Storing and Querying Tree-Structured Records in Dremel

Foto N. Afrati *
National Technical University of Athens
afrati@softlab.ece.ntua.gr

Dan Delorey
Google
delorey@google.com

Mosha Pasumansky
Google
moshap@google.com

Jeffrey D. Ullman
Stanford University
ullman@gmail.com

ABSTRACT

In Dremel, data is stored as nested relations. The schema for a relation is a tree, all of whose nodes are attributes, and whose leaf attributes hold values. We explore filter and aggregate queries that are given in the Dremel dialect of SQL. Complications arise because of *repeated* attributes, i.e., attributes that are allowed to have more than one value. We focus on the common class of Dremel queries that are processed on column-stored data in a way that results in query processing time that is linear on the size of the relevant data, i.e., data in the columns that participate in the query. We formally define the data model, the query language and the algorithms for query processing in column-stored data. The concepts of repetition context and semi-flattening are introduced here and play a central role in understanding this class of queries and their algorithms.

1. Introduction

Systems for managing "big data" often use tree-structured data models. Two important examples of such models are the JSON data format [1] and Google's protocol buffers [2]. Similar models have been studied many years ago as *nested relations* [16, 10]. Two recently developed systems, Dremel [13] and F1 [18] use schemas that combine relational and tree-structured features and have query languages that are dialects of SQL. In these data models, the database consists of one or more relations. The tuples of a relation share a structured schema for that relation. The query language uses SQL-style syntax but the evaluation techniques have to be modified to fit the richer schema.

1.1 Outline of Paper

We begin with a formal definition of tree-schemas and the meaning of SQL-like queries on data matching a schema. The necessary concepts include "flattening" [15] of the tree-structured data, which expands the repeated groups found in the schema, and "dummy" occurrences of repeated groups, a necessary technical contrivance, allowing us to work with unnormalized data without encountering deletion anomalies.

We explain the meaning of SQL-like queries on tree-structured data in terms of the effect of those queries on the flattened data. Unfortunately, not all queries produce results on the flattened data that can be described by modifying the original tree-structured data and then flattening the modified tree. We therefore investigate when it is possible to execute a filter query (selection of relational algebra) by pruning the tree-structured data directly, and have a result, consistent with the execution of the same query on the flattened version of the data. The key techical tool is the "dominance" relation between nodes of the schema tree. It turns out that the class of filter queries allowed in Dremel [13] is a subset of these queries.

We then introduce a representation of tree-structured data called "semi-flattening," which better models the way Dremel processes data and is usually much more compact than flattened data. We show the class of filter queries that can be executed by pruning the tree data can also be executed on semi-flattened data. It is also possible to execute certain queries involving aggregations on semi-flattened data. The last part of the paper examines when it is possible to do so, and relates the condition to the dominance relation on schema nodes.

2. Trees as Data and as Data Types

We define data types recursively as:

1. A *tuple type* is a list of attribute names and a (previously defined) type for each attribute.
2. The type of an attribute is either a *basic type* (integer, real, string, etc.) or a tuple type. Further, attributes within a tuple type can be either *required* (one occurrence), *optional* (zero or one occurrence), *repeated* (zero, one, or more occurrences), or *required and repeated* (one or more occurrences).
3. A *relation type* is a repeated tuple type. We shall refer to the type of tuples (unrepeated) as the *schema* of the relation.

*The work of this author was supported by the project Handling Uncertainty in Data Intensive Applications under the program THALES.

This structure of data appears in many places. Here are some of the most common ones.

1. The *nested relational model* [5, 8, 11, 17].
2. The structure is described by DTD's [7] without the links implied by ID's and IDREF's.
3. It is the structure implied by JSON [1].
4. It is the hierarchical model [6] without links.
5. It is the essence of *protocol buffers* [2].

2.1 Representing Schemas

We use the conventional notation for types. For example, *int* and *string* will denote the basic types integer and string. A tuple type T with attributes $A_1, \ldots, A_n$ whose types are $T_1, \ldots, T_n$, respectively will be denoted

$$T = \{A_1 : T_1, \ldots, A_n : T_n\}$$

The repeated type T will be denoted T^*, the optional type T will be denoted $T?$; we also use T^+ to denote "one or more occurrences."

We shall use trees to represent schemas. The following rules define how a tree is constructed from a data type. Each node represents either the entire type (if the node is the root) or one of the subtypes used to define that type.

1. A node that represents a tuple type has children for each attribute of that tuple type, in order from the left.
2. The children are labeled by their corresponding attribute names.
3. In addition, each attribute has a *repetition constraint*. An attribute that is repeated is labeled with a *; an optional attribute is labeled with a ?, and an attribute that is required and repeated is labeled by a +.
4. The root itself is labeled by the name of the type. Typically, the root type is starred, since it is the type of a relation and the relation consists of zero or more tuples of the root type.
5. Leaf nodes are of basic type. Technically we should attach the type of each leaf to the leaf itself, but in examples these types will all be integers, reals, or strings, and the choice among these will be both obvious and irrelevant to the points we are trying to make.

EXAMPLE 2.1. *In Fig. 1 we see the tree schema for a hypothetical data type that represents advertisers at a search engine. The root is labeled Advertiser, the name of the type. In queries, we shall also use Advertiser as the name of a relation containing tuples of this type. Advertiser is a tuple type, with three attributes: required attributes Name (of the advertiser) and Email (of the advertiser) and a repeated attribute Campaign. Each advertiser can thus have any number of Campaigns, including zero.*

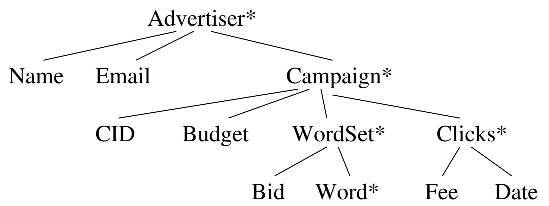

Figure 1: A relation schema represented as a tree

A Campaign comprises CID, a unique identifier for the Campaign, a Budget, zero or more WordSets, and zero or more Clicks. A WordSet is a tuple type consisting of a required Bid and a set of Words. The subtype Click is a tuple type with two required attributes: the Fee collected for the click and the Date of the click.

2.2 Instances of a Schema

An instance of a data type or schema consists of replacement of each subtype by an appropriate number of instances of that subtype. More formally:

1. An instance of a basic type is any single value of the appropriate type.
2. An instance of a tuple type is a node whose children are each instances of one of the types of one of its attributes. The nodes for the attributes are sorted in the same order as the attributes themselves. However, there can be zero or more instances for each attribute, depending on its repetition constraint, as follows.
   a) A required attribute must have exactly one occurrence.
   b) An optional attribute can have zero or one occurrence.
   c) A repeated attribute can have any number of occurrences.
   d) An attribute that is both required and repeated can have one or more instances.

EXAMPLE 2.2. *Figure 2 suggests a possible instance of the relation that is described by the schema of Fig. 1. The root, labeled* o*, represents the relation with this type. For all other instance nodes, we use a naming scheme that indicates to which schema node it belongs. In this example we use one or two letters and a subscript to designate an instance node, and the letters will match the name of the schema node to which the instance belongs.*

The root has two children ad1 *and* ad2*, representing two Advertiser tuples. The second of these is just sketched, so let us concentrate on* ad1*. The node for* ad1 *has children* n1 *and* e1*, the Name and Email for the first Advertiser.*

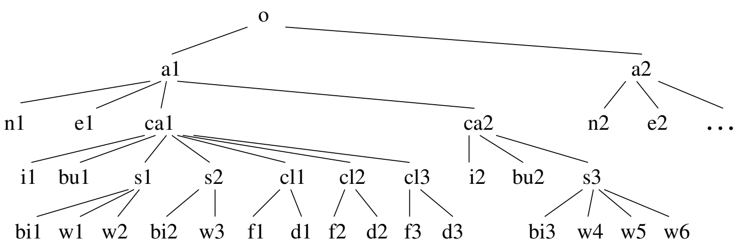

Figure 2: An instance of the schema of Fig. 1

Then we see two instances of the Campaign attribute, represented by nodes ca1 *and* ca2*. The first of these,* ca1*, has two instances of Wordset and three instances of Clicks, whereas the second,* ca2*, has one instance of WordSet and zero instance of Clicks.*

2.3 Dummy Occurrences

For several reasons, including the way we flatten instances in Section 3.1, we shall maintain a fiction about attributes that are repeated or optional. We imagine that there is one *dummy* occurrence of this attribute, all of whose descendant leaves have the value NULL. Since this instance may have descendant interior nodes representing repeated or optional groups, those descendants are consequently treated as if they had only the dummy instance dummy instance.

