CRIUS: User-Friendly Database Design*

Li Qian
Univ. of Michigan, Ann Arbor
eql@umich.edu

Kristen LeFevre
Univ. of Michigan, Ann Arbor
klefevre@umich.edu

H. V. Jagadish
Univ. of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

Non-technical users are increasingly adding structures to their data. This gives rise to the need for database design. However, traditional database design is deliberate and heavy-weight, requiring technical expertise that everyday users may not possess. For this reason, we propose that users of personal data management applications should be able to create and refine data structures in an ad-hoc way over time, thereby "organically" growing their schemas. For this purpose, we develop a spreadsheet-like direct manipulation interface. We show how integrity constraints can still provide value, even in this scenario of frequent schema and data modifications. We also develop a back-end database implementation to support this interface, with a design that permits schema changes at a low cost.

We have folded these ideas into a system, called CRIUS, which supports a nested data model and a graphical user interface. From the user's perspective, the chief advantages of CRIUS are its support for simple schema definition and modification through an intuitive drag-and-drop interface, as well as its guidance towards user data entry based on incrementally updated data integrity. We have evaluated CRIUS by means of user studies and performance studies. The user studies indicate that 1) CRIUS makes it much easier for users to design a database, as compared to state-of-the-art GUI database design tools, and 2) CRIUS makes user data entry more efficient and less error-prone. The performance experiments show that 1) the incremental integrity update in CRIUS is very efficient, making the data entry guidance applicable and 2) the back-end database implementation in CRIUS significantly improves the performance of schema update tasks, without a significant impact on other operations.

1. INTRODUCTION

1.1 Motivation

As technology makes inroads into our daily lives, non-technical people are increasingly discovering the necessity of storing, managing, accessing, and manipulating electronic data. In effect, we are seeing the masses, who lack technical expertise, managing personal and business data, without help from any consultants or DBAs.

*This work was supported in part by NSF grant IIS 1017296.

Proceedings of the VLDB Endowment, Vol. 4, No. 2

Name	City	Address
Orlando	Erie	2251 Elliot
Keith	Erie	3207 Grady

(a) Simple Spreadsheet

City	Name	[Address]
		Address
Erie	Orlando	2251 Elliot
Erie	Keith	3207 Grady
		7943 Walnut

(b) Structured Spreadsheet

Figure 1: Example Address Books

Where the data is a single list, most users have no difficulty with the structure. But even with slightly more complex structures, there frequently are choices to be made, with both performance and extensibility implications. In fact, the requirement for schema specification is a major barrier to database use.

Due to their simplicity, spreadsheets are still the most common data management application used by people without technical training. By design, spreadsheets implement an extremely simple data model, consisting of a single flat table, with rows and columns. However, due to the increasing complexity of personal and small-business data, users often find it necessary to augment this basic data model with additional ad-hoc structure. Implicitly, these users are defining schemas for their data.

EXAMPLE 1. *Consider the basic task of maintaining a personal address book. In the simplest case, this can be done using the basic data model of spreadsheets, as shown in Figure 1(a). However, it is easy to imagine situations in which the user will require more structure in order to manage her data. For example, suppose that the contact named Keith purchases a second home. In this case, the user is likely to capture this by defining some ad-hoc structure, for example as shown in Figure 1(b).*

The above example illustrates the challenge of allowing non-technical users to define and evolve their schemas. One might argue that most data management applications will be designed by professionals, who will construct appropriate schemas for a large class of users. Indeed, this may be sufficient in some scenarios. However, there will always be users who are not completely satisfied with what they can get out of the box and desire something more. Even users who are initially satisfied with an application may wish to enhance or customize it as their requirements change. For instance, after the user has made some international friends, she may wish to record the nationality for each of her contacts. Ideally, the database should respond gracefully to changes like this, in a way that places minimal burden on the user.

In this paper, we study the issue of user-defined structure in data. How can an end user, beginning with a simple spreadsheet, such as an address book with just names and phone numbers, extend this "database" to include more attributes and structure? This sort of organic "schema evolution" may happen because of an intrinsic need to capture some additional information (e.g., nationalities). It will also take place when the user comes upon the need to represent some information that does not quite fit the current structure. For example, when the user realizes that some of her contacts are

married couples, she may decide that it makes more sense to store one address per family, rather than one address per person.

Traditional database design is deliberate – there is extensive gathering of requirements, careful analysis, and methodical step-by-step design, all performed by highly trained personnel. In contrast, the database "design" in our scenario is organic – it is not carefully considered, and it is expected to be modified as new data reveals weaknesses in the current design or exposes assumptions that are now violated. If schema modification is possible at a reasonable cost, this sort of organic schema growth is not a problem – rather it becomes the desired style of design. But to accomplish this, schema modification must be rendered easy and cheap.

Based on our experience, a spreadsheet-like nested-table may naturally support such organic schema growth for non-technical users. Observe, for example, the propensity of non-technical users to define complex spreadsheets (e.g., using Microsoft Excel), rather than migrating their data to relational databases, even those that are designed for personal and small-business use (e.g., Microsoft Access). We believe the reason for this is that a typical relational database is too discrete for end-users to manipulate conveniently. Users have to understand table schemas separately and learn the inter-table relationships. In contrast, having the information of interest folded into a single spreadsheet-like structure makes it much easier to comprehend.

While the users may enjoy the freedom of organically growing their schema, we have to be aware that user-defined schemas are subject to denormalization. Consequently, users have to explicitly deal with duplicated data entries, which may easily produce errors violating integrity constraints. In this paper, we also study how to provide usable guidance towards data entry in such a freestyle environment, by efficiently managing integrity constraints.

1.2 Challenges

In summary, our goal is to support organic schema creation and modification using a natural spreadsheet-like interface, while ensuring efficient and effective data entry on this organic schema. To accomplish this, there are multiple challenges to be addressed:

- **Schema Update Specification:** Specifying a schema update as in Example 1 is challenging with existing tools. For example, using conventional spreadsheet software, it is impossible to arrive at a hierarchical schema such as the one in Figure 1(b). Alternatively, using a relational DBMS, one has to manually split the table and set up the cross-table relationships. This is not easy for end-users, even with support from GUI tools. Instead, we would like to support schema creation and modification via a direct manipulation ("point-and-click") interface.
- **Data Migration:** Once a new schema is specified, there is still a critical task of migrating existing data to the new schema. If the schema is simply augmented, this migration may be easy. However, if the schema structure is changed (e.g, from allowing one address per contact to multiple addresses), then one has to introduce a complex mapping in order to "fit" the existing data into the new schema. Even if spreadsheet software supporting hierarchical schema is provided, the user still has to manually copy data in a cell-by-cell manner to perform such mappings. This process is both time-consuming and error-prone.
- **Data Entry:** As a result of the denormalization due to organic schema growth, data entry may become inefficient and error-prone. One may use integrity constraints to assist in data entry. However, since users cannot understand complex integrity constraints, and constraints are also subject to user update, this cannot be done in a naive way. It other words, we have to construct a positive feedback loop between data entry and constraint update in the usability context, with a practical response time.
- **Schema evolution Performance:** Schema evolution is usually a heavy-weight operation in traditional systems. IT organizations allow days to execute schema evolutions, since they plan them carefully in advance. However, everyday users require a swift feedback when updating schema as their plans are immediate and casual. Thus, we need to develop techniques to support quick schema evolution without giving up other desirable features.

1.3 Contributions

The primary contribution of this paper is the design of a novel system called CRIUS. From the user's perspective, the main benefit of CRIUS lies in the user-friendly interface, which allows non-technical users to create and modify the schemas associated with their data in an organic way. Rather than requiring users to adjust data to the new schema in a cell-by-cell manner, we have designed the UI so that each schema update requires only a single drag-and-drop with the mouse. The details of this interface, addressing the specification challenge, can be found in Section 2.1. The design of direct manipulation operators on the interface, addressing the migration challenge, is covered in Section 2.2.

To address the data entry challenge, we have designed a data entry guidance feature, which auto-completes duplicated values and reacts to possible input errors based on functional dependancies (FDs). To make this feature practical, one has to break the performance bottleneck of repeatedly inducing FDs using traditional algorithms. Thus, we propose the first incremental algorithm for FD induction. How CRIUS assists data entry by FDs, and how these FDs are incrementally computed are discussed in Section 3.

CRIUS uses a relational database as a natural back-end. Ideally, the storage format should support efficient schema evolution, which has not conventionally been a priority in relational databases. For this reason, we suggest a storage format whereby nested relational tables are recursively, vertically partitioned into flat relations. These implementation issues, as discussed in Section 4, address the challenges of data migration and performance.

Finally, in Section 5, we describe the evaluation of our prototype system. Through user studies, we have found that schema development is much easier in CRIUS than using a GUI tool provided by Microsoft SQL Server Management Studio 2008. The study also shows that the integrity-based guidance feature does reduce typing effort and input errors. Our experiments demonstrate that our incremental FD induction algorithm is much faster than traditional approaches, and the schema modification and data reconstruction performance of CRIUS are reasonable.

2. CRIUS DESIGN

2.1 Interface Design

The presentation layer of our system is based on a next-generation spreadsheet, as described in the introduction. Information from multiple related "tables" is combined to present a cohesive nested representation, as shown in Figure 2.

Conventional spreadsheets are not designed to handle complex schemas. If a user wants to add more structure to her data, this must be done in an ad-hoc way. Worse, spreadsheets only support cell-by-cell data modifications. As a result, if the user wishes to modify her schema, this is a complex and error-prone process that often consists of cell-, row-, and column- level copy and paste operations.

