Schema and Tuple Trees: An Intuitive Structure for Representing Relational Data

Appears in *Computing Systems*, Volume 9, Number 2

Eric H. Herrin, II Raphael A. Finkel

University of Kentucky

Abstract

Qddb is a publicly available database suite designed for applications in which the data is a set of records, each of contains hierarchical structure. For example, a database of patients contains a record for each patient; each patient record has multiple copies of visit substructures. Records containing such nested and replicated attributes are equivalent to the join of traditional relational tables. Qddb records therefore allow the data to be recorded in a more natural fashion than relational tables.

The presentation of data in Qddb is unusual but intuitive; the user usually views a subset of a full relational row at any given time.

This paper presents *schema* and *tuple trees*, the underlying structures of a Qddb database. Instead of a set of full relational rows representing the *join* of several tables, the *tuple tree* represents the tables in a compressed form. Related data are stored and displayed together, which allows the application designer to build an application in a relatively small amount of time. The algorithms for search and presentation are quite efficient.

1 Introduction

Conventional relational database implementations can be cumbersome to use and program. A single database often requires many tables of interrelated information. The relationship between the tables is well-defined but can be difficult to translate into a usable application. Allowing an arbitrary number of values in a field generally requires that the field be linked to a separate table built for just that purpose. The links between tables are often unique identifiers that the user must invent and that the application must check for uniqueness and consistency across tables. Each extra table therefore places a burden on the designer, the database programmer, and the end user.

To avoid a proliferation of tables, designers tend to limit the number of values in a field (such as the number of addresses an individual may have). The result can be frustrating to users who need to deal with exceptional cases that do not fit these limits.

This paper shows how Qddb [Herrin, II & Finkel 1991] alleviates most of the tedious work involved in designing relational tables that have fields that may have an arbitrary number of instances. The Qddb technique has other surprising and helpful characteristics.

We begin with a small example to show how a schema tree can be used to define a database. Next, we discuss the relationship between schema trees and tuple trees. Finally, we concentrate on tuple trees and their characteristics, including intuitive presentation forms and algorithms for generating those presentations.

2 A motivating example

A Qddb schema defines the layout of data tuples in the database. The schema is in the form of a tree that we call the *schema tree*. Each individual tuple in the relation is represented in a corresponding *tuple tree*. Each branch of the schema tree may be associated with multiple copies in any given tuple tree. Each duplicate is called an *instance*. The leaves of the tuple tree hold the actual data.

To demonstrate these concepts, consider the following three tables in a traditional relational database:

Client				
Client Id	Name	Address	Home Phone	Work Phone
1	George Goodan	123 Hardy Rd	277-1234	278-5432
2	Joan Goodan	323 Hardy Rd	277-1234	278-5432
3	Henry Zellerman	378 Jaclynne Rd	262-6432	268-9678

Invoice			
Client Id	Invoice Id	Item Id	Price
2	1	1232	$1.20
2	1	1233	$2.40
2	1	1121	$7.30
3	2	3214	$100.00

Invoice Total	
Invoice Id	Invoice Total
1	$10.90
2	$100.00

In traditional relational databases, each table has its own schema. Each client may participate in multiple invoices; the Client and the Invoice tables are linked by the Client Id fields. If a client may have an arbitrary number of addresses or phone numbers, we could build new address or phone number tables and also link them by the Client Id field.

Presentation of data across tables is constructed by an appropriate *join* operation. For example, if we want to present a particular invoice, we (1) find the desired invoice in the *Invoice* table, (2) use the Client Id to find the corresponding client in the *Client* table, (3) use the Invoice Id to find the invoice total in the *Invoice Total* table, and (4) present this data in a usable form. These accesses represent a considerable amount of complex work for the application programmer. Complex work is both expensive and prone to errors, so we would like to eliminate as much of the complexity as possible.

The three tables shown above can be described in Qddb by a single schema tree. Because trees can be represented in a list form, and programmers are generally quite familiar with lists, we choose to represent our schema trees as lists. (End users do not need to see this representation.)

A Qddb schema is a set of attributes, where each attribute may contain other attributes. Here is the appropriate schema:

```
Client ( Id Name Address HomePhone WorkPhone )
Invoice ( Id Item ( Id Price )* Total )*
```

This list has two elements, `Client` and `Invoice.` Each is an *attribute* of the schema. Client is a *structured attribute.* It has five *subattributes*, `Client.Id`, `Client.Name`, and so forth. These have no subattributes of their own, so they are *leaf attributes* and represent slots for data.

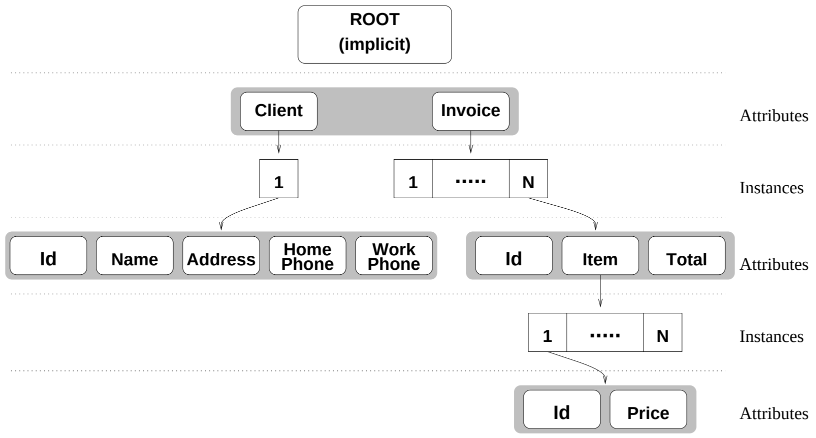

Figure 1: A sample *tuple tree*

The asterisk '`*`' denotes that an attribute or subattribute is *expandable*; in other words, it may have any number of instances[1]. These expandable attributes are equivalent to the joined tables shown earlier. Expandable attributes may be structured; their subattributes may themselves be structured and/or expandable.