We do not show this dummy instance in tree diagrams, although as we shall see, there are reasons why it is useful to imagine it is there, and able to appear when needed. For instance, when we discuss querying, we shall see that sometimes all the occurrences of some repeated attribute are deleted. We do not want anything else in the tree to disappear, so we replace the deleted occurrences by the dummy occurrence. This viewpoint is consistent with the treatment of tree-structured instances as an outerjoin of conventional relations. It is also a reasonable way to avoid deletion anomalies in non-first-normal-form relations.

Example 2.3. *Consider Advertiser tuples with the schema of Fig. 1. Each advertiser has a dummy Campaign. The dummy Campaign has NULL for the values of CID and Budget. It has a dummy WordSet, with value NULL for its Bid. This WordSet also has a dummy set of Words, with only the NULL value for a Word. Finally, this dummy Campaign has a dummy Clicks instance, with NULL as its value for both Fee and Date.*

3. Querying Tree-Structured Data

Query languages like XQuery [4] or XPath [3] are fundamentally navigation languages on trees. The same can be said of older query languages that used the hierarchical model [6]. A more recent approach to query languages attempts to be more SQL-like, and to think of the instances of a tree data type as if they were tuples of a relation. Examples include the SQL variants used in Dremel [13] and F1 [18]. It is this approach that we address in this paper.

3.1 Flattening

Flattening has always been regarded as a fundamental algebraic operation on nested relations [10, 15]. Informally, we flatten an instance of a tree data type by selecting one from each repeated group of values in all possible ways. This selection is made independently at all levels. If an attribute at any level is repeated and has zero values, or an optional attribute is not present, then the attribute's value is taken to be the dummy instance; that is, all its descendant leaves are taken to be NULL. As mentioned in Section 2.3, this expansion using NULL's is necessary to avoid losing information contained in non-first-normal-form relations.

Our goal in this paper is not only to represent tree-structured tuples as rows of a relation. We next use an SQL-like language which evaluates a query on rows using the standard SQL evaluation algorithm. The output is a set of rows which may or may not come from a data tree on a certain schema. Here we investigate when this set of rows actually come from a data tree on the same schema as the input data tree. Since filtering (selection) queries can delete all occurrences of repeated or optional attributes, we are going to want to make explicit the effect of the dummy occurrences discussed in Section 2.3. Thus, we define the "full flattening" (or just "flattening" when there is no ambiguity) of a tree instance to include all the tuples that result when we include all dummy instances in the flattening. The full flattening is what we need to explain the effect of SQL queries on the trees themselves. To save space, we can remove those rows that are *subsumed* by another tuple of the flattened table. A row r_1 is subsumed by row r_2 if r_2 agrees with r_1 wherever r_1 is not NULL. We call this relation the *reduced* flattening of the tree.

More formally, if I is an instance of some schema, we define the (ordinary) relation flatten(I) recursively as follows.

1. If I is a single element of basic type, then flatten(I) is the tuple with a single component; that component is the value of I.
2. If I is an instance of some tuple type with attributes $A_1, \ldots, A_n$: Divide the children of the root of I into n groups, such that the first group is all the nodes that are occurrences of A_1, the second group is all the occurrences of A_2, and so on. For the ith group, construct a relation R_i that has attributes for all the leaves of the schema tree rooted at A_i, as follows:
   a) Recursively apply the flatten operation to the instance represented by each node in the group for A_i. However, if A_i is repeated or optional, include the dummy instance in this set of instances.
   b) Take the union of the relation produced for each instance. The union is the relation R_i.
1. Finally, to get the relation flatten(I), take the Cartesian product $R_1 \times R_2 \times \cdots \times R_n$.

The result is the *full flattening* of the given instance.

Example 3.1. *Let us see how to flatten the instance in Fig. 2, whose schema we saw in Fig. 1. First, let us observe that the relation schema for the flattened relation is*

*(*Name, Email, CID, Budget, Bid, Word, Fee, Date*)*

since these are the attribute names of the leaves of the schema, from the left. In Fig. 3 we present the flattening of the part of the instance with root a1*. The entire result is the union of the relation we get from* a1 *with the relations we get from all the other instances of Advertiser.*

The relation for ca1 *is the cartesian product of the relation for* i1 *(which is* $\{$*(*i1*)*$\}$*), the relation for bu*1*, the union of the relations for s*1 *and s*2 *and, the union of the relations for cl*1*, cl*2 *and cl*3*. The result appears in rows 1 through 25 of Fig. 3. Similarly, for* ca2*, the result appears in rows 26 through 30 of Fig. 3.*

Finally, to construct the relation for a1*, we take the product of the relations* $\{$*(*n1*)*$\}$*,* $\{$*(*e1*)*$\}$*, and the relation of Fig. 3. The result looks similar to Fig. 3, but there are two new attributes at the left, Name and Email, and the value of each of the 21 tuples of Fig. 3 has values* n1 *and* e1 *in those new columns.*

4. Filter Queries

A filter is a conjunction of comparisons $A\theta B$ where A is an attribute, B is an attribute or a constant and θ can be any comparison for which given two values the outcome is "true" or "false." This include inequality, $\neq$, and arithmetic comparisons in $\{\leq, <, \geq, >\}$.

4.1 Querying Flattened Data

If we imagine tree-structured tuples to be flattened, then there is a natural interpretation of SQL-like queries that refer to the attributes at the leaves of the tree: flatten the tuples and apply the query to the ordinary relation that results. There are two problems with this idea:

	CID	Budget	Bid	Word	Fee	Date
1)	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
2)	*i1*	*bu1*	*NULL*	*NULL*	*NULL*	*NULL*
3)	*i1*	*bu1*	*NULL*	*NULL*	*f1*	*d1*
4)	*i1*	*bu1*	*NULL*	*NULL*	*f2*	*d2*
5)	*i1*	*bu1*	*NULL*	*NULL*	*f3*	*d3*
6)	*i1*	*bu1*	*bi1*	*NULL*	*NULL*	*NULL*
7)	*i1*	*bu1*	*bi1*	*NULL*	*f1*	*d1*
8)	*i1*	*bu1*	*bi1*	*NULL*	*f2*	*d2*
9)	*i1*	*bu1*	*bi1*	*NULL*	*f3*	*d3*
10)	*i1*	*bu1*	*bi1*	*w1*	*NULL*	*NULL*
11)	*i1*	*bu1*	*bi1*	*w1*	*f1*	*d1*
12)	*i1*	*bu1*	*bi1*	*w1*	*f2*	*d2*
13)	*i1*	*bu1*	*bi1*	*w1*	*f3*	*d3*
14)	*i1*	*bu1*	*bi1*	*w2*	*NULL*	*NULL*
15)	*i1*	*bu1*	*bi1*	*w2*	*f1*	*d1*
16)	*i1*	*bu1*	*bi1*	*w2*	*f2*	*d2*
17)	*i1*	*bu1*	*bi1*	*w2*	*f3*	*d3*
18)	*i1*	*bu1*	*bi2*	*NULL*	*NULL*	*NULL*
19)	*i1*	*bu1*	*bi2*	*NULL*	*f1*	*d1*
20)	*i1*	*bu1*	*bi2*	*NULL*	*f2*	*d2*
21)	*i1*	*bu1*	*bi2*	*NULL*	*f3*	*d3*
22)	*i1*	*bu1*	*bi2*	*w3*	*NULL*	*NULL*
23)	*i1*	*bu1*	*bi2*	*w3*	*f1*	*d1*
24)	*i1*	*bu1*	*bi2*	*w3*	*f2*	*d2*
25)	*i1*	*bu1*	*bi2*	*w3*	*f3*	*d3*
26)	*i2*	*bu2*	*NULL*	*NULL*	*NULL*	*NULL*
27)	*i2*	*bu2*	*bi3*	*NULL*	*NULL*	*NULL*
28)	*i2*	*bu2*	*bi3*	*w4*	*NULL*	*NULL*
29)	*i2*	*bu2*	*bi3*	*w5*	*NULL*	*NULL*
30)	*i2*	*bu2*	*bi3*	*w6*	*NULL*	*NULL*