In contrast, the CRIUS user interface supports easy schema creation and modification through a simple drag-and-drop interface, as shown in Figure 2. The shaded region at the top of the screen is the schema header, and the region below it displays the data body. The user can modify the schema by simply dragging a cell in the schema header using the mouse. For example, Figure 3(a) shows a

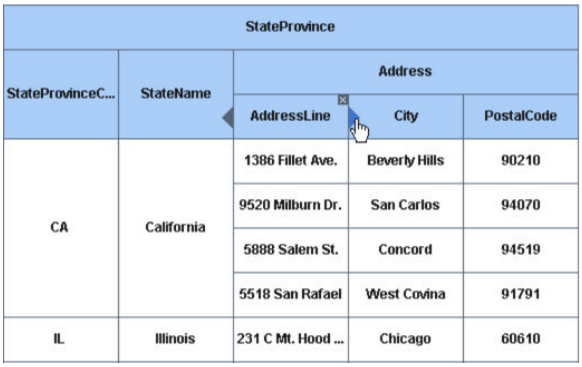
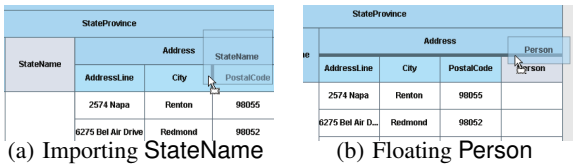


Figure 2: Screenshot of CRIUS



(a) Importing StateName (b) Floating Person

Figure 3: Screenshot of Schema Evolution in CRIUS

screenshot of a user dragging an attribute StateName inward, and making it part of the sub-relation Address. (We refer to this operation as an IMPORT.) Conversely, the user can EXPORT StateName back to the StateProvince relation in a similar way.

The user can also create new sub-relations by dragging attributes up and down. Figure 3(b) shows an example where Person is dragged up to insert a new intermediate level with only itself. Similarly, one can also drag it down, nesting it to a new sub-relation. (We refer to these operations as FLOAT and SINK, respectively.)

2.2 Operator Design

We refer to an instance of the spreadsheet in CRIUS as a *span table*. The UI allows users to restructure the span table schema using drag-and-drop direct manipulations (e.g., IMPORT, EXPORT, FLOAT, and SINK, as described above), and to augment/diminish schema using point-and-clicks (adding/dropping columns). It also supports data manipulation operations (inserting/deleting tuples, updating cells). Collectively, we will refer to this set of operators as the *span table algebra*. (A brief introduction to the span table operators can be found in Appendix A.) In this section, we introduce our key schema update operators in the span table algebra, and compare the expressive power of the algebra to the nested relational algebra.

2.2.1 IMPORT and EXPORT

City	Zipcode	[Person]	
		Name	[Address]
			Address
Detroit	48205	Peter	1023 Westwood Ave
Erie	48109	Orlando	2251 Elliot Avenue
Erie	48105	Keith	3207 S Grady Way
			7943 Walnut Ave

(a) Address Book Before Evolution

City	[Person]		
	Name	[Address]	
		Zipcode	Address
Detroit	Peter	48205	1023 Westwood Ave
Erie	Orlando	48109	2251 Elliot Avenue
	Keith	48105	3207 S Grady Way
		48105	7943 Walnut Ave

(b) Address Book After Evolution

Figure 4: Address Book With Multiple Levels

EXAMPLE 2. *Suppose the user has an address book span table as shown in Figure 4(a), and wishes to associate* Zipcode *with* Address *rather than* City. *The CRIUS UI enables the user to do this by simply pressing the mouse on* Zipcode *, and dragging it onto* [Address]. *The system then needs to execute the schema update and transform the nested relation to the one shown in 4(b). Readers familiar with nested relational algebra may consider an implementation consisting of a series of nest and unnest: 1) unnest [Person], 2) unnest [Address], 3) nest Zipcode, Name and Address, and 4) nest Zipcode and Address. However, this would introduce a large amount of unnecessary computation moving data from unrelated columns (e.g.* Name *and* Address*). Moreover, this series of operations does not semantically conform to the user's intention of the simple drag-and-drop manipulation.*

To overcome the problems with the traditional nested algebra in our scenario, we introduce two new schema modification operators, namely *IMPORT* and *EXPORT*. We set up some notation and then define the basic IMPORT and EXPORT operators.

A nested relation N has schema expression $S(N)$ and schema tree $Tree(N)$. $S(N)$ is the flattened version of $Tree(N)$ by a postorder traversal. For example, $S(N)$ for Fig. 4(a) is {City, Zipcode, {Name, {Address}}}. $t[\overline{X}]$ is a tuple projection, where t is a nested relational tuple and $\overline{X}$ is a list of attributes. $\bigsqcup$ is the relation union operation used in [20].

The IMPORT operator imports an atomic attribute into a nested relation. Intuitively, it pushes down the atomic attribute to become a "child" of a sibling group.

DEFINITION 1 (BASIC IMPORT OPERATOR). *Given a nested relation N with $S(N) = \{\overline{A}\overline{X}P\{\overline{Q}\}\}$, where $\overline{A}$ denotes a list of atomic attributes, $\overline{X}$ denotes a list of relation-valued attributes, P denotes an atomic attribute being transported and $\{\overline{Q}\}$ denotes the target relation-valued attribute, which consists of a list of both atomic and relation-valued attributes, $IMPORT_{P,\{\overline{Q}\}}(N) = N'$, where $S(N') = \{\overline{A}\overline{X}\{P,\overline{Q}\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:*

(1) $t'[\overline{A}] = t[\overline{A}]$

(2) $\forall X$ *in list* $\overline{X}, t'[X] = \bigsqcup\{t''[X] | t'' \in N \wedge t''[\overline{A}] = t[\overline{A}]\}$

(3) $t'[\{P,\overline{Q}\}] = \{t'' | \exists t''' \in N, s.t.\ t''[P] = t'''[P] \wedge t''[\overline{Q}] \in t'''[\{\overline{Q}\}] \wedge t''[\overline{A}] = t'''[\overline{A}]\}$

According to this definition, the manipulation in Example 2 can be executed in two imports: 1) $IMPORT_{Zipcode,\{Name,\{Address\}\}}$ and 2) $IMPORT_{Zipcode,\{Address\}}$.

The EXPORT operator is the inverse of IMPORT. It raises an atomic attribute from a deeper nested level to a shallower nested level and naturally maps existing data to the new schema.

DEFINITION 2 (BASIC EXPORT OPERATOR). *Given a nested relation N with $S(N) = \{\overline{A}\overline{X}\{P,\overline{Q}\}\}$, where $\overline{A}$ denotes a list of atomic attributes, $\overline{X}$ denotes a list of relation-valued attributes, and $\{P,\overline{Q}\}\}$ denotes the source relation from which the atomic attribute P will be extracted and inserted into the target relation $S(N)$, $EXPORT_P(N) = N'$, where $S(N') = \{\overline{A}\overline{X}P\{\overline{Q}\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:*

(1) $t'[\overline{A}] = t[\overline{A}]$

(2) $t'[\overline{X}] = t[\overline{X}]$

(3) $\exists t'' \in t[\{P,\overline{Q}\}] s.t. t'[P] = t''[P]$

(4) $t'[\{\overline{Q}\}] = \{t''[\overline{Q}] | t'' \in t[\{P,\overline{Q}\}] \wedge t''[P] = t'[P]\}$

For instance, executing $EXPORT_{Zipcode}$ twice will bring the span table in 4(b) back to the span table in 4(a).

In practice, our algebra extends IMPORT/EXPORT to span multiple schema levels, which allows the user to import Zipcode from the root to Address in one step. The algebra also includes schema update operators to create new sub-relations (SINK/FLOAT) together with other data manipulation operators. The extensions of IMPORT/EXPORT and the definitions of other operators are detailed in Appendix A.

2.2.2 Expressive Power Analysis

Though nested relational algebra does not naturally lend itself to supporting a direct manipulation interface, we show that the span table algebra and nested relational algebra are actually equivalent in expressive power given a fixed set of universal attribute (recall that nested relational algebra cannot add/drop attributes). To do this, all we have to show is that NEST and UNNEST can both be expressed using the span table algebra, and vice versa. For simplicity, we will restrict IMPORT and EXPORT to their basic versions in lemmas 3 and 4. The general case follows directly by induction.

The IMPORT and EXPORT operators can be expressed in terms of NEST and UNNEST, as given by the following two lemmas. The SINK and FLOAT operators are just restricted versions of NEST.

LEMMA 1. *Let N be a relation with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, then* $IMPORT_{P,\{\overline{Q}\}}(N) = NEST_{P,\{\overline{Q}\}} \cdot UNNEST_{\{\overline{Q}\}}(N)$.

LEMMA 2. *Let N be a relation with $S(N) = \{\overline{AX}\{P,\overline{Q}\}\}$, then* $EXPORT_P(N) = NEST_{\{\overline{Q}\}} \cdot UNNEST_{P,\{\overline{Q}\}}(N)$.

Also, we have the following theorem.

THEOREM 1. *Any NEST or UNNEST can be expressed as a sequence of span-algebra schema update operations.*

The proofs of the lemmas and the theorem are in Appendix A.

Thus, we have not sacrificed expressiveness for usability; anything that can be expressed using nested relational algebra can also be captured using the direct manipulations supported by CRIUS.

3. INTEGRITY-BASED GUIDANCE

Unlike conventional databases where data is carefully normalized according to integrity constraints at design time, in our environment, non-technical users are not able to normalize their data. This increases the user burden to enter duplicated data, as well as the risk of erroneous data entry. To address this problem, CRIUS provides an important set of features that we call *integrity-based guidance*. The basic idea is to induce from the data and maintain a set of "soft" functional dependencies.

These "soft" FDs are then used in two ways to assist in data entry. The first is *inductive completion*, which auto-completes the determined attribute (the righthand side) of an FD according to existing data. For instance, suppose we have inferred the FD $Name \rightarrow Grade$ in Table 1. If the user were to enter a new row for $Peter$, CRIUS would automatically suggest a grade A. In addition, CRIUS uses these FDs for *error prevention*, by warning about possible data entry errors. Following the above example, suppose the user has updated Leo's grade in row 3 from A to B. CRIUS will prompt the user that the update may be a mistake since it violates the inferred FDs. The user may decide whether to undo it or not. In case the user commits the update, CRIUS further asks the user if he wants to: (i) also update the other Leo's grade in row 4 to preserve the FD or (ii) force the update. Option (ii) indicates that the previously induced FD $Name \rightarrow Grade$ is an artifact of the database instance and must be updated.

