On the Difference between Navigating Semi-structured Data and Querying It

Gösta Grahne[1] and Laks V.S. Lakshmanan[2]

[1] Concordia University, Montreal, Canada
grahne@cs.concordia.ca
[2] Concordia University, Montreal, Canada and
IIT - Bombay, Mumbai, India
laks@cs.concordia.ca

1 Introduction

Currently, there is tremendous interest in semi-structured (SS) data management. This is spurred by data sources, such as the ACeDB [29], that are inherently less rigidly structured than traditional DBMS, by WWW documents where no hard rules or constraints are imposed and "anything goes," and by integration of information coming from disparate sources exhibiting considerable differences in the way they structure information. Significant strides have been made in the development of data models and query languages [2, 11, 17, 6, 7], and to some extent, the theory of queries on semi-structured data [1, 23, 3, 13, 9]. The OEM model of the Stanford TSIMMIS project [2] (equivalently, its variant, independently developed at U. Penn. [11]) has emerged as the de facto standard model for semi-structured data. OEM is a light-weight object model, which unlike the ODMG model that it extends, does not impose the latter's rigid type constraints. Both OEM and the Penn model essentially correspond to labeled digraphs. A main theme emerging from the popular query languages such as Lorel [2], UnQL [11], StruQL [17], WebOQL [6], and the Ulixes/Penelope pair of the ADM model [7], is that *navigation* is considered an integral and essential part of querying. Indeed, given the lack of rigid schema of semi-structured data, navigation brings many benefits, including the ability to retrieve data regardless of the depth at which it resides in a tree (e.g., see [4]). This is achieved with programming primitives such as regular path expressions and wildcards. A second, somewhat subtle, aspect of the emerging trend is that query expressions are often dependent on the particular *instance* they are applied to. This is not surprising, given the lack of rigid structure and the absence of the notion of a predefined schema for semi-structured data. In fact, it has been argued [4] that it is unreasonable to impose a predefined schema.

There are two problems with the current state of affairs. First, while classical database theory tells us that queries are computable and generic functions from database instances over a predefined schema to (say) relations over another schema, the state-of-the-art theory of semi-structured databases (SSDB) does not exactly pin down the input/output type of queries. Indeed, it is unclear what it *means* to query SSDBs in a formal sense. Second, the notion of genericity

[16] plays a pivotal role in the theory of queries. We contend that this notion has not been satisfactorily addressed for semi-structured data. Genericity says that the order of columns or rows in a relation is irrelevant for purposes of querying. In its purest form, it also says that the domain values are uninterpreted. One may argue that the structure present in a SSDB, in the form of parent-child relationships and/or left-right ordering, carries inherent information, that should be regarded on a par with the data for purposes of querying. However, drawing lessons from the analogy to the classical notion of genericity, it stands to reason to say that if the same underlying data in one SSDB is modeled differently using different structure in a second SSDB, the two SSDBs should be insensitive to the difference in representation between the two SSDBs. We call this property *representation independence.* These notions are made precise in Section 3. Here we give some examples to illustrate the underlying intuition. We use the OEM model as a representative SS data model and the fragment of the StruQL language presented in [18] as a representative query language for convenience and conciseness. Our observations hold for any of the prominent models/query languages mentioned above.

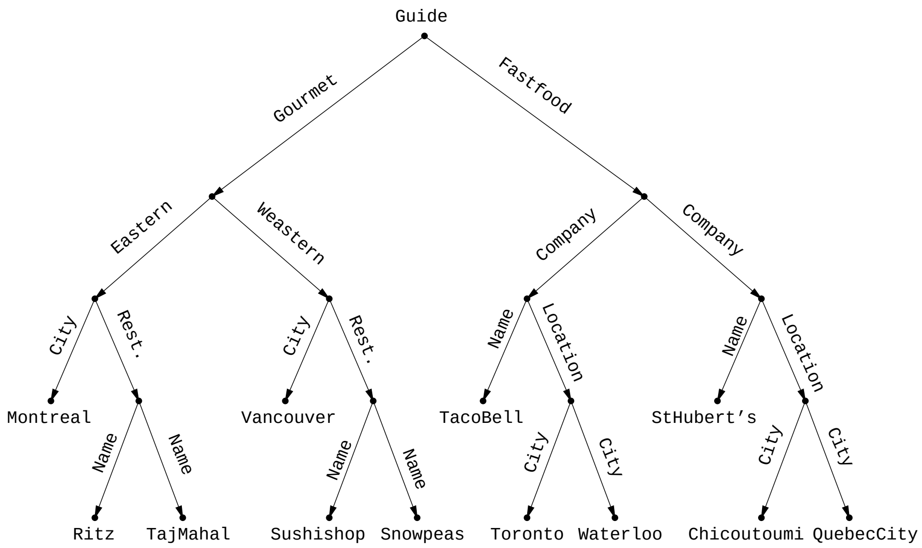

Fig. 1. A Semi-structured Restaurant Guide.

Consider the OEM SSDB in Figure 1, showing (partial) details about restaurants in Canada. It shows information about gourmet and fastfood restaurants and has some heterogeneity in the way the information is represented. This could be the result of integration of information from different sources. Now, consider the query "find pairs of cities and restaurant names in them." One way to express this query in the StruQL fragment is to use two rules:

$$r_1 : q(X,Y) \longleftarrow \texttt{Guide}\ (\texttt{Gourmet.}*)\ Z,\quad Z\ (\texttt{City})\ X,\quad Z\ (*.\texttt{Name})\ Y.$$
$$r_2 : q(X,Y) \longleftarrow \texttt{Guide}\ (\texttt{Fastfood.}*)\ Z,\quad Z\ (\texttt{Name})\ Y,\quad Z\ (*.\texttt{City})\ X.$$

Here, X, Y, Z are node variables, and `Guide` is the entry point (for the root). To write this query expression, one needs to *know* that gourmet restaurants are organized with a different structure than the fastfood ones. If all information had been organized following the gourmet structure, we would have written the query using just rule r_1. If this rule were applied to an "equivalent" database storing the same information using the fastfood structure (for which r_2 would be the right rule) would *not* yield the expected answer. Thus, intuitively, none of these expressions is representation independent. One way out of this might be to write the one-rule expression

$$r_3 :\ q(X,Y) \longleftarrow \texttt{Guide}\ (*)\ Z,\quad Z\ (*.\texttt{City})\ X,\quad Z\ (*.\texttt{Name})\ Y.$$

Since the wildcard $*$ can be mapped to the empty label, this seems to be able to find all city/restaurant pairs in a uniform way, independent of the structural differences. Unfortunately, this does not work (when r_3 is applied to the database of Figure 1: one of the valid substitutions would map Z to the left child of the root, which allows the last two path expressions in the body of r_3 to be mapped to the paths corresponding to the city `Montreal` and the restaurant `Sushishop`. This produces the pair `(Montreal, Sushishop)`, an obviously incorrect answer. It is not clear whether one can write a fixed query expression in this language (or, for that matter, in any of the languages mentioned above) that can correctly find all city/restaurant pairs in a representation independent way.[1]

A possible objection against representation independence would be the following. Consider two nested relational databases $\mathcal{I}$ and $\mathcal{J}$ containing "equivalent" information on restaurants and cities, where $\mathcal{I}$ groups the information by city, whereas $\mathcal{J}$ groups by restaurant names. A query such as "find all city/restaurant pairs in flat form" would require two different expressions against $\mathcal{I}$ and $\mathcal{J}$ in nested relational algebra. If this is acceptable, isn't it unreasonable to impose the requirement of representation independence on SSDB query languages? The difference is that the nested relational model, like all traditional data models, has a well-defined notion of a predefined schema. Thus, the user can be reasonably expected to know the schema in full, which includes, among other things, structural information such as nesting order. Thus, we can live with representation dependence. By contrast, as argued by Abiteboul [4], it is unreasonable to assume a predefined schema or a complete structural/representational knowledge on the part of the user, for SSDBs. *In this context, we believe that representation independence is not only reasonable, but is an extremely important requirement for a SSDB query language to be practical.* We can trace the representation dependent nature of most prominent languages to their heavy

[1] It is even less obvious whether representation independent expressions exist in these languages for *all* queries, since the issue is not addressed in those papers.

reliance on navigation (e.g., via path expressions). Following Ullman [30], we contend that a high-level query language should enable the user to write queries without having to worry about navigational details.

Another source of bewilderment in writing SSDB queries is that the propositions that are supposed to be expressed by a database are not clearly captured by existing SS data models, as these models are largely syntactic. In a traditional model like the relational model, the facts are the tuples in the various relations. Similarly, the fact structure for other traditional models is precisely captured by the data model. In the case of SSDBs, we note from the literature that the facts can sometimes be in the paths [28, 6] and sometimes in certain subtrees (as in the example above; also see [17, 11]). The consequence of the model not capturing the fact structure is that the burden of finding it is shifted to the query expressions the user has to write. Of course, here a possible objection would claim that there *is* no fact structure – that is why the database is "semi-structured." Since there is no fact structure, navigation-based querying is as good as any method. But if the user is able to navigate, he/she has to know the "schema" and since the "schema" is unknown *a priori*, the user must have seen the data in order to have seen the "schema." In this case, a navigational query seems to offer little additional advantage over further browsing by the user. And if the "schema", however primitive, is known *a priori*, then the database *does* express some propositions after all.