Figure 3: Tuples resulting from the full flattening of the dummy Campaign (row 1), *ca1* (rows 2–17) and *ca2* (rows 18–21) in data tree of Fig. 2

1. Flattening can expand greatly the amount of space needed to hold a tuple. For example, instances of the schema of Fig. 1 could have campaigns with hundreds of Words among its WordSets, and many thousands of Clicks. The flattened relation for a tree-structured tuple would then be hundreds of times larger than the original tuple.
2. When you flatten a tree and then apply some filter to the resulting relation, it is common for there to be no way to *prune* (delete nodes of) the original tree to yield a tree that would have produced the result of the filtering the flattened relation. In the study of nested relations, this problem is equivalent to the fact that while one can flatten any nested relation, it is not always possible to apply the inverse of flattening – the *nest* operator [10].

It is the purpose of this paper to resolve these two problems by:

a) Investigating when the result of filtering a flattened relation is what we get by pruning the tree and then flattening (i.e., when do pruning and flattening commute), and
b) Giving an algorithm to perform the filtering on the tree itself, whenever it is possible to do so.

Example 4.1. *To see the problem concretely, consider the schema in Fig. 4(a) and an instance of that schema in Fig. 4(b). The values of B and C are integers, but we give each occurrence of a B-value or C-value a name, such as b_1, to make clear which attribute, B or C, each integer comes from. Suppose we apply to the instance of Figure 4 the query*

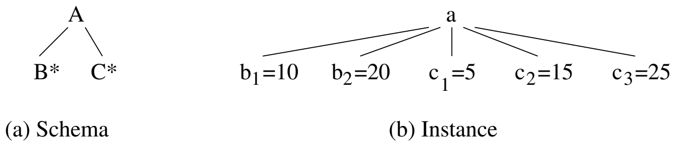

Figure 4: A schema and an instance

```
SELECT B, C FROM A WHERE B<C
```

We use this example to explain why we need a flattened relation where NULL's appear. We apply this query to the flattened version of Fig. 4(b), which is shown in Fig. 5(a). Note that this relation is a reduced flattened version, since we have not shown the tuples where one or both of B and C are NULL. However, in this case, the result would not change if we considered the full flattening. We shall see in Section 4.2 where it becomes essential to use the full flattening.

	B	*C*
1)	10	5
2)	10	15
3)	10	25
4)	20	5
5)	20	15
6)	20	25

(a) Reduced flattening

	B	*C*
1)	10	5
2)	10	15
3)	10	25
4)	20	5
5)	20	15
6)	20	25
7)	NULL	NULL
8)	10	NULL
9)	20	NULL
10)	NULL	5
11)	NULL	15
12)	NULL	25

(b) Full flattening

Figure 5: Reduced and full flattening of Fig. 4(b)

Notice that the second, third and sixth tuples satisfy the filter, while the others do not. Thus, the result of this query is shown in Fig. 6. However, this relation is not the flattening of any tuple with the schema of Fig. 4(a). To see why, notice that such a tree-structured tuple would have B-values 10 and 20 and also have C-values 15 and 25. But then, the flattening of the tree would also yield the tuple $(20, 15)$. We conclude that this SQL query cannot be executed on tree-structured tuples; it can only be executed on the flattened version of the tree, and the result has a schema different from the schema of the input tuples.

	B	*C*
2)	10	15
3)	10	25
6)	20	25

Figure 6: Result of `SELECT B, C FROM A WHERE B<C`

There is an interesting observation in Example 4.1: the fact that Fig. 6 cannot be the flattening of any instance of the given schema may depend on a simple change: If the schema of Fig. 4(a) had a * on the A, then we could view Fig. 6 as an instance of such a schema: this instance would have three occurrences of A, one for each of the three rows in Fig. 6.

However, even in the more common case where the schema has a * at the root and represents a relation with an arbitrary number of tree-structured tuples, we want to rule out the possibility of using more than one tuple to represent the result of a query applied to a single tuple. The motivation comes from the Dremel strategy for processing tree-structured tuples. That is, queries in Dremel are processed by pruning nodes from the trees, not by creating new trees. This strategy is efficient, simple, and avoids explosion in the size of the data (imagine if the tree of Fig. 4(b) were actually part of some much larger tree, which would have to be replicated for each of the three rows in the result).

4.2 Handling NULL's in Query Execution

Now, using the same schema and instance, Fig. 4, suppose we have the query

```
SELECT *
FROM A
WHERE B = 10 AND C = 35;
```

According to the tree-pruning algorithm, the data leaves that do not satisfy the filter will be deleted. The "AND" is interpreted as two filters $B = 10$ and $C = 35$ Thus the first component of the filter is checked on attribute B and the second component on attribute C The first component deletes $b2$ and leaves only $b1$ The second component deletes all leaves on attribute C. Thus the output data tree has only one leaf for attribute B and no leaves for attribute C. The flattening, thus, contains only the row (10 NULL).

But if we apply the query to the relation of Fig. 5(a), we get no tuples, since there are none with $C = 35$. Possibly, we could resolve the problem by starting with the full flattening, because that table has the necessary NULL's. The full flattening corresponding to Fig. 5(a) is shown in Fig. 5(b). We want row (8) to survive the filtering, since it represents the one remaining B-value, paired with the NULL that represents the lack of any C-children. But according to the SQL standard, the truth value of $C = 35$ is UNKNOWN when C is NULL, and thus, when applied to row (8), the truth value of the entire WHERE-clause is UNKNOWN. That is not "true enough," by the SQL standard, to reach the result of the query, so we still get the wrong answer.

The resolution to this dilemma, we believe, is to deviate from the SQL standard by allowing UNKNOWN to be "sufficiently true" to allow a row to reach the result of the query. If we do so, then rows 7 and 8 of Fig. 5(b) pass the filter. However, when we reduce the relation, row 7 is subsumed by row 8, so we get only the latter row as the answer.

5. The Dominance Relation

Now, we are going to show how to distinguish queries that can be implemented on the tree-structured tuples directly, from those that cannot. By "directly," we mean that each tuple is processed by pruning its tree, and not by creating several tuples from one; the distinction and its importance were introduced at the end of Section 4.1. We shall then give several approaches to implementing those queries that can be executed directly on the trees.

5.1 Motivation for the Dominance Relation

To begin, let us take a look at two queries on the schema of Fig. 1 that look almost the same, but in fact behave quite differently. Query Q_1 is:

```
SELECT CID
FROM Advertiser
WHERE Budget < Fee;
```

Now consider query Q_2:

```
SELECT CID
FROM Advertiser
WHERE Bid < Fee;
```

These two queries are very similar but they differ in an important way, so that Q_1 can be computed by tree pruning, while Q_2 can not. First, consider what happens when Q_1 is applied to flattened tuples. All rows that have a Fee no greater than the Budget in that row would be filtered out.