Since user updates may continuously invalidate induced FDs in our environment, we need to frequently update them. While there is a large body of literature on FD induction [8, 12, 15, 26], past solutions are too expensive to be adopted in our scenario where the need to update FD is frequent. To reduce performance cost, CRIUS instead *incrementally maintains* these FDs.

The incremental maintenance can be challenging. For example, if an FD (e.g., $Name \rightarrow Grade$ in Table 1) has been invalidated by a user update (e.g., the grade in tuple 2 from A to B), it is not enough to remove that FD from the minimal set, since one or more weaker FDs (e.g., $Name, Course \rightarrow Grade$) may still hold. These weaker FDs would not have been previously recorded in the minimal set because they were dominated by the now violated FD. Similarly, after a certain update, an FD may be dominated by some stronger FDs, and become no longer minimal.

ID	Name	Course	Grade
1	Peter	Math	A
2	Peter	Physics	A
3	Leo	Math	B
4	Leo	Physics	B
5	Jack	Math	A

Table 1: Student Records

In this paper, we develop the IFDI (Incremental FD Induction) algorithm. To the best of our knowledge, this is the first algorithm to induce FDs incrementally upon value updates. The algorithm works by maintaining an in-memory lattice which contains all the information for FD induction, and incrementally updating part of the lattice. The algorithm differs from existing FD induction algorithms in three ways: 1) Unlike traditional algorithms which fetch data from the database, IFDI only accesses the database once. Subsequent updates can be efficiently processed using the in-memory lattice; 2) IFDI needs to access at most half of the lattice nodes; 3) On average, IFDI only needs to update a very small portion of each lattice node. In this section, we describe the initialization and maintenance phases of the IFDI algorithm separately. In Section 5, we show that the cost of IFDI is significantly smaller than naive approaches.

We also briefly describe how we extend IFDI to the nested scenarios and prove that all the nested FDs are preserved under schema evolution when an appropriate representation is chosen and transformed. Our user study in Section 5 shows that the guidance feature based on these induced nested FDs does improve usability.

3.1 Inducing Initial FDs

Before the incremental maintenance, we first construct a set of important data structures and induce the initial set of FDs.

For flat relations, there is a body of literature on minimal FD induction [8, 12, 15]. As a starting point, we adopt the ideas of *attribute partition* and *attribute lattice* introduced in [8]. An attribute partition on a set of attributes X, denoted by Π_X, is a set of *partition groups*, where each group contains all the tuples sharing the same values at X. For instance, in Table 1, $\Pi_{\{Name\}} = \Pi_{\{Name,Grade\}} = \{\{1,2\},\{3,4\},\{5\}\}$. An FD $X \rightarrow Y$ holds iff $\Pi_X = \Pi_{X \cup Y}$ [8, 26]. (For instance, $Name \rightarrow Grade$ holds since $\Pi_{\{Name\}} = \Pi_{\{Name,Grade\}}$.) We strip partitions by omitting partition groups of size one for simplicity (For instance, $\Pi_{\{Name\}} = \{\{1,2\},\{3,4\}\}$). An attribute lattice is a lattice in which each node corresponds to an attribute set and each edge represents a possible FD. An edge goes from node X to node Y iff Y contains X and exactly one more attribute. The attribute lattice for Table 1 is shown in Figure 5. (For simplicity, hereafter we abbreviate Name by N, Course by C and Grade by G.) The following example demonstrates the initialization phase of the algorithm.

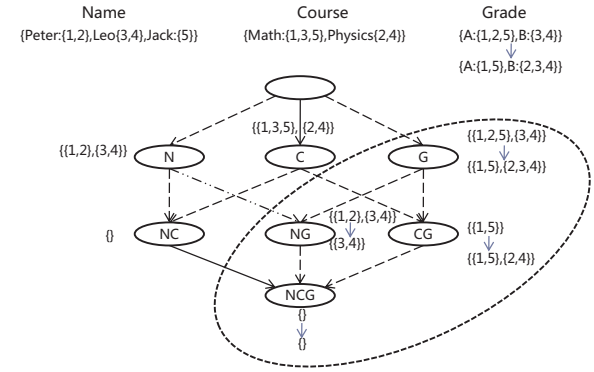


Figure 5: The Set Attribute Lattice of Table 1 in Example 3, and its evolution (as depicted by the arrows) in Example 4

EXAMPLE 3. *The algorithm reads the metadata and generates the stripped partitions for each single attribute. The stripped partitions are then fed into the first level of the lattice (shown at the top in Figure 5). The algorithm processes the lattice in a top-down manner level by level, generating the child partition by taking the product [8] of any two parent partitions (For instance.* $\Pi_C \cdot \Pi_G = \Pi_{CG}$*). The algorithm outputs an FD if its parent and child partitions are identical. For Table 1, only two FDs are discovered:* $Name \rightarrow Grade$ *(the dash and dot line) and* $Name, Course \rightarrow Grade$ *(the solid line). However, since the latter is dominated by the former, it is pruned by the algorithm.*

3.2 Maintaining FDs on Value Update

Once the lattice and the partitions for each node are constructed, the maintenance phase of IFDI is performed for each value update. IFDI checks for FD updates by traversing the lattice and comparing the partitions, in the same way as the existing algorithms. However, IFDI effectively reduces the cost of updating partitions and the number of lattice nodes that must be visited.

Efficient Partition Update: The partition product operation in the traditional FD induction algorithm is a performance bottleneck since it executes a linear scan on the partition, whose size is proportional to the number of tuples in the relation. However, when updating a partition in the incremental case, one does not need to visit most partition groups that are irrelevant to the updated cell.

EXAMPLE 4. *After the Grade of Peter in row 2 in Table 1 is updated from A to B, the lattice evolves as shown in Figure 5. One may use the traditional algorithm to compute the new* Π_{CG} *from* Π_C *and the new* Π_G*: for each group in* Π_G*, assign each tuple in it to the groups in* Π_C *and then collect them group-wise as the new groups of* Π_{CG}*. Specifically, for* $\{1, 5\}$ *in the new* Π_G*, 1 and 5 are assigned to the same group* $\{1, 3, 5\}$ *of* Π_C*. This group thus generates only one new group in* Π_{CG}*:* $\{1, 5\}$*. However, for* $\{2, 3, 4\}$ *in* Π_G*, 3 is assigned to the first group in* Π_C*, while 2,4 are assigned to the second. Thus this group generates two new groups:* $\{3\}$ *and* $\{2, 4\}$*. Collectively, the new* Π_{CG} *becomes* $\{1, 5\}, \{2, 4\}$ *(with* $\{3\}$ *stripped).*

IFDI handles the update of Π_{CG} *in an incremental way. Instead of updating every group in* Π_{CG}*, IFDI only focuses on two groups: the group to which tuple 2 previously belonged, and the group to which it will belong after the update. IFDI removes the updated tuple from the old group and adds it into the new group, with the other groups unchanged. In the example,* $\{2\}$ *is a singleton in the old* Π_{CG}*, so there is no need for removal. When assigning tuple 2 to its new group, IFDI first retrieves the group containing 2 from* Π_C *(*$\{2,4\}$ *in this case), and then scans the group for any other tuple that has the same value on the updated attribute as the modified cell (tuple 4 is returned here since* $\{2, 4\}$ *is in* Π_C *and tuple 4 also has a grade of B). IFDI then retrieves the group containing tuple 4 from* Π_{CG} *(the stripped singleton* $\{4\}$*) and adds tuple 2 to that group (forming a new group* $\{2, 4\}$*). Note that we have finished updating* Π_{CG} *without even touching group* $\{1, 5\}$ *or* $\{3\}$ *in the old* Π_{CG}*.*

Because IFDI maintains indexes for each partition and its groups, group retrieval, insertion and removal cost constant time. The major cost comes from scanning one group until a value-matched tuple is found. In the worst case, this may be as large as the size of the group. However, we prove that for many common cases, the cost is proportional to the number of distinct values in each column and exponential to the number of columns. Detailed analysis can be found in Appendix B.

Less Lattice Traversal: Another advantage of IFDI is that it only walks through the lattice nodes that contain the modified attribute. This is because the other nodes must be unchanged after the update. For instance, in Figure 5, IFDI only visits the nodes within the ellipse, which saves half of the cost.

3.3 Extending IFDI to Nested FDs

In the nested scenario, [26] designed the first system for efficient discovery of XML FDs by extending the algorithm proposed in [8, 12, 15]. In CRIUS, we seamlessly integrate the flat-case IFDI with the discoverXFD algorithm from [26] and propose the incremental nested FD induction algorithm. The algorithm works recursively from the leaves to the root of the schema tree (see Appendix A.1) and builds NFDs from potential flat FDs. It is implemented in CRIUS and studied empirically in Section 5.

3.4 Maintaining NFDs on Schema Evolution

Schema evolution may also affect Nested FDs: as the nested structure changes, certain FDs may be invalidated or satisfied. Unlike value update, the representation of NFDs largely affects its transformation on schema evolution. [7] proposed a NFD representation which can be either *global* or *local*. We prove that: 1) each local NFD can be represented as an equivalent global NFD and 2) each NFD is preserved on schema evolution, after a trivial transformation. Detailed proofs are omitted due to space.

4. RELATIONAL DATABASE BACK-END

Storage management has been so carefully engineered for relational databases that a flat relational storage manager is a natural back-end for a system such as CRIUS. [3] has suggested a possibility to store nested relations using flat tables, but a practical storage plan is left open. There is also excellent related work on storing XML in relational databases, such as [22, 5]. However, such work has traditionally focused on query performance, and has not placed a high priority on the cost of schema evolution.