Our work was motivated by the following questions. Can we make a self-describing light-weight model such as OEM also capture the fact structure in a SSDB? What is an appropriate notion of genericity, and what is an appropriate definition of a query for SSDBs, in a formal sense? Can we design SSDB query languages that are representation independent and are "generic?" These are crucial questions that need to be answered before useful query languages can be designed and implemented, before the goal of *repository independence* [21] can be achieved, and before advances in areas such as expressive power and completeness can be made. More specifically, in this paper we make the following contributions:

1. We formalize the notion of *representation independence* and show that it is an important property for a SSDB query language. We also introduce a notion of *representation preservation* which intuitively says that queries should perturb existing structure as little as possible.[2]
2. We take the core intuitions behind the OEM model and extend the model with a few minor amendments, while preserving its light-weight nature. We make precise the fact structure of a SSDB and give a formal definition of the *contents* of a SSDB.
3. We define the concept of *genericity* for SSDB's. Roughly speaking, genericity means that queries should be representation independent and should not interpret domain values. We give two possible notions of schema – a *conservative* predefined schema similar in spirit to that of traditional data models,

[2] This is not an argument against restructuring. On the contrary, we contend that restructuring should happen by design; not "accidentally" as a result of querying.

and a *liberal* schema, which is essentially a finite set of attributes, not known in advance. We also extend the classical notion of BP-completeness [27, 8] to the SSDB setting.

4. We define a calculus called *component calculus* (CC) and an algebra called *component algebra* (CA) and show that:
   (a) CC and CA are equivalent in expressive power;.
   (b) CC is representation independent but not representation preserving, while CA is both.
   (c) CC and CA are BP-complete in our extended sense.
   (d) In comparison of CC with TRC$^{\perp}$, an extension of classical tuple relational calculus (TRC) to accommodate inapplicable nulls, it turns out that there are queries expressible in CC for which there is no equivalent expression in TRC$^{\perp}$ whose size is polynomially bounded in the size of the conservative schema. When the schema is liberal, there are queries expressible in CC which have no equivalent expressions in TRC$^{\perp}$.

2 Semi-structured Databases

2.1 Schema and Instance

Let $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{V}$ be countable pairwise disjoint sets of *nodes* $\{v, u, w, u_i, v_j, \ldots\}$, *attribute names* $\{A_1, \ldots A_n, \ldots\}$, and *values* $\{a, b, c, \ldots\}$. In general, we may know the "schema" of a semi-structured database either completely, partially, or not at all. However, in practice, when dealing with such databases, based on the application domain, it is sometimes reasonable to assume we at least know what kind of attributes to expect, if not how/where to find them. To reflect this, we use two alternative notions of schema in this paper. In the most general scenario, the *schema* of a semi-structured database can be completely instance dependent, and thus can potentially be any finite subset of $\mathcal{A}$, which is *not* known *a priori*. This is called the *liberal schema*, and is perhaps the most realistic notion. A second, restrictive notion is that of a *conservative schema*, which is a predetermined finite subset $\mathbf{A} \subset \mathcal{A}$ of attributes. This notion may be appropriate when dealing with highly regular databases such as, e.g., bibtex databases (possibly in a fixed source such as [25]), set of machine generated homepages of employees in an organization, etc.

A *database tree* is a 5-tuple $T = \langle V, E, attr, val, label \rangle$, such that:

- $V \subset \mathcal{N}$ is a finite set of *nodes*.
- $E \subseteq V \times V$, is a set of edges such that the graph (V, E) is a rooted tree.
- $attr : V \rightarrow \mathcal{A}$ is a partial function that maps each node $v \in V$ to its attribute $attr(v)$. When $attr(v)$ is undefined, we write $attr(v) = \perp$. We refer to a node u such that $attr(u) = \perp$ as a *null node*.
- $val : V \rightarrow \mathcal{V}$ is a partial function that maps each node $v \in V$ to its value $val(v)$. When $val(v)$ is undefined, we write $val(v) = \perp$. We assume that if $attr(v) = \perp$, then necessarily $val(v) = \perp$.
- $label : V \rightarrow 2^{\{\oplus, \ominus\}}$ maps each node to a subset of labels from $\{\oplus, \ominus\}$.

Sometimes we will assume that the trees are ordered. For this we use a function ω that associates with each node in V a total order on its outgoing edges (see e.g. Beeri and Milo [9]). In essence then, a database tree is an ordered rooted tree whose nodes come with an attribute-value pair $(A : a)$, together with optional labels $\oplus, \ominus$. Either the value a, or both the attribute A and the value a could be null. The labels $\ominus, \oplus$ are called *blockers*, and they enable the query processor in not mixing up associations between different attribute-value pairs, thus capturing the inherent fact structure. Conceptually, this corresponds to transferring the navigational knowledge imposed on the user by navigational languages, to the data model and the source that implements it. Figure 2 shows three database trees.

Informally, a *fact* of a database tree T is any set $\{(A_1 : a_1), \ldots, (A_n : a_n)\}$ of attribute-value pairs from nodes in a subtree of T, such that no attribute is repeated and the subtree does not cross any of the blockers $\ominus$, or $\oplus$. Some of the values a_i might be the null $\perp$. Intuitively, fact $\{(A_1 : a_1), \ldots, (A_n : a_n)\}$ represents the tuple $[A_1 : a_1, \ldots, A_n : a_n]$. For instance, in Figure 2, the tree T_1 contains the facts $\{(A : a), (B : b), (C : \perp), (D : \perp)\}$ and $\{(A : a'), (B : b'), (D : \perp)\}$, while T_2 contains the facts $\{(A : a), (B : b), (E : \perp)\}$ and $\{(A : a'), (B : b'), (C : \perp)\}$. Note that the fact $\{(A : a), (B : b), (C : \perp), (D : \perp)\}$ in T_1 is "subsumed" by the fact $\{(A : a), (B : b)\}$. Tree T_3 in Figure 2 contains the facts $\{(A' : b), (B' : a), (F : \perp)\}$ and $\{(A' : b'), (B' : a'), (F : \perp)\}$. Intuitively, all three trees are "equivalent" in terms of the set of facts they represent. This intuition is captured in Definition 1, where "content equivalence" captures equivalence w.r.t. the underlying set of facts, and the other notions of equivalence there capture the various dissimilarities among the tree of Figure 2.

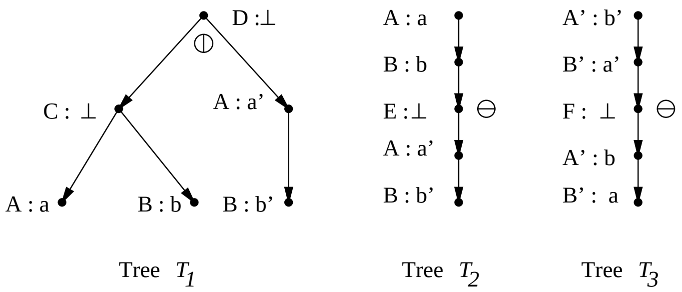

Fig. 2. Three Database Trees.

We can see from the example that $\oplus$ may be regarded as a kind of "light-weight" set node.[3] The role of $\ominus$ is to act as a scoping boundary for attribute names. For instance, if `Name` denotes the name of an employee, and the children's

[3] Notice that the model permits heterogeneous sets in that different tuples can have different sets of attributes.

names of the employee are encoded in a subtree, we can use the blocker $\ominus$ at the root of the subtree to bound the scope of the two different occurences of the attribute `Name`. A tree encoding the facts that `Joe` is an employee with address `Westmount`, and that he has two children, `Sam` and `Brigitte`, aged 5 and 3 resp. is shown in Figure 3. The nodes with $\perp$ as values can be thought of as encoding objects. Thus `EMP` is an object, and `Children` is a subobject of `Emp`. Thus, intuitively, use of the blocker $\ominus$ can be viewed as declaring a new object (corresponding to the subtree) and a reference to the subobject in the parent object.

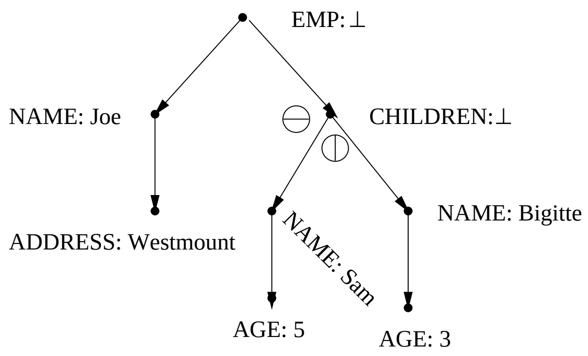

Fig. 3. The use of Blockers.

A semi-structured *database instance* $\mathcal{I}$ is a finite sequence of database trees $\langle T_1, \ldots, T_n \rangle$. An example database appears in Figure 4 in Section 3.3. When we consider instances of a predefined schema $\mathbf{A}$, we will assume that the range of the function *attr* is included in $\mathbf{A}$. The *active domain* of a SSDB $\mathcal{I}$, denoted $adom(\mathcal{I})$, is the set of all values associated with nodes in $\mathcal{I}$'s trees.

2.2 Contents of Databases

It is clear that the same information can be represented as a semi-structured database in several "equivalent" ways. In this section, we formalize four notions of equivalence between SSDBs. Below, we informally explain these notions, first between trees.

- *Content equivalence*: Although two trees might have different "looks" they nevertheless represent the same "generic" information, where attribute names and constants are not interpreted.
- *Subsumption equivalence* (special case of content equivalence): The two trees, in addition, use the same constant and attribute names in the same contexts.
- *Partial isomorphism equivalence*: Full isomrphism would mean that the trees are isomorphic as labeled ordered trees and have the same generic information represented in structurally similar ways. However, since we will choose

to disregard nodes with null values $\perp$ in the isomorphism, we work with *partial isomorphism.*
- *Partial identity* (special case of partial isomorphism equivalence): The trees are identical, except for nodes with null values $\perp$.