In our example, `Invoice` is a structured, expandable attribute, and its subattribute `Invoice.Item` is also both structured and expandable. A tuple that obeys this schema may have many instances of `Invoice`, each of which may have many instances of `Invoice.Item`.

In relational terms, the Qddb database contains the data from the several joined tables shown earlier. Each Qddb tuple corresponds to exactly one entry in the Clients table. That client may be associated with an arbitrary number of entries in the Invoices table. Each invoice contains an arbitrary number of items (each with its price) and a total.

To access an invoice with a particular identifier, we only need to find the unique tuple that has an instance of `Invoice.Id` with the appropriate value. The associated instance of that tuple's `Invoice` attribute has all the information we seek. If we never plan to search invoices, clients, or items by their identifiers, all `Id` fields are extraneous and may be removed.

Figure 1 shows a tuple tree for the schema tree presented above. Every branch of the tuple tree is a copy of a branch in the schema tree with *instance nodes* inserted between each level. If an attribute is expandable,

[1] Non-expandable attributes have exactly one instance, although the instance may be empty. The application programmer can require the user to enter data and can therefore effectively disallow empty instances.

there can be multiple instance nodes ($1...n$), each of which heads its own subtree. Leaf attributes also have instance nodes (not shown) that point to data.

The schema tree presented above is nearly as inflexible as original relational tables, although it is perhaps clearer. Suppose we want to associate each client with an unlimited number of addresses and phones. Further, suppose we want to associate some phone numbers with particular addresses and other phone numbers with no address at all (perhaps for mobile phones). The schema tree can be upgraded to cover these needs:

```
Client (
    Id
    Name
    Residence ( Address Phone* )*
    Phone ( Description Number )*
)
Invoice ( Id Item ( Id Price )* Total )*
```

To represent this schema in a tabular form would require six tables, the original three tables plus two different ones for phones and one for residences. Two new kinds of unique identifiers would be needed, one for residences and one for no-address phones. These identifiers would link the tables.

As the number of expandable fields increases, so does the number of tables required to represent them. For that reason, relational database designers tend to shy away from such data. The Qddb schema tree, however, allows us to specify the relational tables in an easily conceivable form. The application programmer doesn't have to worry about building multiple tables and setting up appropriate links between the tables, so expandable attributes are welcomed if they are appropriate.

3 Schema trees

A *schema tree* describes the structure of tuples in the database. It indicates which attributes are structured and/or expandable. It also contains type information for leaf attributes. As shown above, schema trees represent many linked relational schemas. Schema trees have the following properties.

1. Each node in the schema tree has a unique path name.
2. A subtree of the schema tree headed by any node contains all the attributes logically associated with that node.
3. By traversing the subtree headed at any node, we can find all the relational tables logically associated with that node.

4. By traversing the associated subtree of the tuple tree and collecting values at leaf nodes, we can build the *join* of all the associated relational tables.

5. We can traverse the schema tree to build an intuitive user interface showing the relationship among the attributes.

Subtrees allow us to restrict our attention. For example, suppose we want to add a new residence to a client's record. To perform this task in a database using relational tables, we would (1) find the client in the Client table, read Client Id, (2) add a new row to the Residence table with Client Id and a new unique Residence Id, and (3) add one or more phone numbers to the Phones table, each with the appropriate Residence Id. With schema trees, however, we only need to find the client and add a new section of the tuple tree corresponding to the `Residence` attribute. Multiple phone numbers can be added to the new instance by creating a new section of the tuple tree corresponding to the `Residence.Phones` attribute under the same instance of Residence.

We have demonstrated that a recursively-structured schema tree can describe a set of relational tables where rows in a subordinate table are related to a single entity. In database terminology, these are one-many relations.

Tables that are related in more complex ways, such as many-many relationships, can still be described by a set of schema trees. Two schema trees are linked much in the same way that conventional relational tables are linked. For example, consider the following two schema trees linked by the `SocialSecurityNumber` field:

```
# Person schema tree
Name (First Last)
SocialSecurityNumber
Address (
    Street City State ZipCode
)*

# Company schema tree
CompanyName
CompanyAddress (
    Street City State ZipCode
)*
Employees (
    SocialSecurityNumber
    Position
    SalaryHistory (
        Date
        Salary
    )*
```

```
        )*
```

These particular relations should not be represented by a single schema tree because it is a many-many relationship. A person can be an employee of multiple companies. If you change the `Address` field of the Company relation, you want the `Address` field to change whenever accessing employees of that company.

3.1 Schema trees as full database descriptors

We have seen how schema trees can be used to describe the relationship between relational tables and to create new, logically related, entries in those tables. We now describe how Qddb uses the schema tree to fully describe a database by establishing types, verbose field names, search options, word descriptions and value formatting.

Qddb associates a type with each leaf attribute, either a numeric type (real, integer, or date), or string. Numeric values may be associated with a format for display (not input) purposes. The string type consists of words interspersed with separators. Usually, separators are taken to be any non-alphanumeric characters, but the schema may specify other separators. Strings are of arbitrary length.

Each attribute in the schema may be associated with a verbose (many-word) name for display purposes. By default, database searches cover values in all leaf attributes (in relational database parlance, all attributes are indexed), but the designer can exclude individual attributes from the indices to reduce their space requirements.