We extend column store [24, 6] ideas to nested relations and develop a recursive vertical partitioning approach, which eliminates the need to ever use ALTER TABLE when performing schema updates, delivering low-overhead schema evolution. We implemented this on a row-major RDBMS, because column store requires considerable customization on hierarchical structuring. We observe in our experiments that the storage format efficiently supports other tasks (e.g., data display) that are common in our spreadsheet-like environment.

4.1 Vertical Partitioning

We represent a nested relation as a recursively vertically partitioned relational database. The vertical partitioning is standard. The recursion is the result of the nesting – additional tables are required to link nested tuples with their corresponding nesting tuples. For example, Figure 6 shows a nested relation, and its decomposition, where table (b) links the IDs of nested and nesting tuples. We also maintain a *structure table* to record how the decomposed relations are structured to represent the nested relation.

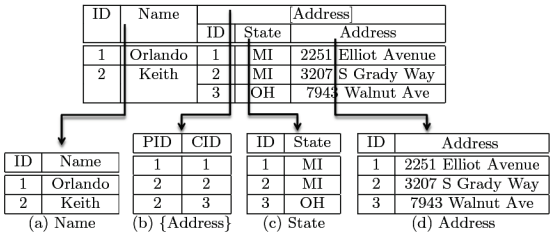


Figure 6: Vertical Partitioning Example

4.2 Upward and Downward Mappings

Using the vertically partitioned storage format, we further define a mapping from the relational database to span tables (the *upward*

mapping) and another mapping of opposite direction (the *downward mapping*). The upward mapping statically maps the relational database to a span table, according to the decomposition described above, while the downward mapping dynamically maps each span table operator to a sequence of manipulations on the underlying relational database. More details can be found in Appendix C.

5. EVALUATION

Our experiments are designed to answer four main questions:

1. How usable is the drag-and-drop interface in CRIUS, compared to state-of-the-art GUI schema design tools?
2. How usable is the integrity-based guidance?
3. How efficient is the incremental FD induction algorithm, compared to traditional FD induction approaches?
4. What are the performance implications of the storage representation, for common tasks (schema modification and data display)? How does the vertically-decomposed format compare to a standard relational storage format?

Due to space, we have focused on the big picture for each experiment. Specifications can be found in Appendix D.

5.1 User Study on Schema Operations

Our first set of experiments measured the usability of schema manipulation in CRIUS. We recruited eight volunteers with no database background, and two database experts for comparison. As a baseline, we compared CRIUS to Microsoft SQL Server Management Studio 2008 (SSMS), which is representative of the state-of-the-art GUI-based relational database design. In the first experiment, users were asked to design (from scratch) a schema for an address book. We asked the same group of subjects to accomplish this task using both CRIUS and SSMS, and recorded the time to define the schema with both tools. The times are shown in Figure 7 (D for database expert, N for non-technical users). Results show that design using CRIUS was about thrice faster. [1] While the first

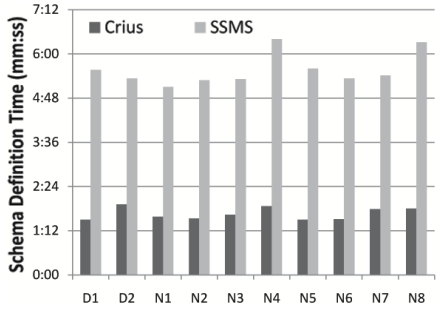


Figure 7: Time defining a schema with CRIUS vs. SSMS.

experiment tested the performance of users constructing a schema from scratch, we also conducted a second experiment, which asked users to modify an existing schema by moving an attribute. To accomplish the move using CRIUS, users only need to specify an IMPORT by a drag-and-drop. However, this task proved difficult in SSMS because, while users could construct a new schema using the GUI tool, migrating data from the old schema to the new one required them to write SQL. As a result, none of the non-technical users were able to complete the task using SSMS. In contrast, all users were able to complete the task within seconds using CRIUS.

Finally, to gain further insight into the usability of CRIUS, we asked the same users to perform the same schema update task using CRIUS and a strawman system that we constructed. The strawman implements a very similar drag-and-drop GUI interface to CRIUS; however, unlike CRIUS's span table algebra, it implements drag-and-drop manipulations that are direct implementations of nested relational algebra operators.

[1]Using the Mann-Whitney test (a standard test of statistical significance), this difference is significant with p-value < 0.0002.

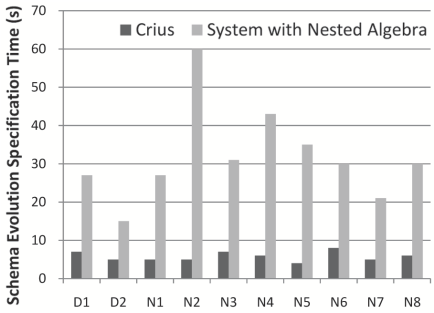


Figure 8: Time specifying an attribute transportation with CRIUS v.s. Nested Algebra GUI.

Figure 8 compares the user performance for this task. [2] All the users were able to accomplish the task almost thrice faster using CRIUS than using the system with a nested algebra interface.[3] This difference supports the intuition that the span table operators supported in CRIUS are more natural direct manipulations for users than nested algebra operators, even though the two are equivalent in expressive power within the schema restructuring domain.

5.2 User Study on Integrity-Based Guidance

Our second user study measured how much the users may benefit from the integrity-based guidance offered by CRIUS. Again, we recruited eight non-technical volunteers and two computer experts. The subjects were asked to complete three tasks on an address book in CRIUS twice, once with the guidance feature on and the other off. Their tasks were: 1) insert a new contact and his address into the address book, 2) update the cell phone number of one contact and 3) update the address of one contact to the address of another contact. For each subject, we measured the time for each task, and the overall count of key strokes and mouse clicks.

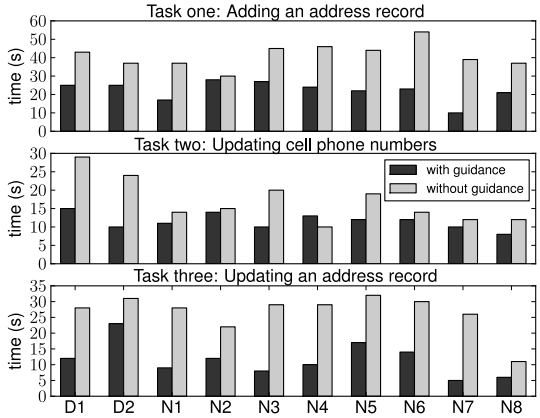


Figure 9: Time for data entry tasks, with guidance on and off

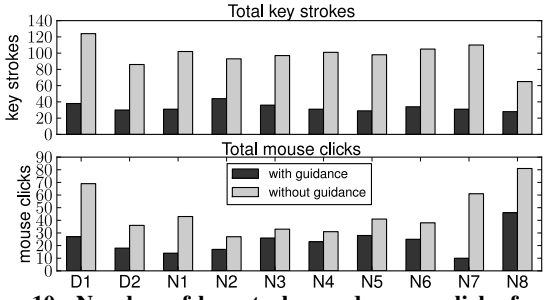


Figure 10: Number of key strokes and mouse clicks for data entry tasks, with and without guidance

All subjects finished all of the tasks. From the time shown in Figure 9, we observe a significant improvement with the guidance

[2]Since we reduced the database size to fit the user study, the query execution time is negligible compared to the user operation time.

[3]This is statistically significant with p-value < 0.0002.

feature on. [4] This demonstrates that CRIUS successfully leverages integrity constraints to save data entry time and improves usability.

We also recorded the number of key strokes and mouse clicks, since they may serve as strong evidence of input errors. The numbers are shown in Figure 10. Although detailed intermediate errors were hard to report, these numbers show that users made many fewer errors with integrity-based guidance. [5] In other words, this supports that CRIUS improves usability by preventing input errors.

5.3 Performance of IFDI

To evaluate the effectiveness of our incremental FD induction algorithm, we conducted experiments to compare the performance for both the naive approach (which recalculates the FDs for each update) and the IFDI algorithm, in a simulated incremental update environment. We measured the average time for both approaches to generate a new set of FDs upon simulated updates in simulated tables, by varying the number of columns and rows.

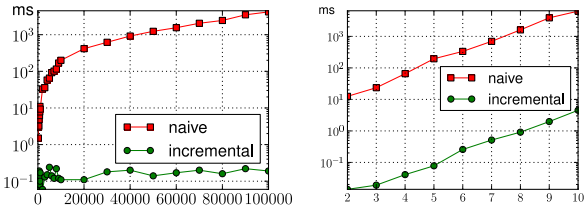


Figure 11: Average time generating new FDs using naive approach and IFDI, for (a) a five-column table with varying row size, and (b) a ten-thousand-row table with varying column size

We first fixed the number of columns to five and recorded the time at each row size. The result is shown in Figure 11(a). While the performance of the naive approach is linear in the number of rows, the cost for IFDI was extremely small and nearly constant. This is because for each lattice node, the traditional algorithm reconstructed each partition by doing a partition product, which involves a linear scan on the input partitions. In contrast, IFDI only updates existing partitions and touches rows with the same values as the updated tuple. The number of such rows has a very small upper bound, as shown in Appendix B.

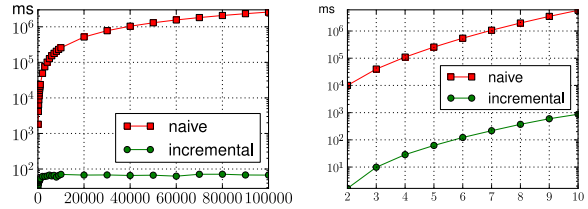


Figure 12: Average number of partition rows accessed when generating new FDs using naive approach and IFDI, for (a) a five-column table with varying row size, and (b) a ten-thousand-row table with varying column size

Our second experiment fixed the number of rows to ten thousand and measured the average time for various number of columns. The result is shown in Figure 11(b). While the costs for both approaches increase exponentially, the IFDI is more than three orders of magnitude faster. Again, the improvement in performance is due in large part to an tremendous decrease in partition rows accessed by the incremental algorithm, as shown in Figure 12. For example, for five columns and ten thousand rows, the incremental algorithm accessed an average of 42.9 partition rows for each update, while the naive approach accessed an average of 257196.9 partition rows.