We next formalize these notions.

A *path* in a database tree T, is a sequence $v_1, v_2, \ldots, v_k$ of nodes related through $E \cup E^{-1}$, subject to the following restrictions: (i) whenever v_i, v_j, v_m are three consecutive nodes on the path, and $(v_i, v_j) \in E^{-1}$ and $(v_j, v_m) \in E$, then $\oplus \notin \mathit{label}(v_j)$, and (ii) if v_i, v_j, v_m are three consecutive nodes on the path, and $(v_i, v_j), (v_j, v_m) \in E$, then $\ominus \notin \mathit{label}(v_j)$. In tree T_1, Figure 2, the sequences of nodes corresponding to $(A : a), (C : \perp), (B : b)$ and that corresponding to $(D : \perp), (C : \perp), (A : a)$ are both paths. A path is *simple* provided no node repeats along the path. Nodes v and w are *connected* if there is a simple path between them. A set of nodes is connected if every pair of nodes in the set is connected (by a simple path). A *connected component* (*component*, for short) of a tree T is a subtree of T induced by a maximal connected subset of nodes of T. Note that unlike in classical graph theory, two components can overlap. The set of attribute-value pairs associated with the nodes of any component of T, such that no attribute repeats, is a *fact* in T. More than one fact may thus come from the same component (if it contains repeated attributes). Thus, the information content of T is the set of facts in T.

Let $T = \langle V, E, \mathit{attr}, \mathit{val}, \mathit{label}\rangle$ and $T' = \langle V', E', \mathit{attr}', \mathit{val}', \mathit{label}'\rangle$ be database trees, and let $U = \{v \in V : \mathit{val}(v) \neq \perp\}$. A mapping $h : U \to V'$ is called a *tree morphism* if for all nodes v_i and v_j in U, the following holds: (i) if $\mathit{attr}(v_i) = \mathit{attr}(v_j)$ then $\mathit{attr}'(h(v_i)) = \mathit{attr}'(h(v_j))$; (ii) if $\mathit{val}(v_i) = \mathit{val}(v_j)$ then $\mathit{val}'(h(v_i)) = \mathit{val}'(h(v_j))$. We sometimes denote a tree morphism as $h : T \to T'$. Basically, tree morphisms are consistent w.r.t. the attributes and values associated with nodes.

Let $\mathcal{A}(T)$ (resp., $\mathcal{V}(T)$) denote the set of attribute names (resp., values) of the nodes in T. Then a tree morphism $h : T \to T'$ induces the (partial) attribute mapping $h_{attr} : \mathcal{A}(T) \to \mathcal{A}(T')$, defined as $h_{attr}(A) = \mathit{attr}'(h(v))$, where v is any node in V for which $\mathit{attr}(v) = A$. Similarly, it induces a value mapping $h_{val} : \mathcal{V}(T) \to \mathcal{V}(T')$, defined as $h_{val}(a) = \mathit{val}'(h(v))$, where v is any node in V for which $\mathit{val}(v) = a$. Clearly, these mappings are well defined.

Definition 1 A tree morphism $h : T \to T'$ is called a *content embedding* if the following conditions hold: (i) the attribute mapping h_{attr} induced by h is injective; (ii) the value mapping h_{val} induced by h is injective; (iii) whenever v_i and v_j are connected in T, then $h(v_i)$ and $h(v_j)$ are connected in T'.

If there is a content embedding from T to T', we say that T is *content-homomorphic* to T' and write $T \leq_{\text{cont}} T'$. If T and T' are content-homomorphic to each other, we say they are *content equivalent*, i.e. $T \equiv_{\text{cont}} T'$ iff $T \leq_{\text{cont}} T'$ and $T' \leq_{\text{cont}} T$. We will also be interested in *special* content embeddings h, which are content embeddings for which the induced mappings h_{attr} and h_{val} are identity mappings. In case there is a content embedding from T to T' via a special mapping $h : T \rightarrow T'$, we say that T is *subsumed* by T' and write $T \leq_{\text{sub}} T'$. If T and T' are subsumed by each other, we say they are *subsumption equivalent*, i.e. $T \equiv_{\text{sub}} T'$ iff $T \leq_{\text{sub}} T'$ and $T' \leq_{\text{sub}} T$.

For nodes u, v in a tree T, u *preorder precedes* v provided u comes before v in the preorder enumeration of T's nodes. It is extended to a SSDB instance $\mathcal{I}$ in the obvious way: u preorder precedes v whenever both come from the same tree and the above condition holds, or u (v) comes from T_i (T_j) and T_i comes before T_j in the sequence of the trees in $\mathcal{I}$.

The next notion we introduce is partial isomorphism equivalence. Thereto, we say that a tree morphism $h : T \rightarrow T'$ is a *tree embedding*, if it is a content embedding and further, whenever both u and v have non-null values, if $(u, v) \in E$, then $(h(u), h(v)) \in E'$, and if u preorder preserves v in T, then $h(u)$ preorder precedes $h(v)$ in T'. If there is a tree embedding from T to T' we write $T \leq_{\text{piso}} T'$, since, loosely speaking, a subtree of T is isomorphic to a subtree of T'. If both $T \leq_{\text{piso}} T'$ and $T' \leq_{\text{piso}} T$, then T and T' are *partial isomorphism equivalent*, denoted $T \equiv_{\text{piso}} T'$. The final notion is partial identity. Suppose there is a tree embedding h from T to T', such that h is special. Then, informally speaking, the non-null portion of T is identical to a subtree of T', so we write $T \leq_{\text{pid}} T'$. If both $T \leq_{\text{pid}} T'$ and $T' \leq_{\text{pid}} T$, then T and T' are *partially identical*, denoted $T \equiv_{\text{pid}} T'$. The various notions of homomorphism are related as follows.

Lemma 1. *(1) If $T \leq_{\text{sub}} T'$ then $T \leq_{\text{cont}} T'$. (2) If $T \leq_{\text{pid}} T'$ then $T \leq_{\text{piso}} T'$. (3) If $T \leq_{\text{piso}} T'$ then $T \leq_{\text{cont}} T'$. (4) None of the reverse implications holds.*

These notions of equivalence are illustrated in Figure 2 from Section 2.1. For the trees in the figure, we have $T_1 \equiv_{\text{sub}} T_2$ and $T_1 \equiv_{\text{cont}} T_3$. We also have $T_2 \equiv_{\text{piso}} T_3$. Clearly $T_1 \not\equiv_{\text{piso}} T_3$.

Finally, let $\leq_*$ denote any of the notions of embedding above. For two database instances $\mathcal{I}, \mathcal{J}$, we define $\mathcal{I} \leq_* \mathcal{J}$ provided for all trees $T \in \mathcal{I}$, there is a tree $T' \in \mathcal{J}$ such that $T \leq_* T'$. We write $\mathcal{I} \equiv_* \mathcal{J}$ iff $\mathcal{I} \leq_* \mathcal{J}$ and $\mathcal{J} \leq_* \mathcal{I}$.

In Section 3.2, in defining operators of our algebra, we will often need to create identical copies of trees or components. An *identical copy* of a database tree T is another database tree T' such that there is a special embedding $h : T \rightarrow T'$ which is 1-1 and onto on the nodes of T, and preserves node labels. The following theorem characterizes the complexity of checking the different notions of embedding.

Theorem 1. *Deciding whether $T \leq_{\text{cont}} T'$ is NP-complete. Deciding whether $\mathcal{I} \equiv_* \mathcal{J}$ can be done in PTIME for $\equiv_* \in \{\equiv_{\text{piso}}, \equiv_{\text{sub}}, \equiv_{\text{pid}}\}$.*

3 Querying Semi-structured Databases

In the classical setting of relational databases, queries are defined as computable functions from database instances of a given schema to relations over another given schema, that are generic, in the sense of commuting with database isomorphisms. For SSDBs, a complication arises because instances do not come with a predefined schema in general. The "right" definition of queries depends on what one perceives as the information content of a SSDB. E.g., consider Figure 4 (Section 3.3), which shows two database trees T_1 and T_2 representing the resources and industries in various Canadian provinces and cities. In T_1, one can argue that the fact that provinces come below regions is significant. A problem with this position, however is that a different presentation of the information in Figure 4 might simply list the information for each province in the form of a set of tuples over region, province, capital, population, and resource. In a SSDB context, it is unreasonable to assume complete user knowledge of the exact structural organization of information. *We postulate that the right level of abstraction of information content in a SSDB is the set of associations between various attribute values as captured by the paths connecting them in the SSDB.* In keeping with this, queries should not distinguish between two SSDBs that are considered equivalent w.r.t. their contents. Let $\mathcal{I}$ be any SSDB, and π any permutation on its active domain $adom(\mathcal{I})$. Then we extend π to $\mathcal{I}$ in the obvious manner: $\pi(\mathcal{I})$ is the database forest obtained by replacing all non-null attribute values a by $\pi(a)$. Then we have

Definition 2 A *query* is a computable function Q from the set of SSDBs to the set of database trees such that: (i) Q is invariant under permutations of the active domain, i.e. for any instance $\mathcal{I}$ and any permutation π on its active domain, $Q(\pi(\mathcal{I})) \equiv_{\mathrm{pid}} \pi(Q(\mathcal{I}))$; and (ii) whenever $\mathcal{I} \equiv_{\mathrm{cont}} \mathcal{J}$, then $Q(\mathcal{I}) \equiv_{\mathrm{cont}} Q(\mathcal{J})$.