Consider the rows with CID = *i1*, like rows 2 through 25 of Fig. 3.[1] All those rows that have a value of Fee no greater than *bu1* will be filtered out. For example, suppose that *f2* is greater than *bu1*, but *f1* and *f3* are not. Then the even-numbered rows $2, 4, \ldots, 16$ survive, while the odd numbered rows $3, 5, \ldots, 25$ are filtered out. Rows 4, 8, 12, 16, 20, and 24 survive because *bu1*<*f2*, while rows 2, 6, 10, 14, 18, and 22 survive because the value of Fee in those rows is NULL, and we have adopted the convention that rows with value UNKNOWN for a filter condition pass the filter.

For the subtree rooted at *ca2*, which is represented by rows 18 through 21 in Fig. 3, notice that there are no Clicks, and so all these rows come from the dummy Click and have NULL for the value of Fee. Therefore, they all survive. The effect on the tree-structured instance of Fig. 2 is that the subtrees rooted at *cl1* and *cl3* are removed, but other than that, the tree, shown in Fig. 7, remains the same.

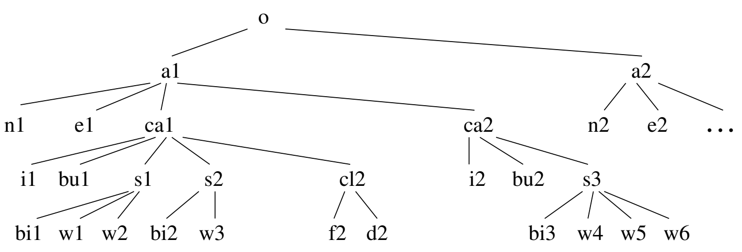

Figure 7: Result of eliminating Fees that are not greater than the Budget for their Campaign

It turns out that regardless of the instance to which it is applied, the effect of query Q_1 can always be implemented by pruning the tree for each tuple. On the other hand, we cannot normally do that for query Q_2. The difference is expressed by the concept of "dominance" between nodes of the schema tree.

5.2 Definition of the Dominance Relation

DEFINITION 5.1. *A path in a schema tree from an ancestor A to a descendant D is* star free *if none of the nodes on the path, with the possible exception of A, is repeated or required-and-repeated.*

[1]Recall that to become rows of the full Advertiser relation, these rows are padded with Name = *n1* and Email = *e1*, but these attributes are irrelevant to our discussion.

Definition 5.2. *An attribute A dominates another attribute B if, in the schema tree, the path from A to the lowest common ancestor (LCA) of A and B is star free.*

Example 5.3. *Consider the schema of Fig. 1. The lowest common ancestor of Budget and Fee is Campaign. The path from Budget to Campaign has no stars, except for the star at Campaign. Since Campaign itself is the LCA, its star is not considered part of the path. Since the path from Budget to Campaign thus is deemed to have no stars, we say that Budget dominates Fee. Fee does not dominate Budget. The reason is that the path from Fee to the LCA includes the node Clicks, which is starred. Now, consider the two attributes Bid and Fee involved in query Q_2. Again the LCA is Campaign. But now, Bid and Fee each have a star on their paths to the LCA, namely the nodes WordSet and Clicks, respectively. Therefore, neither dominates the other.*

For a final example, consider nodes Fee and Date. Their LCA is Clicks. Neither has a star on their path to the LCA; again, the star at the LCA itself does not matter. Therefore, Fee and Date each dominate the other.

The key observation to be made from Example 5.3 is: in any instance of the schema in Figure 1, there is only one Budget node in any subtree rooted at an instance of Campaign, the LCA of Budget, and Fee. This fact makes query Q_1 implementable by tree pruning. But for Q_2 a single instance of Campaign, which is also the LCA of Bid and Fee, can have multiple Bid descendants and also multiple Fee descendants. Since awkward combinations of the Bid and Fee descendants can survive the filtering, it is impossible, in general, to implement Q_2 by tree pruning.

5.3 Tree-Pruning Algorithm for Filter Queries

We can now give an algorithm for modifying the tree in the way suggested by Example 5.3. The algorithm works only for certain filter queries, but for this class of queries it produces a tree whose (full) flattening is the same as what we get by flattening the tree first, and then applying the query to the flattened relation. Moreover, in cases where this algorithm is inapplicable, the result of applying the filter to the flattened relation cannot, in general, be expressed as the flattening of a tree that is derived from the original tree by deleting nodes.

The class of filters allowed by the algorithm is those that are the AND of one or more comparisons. Each comparison is either:

1. A comparison involving only one leaf attribute (e.g., comparison between the attribute and a constant, or some user-defined predicate applied to only that attribute), or
2. A comparison involving two leaf attributes, one of which dominates the other.

If we have the AND of two or more comparisons of these types, we can apply one comparison at a time. As long as each comparison can be implemented by tree pruning, the cascade of pruning steps will result in a tree that satisfies all of the comparisons. Therefore, we shall describe only how to prune the tree for a single comparison.

For either type of comparison, there is a node-deletion step followed by a recursive deletion process for ancestors of the deleted nodes. We shall start with the initial deletion.
Case 1: If the comparison involves only one leaf attribute A, delete all leaves in the instance tree that are instances of A and that do not satisfy the predicate.
Case 2: If the comparison involves leaf attributes A and B, where A dominates B, let C be the LCA of A and B in the schema tree. In the instance tree, look at all occurrences of A and B such that the LCA of these nodes in the instance tree is an occurrence of C. (Intuitively, these are pairs of A- and B-nodes that are sufficiently closely related in the tree that they would be part of the same flattened tuple.) If the values of the A- and B-nodes in the instance tree are such that the comparison is not satisfied, then delete the B-node from the instance tree.

Now, having deleted certain nodes from the instance tree, we need to propagate these deletions up the tree. In particular, if we delete a required node, then we have to delete the entire subtree rooted at its parent. Also, suppose n is a node in the instance tree, and it has some children that are occurrences of some attribute A, which is of kind required-and-repeated. If all these children have been deleted, then n must also be deleted. These rules can propagate up the instance tree indefinitely.

Example 5.4. *Let us consider some possible comparisons involving the schema tree of Fig. 1 and the instance tree of Fig. 2. First, suppose the comparison involves Fee and Date, and we have chosen to regard Fee as dominating Date (note that in this case, either could have been chosen as the dominating attribute). The LCA of these attributes is Clicks. In Fig. 2, the pairs of Fee and Date that have a Clicks node as LCA are $(f1, d1)$, $(f2, d2)$, and $(f3, d3)$. Note, for example, that* f1 *and* d2 *are not compared, because their LCA is* ca1, *which is not a Clicks instance. Suppose further that the first two pairs satisfy the comparison, but $(f3, d3)$ does not. Then, because we have chosen to regard Fee as dominating, we delete the* d3 *node. Now, we must propagate the deletion upward. Because Date is a required child of Clicks, we must delete its Clicks parent,* cl3 *and its entire subtree. That deletion causes* f3 *to be deleted as well.*[2] *We next need to consider the parent of the newly deleted node* cl3. *That parent is the campaign occurrence* ca1. *However, Clicks is a repeated child of Campaign, the deletion of* cl3 *has no effect on* ca1*; the latter simply has one fewer Clicks child in the instance tree.*

The tree pruning algorithm can also handle Boolean formulas of comparisons. It does so by viewing them as a conjunction of disjunctions. The tree-pruning algorithm is modified for each conjunct (that now is not a single comparison but a disjunction of comparisons) as follows: if, for a certain assignment of data leaves at the attributes of the disjunction, the disjunction is not satisfied then, all these data leaves are deleted. For filter queries with filters that are Boolean formulas of comparisons, the following theorem states that if we apply the query Q to flattened data flatten(I) of I then we get in the output the flattening of the output $Q(I)$ of the tree-pruning algorithm.

Theorem 5.5. *Let I be an instance tree and let* flatten(I) *be the flattened relation of I. Then, for any query Q which uses a filter such that one attribute in a comparison dominates the other, the following holds:*

$$\text{flatten}(Q(I)) = Q(\text{flatten}(I))$$

[2]Note that had we chosen to regard Date as dominating Fee, we would have deleted *f3* first, but then would have deleted *cl3* and *d3*, leading to the same tree.

