the exported queries with schema and other metadata information, so that an agent or a human using RSDL Web services over a relational database has a coherent picture of the data and the access methods exported. We rely on typing to accomplish the interconnection of the input and output of Web services with their respective source schema, which is enabled by WSDL's ability to define types once and reference them from multiple points. In detail, RSDL provides the following functionality.

11.1.1 *Query Operations.* For each exported query, RSDL publishes one conventional WSDL operation that is of the general form

$$type_r\ queryName(type_1\ param_1,\ \ldots\ ,type_n\ param_n).$$

Each operation has a distinct name, *queryName*. The output type of each operation, $type_r$, is the relational schema of the `SELECT` clause of the corresponding query expression translated into XML Schema according to the mapping rules specified by the SQL:2003 standard [Melton 2003b]. The types $type_1, \ldots ,type_n$ of the input parameters are XML Schema simple type definitions [Biron and Malhotra 2004] and correspond to the parameters, if any, in the query expressions. The correspondence is positional; the parameters of each operation are ordered according to the positional order within the query expression. The RSDL specification of the operations for the two query expressions exported by the Dell source shown in Figure 2 is given in Appendix A.

11.1.2 *Query Expressions.* RSDL exports the query expressions that are encapsulated by the above query operations. Since queries might be parameterized, the SQL types of the parameters, as well as the relational schema of the query results, match the input and output types of the corresponding query operations. An RSDL Web service provides this functionality by the following required operation:

—`getQueries()` outputs the list of published query expressions over a relational schema. For each query, the corresponding SQL expression is provided along with a unique name among the queries in the list, which matches the name of the corresponding query operation. The SQL expressions follow the SQL:2003 syntax [Melton 2003a] and are independent of a particular implementation. An example output of this operation is shown below for the Dell source, where the list consists of the queries V_1 and V_2 shown in Figure 2. Parameters are denoted by the question mark "?" symbol. The `name` columns match the names of the query operations in the RSDL specification given in Appendix A.

```
<?xml version="1.0" encoding="UTF-8"?>
<queries>
  <query name="ComByCpu">
    SELECT DISTINCT Com.cid, Com.cpu, Com.ram, Com.price
    FROM Computers Com
    WHERE Com.cpu = ?
  </query>
  <query name="ComNetByCpuRate">
    SELECT DISTINCT Com.cid, Com.cpu, Com.ram, Com.price,
                    Net.rate, Net.standard, Net.interface
    FROM Computers Com, NetCards Net
    WHERE Com.cid = Net.cid AND Com.cpu = ? AND Net.rate = ?
  </query>
</queries>
```

11.1.3 *Schema Information.* RSDL exports part of the schema of the underlying relational database. The exported schema information defines the context for the exported query expressions. More specifically, at least the tables and the columns used in the query expressions returned by the `getQueries` operation, along with the relevant referential and integrity constraints, are accessible. An RSDL Web service provides this functionality by the following required operation:

—`getSchema()` outputs an XML Schema representing the exported part of the relational schema, along with primary, unique, and foreign key constraints. The translation is performed according to the SQL:2003 standard mapping rules [Melton 2003b]. Among other things, the output XML Schema contains the catalog name and the schema name of the underlying database, as well as XML Schema datatypes corresponding to the SQL datatypes of each exported column. An example output XML Schema of this operation is shown in Appendix B for the Dell source shown in Figure 2, where the tables and columns, along with their primary and foreign keys constraints, are exported. We opted not to place the translated schema information and the exported query expressions within WSDL. We provide them as output of operations instead, so that authentication and authorization of a caller is possible.

11.1.4 *Syntax and Semantics.* Syntactically, RSDL specifications are valid WSDL documents. The distinction between the two is made by the use of the appropriate namespace. Existing WSDL clients are able to call query operations and receive query results in XML format. The difference between RSDL and WSDL is that the definition of the `getSchema` and `getQueries` operations are required. Moreover, RSDL requires the logical connection between these two operations and the query operations that execute the exported queries, since they refer to each another. According to the WSDL specification, such a logical connection can be established through the use of a port type [Christensen et al. 2001]. Hence, RSDL restricts the WSDL specification in the following aspects:

(1) Each port type should consist of two operations named `getSchema` and `getQueries` with output types as specified above.
(2) Additionally, each port type should consist of one or more query operations (executing the exported queries) with input and output types as specified above.
(3) Operations should be of request-response type only.

RSDL does not restrict any of the implementation specific aspects of WSDL, that is, bindings, ports, and services [Christensen et al. 2001] are left unchanged. The RSDL specification for the Dell source shown in Figure 2 is given in Appendix A.