The schema itself is maintained in a free-format Ascii file that can be created by any text editor. Each attribute or subattribute in the schema is of the form:

```
AttributeName ?<options>? ?(<subattributes>)? ?*?
```

where we use the `??` notation to indicate optional syntax. The `<options>` may be any of:

Option	Purpose
verbosename “string”	Verbose name of attribute
type string	Attribute has type **string** [default]
type date	Attribute has type **date**
type integer	Attribute has type **integer**
type real	Attribute has type **real**
alias AttributeName	A unique alias for the attribute
separators “string”	Words are separated by one of the characters in “string”
format “string”	Format the attribute based on “string”
exclude	Exclude the attribute from indexing
defaultvalue “string”	A default value for each instance

To represent a simple database of potential employees, we might keep their name, addresses, phone numbers, rank, and date of application. A schema for this relation could be:

```
Name ( First Middle Last )
Address (
    Street exclude type string
    City
    State
    Zip verbosename "Zip Code"
)*
Phones ( Desc verbosename "Description" Number )*
Rank
    type integer
    format "%2d"
    verbosename "Prospect ranking"
Date type date format "%D %T"
    verbosename "Date Applied"
```

Real and integer formats are specified in the standard format for *printf()* in the ANSI C standard, and dates are specified in the standard form accepted by the POSIX function *strftime()*. All formats pertain to output conventions; input conventions are those accepted by *atoi()* (integers), *atof()* (reals), and a wide variety of forms (dates).

3.2 Enumerating attributes within a schema tree

It is important to be able to uniquely distinguish attributes in a schema tree. For this reason, a schema must uniquely name attributes at the top level of each subtree. Attributes across levels may have identical names. For example, consider this schema:

```
A ( B C )* B C
```

The leaf attributes are `A.B`, `A.C`, `B`, and `C`. When we refer to `B`, we are referring to the leaf attribute `B`, not `A.B`.

A depth-first traversal of the schema tree generates the set of leaf attributes. This set, displayed with one column for each leaf attribute, is called a *complete row* of the schema. If we wish to elide certain uninteresting parts of the complete row, we may ignore the columns comprising those parts, resulting in a *partial row*. For example, if we are only interested in the leaf attributes `B` and `C`, then we exclude attribute `A` and all its leaves from our traversal.

4 Tuple Trees

A *tuple tree* contains all data for a particular tuple in the database. It represents data from all the joined relational tables described by the associated *schema tree*. Structurally, it is an exact duplicate of the schema tree with a level of instance nodes placed after each level of attribute nodes (including the leaves).

The tuple tree allows programs to manipulate entire sets of related rows with simple operations. We can view, add, delete, or modify branches. We can produce rows relating to a branch by traversing only the part of the tree that is of interest.

For example, an application might create a new branch of a tuple tree. Each attribute is initialized to have a single instance. The application might then build a new instance for an expandable subtree. Then it might assign values to leaf attributes.

We might also want to delete an entire branch of the tuple tree. Suppose we have the following schema and corresponding tuple:

```
# Schema
Name
Address ( Street City State Zip Phones* )*
Phones ( Desc Number )*

# Tuple
Name = "Joe Rohn"
Address (
    Street = "123 Anyway Dr."
    City = "Georgetown
    State = "Virginia"
    Zip = "41908"
    Phones = "(654) 111-1121"
    Phones = "(654) 111-1122"
```

```
)
Address (
    Street = "312 Anyway Dr."
    City = "Georgetown
    State = "Virginia"
    Zip = "41908"
    Phones = "(654) 111-1123"
    Phones = "(654) 111-1124"
)
Phones (
    Desc = "Home"
    Number = "(654) 234-1232"
)
```

If we wish to view a table consisting of the `Address.Street` and `Address.Phones` fields on this tuple, we traverse the tuple tree and produce the following table:

Address.Street	Address.Phones
123 Anyway Dr.	(654) 111-1121
123 Anyway Dr.	(654) 111-1122
312 Anyway Dr.	(654) 111-1123
312 Anyway Dr.	(654) 111-1124

If we delete the branch of the tuple tree corresponding to the `Address` field containing "312 Anyway Dr." we also implicitly delete the two phone numbers "(654) 111-1123" and "(654) 111-1124" associated with that address.

4.1 Identifying leaves

We must be able to uniquely identify individual leaves in the tuple tree. Without this ability, we cannot construct the tree from the values nor can we tell which attribute values are related.

A tuple tree has the following convenient properties: (1) each attribute node has a unique local name (that distinguishes it from its siblings), and (2) each instance node has a unique local number (that distinguishes it from its siblings). In Figures 1 and 2, attribute-node siblings are shown surrounded by a gray box. We number all the attribute-node siblings (within a single gray box) 1, 2, 3, and so forth. Then we assign each leaf a label formed by the concatenation of the local numbers on the path from the root to that leaf, using a dot as a separator, as shown in Figure 2. The superscripts in the figure refer to the attribute number. Therefore, the first instance of `Invoice.Item.Id` (first item in the seventh invoice) is labelled "2.7.2.1.1.1". We call these leaf labels *leaf identifiers*. Prefixes are called *path labels*.