6. Semi-flattening and Repetition Context

In this section we introduce the concepts of semi-flattening and repetition context and then identify a class of filter and aggregate queries computed on semi-flattened data. Semi-flattening is the appropriate model for how Dremel processes tree-structured tuples. As described in [13] (and in Section 7 here), Dremel data is stored by columns (leaf attributes), with structure information to indicate how the values in a column are distributed throughout the tree. When processing a query, the needed columns are read, but at different rates so a value in one column may be "current" while many values of another column are read, one-at-a-time. The semi-flattened representation actually has one row for every step of this column-reading process. That is, every combination of attribute values that exists at some time during Dremel processing is represented by exactly one row of the semi-flattened table.

6.1 Repetition Context

The definition of semi-flattening is based on the dominance relation plus a new concept, the "repetition context." Actually we shall see in this section that if we confine the query to a single repetition context then flattening and semi-flattening coincide. Let us motivate these concepts with an example query that can be implemented by tree pruning but *not* by semi-flattening. Here is a query Q; it refers to our running example schema of Fig. 1:

```
SELECT CID
FROM Advertiser
WHERE CID <> Word AND Budget <> Fee;
```

This query can be answered by applying the tree pruning algorithm to each condition separately. However we shall see it is not answerable from the semi-flattened relation. The reason is there are two dominated attributes Word and Fee among the conditions, but neither of the two dominates the other. We begin by defining the class of queries for which we can apply semi-flattening. We say that all attributes should belong to the same "repetition context" which we define as follows.

Definition 6.1. *The* repetition context *of leaf attribute V, denoted C_V, is the set of leaf attributes that dominate V.*

Example 6.2. *In the query Q Word and Fee do not have the same repetition context. In particular the two contexts are:*

$$C_{Word} = \{Name, Email, CID, Budget, Bid, Word\}$$
$$C_{Fee} = \{Name, Email, CID, Budget, Date, Fee\}$$

As we shall see, the fact that these repetition contexts have an intersection that is not equal to one of them is what makes the query Q ineligible for the semi-flattening algorithm.

Lemma 6.3. *Let V be a leaf attribute and C_V its repetition context. Then the following hold:*

1. *The attributes in C_V can be put in a total order with respect to the dominance relation. That is, the members of C_V can be put in a sequence $V_1, V_2, \ldots, V_m$ such that V_i dominates V_j if $i < j$. Note that there may be several orders possible, since required or optional children of the same node can be placed in the sequence in any order.*
2. *Suppose in some tree schema, V is a leaf and U is a member of C_V. Further, let X be the LCA of V and U, and let Y be any node in the schema tree on the path from U upward to X. Then if T is a subtree of an instance tree that is rooted at an occurrence of Y, then there is only one occurrence of U in T.*

Proof. To prove the first part of the lemma we observe that we can find all attributes that dominate V if we do the following: We focus on the path (in the schema tree) from V to the root, and we call it the *primary path.* If V' dominates V, then the LCA of V and V' is on the primary path, and the path upward from V' to the primary path is star free (except possibly for the node on the primary path). Thus for two attributes that dominate V, the one that meets the primary path higher dominates the other. If they meet the primary path at the same node, then they dominate each other. The proof of the second part of the lemma is a consequence of the fact that there is a star free path from U to Y. □

Example 6.4. *For the schema in Figure 1, the primary path for Budget includes Campaign and Advertiser. In the repetition context of Budget we have Name, Email and CID.*

The primary path for Fee includes Clicks, Campaign, and Advertiser. All leaf attributes belong to the repetition context of Fee except for the attributes Word and Bid. The reason is that both Word and Bid encounter WordSet before their path to the root reaches Campaign (which is on the primary path of Fee). Actually Word would not be in the repetition context of Fee even if the schema were modified by making WordSet required, because Word is itself repeated.

6.2 Semi-Flattening

The properties that render semi-flattening important are stated formally in lemmas 6.7,6.8 and 6.9 where it is shown that, for special schemas, unlike full flattening, uses a number of NULLs in dummy occurrences of attributes that only depends on the size of the schema, not on the size of the data tree. The idea of semi-flattening is illustrated in the following example.

Consider a schema consisting of a root with three children that are leaves: one required (A), and two repeated (B and C). Suppose the data tree has value a for A, $b1$ and $b2$ for B, and $c1$, $c2$, and $c3$ for C. We can represent this information by the following four rows (instead of the 12 rows that would appear in the full flattening):

a	b1	c1
a	b2	c2
a	NULL	c3
a	NULL	NULL

The way we formed these rows, which we shall see is an example of semi-flattening, is by focusing on having all combinations of a with each of the values of B and C. Since B and C are repeated, we need the dummy NULL for each. In fact, since there are fewer B's than C's, we used NULL twice in the B column so each row has a B-value. The combinations of B and C are random, in the sense that $b1$ and $c1$ appear in the same row, but we could just as well have chosen to pair $b1$ with $c2$ or $c3$ instead.

But, now we can not compute a tree query whose filter has two comparisons, one comparing A and B and the other comparing A and C. The reason is that when we delete rows, because a B value fails to satisfy the comparison, we may delete a C value that should not to be deleted, just because it happens to be in the same row.

However, as we will explain next, a quite broad subclass of tree queries can be computed using semi-flattened data. Formally, if I is an instance of some schema, we define the (ordinary) relation s-flatten(I) recursively as follows.

1. If I is a single element of basic type, then s-flatten(I) is the tuple with a single component; that component is the value of I.
2. If I is an instance of some tuple type with attributes $A_1, \ldots, A_n$: Divide the children of the root of I into n groups, such that the first group is all the nodes that are occurrences of A_1, the second group is all the occurrences of A_2, and so on. For the ith group, construct a relation R_i that has attributes for all the leaves of the schema tree rooted at A_i, as follows:
   1. Recursively apply the s-flatten operation to the instance represented by each node in the group for A_i. However, if A_i is repeated or optional, include the dummy instance in this set of instances.
   2. Take the union of the relation produced for each instance. The union is the relation R_i.
3. Finally, to get the relation s-flatten(I), take a "horizontal concatenation" of $R_1, R_2, \ldots, R_n$ as follows. The first row of the result is the concatenation of the first rows of $R_1, R_2, \ldots$; the second row of the result is the concatenation of the second rows of $R_1.R_2, \ldots$, and, in general, the ith row of the result is the concatenation of the ith rows of $R_1, R_2, \ldots$. Of course the R_j's may not have the same number of rows. In this case we pad the short tables with extra rows that contain NULL's.
4. As an exception to the matter mentioned above for padding short tables with NULL's, for each attribute that has a star free path to the current root (the root excluded) we keep its value. Since it has a star free path to the root, it has only one value in all the rows.

The result is the *semi-flattening* of the given instance.

Example 6.5. *Consider the schema and the data in Figure 8. The values, denoted by lowercase letters, correspond to attributes with the corresponding uppercase letter.*

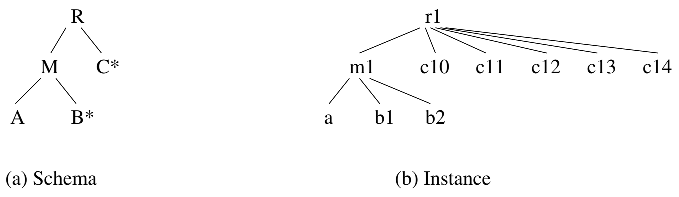

Figure 8: Schema and Data

The root in the schema has two attributes as children, M and C. In the instance, the occurrence $r1$ of the root has one occurrence of M and five occurrences of C. We get the union of the five occurrences of C and get the column in Fig. 9(a). We call this relation F_C. Then we consider the instance subtree with $m1$ as root. Its relation, which we call F_M, is shown in Fig. 9(b). When we horizontally concatenate F_M and F_C, we get the semi-flattened representation for the entire instance, which is shown in Figure 9(c).