11.1.5 *Scope.* An RSDL specification can define multiple port types. Hence, it is possible to export different parts of a single database or multiple databases within a single RSDL specification.

11.1.6 *Communication.* Relational services are stateless. The communication with a relational service is synchronous and is carried out in a request/response fashion.

11.1.7 *Access Control.* Each one of the operations described above takes as additional inputs a username and password in order to authenticate and authorize the caller.

12. CONCLUSIONS

We presented the CLIDE System, a comprehensive proposal for exporting semantically rich Web services on top of relational sources and interactively querying multiple such sources based on their exported services. The Relational Services Description Language (RSDL) exports database functionality on top of relational sources. Each RSDL-based Web service exports and semantically connects a set of operations with the corresponding parameterized views and the source schema. The CLIDE Interface takes as input a set of RSDL services and leads a developer toward feasible queries by employing a color scheme. 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 online use of the CLIDE Interface for a wide class of queries and views. A demonstration is available online at `http://www.clide.info`.

APPENDIX A. RSDL EXAMPLE

The following listing is the WSDL describing the RSDL service for the Dell source of Figure 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
  xmlns="http://clide.info/rsdl/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tns="http://dell.service.ns"
  targetNamespace="http://dell.service.ns">
  <types>
    <xs:schema targetNamespace="http://dell.service.ns">
      <xs:complexType name="schemaType">
        <xs:all>
          <xs:element ref="xs:schema"/>
        </xs:all>
      </xs:complexType>
      <xs:simpleType name="INTEGER">
        <xs:restriction base="xs:integer">
          <xs:maxInclusive value="2147483647"/>
          <xs:minInclusive value="-2147483648"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:simpleType name="VARCHAR_25">
        <xs:restriction base="xs:string">
          <xs:maxLength value="25"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:complexType name="RowType_DellDB_dbo_Query1">
        <xs:sequence>
          <xs:element name="cid" type="tns:INTEGER"/>
          <xs:element name="cpu" type="tns:VARCHAR_25"/>
          <xs:element name="ram" type="tns:INTEGER"/>
          <xs:element name="price" type="tns:INTEGER"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="TableType_DellDB_dbo_Query1">
        <xs:sequence>
        <xs:element name="row"
                    type="tns:RowType_DellDB_dbo_Query1"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="RowType_DellDB_dbo_Query2">
      <xs:sequence>
        <xs:element name="cid" type="tns:INTEGER"/>
        <xs:element name="cpu" type="tns:VARCHAR_25"/>
        <xs:element name="ram" type="tns:INTEGER"/>
        <xs:element name="price" type="tns:INTEGER"/>
        <xs:element name="rate" type="tns:INTEGER"/>
        <xs:element name="standard" type="tns:VARCHAR_25"/>
        <xs:element name="interface" type="tns:VARCHAR_25"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TableType_DellDB_dbo_Query2">
      <xs:sequence>
        <xs:element name="row"
                    type="tns:RowType_DellDB_dbo_Query2"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</types>
<message name="getSchemaIn">
  <part name="username" type="xs:string"/>
  <part name="password" type="xs:string"/>
</message>
<message name="getSchemaOut">
  <part name="out" type="tns:schemaType"/>
</message>
<message name="ComByCpuIn">
  <part name="username" type="xs:string"/>
  <part name="password" type="xs:string"/>
  <part name="cpu" type="tns:VARCHAR_25"/>
</message>
<message name="ComByCpuOut">
  <part name="queryResult" type="tns:TableType_DellDB_dbo_Query1"/>
</message>
<message name="ComNetByCpuRateIn">
  <part name="username" type="xs:string"/>
  <part name="password" type="xs:string"/>
  <part name="cpu" type="tns:VARCHAR_25"/>
  <part name="rate" type="tns:INTEGER"/>
</message>
<message name="ComNetByCpuRateOut">
```