[4]According to the Mann-Whitney test, the p-values are 0.0002, 0.0209 and 0.001 for the three tasks, respectively.

[5]Key strokes has a p-value of 0.0002 and mouse clicks has 0.0019.

5.4 Performance of Vertical Storage

Our last set of experiments compared the performance of the vertically-partitioned storage in CRIUS with a naive approach, which stores tables contiguously using a row-major layout. Recall that in designing the storage system we had two main goals: (1) efficient schema evolution, and (2) efficient data display. Our performance experiments measure each in turn.

5.4.1 Schema Update Performance

Our first experiment measured the time for the same schema update task in 5.1, on both a naive storage and the vertical partitioned storage in CRIUS. The result is shown in Figure 13. As expected, the vertically partitioned storage offered much faster schema modifications than the naive approach, particularly for large databases. This is because for each ALTER TABLE command, the naive method must restructure an entire relation, which greatly degraded performance. In contrast, CRIUS only manipulated the column tables involved in the move.

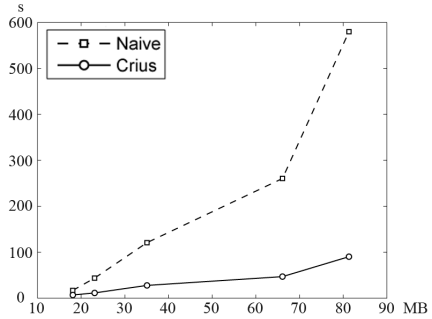


Figure 13: Average time transporting an attribute in CRIUS vs. naive storage, for different database scale

5.4.2 Data Display Performance

In addition to schema update, our vertically partitioned storage should also support other common tasks such as data display. To show this, our last experiment measured the time for loading data and constructing a span table from both the native storage and CRIUS storage. Figure 14 shows our results.

The time cost increased linearly with the number of attributes projected in CRIUS, but remained constant for the naive storage. This is because in CRIUS, the number of required joins grows linearly with the number of columns selected. At the same time, while data display queries are more efficient using the naive storage format, the difference between the naive and CRIUS is not huge, despite the additional joins required by CRIUS. This supports that CRIUS is able to improve the performance of schema modification tasks, without losing much efficiency on the data display task.

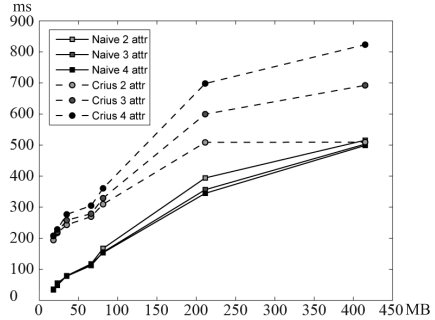


Figure 14: Average data display time in CRIUS vs. the naive storage, for different database scale

6. RELATED WORK

Database schema design has been studied extensively [1, 17]. There is a great deal of work on defining a good schema, both from

the perspective of capturing real-life requirements (e.g., normalization) and supporting efficient queries. However, schema design has typically been considered a heavyweight, one-time operation, which is done by a technically skilled database administrator, based on careful requirements analysis and planning. The new challenge of enabling non-expert user to "give birth" to a database schema was posed recently [10], but no solution was provided.

The focus of this paper has been the challenge of specifying and executing schema modifications. However, once a schema modification has taken place, other interesting challenges may arise, such as answering historical queries on an evolving schema [14].

The theoretical foundation of CRIUS is built on nested relations and their normal forms, which have been studied extensively [20, 16, 13, 2, 19, 18, 20, 25, 4, 21]. In particular, when mapping nested relation views to flat relational databases, our solution is inspired by [3], which treats the mapping as a simulation.

Polaris [23] is an interface for exploration of multidimensional databases that extends the Pivot Table interface to directly generate a rich and expressive set of graphical displays. This inspires our work of organizing a user-specific hierarchical data presentation. While their work focuses on displaying statistical data analysis in a user-friendly and flexible way on top of a static database, our work enables schema evolution in an intuitive manner as the underlying database is dynamically modified.

Recent work [9] proposed a model and architecture for seamlessly combining design-time and run-time aspects of data integration. The idea of such combination is reflected in a schema definition context in our work. However, they are dealing with data integration among multiple datasets, while we are focusing on integrating schema update and data manipulation inside a single database.

7. CONCLUSIONS

In this paper, we described the design and implementation of CRIUS, a system that allows non-technical users to develop and evolve schemas for their data, within the familiar context of a spreadsheet-style interface and data model. In support of the friendly drag-and-drop user interface, we developed a novel span table algebra that is equivalent in expressive power to the nested algebra.

User operations on a spreadsheet are usually error-prone. Integrity constraints, such as functional dependencies can save user input and protect the data from mistakes. However, incremental FD induction is non-trivial when the database is modified frequently. In this paper, we introduced the first algorithm for incrementally updating the set of induced FDs upon value update and schema evolution. CRIUS uses these FDs to recommend auto-completions for updates and to warn the user about potential data entry errors.

While the data model exposed to CRIUS users is a nested span table, the underlying storage model may be different. Because of its pervasiveness, we elected to store our span tables as flat relations in a relational database system. While past work has considered storing nested tables in flat relations, the mapping is only treated as a simulation to establish theoretical results. In contrast, CRIUS requires materialization of such a mapping to support efficient query processing and efficient schema evolution. For these reasons, we implemented a recursively vertically decomposed storage format. We also show how span table algebra operators can be mapped to SQL operations on the relational database.

Finally, we evaluated the CRIUS implementation via both user studies and performance experiments. The user study showed that the drag-and-drop schema modification meets our primary goal of making schema evolution easy to specify, and our integrity-based guidance feature effectively reduces data entry effort and input errors. The performance results indicated that the our incremental FD induction algorithm is much faster compared to the traditional approaches, and the vertical decomposition storage format is considerably more efficient than past techniques for schema evolution, while query processing performance remains reasonable.

8. REFERENCES

[1] J. Biskup. Achievements of relational database schema design theory revisited. In *Semantics in Databases*, pages 29–54, 1998.

[2] L. S. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD*, pages 567–582, 1989.

[3] J. V. den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1-2):363–377, 2001.

[4] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *TODS*, 2(3):262–278, 1977.

[5] D. Florescu, R. INRIA, and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, page 27, 1999.

[6] G. Graefe. Efficient columnar storage in b-trees. *SIGMOD*, 2007.

[7] C. Hara and S. Davidson. Reasoning about nested functional dependencies. In *PODS*, pages 91–100, 1999.

[8] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100, 1999.

[9] Z. G. Ives, C. A. Knoblock, S. Minton, M. Jacob, P. P. Talukdar, R. Tuchinda, J. Ambite, M. Muslea, and C. Gazen. Interactive data integration through smart copy and paste. In *CIDR*, 2009.

[10] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.

[11] M. Jayapandian, A. Chapman, V. Tarcea, C. Yu, A. Elkiss, A. Ianni, B. Liu, A. Nandi, C. Santos, P. Andrews, et al. Michigan molecular interactions (mimi): putting the jigsaw puzzle together. *Nucleic acids research*, 35:D566, 2007.

[12] S. Lopes, P. Jean-Marc, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. *EDBT*, pages 350–364, 2000.

[13] W. Y. Mok, Y.-K. Ng, and D. W. Embley. A normal form for precisely characterizing redundancy in nested relations. *TODS*, 21(1):77–106, 1996.

[14] H. J. Moon, C. A. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and querying transaction-time databases under schema evolution. In *PVLDB*, volume 1, pages 882–895, 2008.

[15] N. Novelli and R. Cicchetti. Functional and embedded dependency inference: a data mining point of view. *Information Systems*, 26(7):477–506, 2001.

[16] Z. M. Ozsoyoglu and L. yan Yuan. A new normal form for nested relations. *TODS*, 12:111–136, 1987.

[17] E. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.

[18] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *TODS*, 17(1):65–93, 1992.

[19] B. Rathakrishnan and J. L. Kim. An extended recursive algebra for nested relations and itsoptimization. In *COMPSAC*, page 145, 1993.

[20] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *TODS*, 13(4):389–417, 1988.

[21] H. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[22] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, page 314, 1999.

[23] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query,analysis and visualization of multi-dimensional relational databases. *IEEE Trans. Vis. Comput. Graphics*, 8(1):52–65, 2002.

[24] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[25] D. Van Gucht and P. Fischer. Multilevel nested relational structures. *Journal of Computer and System Sciences*, 36(1):77–105, 1988.

[26] C. Yu and H. Jagadish. Efficient discovery of xml data redundancies. In *VLDB*, pages 103–114, 2006.

APPENDIX

A. SPAN TABLE ALGEBRA

In this section, we define the operators in the span table algebra. An informal description is in Table 2. The span table algebra differs from the nested relational algebra in the following aspects. First, each span table operator is naturally and directly supported by the UI, while the nested algebra is hard to be implemented using direct manipulation. Second, the nested algebra is designed mainly for structural query, while the span table algebra aims to perform schema evolution. For instance, columns can be added/dropped in the span table algebra, but not in the nested algebra. Finally, data is static under the nested algebra, but can be updated using span table operators. Such update is non-trivial since it may change the table structure and affect further schema evolution.

We also provide the proofs for the lemmas and the theory mentioned in Section 2.2.

Operators	Description
Import(A)	Move A inward into a descendant-relation.
Export(A)	Move A outward into an ancestor-relation.
Sink(A)	Push A to create a new leaf relation.
Float(A)	Lift A to create a new intermediate relation.
Add/Drop(A)	Add/drop attribute A.
Insert/Remove/Update(T)	Insert/remove/update a tuple T.