c10
c11
c12
c13
c14
NULL

(a) Table for C

a	b1
a	b2
a	NULL

(b) Table for M

a	b1	c10
a	b2	c11
a	NULL	c12
a	NULL	c13
a	NULL	c14
a	NULL	NULL

(c) Table for R

Figure 9: Construction of semi-flattening for data in Fig. 8

Theorem 6.6. *The number of rows in the semi-flattening of instance I is at most equal to the product of the number of leaves in the data tree multiplied by the depth of the schema tree and by the number of leaves in the schema tree.*

Call a schema *linear* if the only non star free path is a single path from the root to a single leaf; we call this leaf the *primary leaf.*

Lemma 6.7. *For any data tree in a linear schema, full flattening and semi-flattening coincide.*

Proof. The first observation is that when we have a linear schema in each inductive step of the definition of flattening (full or semi) we have only one regular relation and the other children have degenerate relations with only one tuple for each such relation. The second observation is: The only difference between the definitions of full flattening and semi-flattening is: in full flattening we have a cartesian product whereas in semi flattening we have a horizontal concatenation. In the case of linear schema, the cartesian product in the definition of full flattening reduces to the horizontal concatenation in the definition of the semi-flattening. □

Lemma 6.8. *The schema subtree of any repetition context is a linear schema.*

Lemma 6.9. *The number of NULLs that appear in dummy instances of repeated attributes in semi-flattening is at most quadratic on the number of nodes in the schema.*

Semi-flattening can be used in a limited way for filter queries. For instance, in Example 6.5, suppose the filter compares attributes A and B, and suppose value $b1$ is filtered out, then the output of the filter will be $(a, b2, c11)$. It is still useful information because it give us the correct answer if we have only attributes A and B in the SELECT clause. This is also true if the filter is on attributes A and C and we have only A and C in the SELECT clause. Its usefulness is made formal in the following lemma which is a consequence of Lemma 6.7:

Lemma 6.10. *If we restrict the semi-flattened data only to the columns that comprise a repetition context C (in which case, they can be thought of as being on a linear schema), then the set of rows that we get is the same as the set of rows we get if we restrict full-flattened data (of the same data tree) to repetition context C.*

The way we answer queries in context C on semi-flattened data is: a) first we restrict the columns to only the columns of C and b) then we answer the query as we explained in earlier sections. Hence, the above lemma and Theorem 5.5 give us the following theorem:

THEOREM 6.11. *Let I be an instance tree and let s-flatten(I) be the semi-flattened relation of I. Then, for any query Q which uses a filter in a single repetition context the following holds:*

$$s\text{-}flatten(Q(I)) = Q(s\text{-}flatten(I))$$

6.3 Aggregate Queries

The aggregate functions we consider are SUM, MAX, MIN, COUNT, AVG and COUNT-DISTINCT, under the following constraints:

(a) All aggregated attributes should be dominated by all grouping attributes, and

(b) The SELECT clause should include only the grouping attributes and the aggregation(s).

Note that (a) implies that the repetition contexts of the aggregated attributes have an intersection which contains the grouping attributes. When these constraints are met in queries, we call them *legitimate aggregate queries.*

We give the algorithm to compute an aggregate query with grouping attribute $A1$ and aggregated attribute $A0$, where $A1$ dominates $A0$. The output will be a normal relation with two attributes, one attribute is the grouping attribute $A1$ and the other is a new attribute A_{agg} which stores the result of applying the aggregate function on bags, one bag for each value of $A1$. This is the description of the *tree-aggregating algorithm* that does the computation:

1. Suppose the attribute that is the lowest common ancestor of $A1$ and $A0$ in the schema tree is $A01$. For each value u of $A1$, let $\{v1, v2, \ldots, \}$ be all nodes in the data tree with value u, and let $\{v01, v02, \ldots, \}$ be the "corresponding" values of attribute $A01$ (i.e., $v1$ has ancestor $v01$, $v2$ has ancestor $v02$ and so on).
2. For each value u of $A1$, we form a bag of values of the aggregated attribute $A0$. This bag stores all values for each data leaf which is a) an occurrence of $A0$ and b) is a descendant of a node in $\{v01, v02, \ldots, \}$.
3. Then we aggregate over the values in each bag (which corresponds to a value of $A0$) and store the result in the new attribute A_{agg}.

Of course, we need not form bags explicitly. We compute the aggregation function on the fly, except for the average function, where we need to compute both count and sum on the fly and divide at the end and the count-distinct function where we need to compute a set instead of a bag.

EXAMPLE 6.12. *We refer to the data tree in Figure 2 and we consider the query:*

```
SELECT Budget, SUM(Bid)
FROM Advertiser
GROUP BY Budget;
```

If $bu1 \neq bu2$*, the answer to this query is on the left below. However, if* $bu1 = bu2$ *then the answer to the query is on the right (it contains only one row):*

Budget	*SUM(Bid)*
bu1	*bi1+bi2*
bu2	*bi3*

Budget	*SUM(Bid)*
bu1	*bi1+bi2+bi3*

When there is more than one grouping attribute, there is at least one grouping attribute that is dominated by all other grouping attributes; call one of them arbitrarily the *most dominated attribute.* We form one bag for each tuple of values of the grouping attributes. In this case, the computation is led by the most dominated attribute as to which subtrees we consider for all their aggregated attribute values to go in the same bag. That is, the tree-aggregating algorithm considers the lowest common ancestor of the aggregated attribute and the most dominated attribute.

All legitimate aggregate queries can be conceptually computed on semi-flattened data. The way to compute them is: First NULLs are ignored. Second, for MIN and MAX we apply standard SQL semantics. However, for SUM and other duplicate-sensitive aggregate functions, we need to be more careful. We observe that flattening (full or semi) may use the same data leaf in more than one rows. E.g., in Figure 10, we have the semi-flattening of Figure 2 and we observe that $bi1$ appears in 3 rows, whereas in an aggregate with *Bid* as the aggregated attribute we should take $bi1$ only once. Thus we need to do duplicate elimination in that sense. Conceptually, this is achieved by adding a new attribute for each aggregated attribute, we call this attribute *Tag*. The value of *Tag* is either 0 (meaning we ignore this value of the aggregated attribute because it is a duplicate) or 1 (meaning we include this value). The value of *Tag* is 1 in a row of semi-flattened data if the value of the aggregated attribute in the query is the value of data leaf a and it is the first row (we conveniently imagine a total order on the rows) where the value of data leaf a appears in the semi-flattened data. Otherwise it is 0. Thus, when we compute the aggregate function, we compute it on two grouping attributes: the grouping attribute we started with and the additional (rather trivial) one, which we constrain to have value equal to 1. This does not introduce any additional complication, because, as we will explain in Section 7, we store the data in columnar storage in such a way that when we retrieve them it is easy to treat this extra attribute implicitly without having it being materialized.

EXAMPLE 6.13. *We compute aggregate query of Example 6.12 on semi-flattened data. Semi-flattened relation of the data tree in Figure 2 is in Figure 10. We do not record all the tags (in the first column here), only the one that refer to the aggregated attribute in our query, i.e., for Bid.*

Tag	CID	Budget	Bid	Word	Fee	Date
1	*i1*	*bu1*	*bi1*	*w1*	*f1*	*d1*
0	*i1*	*bu1*	*NULL*	*NULL*	*f1*	*d1*
0	*i1*	*bu1*	*bi1*	*w2*	*f2*	*d2*
0	*i1*	*bu1*	*NULL*	*NULL*	*f2*	*d2*
0	*i1*	*bu1*	*bi1*	*NULL*	*NULL*	*NULL*
1	*i1*	*bu1*	*bi2*	*w3*	*f3*	*d3*
0	*i1*	*bu1*	*NULL*	*NULL*	*f3*	*d3*
0	*i1*	*bu1*	*bi2*	*NULL*	*NULL*	*NULL*
0	*i1*	*bu1*	*NULL*	*NULL*	*NULL*	*NULL*
1	*i2*	*bu2*	*bi3*	*w4*	*NULL*	*NULL*
0	*i2*	*bu2*	*bi3*	*w5*	*NULL*	*NULL*
0	*i2*	*bu2*	*bi3*	*w6*	*NULL*	*NULL*
0	*i2*	*bu2*	*bi3*	*NULL*	*NULL*	*NULL*
0	*i2*	*bu2*	*NULL*	*NULL*	*NULL*	*NULL*

Figure 10: Semi-flattening for the data tree in Fig. 2; the Tag is with respect to Bid.