```
    <part name="queryResult" type="tns:TableType_DellDB_dbo_Query2"/>
  </message>
  <portType name="RSDLPortType">
    <operation name="getSchema">
      <input message="tns:getSchemaIn"/>
      <output message="tns:getSchemaOut"/>
    </operation>
    <operation name="ComByCpu">
      <input message="tns:ComByCpuIn"/>
      <output message="tns:ComByCpuOut"/>
    </operation>
    <operation name="ComNetByCpuRate">
      <input message="tns:ComNetByCpuRateIn"/>
      <output message="tns:ComNetByCpuRateOut"/>
    </operation>
  </portType>
  <binding name="HTTPBinding" type="tns:RSDLPortType">
    <http:binding verb="POST"/>
    <operation name="getSchema">
      <http:operation location="getSchema"/>
      <input>
        <mime:content type="application/x-www-form-urlencoded"/>
      </input>
      <output>
        <mime:mimeXml part="Body"/>
      </output>
    </operation>
    <operation name="ComByCpu">
      <http:operation location="/ComByCpu"/>
      <input>
        <mime:content type="application/x-www-form-urlencoded"/>
      </input>
      <output>
        <mime:mimeXml part="Body"/>
      </output>
    </operation>
    <operation name="ComNetByCpuRate">
      <http:operation location="/ComNetByCpuRate"/>
      <input>
        <mime:content type="application/x-www-form-urlencoded"/>
      </input>
      <output>
        <mime:mimeXml part="Body"/>
      </output>
    </operation>
  </binding>
  <service name="DellService">
```

```
    <port name="DellPort" binding="tns:HTTPBinding">
      <http:address location="https://www.dell.com/service"/>
    </port>
  </service>
</definitions>
```

APPENDIX B. GETSCHEMA OPERATION

The following listing is the output of the `getSchema` operation for the Dell source of Figure 2 described by the RSDL service of Appendix A.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="INTEGER">
    <xs:restriction base="xs:integer">
      <xs:maxInclusive value="2147483647"/>
      <xs:minInclusive value="-2147483648"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="VARCHAR_25">
    <xs:restriction base="xs:string">
      <xs:maxLength value="25"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="RowType_DellDB_dbo_Computers">
    <xs:sequence>
      <xs:element name="cid" type="INTEGER"/>
      <xs:element name="cpu" type="VARCHAR_25"/>
      <xs:element name="ram" type="INTEGER"/>
      <xs:element name="price" type="INTEGER"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TableType_DellDB_dbo_Computers">
    <xs:sequence>
      <xs:element name="row"
                  type="RowType_DellDB_dbo_Computers"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="RowType_DellDB_dbo_NetCards">
    <xs:sequence>
      <xs:element name="cid" type="INTEGER"/>
      <xs:element name="rate" type="INTEGER"/>
      <xs:element name="standard" type="VARCHAR_25"/>
      <xs:element name="interface" type="VARCHAR_25"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TableType_DellDB_dbo_NetCards">
    <xs:sequence>
      <xs:element name="row"
                  type="RowType_DellDB_dbo_NetCards"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="SchemaType_DellDB_dbo">
    <xs:all>
      <xs:element name="Computers"
                  type="TableType_DellDB_dbo_Computers"/>
      <xs:element name="NetCards"
                  type="TableType_DellDB_dbo_NetCards"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="CatalogType_DellDB">
    <xs:all>
      <xs:element name="dbo" type="SchemaType_DellDB_dbo">
        <xs:key name="Computers_PK">
          <xs:selector xpath="Computers/row"/>
          <xs:field xpath="cid"/>
        </xs:key>
        <xs:keyref name="Computers_FK"
                   refer="Computers_PK">
          <xs:selector xpath="NetCards/row"/>
          <xs:field xpath="cid"/>
        </xs:keyref>
      </xs:element>
    </xs:all>
  </xs:complexType>
  <xs:element name="DellDB" type="CatalogType_DellDB"/>
</xs:schema>
```

ACKNOWLEDGMENTS

The authors would like to thank Alan Nash and Michael J. Carey for their valuable contribution.

REFERENCES

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley, Reading, MA.

Biron, P. V. and Malhotra, A. 2004. XML Schema part 2: Datatypes second edition. W3C Recommendation 28 October 2004. Go online to http://www.w3.org/TR/xmlschema-2/.

Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., and Siméon, J. 2007. XQuery 1.0: An XML query language. W3C Recommendation 23 January 2007. Go online to http://www.w3.org/TR/xquery/.

Borkar, V. R., Carey, M. J., Lychagin, D., Westmann, T., Engovatov, D., and Onose, N. 2006. Query processing in the aqualogic data services platform. In *Proceedings of VLDB*. 1037–1048.

Braga, D., Campi, A., and Ceri, S. 2005. *QBE* (query y xample): A visual interface to the standard xml query language. *ACM Trans. Database Syst. 30*, 2, 398–443.

Carey, M. J. 2006. Data delivery in a service-oriented world: The Bea aqualogic data services platform. In *Proceedings of the SIGMOD Conference*. 695–705.