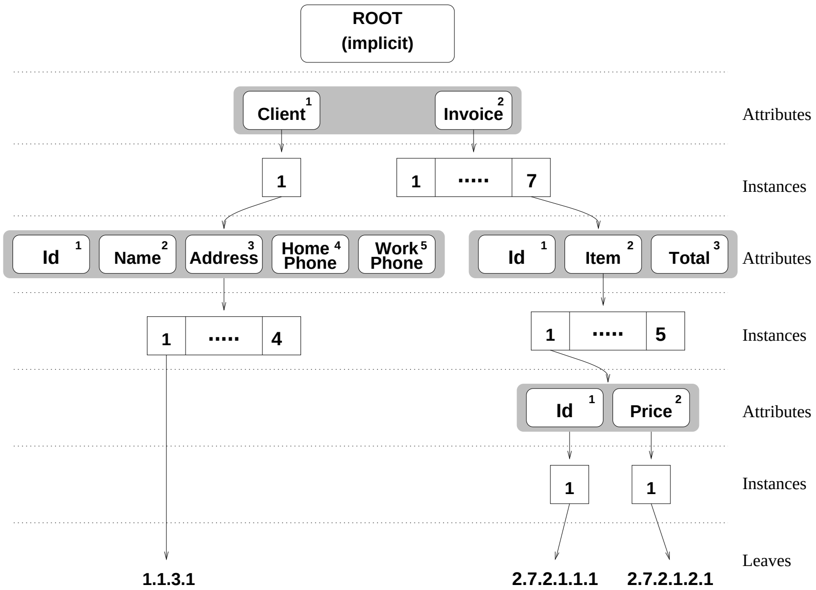

Figure 2: Enumerating the leaves of a tuple tree

Integers at odd positions in a path label (that is, 2, 2, 1) are due to attribute numbers; integers at even positions (7, 1, 1) are due to instance numbers.

4.2 Expandable attributes and their relation to tables

We can think of each expandable Qddb attribute `A` as a separate relational table. The subattributes of `A` form the columns of the table. (If `A` is not structured, the table has one column.) In addition, the equivalent relational tables must contain a link column with a unique identifier for joining with other tables. For example, an attribute `A` whose schema is `A ( B C D )*` is a table with columns B, C, and D. A single tuple gives rise to several rows in that table, one for each instance of attribute `A`.

Since both structured and non-structured attributes can be expandable, we can easily describe very complicated relationships between the tables. For example, we might have the following schema:

```
Location (
    Name*
    Address ( Street City State Zip Contact* )*
    Phone ( Desc AreaCode Number )*
)
```

This Qddb example is equivalent to a relational Location table containing columns Name, Address, and Phone. The Address column contains unique identifiers linking it to a separate Address table, which contains, besides the link column, columns Street, City, State, Zip, and Contact. The Contact column contains unique identifiers linking it to a separate Contact table, which contains, besides the link column, a single data column. Similarly, the Name and Phone columns of the Location table are links to other tables. In all, there are five tables, interlinked with a web of unique identifiers.

The tuple tree is a convenient abstraction for manipulating related rows in possibly many relational tables. Relational rules may be enforced at the application level to ensure full compliance with the relational model (for example, to enforce non-empty fields or indivisible search values.) Any set of tuple trees can be decomposed into proper relational tables, although unique link fields must be invented when performing this decomposition. Any set of relational tables may also be composed into a set of tuple trees; existing link fields are extraneous and may be removed.

4.3 Generating all rows

The tuple tree lets us construct all rows pertaining to the tuple without expensive *join* operations on large tables. The following algorithm produces all rows from a given node in a tuple tree:

```
procedure ProduceAttributes (AttributeNode) : set of rows
    answer := NULL
    for each Inst := instance of AttributeNode do
        answer := union(answer, ProduceInstances(Inst))
    return answer

procedure ProduceInstances (InstanceNode) : set of rows
    if leaf(InstanceNode) then
        return InstanceNode.Value
    else
        answer := NULL
        for each Attr := attribute of InstanceNode do
            answer := cartesian product(answer,
                ProduceAttributes(Attr))
        return answer
```

The `union` operator returns a table that combines all the rows of its two operands, which agree on number and type of columns. The `cartesian product` operator returns a table that combines all the columns of its two operands, with as many rows as the product of the rows of the two operands.

We can build all the complete rows of the tuple tree for a given tuple:

ProduceInstances(RootNode)

We can also build partial rows pertaining to subtrees. For example, we could find all information about the 7th invoice for a particular tuple as follows:

ProduceInstances(2.7)

The result is a table with columns:

```
Invoice.Id Invoice.Item.Id Invoice.Item.Price Invoice.Total
```

The values for the `Invoice.Id` and `Invoice.Total` columns are the same for all rows. We can narrow our focus and consider only items in the 7th invoice:

ProduceAttributes(2.7.2)

The result is a table with columns `Invoice.Item.Id` and `Invoice.Item.Price`.

Figure 3 depicts a tuple tree for the schema `A ( B C* D )*`. An invocation of

ProduceAttributes(RootNode)

on this tuple tree returns the following table of complete rows:

A.B	A.C	A.D
once	upon	dreary
once	midnight	dreary
while	pondered	weak
over	many	curious
over	quaint	curious

Now suppose we aren't interested in the `A.C` attribute. Excluding the undesired attributes from the traversal builds the following table, which accurately restricts the previous one:

A.B	A.D
once	dreary
while	weak
over	curious

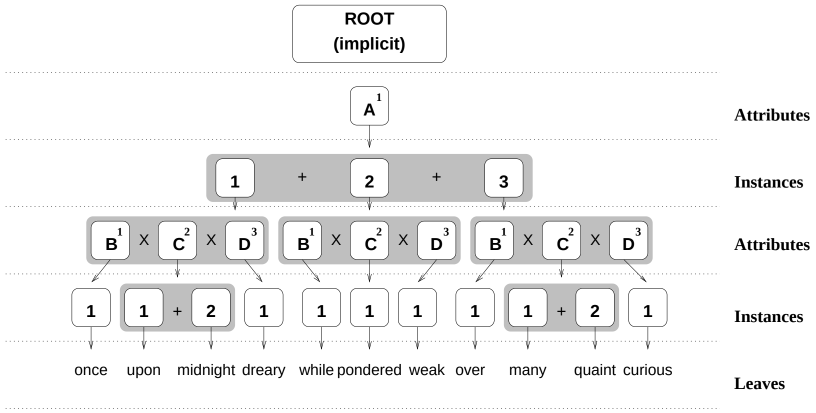

Figure 3: Traversing a tuple tree

4.4 Sparse data

Many database applications manipulate sparse data; that is, many fields are empty in any given tuple. One advantage of the tuple tree is that we can reconstruct the entire tuple tree from only those leaves containing data. Qddb's external representation only stores attributes with non-empty data. For example, consider the following modification of the tuple tree in Figure 3 to include fewer populated leaves:

1.1.2.1	upon
1.1.2.2	midnight
1.2.1.1	while
1.3.2.1	many
1.3.2.2	quaint
1.3.3.1	curious

From these leaves, we know there are three instances of attribute `A`, the first and third of which contain two instances of `A.C`. When Qddb reads in these leaves, it can build an entire internal tuple tree. The algorithm for building tuple trees from a given set of leaf values (each with a leaf identifier) is as follows:

1. Construct a tuple tree containing one instance for every attribute in the schema. All leaves start with empty values.

2. For each given leaf value, follow its path label, diving down the tuple tree according to the label's components.

   (a) If we we reach an instance that doesn't exist, create the instance and its entire subtree, giving all leaves empty values.

(b) When we reach a leaf, set its value to the given value.

The implementation of this algorithm is most efficient when the leaves are presented in left-to-right order by leaf identifier. The external representation of Qddb data always maintains order because it is generated from a left-to-right depth-first traversal of an internal tuple tree.

4.5 Locking

Locking rows in a traditional relational database can be a non-trivial task. A set of related rows may be spread across many tables, requiring many locks. Locating the rows to lock may require queries on many tables.

A tuple tree is stored externally as a contiguous list of leaves. The entire contiguous region of the database file can be locked to achieve a lock on a given tuple. Qddb does not lock at a finer grain (such as a subtree of the tuple tree), although subtrees are also stored contiguously, and Qddb therefore could easily accomplish finer-grained locking. It is straightforward to lock multiple tuple trees if the need arises.

5 Indexing and searching

Locators are pointers into data. They contain two components: a *tuple identifier* and a *leaf identifier*. Each tuple in a relation can be located with its tuple identifier. Each leaf within a given tuple is uniquely identified by its leaf identifier.

Qddb associates a list of locators with every *key*, that is, every searchable value found in the tuples. We call this association the *index* into a Qddb relation. The index can be accessed in three ways. The most direct way is by hashing the given key to find the associated locator list. Word-range and numeric-range searches are performed by binary search in a sorted key or number file, leading to a contiguous set of entries, each of which points to a locator list in the index. Regular-expression searches use finite-automaton search through the sorted key file, leading to a set of matching entries, each of which points to a locator list in the index.

In all cases, searches yield a list of locators. If only the list of tuples is needed, this list can be pruned by discarding duplicate locators that have the same tuple identifier. If the search is to be constrained to particular attributes, the list of locators is pruned by discarding all locators with irrelevant leaf identifiers.

Complex searches are constructed by applying simple searches and combining the results. We will demonstrate several examples based on this schema tree:

```
Name (First Last) Address (Street City)*
```

A sample complex query might be: Find all tuples with the value "Joe" in the `Name.First` field and "Harrison" in the `Name.Last` field. In other words, we want to find tuples that satisfy the following expression:

```
(Name.First = "Joe") and (Name.Last = "Harrison")
```

This query requires three steps: (1) Search for all tuples with `Name.First` containing "Joe", (2) Search for all tuples with `Name.Last` containing "Harrison", and (3) construct a weak intersection of the matching locators. A *weak intersection* ignores the leaf identifiers of the locators and uses only the tuple identifiers for comparisons. Only tuples that match both searches remain after the intersection.

Strong operations on locator lists do not ignore the leaf identifiers. Strong operations use both the tuple identifier and the leaf identifier for comparison purposes. Two locators are equivalent if both the tuple identifiers and leaf identifiers are identical. For example, a strong intersection of sets A and B produces all locators that reside in both sets; a weak intersection produces all locators with tuple identifier t if at least one locator with tuple identifier t resides in both A and B. Strong operations generally operate on lists that contain locators associated with a particular attribute.

We can also do binary union and exclusion operations on locator lists. Binary union merges two sets of locators. Binary exclusion of two sets A and B produces all locators that are contained in set A but not in set B. Binary exclusion has both weak and strong forms. Binary union does not have a weak or strong form because no comparisons are used in the operation.

Suppose, for example, we want to find all occurrences of "Joe" in the `Name.First` field where the `Name.Last` field is *not* "Harrison". In other words, we want

```
(Name.First = "Joe") and ((Name.Last = ".*")
    minus (Name.Last = "Harrison"))
```

The syntax ".*" means "any value." The search first builds the locator lists (A, B, and C) for the three subexpressions from left to right; it then performs a strong exclusion $(B - C)$ and a weak intersection $(A \cap (B - C))$ to produce the result.

5.1 Finding tuples with matching rows

In our previous example, the attributes of interest were not expandable. Expandable attributes provide an interesting dilemma: how do we know if two matching locators belonging to the same tuple are in the same row? To illustrate this problem, consider the query:

```
(Address.Street = "Rainbow") and (Address.City = "Lexington")
```

A tuple may have the following values for the `Address` attribute:

`Address.Street`	`Address.City`
Rainbow	Pittsburgh
Nichols	Lexington

Such a tuple will be produced by the weak intersection of the two expressions, but is not a proper result, because the two expressions fail to match on the same row.

For two leaves to be in the same row, they must be in the same instance of their deepest common attribute ancestor. In our case, they must be in the same instance of `Address`. (In the previous cases, they must be in the same instance of `Name`, but this attribute is not expandable, so all results were in the same row.)