If $bu1 \neq bu2$*, then for grouping-attribute value* $bu1$ *we use the first and the third rows and have as a result of the aggregation* $b1+b2$*. For the value* $bu2$*, we use only the fourth* *column and have as a result* $b3$. *If, however,* $bu1 = bu2$, *then we have only one tuple in the result with values* $bi1+bi2+bi3$.

6.3.1 Filter And Aggregate Queries

So far we have discussed aggregate queries without filters. We can also have a filter in the query but we allow comparisons among the grouping attributes only. Because of Lemma 6.3 any comparison is guaranteed to be among two attributes where one of them dominates the other attribute.

The computation algorithm now, applies first the filter by using the tree pruning algorithm and in the output data tree applies the tree-aggregating algorithm to obtain the final output. Semi-flattening can be used to compute aggregate queries with filters. The following theorem is a straightforward consequence of the definitions of the algorithm for computing aggregates and semi-flattening. By $Q(I)$ we mean the output of the query when we apply the tree pruning followed by the tree-aggregating algorithm on the data tree.

THEOREM 6.14. *Let* I *be an instance tree and s-flatten*(I) *be the semi-flattened relation of* I. *Then, for any legitimate aggregate-and-filter query* Q, *the following holds:*

$$Q(\textit{s-flatten}(I)) = Q(I)$$

7. Efficient Data Storage and Retrieval

Dremel is designed to deal with relations that have many attributes, only a few of which are referenced in any one query. In such an environment, columnar storage is the appropriate structure, and Dremel is optimized for when the data data is stored column-wise. We explain in this section:

- How Dremel stores a data tree in columnar storage in a space-efficient way.
- How to retrieve the data from the columnar storage to implement legal Dremel SQL queries.

We shall refer to a schema and its instances by its sequential description, thus maintaining an order that will be implicit when we store the data in columnar storage. Thus the schema in Figure 8 is referred to as $R\{M\{A, B*\}, C*\}$. The data in Figure 8 is referred to as $R\{M\{A : a, B : b1, b2\}, C : 10, 11, 12, 13, 14\}$

For a given data schema, we view columnar storage as a number of lists (it is rather a stack because of the way it is retrieved), one list for each column/attribute. Given a data tree over the schema, we store the values of the leaf attributes in the appropriate column/list. The columns are synchronized. This is not sufficient information to retrieve the data tree for the purpose of computing queries in the class we described. So, together with the value of each leaf node we store its repetition and definition levels. They tell us how to traverse the data tree to go from an occurrence of leaf attribute A to its next occurrence in the tree. E.g., in Fig. 2, the next occurrence of attribute Word after $w2$ is $w3$. In order to go from $w2$ to $w3$ in the data tree, we have to go up to $ca1$ (which is an occurrence of Campaign, so we say that the repetition level of $w3$ is Campaign), and then travel down to $s2$ and $w3$ is a child of an occurrence of WordSet.

7.1 Repetition Level

- The *repetition level* of a data leaf v is the attribute name of the lowest common ancestor of v and the previous data leaf stored it its column (i.e., the previous leaf with the same attribute as v). By convention when a leaf is the first for its attribute in the record, its repetition level is *root*.

CID	Budget	Bid	Word	Fee	Date
i1 r *i2 Ad*	*bu1 r* *bu2 Ad*	*bi1 r* *bi2 Ca* *bi3 Ad*	*w1 r* *w2 WS* *w3 Ca* *w4 Ad* *w5 WS* *w6 WS*	*f1 r* *f2 Ca* *f3 Ca*	*d1 r* *d2 Ca* *d3 Ca*

Figure 11: Columns and repetition levels

EXAMPLE 7.1. *For the tree in Figure 2, the values are stored in 6 columns, for CID, Budget, Bid, Word, Fee, and Date. The column CID stores the values* $i1, i2$, *the column Budget stores the values* $bu1, bu2$, *etc. Columns and repetition levels are shown in Fig. 11 (we use* r *for root,* Ad *for Advertiser, etc.). E.g., the repetition level for* $w2$ *is* $WordSet$ *because the LCA of* $w2$ *and* $w1$ *(its previous element in the column) is* $s1$ *(an occurrence of* $WordSet$*).*

THEOREM 7.2. *The repetition level suffices to reconstruct the data tree if for each occurrence of an attribute, there is at least one occurrence (in the data tree) for each of its children (in the schema tree).*

7.1.1 Producing the Semi-flattening

When the conditions of Theorem 7.2 are met, we can build semi-flattening of a data tree stored in columns by using only the repetition level. We describe here the algorithm.

Each time we read the next element from a column, we will say that this column *makes a move*. We produce a semi-flattening of a relation from its columns by selecting, for each row produced, from which columns we should read a new value and from which columns we should take the current value. There is a *reader* that gets a new row of the semi-flattening whenever it is called upon to do so by the query-processing engine. The algorithm by which the reader decides whether to use the current value from a column V or to move to the next value in its column is as follows:

1. As long as there is a column dominated by V whose current repetition level does not go above or at the repetition level of V, the reader for V remains in the same place and outputs in each constructed row the current value of V.
2. Otherwise, if all its dominating attributes move to current repetition level, it goes to step 3 below. If not, it contributes NULLs (and repeats this step).
3. If V is a required attribute, the reader first produces an extra row with the current values in column V and in all the columns dominating V, whereas all other columns have NULL's. If V is not required it does not produce this row. In either case, the reader then moves to the next value in the column for V.

EXAMPLE 7.3. *We consider the data tree in Fig. 2. For better insight, we confine ourselves to one repetition context, the repetition context of Word. Then we can ignore Fee and Date and we get the semi-flattening of the rest, shown in Fig.12 (N stands for NULL) which also is part of Fig. 10.*

Figure 12 has the attributes of the repetition context in the "correct" order as concerns dominance; i.e., CID dominates

	CID	Budget	Bid	Word
1)	*i1 r*	*bu1 r*	*bi1 r*	*w1 r*
2)	*i1*	*bu1*	*bi1*	*w2 WS*
3)	*i1*	*bu1*	*bi1*	*N*
4)	*i1*	*bu1*	*bi2 Ca*	*w3 Ca*
5)	*i1*	*bu1*	*bi2*	*N*
6)	*i1*	*bu1*	*N*	*N*
7)	*i2 Ad*	*bu2 Ad*	*bi3 Ad*	*w4 Ad*
8)	*i2*	*bu2*	*bi3*	*w5 WS*
9)	*i2*	*bu2*	*bi3*	*w6 WS*
10)	*i2*	*bu2*	*bi3*	*N*
11)	*i2*	*bu2*	*N*	*N*

Figure 12: Rows generated, in order

Budget, Budget dominates Bid, and Bid dominates Word. The repetition levels are also shown in Fig. 12. We will show how to use repetition level to produce the semi-flattening in Fig.12. The first row in Fig.12 is formed by the top elements in each column.

Producing the Second Row

The repetition level for each of the second elements in columns Bid and Word stays below[3] *Advertiser, which is the repetition level of the second element in columns CID and Budget. Thus these two columns stall (according to step 1 of the algorithm) and emit $i1$ and $bu1$, respectively, the second time the reader is called. The second element in column Word has repetition level WordSet, which is below the repetition level of the second element in column Bid (which is Campaign). Hence for the second row, column Bid stalls too and emits $bi1$. Column Word is allowed by steps 1 and 2 of the algorithm to go to step 3 and make a move. Thus second row is formed.*

Producing Rows 3 and 4

$w3$'s repetition level is Campaign, and so is the repetition level of $bi2$. So, since Bid dominates Word, the column Bid can now make a move to $bi2$ because all its dominated columns (actually, only the one column Word) have repetition levels at or above its own repetition level (step 1). Bid is required and it forms the extra row before moving to the next element (this is row 3). Row 4 includes the new values after the moves that are allowed at this stage. Columns CID and Budget still stall since the repetition levels of their next elements ($i2$ and $bu2$) are above Campaign (which is the current repetition level of some of its dominated attributes).