Carey, M. J., Haas, L. M., Maganty, V., and Williams, J. H. 1996. PESTO: An integrated query/browser for object databases. In *Proceedings of VLDB*. 203–214.

Chandra, A. K. and Merlin, P. M. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of STOC*. 77–90.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. 2001. Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001. Go online to `http://www.w3.org/TR/wsdl`.

Dong, X., Halevy, A. Y., Madhavan, J., Nemes, E., and Zhang, J. 2004. Simlarity search for Web services. In *Proceedings of VLDB*. 372–383.

Duschka, O. M., Genesereth, M. R., and Levy, A. Y. 2000. Recursive query plans for data integration. *J. Log. Program. 43*, 1, 49–73.

Fallside, D. C. and Walmsley, P. 2004. XML schema part 0: Primer second edition. W3C Recommendation 28 October 2004. Go online to `http://www.w3.org/TR/xmlschema-0/`.

Fan, W., Chan, C. Y., and Garofalakis, M. N. 2004. Secure XML querying with security views. In *Proceedings of the SIGMOD Conference*. 587–598.

Garcia-Molina, H., Ullman, J. D., and Widom, J. D. 2001. *Database Systems: The Complete Book*. Prentice Hall, Englewood Cliffs, NJ.

Halevy, A. 2001. Answering queries using views: A survey. *VLDB J. 10*, 4, 270–294.

Jordan, D. and Evdemon, J. 2006. Web Services Business Process Execution Language Version 2.0. OASIS Public Review Draft, 23th August, 2006. Go online to `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html`.

LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., and DeWitt, D. J. 2004. Limiting disclosure in hippocratic databases. In *Proceedings of VLDB*. 108–119.

Levy, A. Y., Rajaraman, A., and Ullman, J. D. 1996. Answering queries using limited external processors. In *Proceedings of PODS*. 227–237.

Li, C. and Chang, E. Y. 2001. Answering queries with useful bindings. *ACM Trans. Database Syst. 26*, 3, 313–343.

Melton, J. 2003a. Database languages—SQL—part 14: XML-related specifications (SQL/XML). In *Proceedings of ISO/IEC 9075-14:2003*.

Melton, J. 2003b. Database languages—SQL—part 2: Foundation (SQL/foundation). In *Proceedings of ISO/IEC 9075-2:2003*.

Microsoft, Inc. Native XML Web services for Microsoft SQL server. 2005. Go online to `http://msdn2.microsoft.com/en-us/library/ms345123.aspx`.

Microsoft, Inc. 2004. Microsoft BizTalk Server. Go online to `http://www.microsoft.com/biztalk/`.

Nash, A. and Ludaescher, B. 2004. Processing unions of conjunctive queries with negation under limited access patterns. In *Proceedings of EDBT*.

Nielsen, J. 2000. *Designing Web Usability*. New Riders Publishing, Berkeley, CA.

Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., and Fagin, R. 2002. Translating Web data. In *Proceedings of VLDB*. 598–609.

Pottinger, R. and Halevy, A. Y. 2001. Minicon: A scalable algorithm for answering queries using views. *VLDB J. 10*, 2-3, 182–198.

Rahm, E. and Bernstein, P. A. 2001. A survey of approaches to automatic schema matching. *VLDB J. 10*, 4, 334–350.

Rajaraman, A., Sagiv, Y., and Ullman, J. D. 1995. Answering queries using templates with binding patterns. In *Proceedings of PODS*. 105–112.

Rizvi, S., Mendelzon, A. O., Sudarshan, S., and Roy, P. 2004. Extending query rewriting techniques for fine-grained access control. In *Proceedings of SIGMOD Conference*. 551–562.

Roth, M. T. and Schwarz, P. M. 1997. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proceedings of VLDB*. 266–275.

Shipman, D. W. 1981. The functional data model and the data language daplex. *ACM Trans. Database Syst. 6*, 1, 140–173.

Tufte, E. R. 1997. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, CT.

Vassalos, V. and Papakonstantinou, Y. 1997. Describing and using query capabilities of heterogeneous sources. In *Proceedings of VLDB*.

Yannakakis, M. 1981. Algorithms for acyclic database schemes. In *Proceedings of VLDB*. 82–94.

Yerneni, R., Li, C., Garcia-Molina, H., and Ullman, J. D. 1999. Computing capabilities of mediators. In *Proceedings of SIGMOD Conference*. 443–454.

Zloof, M. 1975. Query by example. *AFIPS NCC 44*, 431–438.

Received October 2006; revised May 2007; accepted August 2007