As illustrated in Section 1, one of the desirable properties for a query language for SSDBs is the "robustness" with which it can accommodate modifications to the structural presentation of data while the contents are left intact. More formally, we have

Definition 3 A query language $\mathcal{L}$ for SSDBs is *representation independent* provided for every expression E in $\mathcal{L}$, and for any two SSDBs $\mathcal{I}$ and $\mathcal{J}$, $\mathcal{I} \equiv_{\mathrm{cont}} \mathcal{J}$ implies $E(\mathcal{I}) \equiv_{\mathrm{cont}} E(\mathcal{J})$.[4]

A user of a representation independent query language can remain oblivious to differences in the actual representation of a given piece of information. None of the langauges surveyed in Section 1 is representation independent.

A second desirable property of query languages is representation preservation. Intuitively, it means the expressions in the language should perturb the original structure present among elements of the output, *as little as possible*. We believe queries expressed in a representation preserving language, requiring minimal structural alterations, will lead to efficient implementation. We distinguish between querying and restructuring in the sense of [20] and postulate that the spirit of pure querying is captured by representation preservation.

[4] Note that this is stronger than saying that for every query Q expressible in $\mathcal{L}$, there is an expression E in $\mathcal{L}$ that computes Q and returns equivalent answers on content equivalent databases.

Definition 4 A query language $\mathcal{L}$ is *representation preserving* provided for every expression E in $\mathcal{L}$, and for every instance $\mathcal{I}$, there is a partial isomorphic embedding of $E(\mathcal{I})$ into $\mathcal{I}$, that is, $E(\mathcal{I}) \leq_{\text{piso}} \mathcal{I}$.

3.1 Component Calculus

In our model for semi-structured data, we regard the connected components of the database forest as the basic information carrying units which define the fact structure of the database. In the component calculus, we give primacy to these components, by letting the variables range over them. Let $\mathbf{V}$ be a countable set $\{s, t, \ldots\}$ of *component variables*, or variables for short. To accommodate inapplicable nulls which are natural in SSDBs, following Manna [22], we use two kinds of equalities, namely $=$ and $\doteq$. An *atom* of component calculus is an expression of the form $t.A = a$, $t.A \doteq a$, $t.A \doteq \bot$, $t.A = s.B$, $t.A \doteq s.B$, $t \sqsubseteq s$, $t \sqsubseteq_{\bar{A}} s$, $t[a]$, or $T(t)$, where T is a tree, t and s are variables, A and B are attribute names, and a is a domain value. [5] The set of all *well-formed formulas* of component calculus is formed by closing the atoms under $\wedge$, $\neg$, and $\exists$ in the usual way. We will use the shorthands: $t.A \neq a$ for $\neg(t.A = a)$, $t.A \not\doteq a$ for $\neg(t.A \doteq a)$, etc. We will use the special abbreviation $t[A]$ for $t.A \not\doteq \bot$, which in turn abbreviates $\neg(t.A \doteq \bot)$.

Throughout the paper, when we construct new trees, by a "new" node, we mean a node that appears nowhere else before. A tree is said to be *multiple-free* w.r.t. a set of attributes X if no attribute in X occurs more than once in T. It is multiple-free if no attribute occurs more than once. Let T be a tree. By the *universe of* T we mean the set of all multiple-free line trees, constructed in such a way that each attribute-value pair in it is equal to some attribute-value pair in T, and the line trees do not contain null nodes. The notion of universe is extended in the obvious way to a database instance $\mathcal{I}$. Furhtermore, let φ be a formula in component calculus, and $\mathcal{I}$ and instance. Then the universe of $(\mathcal{I}, \varphi)$ is defined as the universe of $\mathcal{I}$ except that it additionally has components obtained as folllows: for each line tree τ in the universe of $\mathcal{I}$, rename any subset of attributes of τ with distinct attributes appearing in φ.

Let θ be a valuation into $\mathcal{I}$, i.e. a mapping from the variable set $\mathbf{V}$ to the universe of $\mathcal{I}$. We now define *satisfaction* of a component calculus formula φ by a database $\mathcal{I}$ under a valuation θ, in symbols $\mathcal{I} \models \varphi\theta$.

- $\mathcal{I} \models (t.A = a)\theta$, if $attr(v) = A$ and $val(v) = a$, for some node v in $\theta(t)$.
- $\mathcal{I} \models (t.A = s.B)\theta$, if $attr(u) = A$, $attr(v) = B$, $val(u)$ and $val(v)$ are defined, and $val(u) = val(v)$, for some nodes u in $\theta(t)$ and v in $\theta(s)$.
- $\mathcal{I} \models (t.A \doteq \alpha)\theta$, if $attr(v) = A$, and either $val(v)$ is defined, α is a domain value, and $val(v) = \alpha$, or $val(v)$ is undefined and α is $\bot$, for some node v in $\theta(t)$.

[5] For expressive power issues, following standard practice, we focus attention on the fragment of CC without the atoms $t.A = a$ and $t.A \doteq a$.

- $\mathcal{I} \models (t.A \doteq s.B)\theta$, if $attr(u) = A$, $attr(v) = B$, and either $val(u)$ and $val(v)$ are defined, and $val(u) = val(v)$, or $val(u)$ and $val(v)$ are both undefined, for some nodes u in $\theta(t)$ and v in $\theta(s)$.
- $\mathcal{I} \models (t[a])\theta$, if $val(v) = a$, for some node v in $\theta(t)$.
- $\mathcal{I} \models (t \sqsubseteq s)\theta$, if $\theta(t) \leq_{\text{sub}} \theta(s)$.
- $\mathcal{I} \models (t \sqsubseteq_{\bar{A}} s)\theta$, if $\tau \leq_{\text{sub}} \theta(s)$, where τ is a $\leq_{\text{sub}}$-maximal component without any occurences of attribute A, such that $\tau \leq_{\text{sub}} \theta(t)$.
- $\mathcal{I} \models (T(t))\theta$, if $\theta(t) \equiv_{\text{sub}} C$ for some component C of T.

The semantics of $\wedge$ and $\neg$ is as usual, and for quantification we need to say that $\theta(t/\tau)$ is the valuation θ except that it maps the variable t to the line tree τ in the universe of $(\mathcal{I}, \varphi)$. Then

- $\mathcal{I} \models (\exists t\varphi)\theta$, if $\mathcal{I} \models \varphi\theta(t/\tau)$, for some τ in the universe of $(\mathcal{I}, \varphi)$.

It will be convenient to work with finite subuniverses of a pair (T, φ). Let k be a natural number. Then the *k-universe* of (T, φ) is the subuniverse of (T, φ) consisting of line trees with at most k nodes. The definition of the k-universe of the pair $(\mathcal{I}, \varphi)$ for an instance $\mathcal{I}$ is similar.

A *component calculus query expression* E is an expression of the form $(t|\varphi(t))$, where $\varphi(t)$ is a component calculus formula with t as its only free variable. Let the length (in number of symbols) of E be k. When applied to an instance $\mathcal{I}$, E defines a tree $E(\mathcal{I})$ with a "new" node v as its root, where $attr(v) = \perp$ $val(v) = \perp$, and $label(v) = ⦶$. The subtrees of the root are determined as follows: Let $\theta_1, \ldots, \theta_q$ be all assignments into the k-universe of $\mathcal{I}$ such that for $i \in [1, q]$, $\mathcal{I} \models \varphi(t)\theta_i$. Now take the forest $\{\theta_i(t)\}_{i=1}^{q}$ and reduce this forest under $\leq_{\text{sub}}$. If there are several equivalent $\leq_{\text{sub}}$-minimal line trees, choose one representative of each equivalence class, say the line tree that has the smaller attributes (in some standard enumeration of the attributes) higher up. This will yield a subsumption equivalent forest $\{T_1, \ldots, T_p\}, p \leq q$. The root then has subtrees $T_1, \ldots T_p$. Sample queries can be found in Section 3.3.

3.2 The Component Algebra

In this section, we define an algebra called the *component algebra* (CA), whose expressions map SSDBs to database trees. This is achieved by extending the classical relational operators to the SSDB setting. We shall find it convenient to have the notion of a canonical form an instance. First we recall that a *null node* is a node u where both $attr(u)$ and $val(u)$ equal $\perp$. A database instance $\mathcal{I}$ is in *canonical form* if every node in $\mathcal{I}$ that has a blocker is a null node. It turns out that every instance $\mathcal{I}$ can be easily converted into an "equivalent" instance that is in canonical form, as follows. (1) For every non-null node u with attribute-value pair $A : a$ [6] and a horisontal blocker, replace u with three nodes x, y, z such that: (i) x and z both have the attribute-value pair $A : a$ associated with them, while y is a null node; (ii) the parent of u becomes the parent of

[6] a can be $\perp$.

x, while every child of u becomes a child of z; (iii) y gets a horisontal blocker, is the only child of x, and is the parent of z; and (iv) if u *additionally* had a vertical blocker, then z gets a vertical blocker. (2) For every non-null node u with attribute-value pair $A : a$ and a vertical blocker, let $v_1, ..., v_n$ be the children of u. Then replace u with $n+1$ nodes $r, x_1, ..., x_n$ such that: (i) x_i is a child of r, has attribute-value pair $A : a$, and is the parent of v_i, $i = 1, ..., n$; (ii) r is a null node, has a vertical blocker, and is a child of u's parent. It is straightforward to verify that the instance so constructed is partially isomorphic to $\mathcal{I}$. *In the sequel, we shall assume without loss of generality that the instances we consider are in canonical form.*

We begin with the simplest operator, union. Let T_1 and T_2 be any database trees (which may be the result of applying CA expressions to a database). Let T_i' denote a copy of T_i, $i = 1, 2$. Then $T_1 \cup T_2$ is the tree T with a new null node u as its root, with $label(u) = \{\oplus\}$, which has the roots of T_1' and T_2' as its children, in that order. Note that the operands are not required to be union-compatible, as such an assumption would be inappropriate for the SSDB setting.