Producing Rows 5, 6 and 7

Next, all current (i.e., for data values $i2$, $bu2$, $bi3$ and $w4$) repetition levels are at or above Advertiser, so columns CID, Budget and Bid move to their next element. So does column Word. Columns 5 and 6 are the extra columns created by the move of CID, Budget and Bid (they are all required attributes). Column 7 contains the new values.

Producing Rows 8, 9, 10 and 11

Rows 10 and 11 are the extra rows. Rows 8 and 9 are formed by Word making one move for each column and CID and Budget stalling.

[3]we use "below" and "above" to mean descendant and ancestor respectively

Interesting observation: Observe how in each row in Fig.12, the repetition levels from left to right are in non-ascending order, e.g., in row 1, we have (r,r,r,r), in row 2 we have (r,r,r,ws), in row 3 we have (r,r,Ca,-), etc. This is always the case if we confine ourselves to a single repetition context.

Finally, the restriction in step 2 of the algorithm only manifests itself if we go beyond a single repetition context. E.g., imagine that $w3$ had five siblings. It that case, for five rows, columns Fee and Date would emit NULL.

We say that a data leaf v *covers* another data leaf u if the attribute V for v dominates attribute U for u, and data nodes u and v are descendants of the same occurrence of their LCA in the schema tree. When a data leaf covers another data leaf, the repetition level of the former is the same as or above the repetition level of the latter. For instance, $b1$ covers $w1$ and $w2$.

The algorithm we presented to produce semi-flattening from columnar storage only retains the covering relation. It does not care, for example, to show any interrelationship between data leaves for Fee and Word, because in the class of queries supported by semi-flattening, we do not have a query where values of Word and Fee are compared or where Word is the grouping attribute and Fee is the aggregated attribute.

Finally, a note about the functionality of the extra row that the algorithm creates before a column makes a move. An example of such an extra row in Fig. 12 is (*i1 bu1 NULL NULL*). Suppose this row did not exist. Then if neither *bi*1 nor *bi*2values of the Bid attribute satisfy the filter, the values *i1* and *bu1* would not appear in the result of the query. This is wrong according to the tree-pruning algorithm. It is worth mentioning here that this extra row is in accordance with discussion Sec. 2.3 where we mentioned how we treat NULLs (and why we treat them this way). The correctness of the algorithm is a consequence of the lemma:

LEMMA 7.4. *If leaf v covers leaf u in the data tree, then the repetition level of u is either the same as the repetition level of v, or a descendant of the repetition level of v.*

7.2 Definition Level

The definition level is a second parameter stored along with the repetition level. Its purpose is to avoid having to store NULL's explicitly in the columns. The *definition level* tells how many subtrees between the current value and the previous value of the same column have zero occurrences. We must specify the root of these subtrees for attribute A. The root, denoted LCA_A, is the "closest" nonleaf attribute to A which has a descendant that dominates A.

LEMMA 7.5. *Let B be an attribute that dominates A. Let $LCA(A, B)$ be the LCA of B and A. Suppose a data leaf v of A and a data leaf u of B occur within (i.e., are descendants of) the same occurrence x of $LCA(A, B)$. The repetition and the definition levels are sufficient to tell that v and u are descendants of the same occurrence of $LCA(A, B)$.*

Producing the Semi-flattening

The algorithm in Sec. 7.1 is now modified so that whenever a column A is due for a move, it stalls for as many moves of star-free descendants of LCA_A as tells the definition level of A. Given a query Q, we call the attributes appearing in the query *relevant* attributes of Q. Given a semi-flattening of an instance I and a query Q, suppose we delete from the semi-flattening all columns of attributes that are not relevant to Q. What remains may contain duplicate rows, which we delete and obtain the *relevant semi-flattening* of Q.

THEOREM 7.6. *The data retrieval from columnar storage (of a data instance I) as described in this section for the purposes of computing a query Q requires a number of data-value retrievals that is at most equal to the number of rows in the relevant semi-flattening of Q multiplied by the number of relevant attributes.*

Finally, it is easy to observe that by the techniques explained in this paper, we can handle queries on multiple repetition contexts as long as these contexts are pairwise disjoint.

Experiments We run experiments to compare the processing time of a query on flattened data versus semi-flattened data. First, we use artificial schema and data: the schema of Fig. 1 and the data tree of each record is of rather small size, similar to Fig. 2. We run: a) Filter query Q_1 in Sec 5.1. The ratio of the processing times was 1.3:1. Observe that for the relevant attributes of this query flattening and semi-flattening coincide. b) Aggregate query like in Example 6.12 only with two aggregated attributes. The ratio is 4:1. The second set of experiments uses Google logs data and query is from Google production: we had a larger schema where the tree is of depth 6 and some of the attributes having as many as a few tens children. Each record had often as many as a few hundreds data leaves and rarely a few thousands. Then, depending on the number of repetition contexts we have a remarkable improvement (as expected): a) 2 contexts, ratio is 7:1, b) 4 context, ratio 25:1, c) 5 contexts, ratio 250:1.

8. Conclusions and Related Work

We presented the data model and query language of Dremel, a query engine used in Google. We defined the conditions for queries to be legitimate, introducing, for this purpose, the dominance relation. Dremel was first discussed in [13] where the focus was on the systems part of the engine. The schema and the data tree in Dremel are described as protocol buffers. The data tree is stored in columnar storage associating with each data value the two levels. When a query is issued, the data of the relevant (to the query) columns are retrieved in semi-flattening format, and a standard SQL evaluation algorithm is applied to compute the query. Thus the syntax of the Dremel language is SQL syntax, the evaluation algorithm on the data tree is the tree-pruning and the tree-aggregating algorithm, but the actual computation is carried out as standard SQL computation.

There is a large body of work about tree-like structures, storing and querying them. Much of it stems from XML documents, and the query languages examined are fragments of XQuery, not SQL. Dominance relation and its manifestation in allowing semi-flattening are introduced for the first time in this paper. A flattening technique is proposed in [12] to map XML data into relational tables. [19] builds on[12] and proposes three approaches to map efficiently an XML document to a relational database. Other approaches have been used, e.g., [14] is native algebraic based. As concerns columnar storage, [9] supports a uniform interface for querying efficiently both the structure and the data values for highly regular data. Techniques to avoid storing NULLs are developed.

9. References

[1] json. *http://www.json.org/*.

[2] Protocol Buffers. *https://code.google.com/p/protobuf/*.

[3] XPath. *http://davidezechukwu.com/docs/xpath.pdf*.

[4] XQuery. *http://www.w3.org/TR/xquery/*.

[5] S. Abiteboul, C. Beeri, M. Gyssens, and D. V. Gucht. An introduction to the completeness of languages for complex objects and nested relations. In *NF^2*, pages 117–138, 1987.

[6] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchical organized data. In *PODS*, pages 191–200, 1984.

[7] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[8] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *VLDB*, pages 263–269, 1982.

[9] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large xml repositories. In *ICDE*, pages 261–272, 2005.

[10] L. S. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD Conference*, pages 273–283, 1989.

[11] V. Deshpande and P.-Å. Larson. The design and implementation of a parallel join algorithm for nested relations on shared-memory multiprocessors. In *ICDE*, pages 68–77, 1992.

[12] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[13] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[14] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of xquery. In *SIGMOD Conference*, pages 71–82, 2004.

[15] J. Paredaens and D. V. Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *PODS*, pages 29–38, 1988.

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

[17] M. H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *VLDB*, pages 137–146, 1987.

[18] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[19] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.