Table 2: Span Table Operators

A.1 Basics

We will assume the reader is familiar with the basic concepts of nested relational algebra (e.g., nest and unnest) [2, 20, 25, 21]. In CRIUS, we chose to adopt partitioned normal form (PNF) [20], which asserts functional dependencies from the set of all the atomic attributes at each schema level, since it is consistent with our need to preserve data integrity. We also adopt the schema tree described in [25]. To summarize, each node in the schema tree represents either an atomic attribute or a relation-valued attribute in the corresponding nested relation. An edge from one node to another indicates that the parent node contains the attribute represented by the child node. For example, the schema updating process in Figures 4(a) and 4(b) can be expressed by a schema tree evolution shown in Figure 15. We also define the *schema level* of a certain relation to be the depth of its corresponding node in the schema tree, with zero for the root.

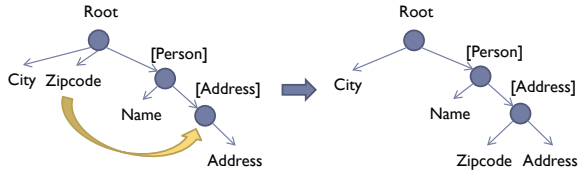


Figure 15: Address Book Schema Evolution

A.2 Schema Update Operators

The span table algebra has four schema restructuring operators: IMPORT, EXPORT, SINK, and FLOAT. Each operator not only updates the schema but also maps existing data to the updated schema.

- **IMPORT:** The basic IMPORT, as defined in Section 2.2, can only take place at the root of a schema tree and can only push attributes down to a schema level that is exactly one level deeper. These limitations must be lifted if we are to support drag-and-drop driven organic schema modification. In the following, we upgrade the basic IMPORT so that it is able to reach an arbitrarily deep schema level starting from anywhere in the schema tree. We denote the parent of a given attribute Q by $Parent(Q)$. $Parent(Root)$ is null. We denote the ancestor and descendent of Q by $Ancestor(Q)$ and $Descendent(Q)$, respectively. Both $Ancestor(Root)$ and $Descendent(Leaf)$ are nulls. If $T \in Descendent(L)$, we define $Path(L,T)$ to be the ordered list of schema nodes from L to T, excluding L and including T. Then, the full IMPORT can be defined as below.

  DEFINITION 3 (IMPORT OPERATOR). *Assume N is a nested relation with schema tree $Tree(N)$. Let $L = \{\overline{A}, \overline{X}, P\}$ and T be the schema nodes in $Tree(N)$ such that $T \in Descendent(L)$ and T is relation-valued, then $IMPORT_{P|L,T}(N) = N'$, where $S(N')$ differs from $S(N)$ by removing P from L and inserting it to T. Also, N' is the result relation of the following operations: for each Q on $Path(L,T)$ following the order, on each sub relation with schema $Parent(Q)$ in N, execute $IMPORT_{P,Q}$.*

  In practice, by observing some transitive features of the upgraded IMPORT sequence, we can avoid repeated copying and merging, and thereby implement it cheaply compared to a literal implementation of Definition 3.

- **EXPORT:** The basic EXPORT has been defined in Section 2.2. Now we extend it to its full version similar to IMPORT. Due to space, we omit the full definition and illustrate IMPORT and EXPORT by the following example.

  EXAMPLE 5. *Consider an address book in Figure 16(a). The user may call EXPORT on* State[6] *to augment categorizing information by state name. The resulting relation is pictured in Figure 16(b), with the value "Keith" duplicated. Finally, the user may decide to categorize data only by State. So she calls IMPORT on* Name[7]*, resulting in the final address book in Figure 16(c), with the two "*MI*"s merged.*

Name	[Address]	
	State	Address
Orlando	MI	2251 Elliot Avenue
Keith	MI	3207 S Grady Way
	OH	7943 Walnut Ave

(a) Categorized by Name

Name	State	[Address]
		Address
Orlando	MI	2251 Elliot Avenue
Keith	MI	3207 S Grady Way
Keith	OH	7943 Walnut Ave

(b) Categorized by Name and State

State	[Address]	
	Name	Address
MI	Orlando	2251 Elliot Avenue
	Keith	3207 S Grady Way
OH	Keith	7943 Walnut Ave

(c) Categorized by State

Figure 16: Categorizing an Address Book

- **SINK:** IMPORT and EXPORT serve to transport atomic attributes across different levels of the schema tree. However, they do not create new schema levels, as is normally done by NEST in traditional nested algebra. A traditional NEST operator combines multiple attributes into a nested relation. This is difficult to do in a single drag-and-drop. A multi-step operation, with selecting attributes first and then nesting and moving them, is much less user-friendly. To solve this problem, we restrict the NEST operator to a single attribute, and name it SINK to distinguish it from NEST. SINK is also defined from the basic version.

  DEFINITION 4 (BASIC SINK OPERATOR). *Given a nested relation N with $S(N) = \{\overline{A}\overline{X}P\}$, where $\overline{A}$ and $\overline{X}$ have the same meaning as before, and P denotes the atomic attribute to be nested, $SINK_P(N) = N'$, where $S(N') = \{\overline{A}\overline{X}\{P\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:*

  (1) $t'[\overline{A}] = t[\overline{A}]$

[6] $EXPORT_{State|\{State,Address\},Name}$

[7] $IMPORT_{Name|\{Name,State,\{Address\}\},\{Address\}}$

(2) $t'[\overline{X}] = \bigsqcup\{t''[X] | t'' \in N \wedge t''[\overline{A}] = t[\overline{A}]\}$

(3) $t'[\{P\}] = \{t''[P] | t'' \in N \wedge t''[\overline{A}] = t[\overline{A}]\}$

DEFINITION 5 (SINK OPERATOR). *Assume N is a nested relation with schema tree $Tree(N)$. Let $L = \{\overline{A}, \overline{X}, P\}$ be a certain schema node in $Tree(N)$, then $SINK_{P|L}(N) = N'$, where $S(N')$ differs from $S(N)$ by replacing L with $\{\overline{A}, \overline{X}, \{P\}\}$. Also, N' is the result relation after executing $SINK_P$ on all sub relations with schema L within N.*

- **FLOAT:** Theoretically, IMPORT, EXPORT and SINK form an orthogonal and complete (with respect to NEST and UNNEST in nested relational algebra) set of operators. However, in some cases, the users may wish to create a new schema level by sinking most of the attributes on the current level. For the sake of usability, we propose another operator, namely FLOAT, to complement SINK. FLOAT relatively lifts an atomic attribute up by sinking all its siblings by one level.

  DEFINITION 6 (BASIC FLOAT OPERATOR). *Given a nested relation N with $S(N) = \{P\overline{Q}\}$, where $\overline{Q}$ is a set of attributes, and P denotes the atomic attribute to be floated, $FLOAT_P(N) = N'$, where $S(N') = \{P\{\overline{Q}\}\}$ and N' is the set of all t' for which there exists $t \in N$, such that:*

  (1) $t'[P] = t[P]$

  (2) $t'[\{\overline{Q}\}] = \{t''[\overline{Q}] | t'' \in N \wedge t''[P] = t[P]\}$

  DEFINITION 7 (FLOAT OPERATOR). *Assume N is a nested relation with schema tree $Tree(N)$. Let $L = \{\overline{A}, \overline{X}, P\}$ be a certain schema node in $Tree(N)$, then $FLOAT_{P|L}(N) = N'$, where $S(N')$ differs from $S(N)$ by replacing L with $\{P, \overline{X}, \{\overline{A}\}\}$. Also, N' is the result relation after executing $FLOAT_P$ on all sub relations with schema L within N.*

  Note that FLOAT and SINK can be applied at any nesting level but move the affected attribute by only one level. We do not extend to multi-level as we did for IMPORT and EXPORT because multi-level FLOAT/SINK can result in an intermediate sub relation with no atomic child, which violates PNF.

In addition to the four schema restructuring operators, the algebra also includes operators to augment/diminish schema, such as adding/dropping/permuting columns. All these operators are restricted to a single schema level and behave identically to their flat versions. In the interest of space, we skip their formal definitions.

A.3 Data Manipulation Operators

Besides the schema update operators, three data manipulation operators, namely INSERT/DELETE/UPDATE, are also provided with the algebra. These are consistent with their traditional semantics except that i) all of them are extended to the nested scenario so that manipulating data at arbitrary schema level becomes feasible, and ii) insertion and deletion trivially guarantee foreign key constraints in a cascading manner.

A.4 Proofs

LEMMA 3. *Let N be a nested relation with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, then $IMPORT_{P,\{\overline{Q}\}}(N) = NEST_{P,\{\overline{Q}\}} \cdot UNNEST_{\{\overline{Q}\}}(N)$.*

Proof of Lemma 3: Let N be a nested relation with $S(N) = \{\overline{AX}P\{\overline{Q}\}\}$, $N' = UNNEST_{\{\overline{Q}\}}(N)$ and $N'' = NEST_{P,\{\overline{Q}\}}(N')$. According to definition,

$$UNNEST_{\{\overline{Q}\}}(N) = \{t' | \exists t_{t'} \in N \; s.t. \; t'[\overline{A}] = t_{t'}[\overline{A}] \wedge t'[P] = t_{t'}[P] \wedge t'[\overline{X}] = t_{t'}[\overline{X}] \wedge t'[\overline{Q}] \in t_{t'}[\{\overline{Q}\}]\} \quad (1)$$

where $t_{t'}$ is the tuple in N with the same values at $\overline{A} \cup P$ as t' (and is thus unique).

$$NEST_{P,\{\overline{Q}\}}(N') = \{t'' | \exists t' \in N' \; s.t. \; t''[\overline{A}] = t'[\overline{A}] \wedge t''[\overline{X}] = \bigsqcup\{t'[\overline{X}] | t'[\overline{A}] = t''[\overline{A}]\} \wedge t''[\{P, \overline{Q}\}] = \{t'[P, \overline{Q}] | t'[\overline{A}] = t''[\overline{A}]\}\} \quad (2)$$