Each of our remaining operators can be conceptually regarded as involving two steps.[7] First, one marks a set of nodes and erases the attribute/value pairs associated with them. Second, one makes the resulting tree compact by contracting "redundant" nodes. For conceptual clarity, we thus define these auxiliary "operations" first. Let $T = \langle V, E, label, attr, val\rangle$ be a database tree and N a subset of T's nodes. Then $\text{ERASE}(N, T)$ is obtained by taking an identical copy of T and erasing for each node $u \in N$, the attribute/value pair associated with u. More precisely, let $T' = \langle V', E', label', attr', val'\rangle$ be an identical copy of T, $h : T \rightarrow T'$ be an 1-1 onto special tree embedding from T to T', and let $h(N) = \{h(u) \mid u \in N\}$. Then $\text{ERASE}(N, T)$ is the database tree $T" = \langle V', E', attr", val", label'\rangle$, such that $\forall u \in h(N)$, $attr"(u) = val"(u) = \perp$, and $\forall u \in V' - h(N)$, $attr"(u) = attr'(u)$, and $val"(u) = val'(u)$. In the sequel, we will abuse terminology and say erase a node to mean erase the attribute/value pair associated with it.

The second auxiliary operator is REDUCE. Applying it to a database tree results in a tree which has no more nodes than T, but which is content equivalent to it. Intuitively, whenever T contains a parent node u and a null child node v, both belonging to a common component, then u and v can identified (i.e. merged). This will leave the contents unchanged as long as the child (null) node does not have any associated horisontal blocker.[8] Formally, $\text{REDUCE}(T)$ is obtained by repeatedly applying the following step to an identical copy T' of T, until there is no change: whenever T' has a null node u whose parent is v, and u, v belong to a common component, and u does not have a horisontal blocker, identify (i.e. merge) u and v. Note that this step only applies to null nodes different from the root, and also that the result of applying REDUCE to T is always unique.

[7] Join also involves something more, of course.

[8] Identifying two adjacent nodes is similar to the edge contraction operation in the theory of graph minors.

Renaming is defined in a straightforward manner. Given a tree T, the operator $\rho_{B \leftarrow A}(T)$ replaces every occurrence of an attribute A (at any node of T) by attribute B. When there is no occurrence, this is a nop.

The allowable selection conditions in CA are of one of the following forms: (i) $A\theta a$, (ii) $A\theta B$, (iii) $A \doteq \perp$, $A \not\doteq \perp$, and (iv) $\_ = a$, where A, B are attributes, a is a value, and θ is one of $=, \neq, \doteq, \not\doteq$, and their boolean combinations. [9] We include conditions of form (iv) since in a SSDB where a predefined schema may not exist, it is often convenient to search for occurrence of values without knowing *a priori* their associated attributes.[10] Let T be a database tree and C any allowable selection condition of the form, say $A\theta B$. Then a component C of T satisfies this selection condition exactly when C contains two nodes u, v such that $attr(u) = A$, $attr(v) = B$, and $val(u)$ and $val(v)$ stand in the relationship indicated, as per the definition of satisfaction given for CC. Satisfaction of other conditions is similar. In particular, C satisfies $\_ = a$, provided C contains some node with associated value a. Our definition of selection follows. Recall that we assume database trees are in canonical form.

Definition 5 Let T be a database tree, C any allowable selection condition. Then

$$\begin{aligned} &\sigma_C(T) = \text{REDUCE}(\text{ERASE}(S, T)), \text{ where} \\ &\qquad S = \{u \in T \mid u \text{ belongs to component } C \text{ of } T \;\&\; C \text{ does not satisfy } C\}. \end{aligned}$$

Intuitively, selection erases the attribute/value information on all nodes belonging to those components of T which do not satisfy C and reduces the resulting tree.

Definition 6 Let T be a database tree, X any set of attributes. Then

$$\pi_X(T) = \text{REDUCE}(\text{ERASE}(\{u \in T \mid attr(u) \notin X\}, T)).$$

Projection on a set of attributes X erases all nodes whose attribute is not in X, and reduces the resulting tree. When we work with liberal schema, we will find it convenient to use dual projection, which amounts to dropping attributes.

Definition 7 Let T be a database tree, A any attribute. Then

$$\overline{\pi}_A(T) = \text{REDUCE}(\text{ERASE}(\{u \in T \mid attr(u) = A\}, T)).$$

This operator erases all nodes with associated attribute A (if any) and reduces the resulting tree.

The minus operator is based on the notion of subsumption. A component C of a tree T is subsumed by a component C' of a tree T' exactly when C as a tree is subsumed by C' (cf. Section 2.2).

[9] As mentioned for CC, when considering expressive power issues, we will drop form (i).

[10] This feature is exhibited by previous SSDB languages.

Definition 8 Let T_1, T_2 be any database trees. Then

$$T_1 - T_2 = \text{REDUCE}(\text{ERASE}(S, T_1)) \text{ where}$$
$$S = \{u \in C \mid C \text{ is a component of } T \ \& \ \exists\, C' \text{ of } T_2, \text{ s.t. } C \leq_{\text{sub}} C'\}$$

Basically, $T_1 - T_2$ is obtained by erasing (the nodes in) those components of T_1 which are subsumed by some component of T_2 and then reducing the resulting tree.

The last operator is join. Since we are operating in a model where we use attribute names (as opposed to position numbers as in classical relational model), it is more appropriate to define join as a primitive operator. Thereto, we need the following notions.

The preorder precedes relation Section 2 is extended to components of trees: a component C of T preorder precedes another component C' of T provided for every non-null node v in C', there is a non-null node u in C such that u preorder precedes v. Recall that components of trees in canonical form never share non-null nodes. Consequently, on such trees, the preorder precedes relation is a total order on components.

Let T and T' be any trees. Then we can form a new tree $T \odot T'$ by taking a new node v as root, with $attr(v) = \perp$, $val(v) = \perp$, and $label(v) = \{\}$, and by making copies of T and T' subtrees of v, in that order.

For an enumeration of trees $T_1, \ldots, T_n$, we abuse notation and let the expression $\oplus(T_1, \ldots, T_n)$ denote the tree with a new null node u as its root with label $\oplus$, and the roots of copies of $T_1, \ldots, T_n$ as its children, in that order.

Let C be a component of a tree T and let X be a set of attributes appearing in C. Suppose C has one or more nodes corresponding to each of the attributes in X. A *cutset* of C w.r.t. X is any set of nodes V_X in C such that there is exactly one node in V_X corresponding to each attribute in X. Clearly, a cutset need not be unique. Each cutset V_X leads to a component which is multiple-free w.r.t. X, as follows. Let $V_X^2 = \{u \mid u \in \text{ nodes}(C) - V_X \ \& \ attr(u) \in X\}$, i.e. V_X^2 consists of precisely those nodes in C not in V_X, which contain an attribute in X. Then $\text{REDUCE}(\text{ERASE}(V_X^2, C))$ denotes the component obtained from C by erasing all nodes outside V_X which contain an attribute in X, and then reducing the resulting component. Finally, the decomposition of C w.r.t. X is defined as $\text{DECOMP}(C, X) = \{\text{REDUCE}(\text{ERASE}(V_X^2, C)) \mid V_X \text{ is a cutset of } C \text{ w.r.t. } X\}$. Clearly, when C is multiple-free w.r.t. X, $\text{DECOMP}(C, X) = C$.

Let C, C' be any components of trees T_1, T_2. Suppose C (resp., C') is multiple-free w.r.t. A (resp., B). Then the pair (C, C') satisfies the condition $A = B$ provided there is a pair of nodes $u \in C$ and $v \in C'$ whose associated attributes are A and B respectively, associated values are defined, and are equal. Satisfaction for $A \neq B, A \doteq B, A \not\doteq B$ is similarly defined. In particular, the pair (C, C') satisfies $A \doteq B$ provided there is a pair of nodes $u \in C$ and $v \in C'$ whose associated attributes are A and B respectively, and either both their associated values are undefined, or both are defined and equal. Whenever (C, C') satisfies $A\theta B$, we say (C, C') is *(theta-)joinable*. When C and/or C' is not multiple-free w.r.t the appropriate attribute, we work with the components in their decomposition. Let

$\text{DECOMP}(C, A) = \{C_1, \ldots, C_k\}$, where C_i preorder precedes $C_{i+1}, 1 \leq i < k$, and $\text{DECOMP}(C', B) = \{C'_1, \ldots, C'_m\}$, where and C'_j preorder precedes $C'_{j+1}, 1 \leq j < m$. Then define $C \star_{A\theta B} C' = \oplus(C_i \odot C'_j \mid C_i \text{ and } C'_j \text{ are joinable })$. Here, the pairs of components are enumerated in a manner that respects the preorder sequence.