We can determine whether two leaves are in the same row by (1) comparing the leaf identifiers of the two leaves from left to right, and (2) noticing the first position that the attribute numbers (odd positions) or instance numbers (even positions) differ. If the first difference is an attribute number, then the two leaves are in the same row. If the first difference is an instance number, then the two leaves are not in the same row.

Suppose we perform a query Q that returns a set of locators L describing the results of our query. We want to find all rows that match our query Q. The following algorithm accomplishes this task:

1. Let T be the set of all tuple trees in the relation.
2. Let A represent all attributes in the schema tree that were used in our query Q.
3. Partition the set L into a set S of subsets. Each subset $s \in S$ contains all locators $l \in L$ describing a single tuple $t \in T$; that is, there is a one-to-one correspondence from each element $s \in S$ to an element $t \in T$.
4. For each $s \in S$,
   (a) For each attribute $a \in A$, place all locators $l \in s$ describing

attribute a (regardless of instance) into a set s_a. A set $s_a = \emptyset$ if and only if there is no locator $l \in s$ describing a match on attribute a.

(b) If for any attribute $a \in A$, $s_a = \emptyset$, then the tuple tree t described by s does not contain a row that satisfies Q. Let $S = S - s$, then continue with the next $s \in S$.

(c) Otherwise, perform the cartesian product $R = \bigotimes_{a \in A} s_a$.

(d) For each $r \in R$, check to see if all locators in the components of r are in the same row. If no $r \in R$ satisfies this condition, then the tuple tree t described by s does not contain a row matching query Q; let $S = S - s$ and continue with the next $s \in S$. Otherwise, do not modify S and continue with the next $s \in S$.

At the completion of this algorithm, each remaining $s \in S$ describes all rows in a single tuple $t \in T$ that satisfy Q. For each $s \in S$, we can find the all tuples that satisfy Q by the tuple identifier contained in any locator $l \in s$. We now have a set of tuple identifiers completely describing the locations of the tuple trees satisfying query Q.

This algorithm finds all tuples with *at least* one matching partial row (partial in that it considers only the attributes in the query) without reading the actual tuple tree data. Each matching row may represent several complete rows; this expansion can be determined only by reading the data. For most purposes, this is a very efficient mechanism for determining which tuples match a particular query; we only need to read the tuples that contain at least one matching row.

5.2 Generating matching rows

After applying the algorithm presented in Section 5.1, we have a set S that contains a subset of locators $s \in S$ for each tuple $t \in T$ satisfying our query Q. Each subset s describes all the locators for each attribute in a particular tuple matching our query. We know each tuple is guaranteed to contain *at least* one row satisfying our query. We must now read the tuple's leaves from the disk to build the tuple tree and produce all matching rows. Once we have the tuple in the form of a tuple tree, we mark all leaves matching our query as *good*. We mark *bad* all leaves of searched attributes that are not marked good, because we know that these leaves did not match our query. A traversal of the tuple tree will produce all rows matching the query if we exclude any row containing a leaf that is marked *bad*.

For example, suppose we have the tuple tree shown in Figure 4 and we search on the attributes `A.B` and `A.C`. Suppose we know this tuple matches because the algorithm presented in Section 5.1 produces a subset $s \in S$

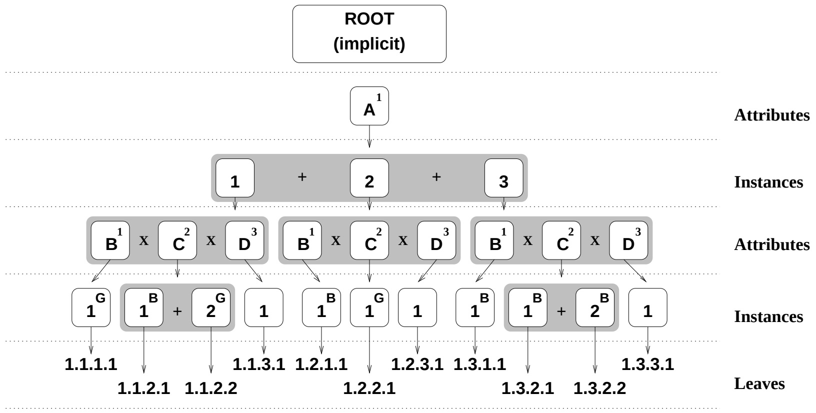

Figure 4: Generating rows in a *tuple tree* matching a query.

corresponding to this tuple. Suppose our set s describes the leaves (1.1.1.1 1.1.2.2 1.2.2.1), so we mark those leaves as good. (Figure 4 shows them with a superscript 'G'.) The searched attributes are `A.B` and `A.C`, so we mark all instances of `A.B` and `A.C` that are not marked good as bad (shown with a superscript 'B'.)

To produce all rows matching the query for this tuple, we use the following modification of the algorithm presented in Section 4.3:

```
procedure ProduceAttributes (AttributeNode) : set of rows
    answer := NULL
    for each Inst := instance of AttributeNode do
        answer := union(answer, ProduceInstances(Inst))
    return answer

procedure ProduceInstances (InstanceNode) : set of rows
    if leaf(InstanceNode) then
        if bad(InstanceNode) then
            return NULL
        else
            return InstanceNode.Value
    else
        answer := NULL
        for each Attr := attribute of InstanceNode do
            answer := cartesian product(answer,
                ProduceAttributes(Attr))
        return answer
```

Using the above algorithm on the tuple shown in Figure 4, we get the following single row:

1.1.1.1   1.1.2.2   1.1.3.1

The other rows were eliminated because one or more of their leaves were marked bad. This algorithm will not generate false matches because each leaf that represents a searched attribute is marked bad unless a locator is found for that particular leaf.