Since for each $t' \in N'$, we have a $t_{t'} \in N$, $\exists t' \in N'$ always implies $\exists t_{t'} \in N$. Furthermore, according to the first line in equation (1), $t'[\overline{A}] = t_{t'}[\overline{A}]$. Thus, the first line in equation (2) can be translated to $\exists t_{t'} \in N \; s.t. \; t''[\overline{A}] = t_{t'}[\overline{A}]$. Moreover, we define $E_{\overline{V},W}$ to be the set of t' which map to the same $t_{t'}$, with $t'[\overline{A}] = \overline{V} \wedge t'[P] = W$. Then, $\{t' | t'[\overline{A}] = t''[\overline{A}]\}$ can be classified into a group of $E_{t''[\overline{A}],W}$, each of which corresponds to a $t_{t'} \in N$ with $t_{t'}[P] = W$. We denote this group by $R_{t''[\overline{A}]}$. Now we can rewrite $\bigsqcup\{t'[\overline{X}] | t'[\overline{A}] = t''[\overline{A}]\}$ as $\bigsqcup_{R_{t''[\overline{A}]}}\{\bigsqcup\{t'[\overline{X}] | t' \in E_{t''[\overline{A}],t''[P]}\}\}$. According to (1), $t'[\overline{X}]$ are the same as long as their values at $\overline{A}$ and P are the same, which is the same as that of their corresponding $t_{t'}$. Thus, $\bigsqcup\{t'[\overline{X}] | t' \in E_{t''[\overline{A}],t''[P]}\} = t_{t'}[\overline{X}]$. So the second line in (2) can be translated to $t''[\overline{X}] = \bigsqcup_{R_{t''[\overline{A}]}}\{t_{t'}[\overline{X}]\} = \bigsqcup\{t_{t'}[\overline{X}] | t_{t'}[\overline{A}] = t''[\overline{A}]\}$. Similarly, $\{t'[P, \overline{Q}] | t'[\overline{A}] = t''[\overline{A}]\} = \bigcup_{R_{t''[\overline{A}]}}\{t'[P, \overline{Q}] | t' \in E_{t''[\overline{A}],t''[P]}\}$. According to (1), $t'[P] = t_{t'}[P]$ and $t'[\overline{Q}] \in t_{t'}[\{\overline{Q}\}]$ for $t_{t'}$ corresponding to each $E_{t''[\overline{A}],t''[P]}$. So, $\bigcup_{R_{t''[\overline{A}]}}\{t'[P, \overline{Q}] | t' \in E_{t''[\overline{A}],t''[P]}\} = \bigcup_{R_{t''[\overline{A}]}}\{t'[P, \overline{Q}] | t'[P] = t_{t'}[P] \wedge t'[\overline{Q}] \in t_{t'}[\{\overline{Q}\}]\}$. In other words, the third line in (3) can be translated to $t''[\{P, \overline{Q}\}] = \{t' | \exists t''' \in N, \; s.t. \; t'[P] = t'''[P] \wedge t'[\overline{Q}] \in t'''[\{\overline{Q}\}] \wedge t'[\overline{A}] = t'''[\overline{A}]\}$. In all, by substituting (1) into (2), we obtain the following equation:

$$\begin{aligned} &NEST_{P,\{\overline{Q}\}} \cdot UNNEST_{\{\overline{Q}\}}(N) \\ &= \{t'' | \exists t_{t'} \in N \; s.t. \; t''[\overline{A}] = t_{t'}[\overline{A}] \\ &\wedge t''[\overline{X}] = \bigsqcup\{t_{t'}[\overline{X}] | t_{t'}[\overline{A}] = t''[\overline{A}]\} \\ &\wedge t''[\{P, \overline{Q}\}] = \{t' | \exists t''' \in N, \; s.t. \; t'[P] = t'''[P] \\ &\wedge t'[\overline{Q}] \in t'''[\{\overline{Q}\}] \wedge t'[\overline{A}] = t'''[\overline{A}]\}\} \end{aligned} \quad (3)$$

Which is identical to the definition of IMPORT.

LEMMA 4. *Let N be a nested relation with $S(N) = \{\overline{AX}\{P, \overline{Q}\}\}$, then $EXPORT_P(N) = NEST_{\{\overline{Q}\}} \cdot UNNEST_{P,\{\overline{Q}\}}(N)$.*

The proof is similar to the proof of Lemma 3.

THEOREM 2. *Any NEST or UNNEST can be expressed as a sequence of span-algebra schema update operations.*

PROOF. Let N be a nested relation with schema $S(N) = \{\overline{Q}, \{A_1, A_2, ..., A_n\}\}$, and denote $G_i = A_i, A_{i+1}, ...A_n$, then according to Lemma 4:

$$\begin{aligned} &UNNEST_{\{A_1,A_2,...,A_n\}}(N) = UNNEST_{G_n} \cdot \\ &NEST_{G_n} \cdot UNNEST_{G_{n-1}} \cdot ... \cdot NESTG_3 \cdot \\ &UNNEST_{G_2} \cdot NESTG_2 \cdot UNNEST_{G_1}(N) \\ &= EXPORT_{A_n} \cdot EXPORT_{A_{n-1}} \cdot ... \\ &\cdot EXPORT_{A_2} \cdot EXPORT_{A_1}(N) \end{aligned} \quad (4)$$

Similarly, for NEST, suppose the nested relation being operated on is N with schema $S(N) = \{\overline{Q}, A_1, A_2, ..., A_n\}$, using the same notation for G, we have:

$$\begin{aligned} &NEST_{A_1,A_2,...,A_n}(N) = \\ &NESTG_1 \cdot UNNEST_{G_2} \cdot ... \cdot NEST_{G_{n-2}} \cdot \\ &UNNEST_{G_{n-1}} \cdot NESTG_{n-1} \cdot UNNEST_{G_n}(N) \cdot SINK_{G_n}(N) \\ &= IMPORT_{A_1} \cdot IMPORT_{A_2} \cdot ... \\ &\cdot IMPORT_{A_{n-2}} \cdot IMPORT_{A_{n-1}}(N) \cdot SINK_{A_n}(N) \end{aligned} \quad (5)$$

□

B. THE IFDI ALGORITHM

Here we describe in detail our algorithm for incremental FD induction (IFDI). The algorithm consists of two phases: the initialization phase (IFDI-Initialize) which generates the original set of minimal FDs and the update phase (IFDI-Update) which updates the FDs incrementally. For incremental induction purposes, we maintain two data structures generated by the initialization process: the $initialPartitions$ which records the unstripped partitions for each attribute with data values as group keys, and P which stores all the stripped partitions. After a cell is updated to a different value, we pass the attribute associated with the cell, the tuple id of the cell and the new value to IFDI-Update. The algorithm first updates

the initial partition of the modified attribute by invoking Update-Initial-Partition(line 1). It then traverses the sub-lattice formed by nodes whose labels contain the modified attribute, and checks the validity of FDs represented by the edges within the sub lattice by checking if the attribute partitions of the LHS and RHS of the FD are identical. In practice, this is done by introducing an *error rate* which is defined to be the total size of the partition groups minus the total number of partition groups. For example, the error rate of Π_{Name}, denoted as $P(Name).e$, is $5-3=2$. Stripping has no effect on error rates. And $P(X \setminus \{A\}) = P(X)$ iff $P(X \setminus \{A\}).e = P(X).e$ (line 6). Before the algorithm moves forward to the next level, it computes the partitions in the next level by invoking Update-Partition(line 15). The traditional pruning technique for non-minimal FDs still works(line 5-11).