Let C be a component of T_1 and let T_2 be any tree. Then $C \star_{A\theta B} T_2 = \oplus(C \star_{A\theta B} D_1, \ldots, C \star_{A\theta B} D_n)$ where $D_1, \ldots D_n$ are all the components of T_2 and D_i preorder precedes $D_{i+1}, 1 \leq i < n$. For a tree T and a binary relation $r = \{(C, D) \mid C \text{ is a component of } T, D \text{ is any tree}\}$, let $\text{REPL}(T, r)$ denote the result of replacing, in a copy of T, (the corresponding copy of) each component C of T, by the tree D, whenever $r(C, D)$ holds.

Definition 9 Let T_1 and T_2 be any database trees, $A \in \mathcal{A}(T_1), B \in \mathcal{A}(T_2)$ be any attributes appearing in them, and let $\theta \in \{=, \neq, \doteq, \not\doteq\}$. Then

$$T_1 \bowtie_{A\theta B} T_2 = \text{REPL}(T_1, \{(C, C \star_{A\theta B} T_2) \mid C \text{ is a component of } T_1\}).$$

In words, join works as follows. Create a copy T'_1 of T_1. For each component C of T'_1 do the following: (i) concatenate (in the sense of $\odot$) C with each component C' of T_2 with which it is joinable; (ii) then "merge" the resulting set of trees using the $\oplus$ operator above; this gives rise to the tree $C \star_{A\theta B} T_2$ above; (iii) next, replace C by the tree $C \star_{A\theta B} T_2$ in T'_1. A technicality arises due to the fact that C (resp., C') may not be multiple-free w.r.t. attribute A (resp., B), which is handled by decomposing C (C') w.r.t. A (B) and "processing" the resulting pieces w.r.t. their preorder enumeration.

We note that the definition of natural join is analogous. The main issue, however, is when we can consider a pair of components natural joinable. There are several ways to define this, depending on which notion of equality is adopted. We argue that when no notion of predefined schema is applicable (as is common with most SSDB applications), practically the most likely useful definition is one that says whenever both components have defined values for any common attribute, then they must agree. We note that this definition of natural join is in the spirit of the polymorphic join defined by Buneman and Ohori [12]. Formally, we say that a pair (C, C') of components of T_1, T_2 respectively, is *natural joinable*, provided for every attribute A, *if* there are nodes $u \in C$ and $v \in C'$ such that their associated attribute is A and their associated values are both defined, *then* they are equal. Define the notions $C \star C'$, $C \star T_2$ analogously to $C \star_{A\theta B} C'$ and $C \star_{A\theta B} T_2$, but incorporating the condition for natural joinability instead of theta-joinability. Necessary decompositions must be done w.r.t. the set of common defined attributes for the pair of components being joined. We have the following definition of natural join.

Definition 10 Let T_1 and T_2 be any database trees. Then

$$T_1 \bowtie T_2 = \text{REPL}(T_1, \{(C, C \star T_2) \mid C \text{ is a component of } T_1\}).$$

Samples of algebraic queries can be found in Section 3.3 where an example of natural join is also included.

3.3 Examples

Here we illustrate the query languages. Consider the following database forest representing information on Canadian provinces (Figure 4). In tree T_1, the provinces are grouped into two subtrees: the Eastern provinces and the Maritimes. For each province, the database records its name. `PEI` stands for Prince Edward Island. For some provinces the main resources (`RES`) are recorded, and likewise for the capital (`CAP`), and population (`POP`). Only for New Foundland is the main import goods (`IMP`) recorded. The tree T_2 shows major cities and industries in each province.

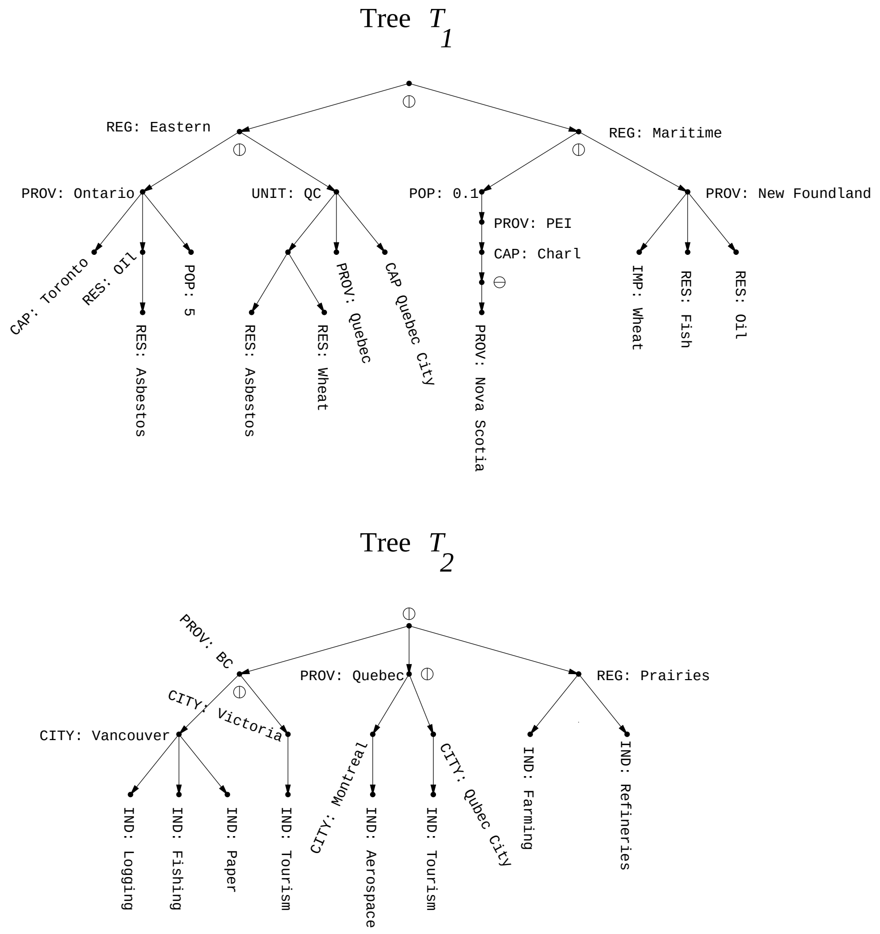

Fig. 4. A Canadiana Forest.

Consider now the following queries.

Query 1 Suppose we want the name and capital of all recorded provinces in tree T_1. Note that the user does not necessarily have any knowledge about the structure of the database. This query is expressed by

$$(t \mid \exists s : T_1(s) \ \wedge \ t.\texttt{PROV} = s.\texttt{PROV} \ \wedge \ t.\texttt{CAP} \doteq u.\texttt{CAP}).$$

This CC expression will return (a tree with) the following set of components (as branches): $\{(\texttt{PROV}:\texttt{Ontario},\ \texttt{CAP}:\texttt{Toronto}), (\texttt{PROV}:\texttt{Quebec},\ \texttt{CAP}:\texttt{QuebecCity}), (\texttt{PROV}:\texttt{PEI},\ \texttt{CAP}:\texttt{Charlottetown}), (\texttt{PROV}:\texttt{NovaScotia}), (\texttt{PROV}:\texttt{NewFoundland})\}$. In component algebra this query is formulated as

$$\pi_{\texttt{PROV},\texttt{CAP}}(\sigma_{\texttt{PROV} \not\doteq \perp}(T_1)).$$

Note that the Component Algebra query would return a "substructure" of T_1, induced by the PROV and CAP nodes (if any), as the answer.

Query 2 List those provinces in T_1 that are in any way associated with `wheat`, and their capitals.

$$(t \mid \exists s : T_1(s) \ \wedge \ t.\texttt{PROV} = s.\texttt{PROV} \ \wedge \ t.\texttt{CAP} \doteq s.\texttt{CAP} \ \wedge \ s[\texttt{wheat}]).$$

The answer will include the following set of components $\{(\texttt{PROV}:\texttt{Quebec},\ \texttt{CAP}:\texttt{QuebecCity}), (\texttt{PROV}:\texttt{NewFoundland})\}$. In component algebra, this query can be expressed as

$$\pi_{\texttt{PROV},\texttt{CAP}}(\sigma_{\texttt{PROV} \not\doteq \perp \wedge \_ = \texttt{wheat}}(T_1)).$$

Query 3 List all information about those provinces in T_1 for which the capital and population is recorded

$$(t \mid T_1(t) \ \wedge \ t[\texttt{PROV}] \ \wedge \ t[\texttt{CAP}] \ \wedge \ t[\texttt{POP}]).$$

The answer will be $\{(\texttt{PROV}:\texttt{Ontario},\ \texttt{CAP}:\texttt{Toronto}, \texttt{RES}:\texttt{Oil},\ \texttt{POP}:5), \{(\texttt{PROV}:\texttt{Ontario},\ \texttt{CAP}:\texttt{Toronto}, \texttt{RES}:\texttt{Asbestos},\ \texttt{POP}:5), (\texttt{PROV}:\texttt{PEI},\ \texttt{CAP}:\texttt{Charlottetown}, \texttt{POP}:0.1)\}$ In algebra this would be written as

$$\sigma_{\texttt{PROV} \not\doteq \perp \wedge \texttt{CAP} \not\doteq \perp \wedge \texttt{POP} \not\doteq \perp}(T_1).$$

Query 4 Join the information in T_1 and T_2.

$$(t \mid \exists u,\ v : T_1(u) \wedge T_2(v) \wedge u \sqsubseteq t \wedge v \sqsubseteq t).$$