5.3 Efficiency

Tuple trees are generally stored contiguously. Each tuple tree contains all the relational rows logically related to that tuple, so the following tasks are greatly simplified:

1. Reading - tuple trees can be read in a single file system read operation, so all rows related to that tuple are available for processing.
2. Locking - tuple trees can be locked by a single file system lock operation.
3. Writing - tuple trees can be written with a single file system write operation.
4. Addition - tuple trees (or any sub-tree) can be added with a single operation on an in-memory tuple tree.
5. Deletion - tuple trees (or any sub-tree) can be deleted with a single operation on an in-memory tuple tree.

Tuple trees need not store empty attributes, either in memory or in a file, so sparse tables may be represented in a very compressed form. The main disadvantage of the tuple tree is that leaf identifiers must also be stored with each attribute value in the tuple. To eliminate much of this overhead, we can enumerate all possible leaf identifiers with base-36 indices (

$$a - zA - Z0 - 9$$

) and store the index instead of the full leaf identifier. For example, 36^4 (1,679,616) unique leaf identifiers can be represented by 4 digits (a 4-byte overhead per attribute). In practice, tuple trees tend to have a high level of leaf identifier replication between different tuple trees, so the number of unique leaf identifiers is typically quite small.

6 Presentation

Tuple trees lend themselves to various representations for different purposes. The *external form* used by Qddb in its database files prefaces each value with its leaf identifier, omitting those attributes that have empty values.

The *readable form* is a textual representation of the tuple tree with attributes given their local (textual) names and with structured attributes surrounded by parentheses.

For example, consider the following schema tree: `A ( B* C )* D`. A simple readable form of a tuple tree might be:

```
A (
    B = "10"
    C = "20"
)
D = "30"
```

`A` is a structured attribute containing subattributes `B` and `C`. Since the attributes `A` and `A.B` are expandable and values can be empty, tuples can be more complex:

```
A (
    B = "10"
    B = "80"
    C = "20"
)
A (
    B = "40"
)
D = "30"
```

Multiple instances of expandable attributes are always adjacent in the readable form. Attributes with empty values are omitted. Qddb uses the readable form for a text-editor interface to data.

For the convenience of Tcl programmers, Qddb supports the *tcl form* of tuple trees. In this form, a tuple tree is a list of (attribute, value) pairs, where the attribute is a leaf identifier and the value is a string.

The newest presentation form of a tuple tree is the *graphical form*. The graphical form is displayed in a window (in the X Window System [Scheifler & Gettys 1986]). Given the following schema tree, Qddb tools automatically build the graphical form of the tuple shown in Figure 5.

```
Name ( First Last )
Address ( Street City State Zip Phones (Desc Number)* )*
Phones ( Desc Number )*
```

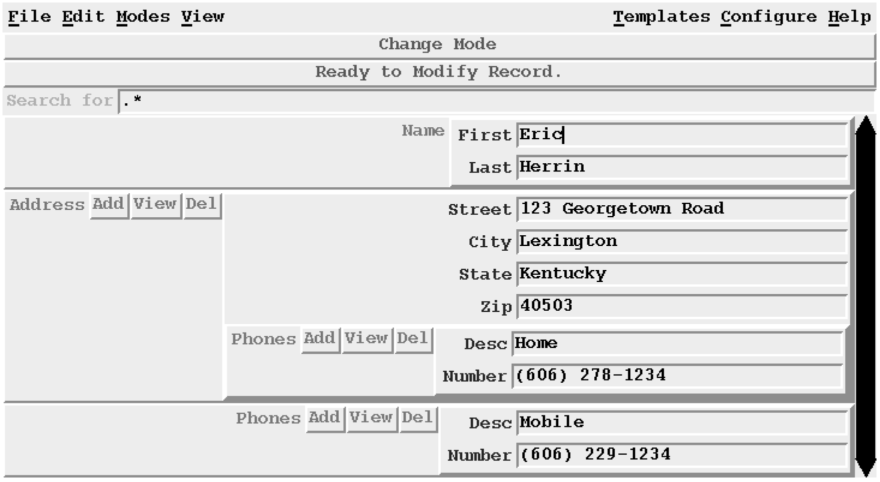

Figure 5: The graphical form of a tuple tree

The graphical form looks very similar to the schema tree [2]. At any given time, the graphical form displays one complete row in a single tuple.

Rows of an expandable attribute associated with the current complete row are accessible by interactively selecting a "View" button. This button invokes the ProduceAttributes algorithm introduced in Section 4.3. For example, suppose we are in the 2nd instance of `Address` and wish to view `Address.Phones`. The viewed rows are produced by evaluating:

```
r = ProduceAttributes(Address.2.Phones)
```

If we choose to view the `Address` attribute, the viewed rows are produced with:

```
r = ProduceAttributes(Address)
```

In other words, choosing the `Address` attribute's "View" button shows all possible rows beginning at the `Address` node in the tuple tree. Choosing the `Address.Phones` "View" button shows all rows beginning at the `Address.Phones` node in the current instance of `Address` in the tuple tree. All views are configurable, that is, the user can choose to view only certain columns, sort by given columns, and order the columns.

The user can add a new instance of an expandable attribute by selecting the "Add" button. This button adds an entire branch (with initially empty

[2] Qddb comes with interactive tools that allow users to modify the appearance of this graphical form and programming tools that allow programmers to make more substantial modifications.

contents or default values) to the tuple tree. Similarly, the user can delete an entire branch of the tuple tree by selecting the "Del" button.

The graphical form provides a clean interface for attributed searching. The user may specify one or more keys for all attributes. If multiple keys are specified for a particular attribute, each constitutes a search. The results of the searches within a particular attribute are combined by strong operations. The results of searches across attributes are then combined using weak intersection, and the result is pruned by the algorithm presented in Section 5.2. The matching tuples are then read, and the rows that satisfy the query are constructed and displayed.