We analyze the time complexity of IFDI on top of the following general assumption. The table has M columns and N rows, data from each column uniformly distributes over a value pool of size R, and different columns are independent and identically distributed. For each update, traditional FD induction algorithm constructs each partition by a partition product, whose cost is proportional to N. Since in the worst case, the traditional algorithm traverses the whole lattice and perform such construction for each node, it has a worst-case time complexity of $O(N2^M)$. For the IFDI algorithm, we reconstruct each partition by executing Update-Partition. Since we maintain indexes on partitions, to remove the modified tuple from the old group costs constant time (line 1-8). The time complexity of Update-Partition is thus dominated by the for loop which seeks another tuple that has the same values as the modified tuple at the columns associated with that partition (line 10-15). The cost of this loop is upper-bounded by the size of g_x (line 10) and estimated by the expected number of executions of the loop body. On one hand, under our assumption, the expectation of the size of g_x decreases geometrically with respect to its depth in the lattice. Thus, the overall cost is upper-bounded by the summation of such expectations over the lattice, which is $kN((1+1/R)^M - 1)$ (where k is the constant time for one execution of the loop body). On the other hand, the loop terminates immediately after a match (line 14-15), which means the cost estimates to the expected number of tries before the match is found. Under our assumption, these numbers conform to a geometric distribution, which has an expectation of R. Since IFDI traverses half of the lattice, the average-case complexity is $kR2^{M-1}$. To summarize, the time-complexity of IFDI cam be estimated by $kR2^{M-1}$ when the value pool is small ($R << N$), and upper-bounded by $kN((1+1/R)^M - 1) \approx kMN/R$ when the value pool is large ($R >> M$). This explains why the IFDI algorithm is much faster than the traditional algorithm and the cost for the former is extremely small and nearly constant.

```
IFDI-INITIALIZE()
 1  {R, n} ← RETRIEVE-METADATA()
 2  initialPartitions ← INITIALIZE-PARTITIONS()
 3  strippedPartitions ← STRIP-PARTITIONS()
 4  L_0 ← {{∅}}
 5  C({∅}) ← R
 6  L_1 ← {{A}|A ∈ R}
 7  for each A ∈ R
 8     do P({A}) ← strippedPartitions(A)
 9  for i ← 1 to n
10     do for each X ∈ L_i
11           do C(X) ← ⋂_{A∈X} C(X \ {A})
12              for each A ∈ X ∩ C(X)
13                 do if P(X \ {A}).e = P(X).e
14                       then FD ← FD ∪ (X \ {A} → A)
15                            C(X) ← C(X) \ {A}
16                            C(X) ← C(X) ∪ X
17        if i < n
18          then L_{i+1} ← {X|X \ {A} ∈ L_i, A ∈ X}
19        for each X ∈ L_{i+1}
20           do Let A be an arbitrary element in X
21              P(X) ← P({A}) · P(X \ {A})
```

```
IFDI-UPDATE(M, id, value)
 1  UPDATE-INITIAL-PARTITION(M, id, value)
 2  L_1 ← {{M}}
 3  for i ← 1 to n
 4     do for each X ∈ L_i
 5           do C(X) ← ⋂_{A∈X} C(X \ {A})
 6              for each A ∈ X ∩ C(X)
 7                 do if P(X \ {A}).e = P(X).e
 8                       then FD ← FD ∪ (X \ {A} → A)
 9                            C(X) ← C(X) \ {A}
10                            C(X) ← C(X) ∪ X
11                       else FD ← FD \ (X \ {A} → A)
12        if i < n
13          then L_{i+1} ← {X|X \ {A} ∈ L_i, A ∈ X}
14        for each X ∈ L_{i+1}
15           do UPDATE-PARTITION(initialPartitions(M),
16              P(X \ {M}), P(X), id, value)

UPDATE-INITIAL-PARTITION(A, id, value)
 1  p_ai ← initialPartitions(A)
 2  value_old = p_ai.T(id).value
 3  g ← p_ai(value_old)
 4  g ← g \ id
 5  if g.isEmpty()
 6    then p_ai ← p_ai \ g
 7    else p_ai.e ← p_ai.e − 1
 8  g ← p_ai(value)
 9  if g ≠ NIL
10    then g ← g ∪ id
11         p_ai.e ← p_ai.e + 1
12    else g ← id
13         p_ai ← p_ai ∪ g
14  p_ai.T.remove(id)
15  p_ai.T.put(id, g)
16  p_a = p_ai.strip()
17  P(A) ← p_a;

UPDATE-PARTITION(p_a, p_x, p_ax, id, value)
 1  g_ax ← p_ax.T(id)
 2  if g_ax ≠ NIL
 3    then
 4         p_ax.e ← p_ax.e − 1
 5         if g_ax.size = 2
 6           then p_ax ← p_ax \ g_ax
 7           else g_ax ← g_ax \ id
 8  g_x ← p_x.T(id)
 9  rid ← NIL
10  for each tid ∈ g_x
11     do if tid = id
12           then continue
13        if p_a.T(tid).value = value
14          then rid ← tid
15               break
16  if rid ≠ NIL
17    then p_ax.e ← p_ax.e + 1
18         if p_ax.T(rid) = NIL
19           then g ← rid ∪ id
20                p_ax ← p_ax ∪ g
21                p_ax.T.put(rid, g)
22                p_ax.T.put(id, g)
23           else g_ax ← p_ax.T(rid)
24                g_ax ← g_ax ∪ id
```

C. DOWNWARD MAPPING

We describe the downward mapping for IMPORT with a few key points here. Other mappings are similar.

C.1 Mapping IMPORT

A multi-level IMPORT can be evaluated in two stages: (1) transporting source column with value naively duplicated and (2) the merge operation, which corresponds to $\bigsqcup$ operator in PNF. The first stage is making the schema change, and the second stage is adjusting the data to this changed schema.

Let $Source$ denote the source attribute, and $Target$ denote the target relation. According to the tree structure, we can obtain a unique ordered path from $P(Source)$ to $Target$. Let the relations on this path, excluding $P(Source)$, be $\{R_1, R_2 ... R_n\}$. Also, for any attribute A, denote its corresponding flat table with $Flat(A)$. Suppose $S(Flat(Source)) = \{ID, A_S\}$. Then, the first stage can

be formalized by:

$$Flat(Source) \leftarrow \Pi_{ID,A_S}(\rho_Y(Flat(Source)) \bowtie_{Y.ID=Z.PID} \rho_{Z(PID,ID)}(\Pi_{L_i.PID,L_n.CID} (\bowtie_{L_i.CID=L_{i+1}.PID} \{L_i = Flat(R_i), L_{i+1} = Flat(R_{i+1}) : i = 1..n-1\}))) \quad (6)$$

This formula finds the tuple relationship by joining all the linking tables from source parent to target. It then joins the result with the original source column, which is a value table, achieving the goal for replicating A_S according to tuple ID correspondence.

Also, the structure table has to be updated. However, the update is extremely simple in that we only need to update the parent relation of the source attribute to the target relation, and do the obvious bookkeeping on the auxiliary information.

For the second stage, a merge is applied recursively, on each relation merging all tuples whose set of atomic attributes at this schema level are the same. This involves three steps: updating atomic values, updating the parent link table, and updating the link tables for all relation-valued children.

We first define the ungrouped table, which is a join of parent link table and all atomic value tables at current level. Suppose the set of atomic attributes in current level are $\{R_1, R_2, ...R_n\}$, with $S(Flat(R_i)) = \{ID, A_i\}$. Also suppose the parent relation of current level is $Parent$, then:

$$ungrouped(Parent) \leftarrow \Pi_{PID,CID,A_1,A_2,...,A_n}(\rho_Y Flat(Parent) (\bowtie_{Y.CID=V_i.ID} \{V_i = Flat(R_i)\})) \quad (7)$$

After merge, the corresponding table would be:

$$grouped(Parent) \leftarrow ungrouped(Parent)\ group\ by\ \{PID, A_1, A_2, ..., A_n\} \quad (8)$$

If the size of ungrouped and grouped table is the same, than there is no need to merge. Otherwise:

$$Flat(R_i) \leftarrow \Pi_{CID,A_i}(grouped(Parent)) \quad (9)$$

For $i = 1..n$, and:

$$Flat(Parent) \leftarrow \Pi_{PID,CID}(grouped(Parent)) \quad (10)$$

The link tables of child relation-valued attributes are updated in a similar manner. The merge stage is done by following this procedure recursively. The recursion will never exceed the target relation since schema elements below that relation are untouched.

In general, the whole structure of the flat database needs very few changes, since most of the structural updating is reflected by the structure table. Further, no "ALTER TABLE" command is required, which avoids the high cost of traditional schema updates.

C.2 Mapping other operators

The EXPORT works quite similarly to IMPORT in terms of specifying tuple relationship by joining linking tables, except no merge is required. FLOAT and SINK also work in a similar manner, with much simpler procedures required.

For other schema update operators, Adding/Dropping columns requires only creating/dropping flat tables corresponding to the attributes and insertion/deletion of tuples in the structure table. Permutation is trivially exchanging attribute IDs in the structure table.

Data manipulation operations are much simpler. INSERT and DELETE update the corresponding value tables and link tables. UPDATE is similar to INSERT except an additional merge is required for preserving PNF.

D. EXPERIMENT SPECIFICATIONS

D.1 User Study on Schema Operations

For the schema design task, we asked the users to define three relations: 1) a Person relation with two attributes: FirstName and LastName, 2) an Email relation with one attribute Email. and 3) a Phone relation with one attribute Phone. We also required the users to structure the database hierarchically so that each person may have multiple emails and phones. We taught the users how to do this in both CRIUS and SSMS. In CRIUS, we taught them to use the Span Table operators to create the structure. In SSMS, we taught them to specify foreign key references from both Email and Phone to Person, by creating ID columns and dragging links between them to indicate foreign key references using the database diagram UI in SSMS.

For the schema modification task, we used a more complex schema from MiMI [11]. We focused on two relations: the Gene relation and the Interaction relation. Gene records individual gene information. It consists of five attributes: gene_id, symbol, type, taxid, and description. Interaction stores the basic information describing how two genes which interact with each other, including nine attributes: int_id, gid1, symbol1, type1, taxid1, gid2, symbol2, type2, and taxid2. gid1 and gid2 are foreign keys referencing gene_id. We nested Interaction inside Gene by gene_id to bring them into a single spreadsheet. We asked the users to move one attribute (Description) from Gene to Interaction, which was a practical need from current MiMI users.

For both tasks, we equally alternated the order of which system is used first, in order to counterbalance the learning effect.

D.2 User Study on Integrity-Based Guidance

For this user study, we designed an address book with schema $Name, \{Address, Zipcode, HomePhone, CellPhone\}$. It contained three contacts and six addresses, and induced these FDs: $Name \rightarrow CellPhone$, $HomePhone \rightarrow Zipcode$, $Address \rightarrow HomePhone$, $HomePhone \rightarrow Address$ and $Address \rightarrow Zipcode$. A brief tutorial on how to use CRIUS was given to each subject prior to the study. We started timing after the subject was clear about each task and ready to execute it. In order to counterbalance the learning effect, the guidance feature was turned on first for half of the studies, and turned off first for the other half.

D.3 Performance of IFDI

For both experiments, we simulated the table to have one near-FD (50% of the rows satisfied the FD) from the first column to the second. All the data (except in the determined column) was randomly generated from a value pool of size ten. Each cell update changed an FD-violating row to a FD-satisfying row by updating its second column. We repeated the experiment by varying the number of rows and columns. Time in both tests were averaged upon ten tables and one hundred updates for each table.

D.4 Performance of Vertical Storage

Both performance tasks used data from MiMI, as described in Appendix D.1. For the schema update experiment, we set up two versions of the MiMI: The first (Naive) stores the two relations just as they are stored in MiMi, with gene_id and int_id as the primary keys. The second (CRIUS) partitions the two relations in a per-column manner similar to that depicted in Figure 6, with a primary key ID column associated with each column table. In both cases, only a clustered index on the primary key is constructed for each relation. Our experiment repeatedly moved the description attribute between the Gene and Interaction relations in each database.

The data display experiment is designed to verify that the vertical partitioned storage still offers reasonable performance for other common tasks in this environment. Specifically, we focused on a query to retrieve the gene_id, symbol, type, and taxid of all the genes that interact with a given gene whose symbol matches a random pattern. We executed this query on both the naive and CRIUS databases.