In Component Algebra we would simply write $T_1 \bowtie T_2$. The result of the algebraic join is shown in Figure 5

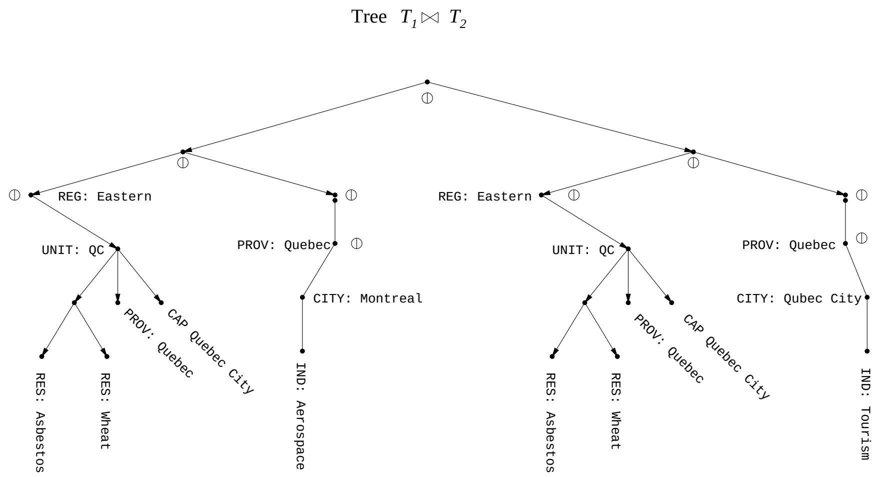

Fig. 5. The result of a natural join.

4 On the Expressive Power of Component Calculus and Algebra

4.1 Calculus vs. Algebra.

In this section, we study the expressive power of Component Calculus/Algebra. First, we establish that while CC is representation independent but not representation preserving, CA is both. Next we show that CC is *instance complete.* Instance completeness is a notion that generalizes the BP-completeness of Bancilhon and Paredaens to semi-structured databases. The natural next question is whether Component Calculus collapses to Relational Calculus when inputs and outputs are relational. We show that when we consider instances of a predefined (i.e. conservative) schema **A**, then CC and Tuple Relational Calculus (TRC) indeed have the same expressive power. However, there are queries expressible in CC, such that no equivalent TRC expression has size polynomial in the size of **A**. In the case where instances can have arbitrary (i.e. liberal) schemas, we show that there are queries expressible in CC that are not expressible in TRC.

Theorem 2. *(1) The component calculus is representation independent, but not representation preserving. (2) The component algebra is both representation independent and representation preserving.*

This phenomenon is due to the declarative/procedural dichotomy between calculus and algebra. The Component Calculus "talks" about (connected) components of the input databases. It does not distinghuish between content equivalent representations of the same components. Therefore the calculus is representation independent. On the other hand, the calculus always returns components in a canonical form (the minimal line tree). The calculus is thus not representation preserving. The component algebra operates directly on the tree underlying the contents of the database. The algebra is designed to upset the existing structure as little as possible. On the other hand, the algebra is not syntax-dependent, in the sence that any query will produce content equivalent ouputs on content equivalent inputs.

Notice that the semantics of CC is based on the extended active domain semantics. Thus, the issue of safety does not arise. We can show the equivalence of CC and CA.

Theorem 3. *The component calculus and algebra are equivalent in expressive power. That is, for every expression E_c of the component calculus, there is an expression E_a in the component algebra, such that for every instance $\mathcal{I}$, $E_c(\mathcal{I}) \equiv_{\text{cont}} E_a(\mathcal{I})$, and vice versa.*

It is important to notice that calculus and algebra expressions will not produce identical results, although the results will always be content equivalent.

4.2 Instance Completeness.

In this section, we extend the classical notion of BP-completeness due to [8, 27] to SSDBs. Recall that in the classical case, a language $\mathcal{L}$ is BP-complete provided for every instance $\mathcal{I}$, for every query Q (that is computable and is invariant under automorphisms), there is an expression E in $\mathcal{L}$ such that $E(\mathcal{I}) = Q(\mathcal{I})$. In the case of SSDBs, because of structural variations among equivalent representations of the same information, we modify this notion as follows. In order to avoid confusion, we term the extended notion instance completeness, rather than BP-completeness. A query language $\mathcal{L}$ for SSDBs is *instance complete* provided for every instance $\mathcal{I}$, for every query Q, there is an expression E in $\mathcal{L}$ such that $E(\mathcal{I}) \equiv_{\text{cont}} Q(\mathcal{I})$. We have the following results.

Theorem 4. *The component calculus and component algebra both compute only queries.*

Theorem 5. *The component calculus is instance complete.*

Corollary 1. *The component algebra is instance complete.*

Our proof of Theorem 5 is based on an extension of the technique developed by Paredaens. One of the novelties is finding a tree representation for the auto-embedding group of an instance [19].

4.3 Component vs. Relational Calculus.

A semi-structured database has its schema implicitly defined in the extension. Thus the schema is not known before processing the instance. The Component Calculus is designed so that it can deal with the situation in an instance independent manner. All we know is that the attribute names come from a countable set

$\mathcal{A}$. Thus, the relevant notion of schema in this case is that of a liberal schema. On the other hand, if we have a predetermined collection of semi-structured databases, the set of all attributes occurring in the collection is finite, and one can apply the notion of the so-called conservative schema.

To understand how CC compares with tuple relational calculus (TRC) in terms of expressive power, we need to extend the latter to handle inapplicable nulls [24]. This we do by introducing the equality $\doteq$ and adding the symbol $\perp$ to TRC's vocabulary. Specifically, the extended language, TRC$^{\perp}$, consists of all atoms of TRC, together with $t.A \doteq s.B$, and $t.A \doteq \perp$, where t, s are tuple variables and A, B are attributes. Their semantics is exactly that in CC.

Consider a set of attributes $\mathbf{A} = \{A_1, \ldots, A_n\}$, and relations over $\mathbf{A}$, possibly containing inapplicable nulls. Define a query $Q^{\mathbf{A}}_{\mathrm{rel}}$ over such relations r as follows.

$$Q^{\mathbf{A}}_{\mathrm{rel}}(r) = \begin{cases} \{()\} \text{ if } \exists t \in r : t.A_1 \not\doteq \perp \ \&\ t.A_n \not\doteq \perp \\ \qquad \text{or } \exists t_1, t_2 \in r : t_1[A_1] \ \&\ t_2[A_n] \ \& \\ \qquad\quad \forall A \in \mathbf{A} : [t_1.A \not\doteq \perp \ \&\ t_2.A \not\doteq \ \Rightarrow t_1.A = t_2.A]. \\ \{\} \quad \text{otherwise.} \end{cases}$$

By adopting the technique for simulating relational databases as SSDBs (e.g., see [11]) with a $\oplus$ placed at the root node, we can see that $Q^{\mathbf{A}}_{\mathrm{rel}}$ is a query over SSDBs in the sense of Definition 2. For simplicity, below we blur the distinction between a relational database and its SSDB simulation. We have the following results.

Theorem 6. *There is an expression in the component calculus, with size linear in the number of attributes in* $\mathbf{A}$*, that computes* $Q^{\mathbf{A}}_{\mathrm{rel}}$.

Theorem 7. *(1) There is an expression in TRC*$^{\perp}$ *that computes* $Q^{\mathbf{A}}_{\mathrm{rel}}$. *(2) However, there is no expression* E *in TRC*$^{\perp}$*, with size polynomial in the number of attributes in* $\mathbf{A}$*, such that* E *computes* $Q^{\mathbf{A}}_{\mathrm{rel}}$.

Recall that $\mathcal{A}$ denotes the countable set of all possible attributes. In most "real-life" SS applications, the schema of the potential input databases is not known, just as the number of tuples in a relation in the classical model is not known in advance. In this case, the notion of liberal schema, i.e. an arbitrary finite subset of $\mathcal{A}$, not known *a priori*, applies. We extend the above query $Q^{\mathbf{A}}_{\mathrm{rel}}$ to this situation as follows. Let T be a relation over an arbitrary finite subset $\mathbf{A} \subset \mathcal{A}$, which is not known *a priori*. The definition of $Q^{\mathcal{A}}_{\mathrm{rel}}$ is exactly the same as that of $Q^{\mathbf{A}}_{\mathrm{rel}}$ for the conservative schema. The CC query expression

$$(t \mid (\exists t : T(t) \wedge t[A_1] \wedge t[A_n]) \vee \\ \qquad (\exists t, s_1, s_2 : T(s_1) \wedge T(s_2) \wedge s_1[A_1] \wedge s_2[A_n] \wedge s_1 \sqsubseteq t \wedge s_2 \sqsubseteq t))$$

clearly computes $Q^{\mathcal{A}}_{\mathrm{rel}}$. Indeed, the same formula computes $Q^{\mathbf{A}}_{\mathrm{rel}}$ when the schema is conservative. We have

Theorem 8. *The query* $Q^{\mathcal{A}}_{\mathrm{rel}}$ *is expressible in the component calculus. However, there is no expression in TRC*$^{\perp}$ *which can compute this query.*

A restricted version of this result has been independently discovered recently by Van den Bussche and Waller [15].

5 Where do the labels come from?

Our data model assumes that the trees are appropriately labeled with blockers. How does the system obtain these blockers? There are two answers to this question. First, the data may be "raw data" obtained from the web. In this case, schema mining techniques similar to [26] can be used. We do not explore this approach further in this context. Second, the trees might be the result of exporting structured or semi-structured data into a common exchange format. We contend that our enhanced OEM model is well suited for this purpose. To substantiate this claim, we briefly sketch how relations, nested relations, and XML data can represented in our enhanced OEM model in a manner that makes the fact structure explicit.