7 Related Work

Schema and tuple trees are related to nested relations and object-oriented databases. *Nested relations* [Makinouchi 1977; Jaeschke & Schek 1982] are relations that have non-atomic attributes, that is, they are not in the First Normal Form ($1NF$). Nested relations are also called NF^2 (Non First Normal Form) relations. It has been realized for some time that NF^2 relations can be decomposed into $1NF$ relations [Korth & Silberschatz 1991].

Other researchers have used the term *scheme tree* in a way different from ours. For example, they can represent nested relations, where the nodes are pairwise-disjoint sets of non-nested attributes and the edges represent multivalued dependencies [Ozsoyoglu & Yuan 1987]. Some implementations of nested relations include AIM-P [Pistor & Dadam 1989], VERSO [Schoil et al. 1989], ANDA [Deshpande & Gucht 1989], and DASDBS [Schek & Scholl 1989]. These implementations appear to use linked tables as the underlying data structure, although DASDBS seems to perform some optimization by storing related rows within a table close together.

Object-oriented databases [Kroha 1993; Korth & Silberschatz 1991] mix object-oriented programming practices with databases. Typically, objects are schemas written in a fashion similar to a `C++` *class* containing members such as constructors, methods, and variables. Objects may be derived from other objects through inheritance. Since object-oriented databases allow arbitrary recursive structures, it can be quite difficult to map them into the relational model. Object-oriented programming techniques can be used without inheritance or extensible data types in a relational database [Premerlani et al. 1990]. Typically, object-oriented databases are used for applications that do not fit the relational model well, such as CAD (computer aided design). Some implementations of object-oriented databases include O_2 [Lecluse et al. 1988] and Gemstone [Maier et al. 1986].

8 History and Current Status

We developed schema and tuple trees in 1989 when we first developed Qddb. At that time, Qddb was implemented in around 10,000 lines of C code and performed very basic searches and operations. Qddb was released to the public, but the presentation utilities were so crude that only a few hundred people bothered to learn about it. Qddb development stalled until 1994 because of other projects. In 1994, we began building a generic graphical application based on the schema tree. With the advent and maturation of Tcl and the Tk toolkit [Ousterhout 1994], we have been able to build a significantly improved interface that interacts with the X Window System [Scheifler & Gettys 1986].

Today, Qddb sports a fancy graphical user interface allowing users to conveniently navigate and search for rows in tuple trees. As a result, the Qddb user community has grown to thousands of people world-wide. Applications include World-Wide Web (WWW) servers, inventory management, client-invoice management, point-of-sale terminals, experimental data analysis, project tracking, bug reports, journal abstracts, dictionaries, and student records.

Qddb is implemented in around 45,000 lines of C and Tcl code. It runs on any Unix-based computer. Qddb is always publicly available from:

ftp.ms.uky.edu:pub/unix/qddb/qddb-<version>.tar.gz

The distribution contains full documentation and source. Sample databases and applications are also available.

9 Conclusion

Tuple trees provide a convenient and efficient structure for storing conventional relational rows in a prejoined fashion. Schema trees provide an easy, intuitive way for programmers (and even non-technical people) to define a database with fairly complex relationships among the various attributes. Tuple trees have many nice properties: they are easy to build, easy to manipulate, efficient to store and search, and easy to view. Since a tuple tree's leaves can be stored contiguously, locking a set of related rows (one tuple tree) is straightforward.

Since schema and tuple trees are easy to understand, we have found that many people incapable of defining simple databases with standard relational tools can now build fairly complex databases in a matter of minutes.

References

1. A. Deshpande and D. V. Gucht. A Storage Structure for Nested Relational Databases. In S. Abiteboul, P. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 50–68. Springer-Verlag 1989.

2. E. H. Herrin, II and R. A. Finkel. An ASCII Database for Fast Queries of Relatively Stable Data. *Computing Systems*, 4(2):127–155 (1991).

3. G. Jaeschke and H. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 124–138 (1982).

4. H. F. Korth and A. Silberschatz. *Database System Concepts.* Computer Science Series. McGraw-Hill 1991. ISBN 0-07-044754-3.

5. P. Kroha. *Objects and Databases.* International Series in Software Engineering. McGraw-Hill 1993. ISBN 0-07-707790-3.

6. C. Lecluse, P. Richard, and F. Velez. O2: An Object-Oriented Data Model. In *Proceedings of the ACM International Conference on the Management of Data*, pages 424–433 (1988).

7. D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482 (1986).

8. A. Makinouchi. A Consideration of Normal Form on Not-necessarily Normalized Relations in the Relational Data Model. In *Proceedings of the International Conference on Very Large Databases*, pages 447–453 (1977).

9. J. K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley 1994. ISBN 0-201-63337-X.

10. Z. M. Ozsoyoglu and L.-Y. Yuan. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems*, 12(1):111–136 (1987).

11. P. Pistor and P. Dadam. The Advanced Information Management Prototype. In S. Abiteboul, P. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 3–26. Springer-Verlag 1989. ISBN 3-540-511171-7.

12. W. J. Premerlani, M. R. Blaha, J. E. Rumbaugh, and T. A. Varwig. An Object-Oriented Relational Database. *Communications of the ACM*, 33(11):99–109 (1990).

13. M. A. Roth and H. F. Korth. The Design of ¬1NF Relational Databases into Nested Normal Form. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 143–159 (1987).

14. R. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109 (1986).

15. H.-J. Schek and M. H. Scholl. The Two Roles of Nested Relational Databases. In S. Abiteboul, P. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 50–68. Springer-Verlag 1989.

16. M. Schoil, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A Database Machine Based on Nested Relations. In S. Abiteboul, P. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 27–49. Springer-Verlag 1989.