Relational data. A tuple $[A : a, B : b, C : c]$ can be modelled as any connected component with three nodes, having atttribute-value pairs $(A : a)$, $(B : b)$, and $(C : c)$, respectiveley. A relation $R = \{t_1, \ldots, t_n\}$ is modelled as a tree with root with attribute-value pair $(R : \perp)$ and label ⦶, and subtrees corresponding to the components for the tuples $t_1, \ldots, t_n$.

Nested Relational data. Relational tuples and relations are modelled as above. A possibly nested tuple $[A_1 : v_1, \ldots, A_n : v_n]$ is represented as a tree with a null node as root. The root has children representing each pair $(A_i : v_i)$ If v_i is an atomic value, then the pair is modelled as above. Otherwise we create node with value $(A_i : \perp)$, and a horisontal blocker If v_i is a tuple, then the created node has one child, which is the root of the component representing the tuple v_i. If v_i is a set, then the created node also has a vertical blocker, and children representing the elements.

XML data. The structure of XML documents are described through Data Type Definitions (DTD's) [10]. In Figure 6 we see a simple example of a DTD specification for a course calendar. The specification states that a calendar consists of graduate and undergraduate courses. A course entry gives a title and a schedule for a course, or just the title.

```
<DOCTYPE Calendar
   <!ELEMENT Calendar (Gradcourse, Ugradcourse)>
   <!ELEMENT Gradcourse (Entry*)>
   <!ELEMENT Ugradcourse (Entry*)>
   <!ELEMENT Entry (Title, Schedule) | Title>
   <!ELEMENT Title CharData>
   <!ELEMENT Title Chardata>
]>
```

Fig. 6. A DTD Specification.

Using the DTD in Figure 6, an XML instance can be parsed into a labeled tree in the obvious way. All we need to add is the placement of the blockers

$\obar, \ominus$ in the parsed instance. For this we use the following (informal) rules: (1) A node labeled with an element name that appears in a list specification (e. g. `Gradcourse`) is labeled with $\ominus$. (2) The parent of a node labeled with an element name that appears in a star specification (e. g. `Gradcourse`) is labeled with $\obar$. (3) Leaves are not labeled with blockers.

Following the method sketched above, the XML instance in Figure 7 would result in the database tree in Figure 8. Note that the method is informal. We are currently investigating automated wrapping of XML data into our enhanced OEM-model.

```
<CALENDAR> <GRADCOURSE> <ENTRY> Databases, Mo-We-5-9 </ENTRY>
                        <ENTRY> OLAP<\ENTRY>
           </GRADCOURSE>
           <UGRADCOURSE> <ENTRY> Datastructures </ENTRY>

           </UGRADCOURSE>
</CALENDAR>
```

Fig. 7. An XML instance.

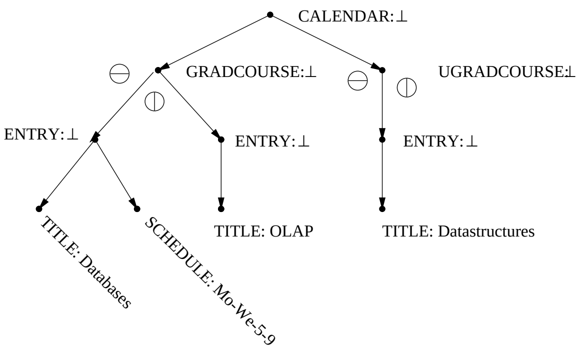

Fig. 8. The Labeled Tree.

6 Related Work

In a recent paper, Abiteboul and Vianu [1] study generic computations over the world wide web. They model the web as a simple relational database with links as relations, and equipped with certain functional and inclusion dependencies. Their notion of genericity is based on the classical database isomorphisms on the simple relational database. Abiteboul and Vianu study the expressive power of existing languages, such as FO and Datalog. In a related paper, Mendelzon and Milo [23] propose essentially an object-based version of the model of Abiteboul and Vianu. The notion of genericity is still confined to the classical domain permutations. Mendelzon and Milo propose a web-calculus that operates on web-objects. Both aforementioned works [1, 23] deal only with world wide web computations, and their analyses do not spell out the fact structure of SSDBs. As far as we know, the notions of representation independence and preservation have not been addressed before.

7 Summary and Future Work

BP-completeness, originally proposed in [8, 27] was a breakthrough which led to the subsequent development of the notions of genericity and complete query languages by [16], which in turn plays a central role in the theory of classical database queries. BP-completeness essentially provides an instance dependent characterization of the expressive power of a query language. The analog of BP-completeness for object creating languages was developed by [5] which subsequently led to the development of an appropriate extension of the notion of genericity as well as a complete query language for the graph-based object-oriented GOOD data model [14].

In this paper, we have extended BP-completeness for SSDBs. In this connection, we have introduced the notion of representation independence for SSDB query languages, which can be regarded as a first important step toward achieving the repository independence envisioned by Hull [21]. We have also introduced a notion of genericity and representation preservation for SSDB query languages. The latter captures the spirit of pure querying as opposed to restructuring. We have defined a calculus and an equivalent algebra for SSDBs, and shown that both languages are generic and representation independent, and that the algebra is also representation preserving, thus lending itself to efficient implementation. Both languages are instance complete. Finally, the component calculus can express queries much more concisely than $\mathrm{TRC}^{\perp}$, an extension to classical TRC for handling inapplicable nulls. Moreover, when the schema is not predefined, CC can express queries that $\mathrm{TRC}^{\perp}$ cannot.

It is our hope that work presented in this paper will be useful in the development of a comprehensive theory of queries for SSDBs. Clearly, much work remains to be done. In this paper we have concentrated on SSDBs which are forests. Handling more general SSDBs is an important problem. In this vein, a particularly promising direction is to explore a model in the spirit of the hypertrees of [6] with cross-links between trees, possibly with cycles, rather than a completely arbitrary digraph. Another interesting direction is to extend the theory to include restructuring, a feature already exhibited by many of the languages surveyed in Section 1.

References

1. Abiteboul, S., and V. Vianu. Queries and computation on the Web. *ICDT '97.*
2. Abiteboul, S. et al. The Lorel query language for semistructured data. *Intl. J. on Digital Libraries*, 1:68–88, 1997.
3. Abiteboul, S., and V Vianu. Regular path queries with constraints. *PODS '97.*
4. Abiteboul, S. Querying Semi-Structured Data. *ICDT '97.*
5. Andries, M., and Paredaens, J.. On instance-completeness for database query languages involving object creation. *J. Comp. Syst. Sci.*, 52(2):357–373, 1996.
6. Arocena, G., and A. O. Mendelzon. WebOQL: Restructuring documents, databases and webs. *ICDE '98.*
7. Atzeni, P., G. Mecca, and P. Merialdo. To weave the web. *VLDB '97.*
8. Bancilhon, F.. On the Completeness of Query Languages for Relational Data Bases. *MFOCS '78.*
9. Beeri, C., and T. Milo. Schemas for Integration and Translation of Structured and Semi-Structured data. *ICDT '99.*
10. Brey, T., J. Paoli, and S. Sperberg-McQuenn. Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml`
11. Buneman,P. et al. A query language and optimization techniques for unstructured data. *SIGMOD '96.*
12. Buneman, P., and A. Ohori. Polymorphism and Type Inference in Database Programming. *ACM TODS*, 21:30–76, 1996.
13. Buneman,P. et al. Adding structure too unstructured data. *ICDT '97.*
14. Bussche, J. Van den et al. On the Completeness of Object-Creating Database Transformation Languages. *JACM*, 44(2):272–319.
15. Bussche, J. Van den, and E. Waller. Type inference in the polymorphic relational algebra. *PODS 99.*
16. Chandra A. K. and D. Harel. Computable queries for relational data bases. *J. Comp. Syst. Sci.*, 21:156–178, 1980.
17. Fernandez, M., D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.
18. Florescu, D., A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. *PODS '98.*
19. Grahne, G., and L.V.S. Lakshmanan. On the Difference between Navigating Semi-structured Data and Querying It. Technical report, Concordia University, Montreal, Canada, August 1999. in preparation.
20. Gyssens, M., L.V.S. Lakshmanan, and I. N. Subramanian Tables as a paradigm for querying and restructuring. *PODS '96.*
21. Hull, R.. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. *PODS '97.*
22. Manna, Z., *Mathematical Theory of Computation.* McGraw-Hill, New York, Montreal, 1974.
23. Mendelzon, A. O., and T. Milo. Formal models of web queries. *PODS '97.*
24. Lerat, N., and W. Lipski. Nonapplicable nulls. *TCS*, 46:67–82, 1986.
25. Ley, M.. Database Systems and Logic Programming Bibliography Server. `http://www.informatik.uni-trier.de/ ley/db/.`
26. Nestorov, S., S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. *SIGMOD '98.*
27. Paredaens, J. On the expressive power of the relational algebra. *IPL*, 7:107–110, 1978.

28. Rajaraman, A., and J. D. Ullman. Integrating Information by Outerjoins and Full Disjunctions. *PODS '96*.
29. Thierry-Mieg, J., and R. Durbin. A C.elegans database: syntactic definitions for the ACEDB data base manager, 1992.
30. Ullman, J. D. Database Theory - Past and Future. *PODS '87*.