Dealing with (un)structuredness in XML Data and Queries Using Relational Databases

Rajasekar Krishnamurthy **Jeffrey F. Naughton**

Jayavel Shanmugasundaram*

University of Wisconsin-Madison

(sekar,naughton,jai)@cs.wisc.edu

Eugene Shekita

IBM Almaden Research Center

shekita@almaden.ibm.com

Abstract

An XML database can contain documents with varying degrees of schema information. The queries can also range from fully specified structured SQL like queries to partially specified regular path expression queries. Relational databases are widely used to store and query XML data and various schemes have been proposed to this end. They either use DTDs or assume schemaless data. We show how more advanced schema information (like XMLSchema), even if partially available, can be used effectively to answer queries. We also show how the interaction between the amount of schema information available and the query workload plays an important role in choosing a decomposition strategy into relational tables, suited to that workload. We propose a simple cost metric for this purpose. Our experiments indicate that this metric can provide a reasonable choice among the different alternatives. We also show that using the schema at query time, irrespective of the decomposition strategy, can benefit considerably in improving query response times.

1 Introduction

Data used by many applications can be broadly classified into structured and unstructured data. Structured data, like data in relational databases, includes data that corresponds to a pre-defined

*Also at IBM Almaden Research Center

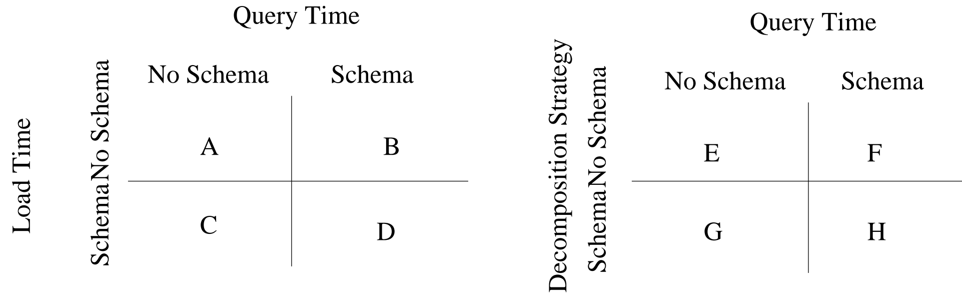

Figure 1: Alternate scenarios for presence of schema

schema. On the other hand, a typical example of unstructured data is the collection of documents found on the internet that do not have a common schema. The advent of XML has brought semistructured databases, whose structure may be known in advance to varying degrees, in the limelight.

The queries in structured databases (e.g. SQL queries) tend to be fully structured queries, where the complete information about the data required is specified and there are no wild-cards in the query. The queries in unstructured databases, like regular path expression queries, tend to involve quite a few wild-cards as no schema information is usually available. XML query languages tend to have a mixture of both flavors and provide a lot of flexibility in framing queries.

There has been a lot of work in using relational databases to efficiently answer queries on XML documents. This work has been in the two extremes: a lot of work has been done on storing XML in RDBMS, without using any schema information [4, 5, 12]. There has also been some work on using DTD information to decide on a mapping [10, 9] which can provide better performance in many cases. The main advantage of the former approaches is that they treat all XML data uniformly, irrespective of the amount of schema information available, while the latter approaches require a DTD to be present. Also, using the DTD can result in a large number of relations that can be avoided in some of the schemaless approaches. In this paper, we show how the efficient use of schema information that goes beyond DTDs, like XMLSchema [15], can help in answering queries. We propose 3 decomposition strategies, *xschema, kxschema and kxanyschema* that use different amount of *XSDL* schema information and discuss some of the benefits that can be leveraged from these additional features. While *xschema* uses the main features of an *XSDL* schema, *kxschema* uses integrity constraints effectively and *kxanyschema* can work with partial schema information and treats all XML documents uniformly, using schema information where available.

The presence of a schema can present various alternatives when the XML documents are shred-

ded into relational tables. It can also play a vital role at query time. Let us look at the various alternatives in Figure 1. The first table shows the various alternatives at load time and query time. The second table shows the interaction between the decomposition strategy and querying. The schema, if present, may be partial (through the use of ANY elements). While quadrants A and D are common, it is possible that schema information, which wasn't available at load-time, becomes available later or schema inference provides a partial schema during query time resulting in quadrant B[1]. When the schema is not present at load-time, like in quadrants A and B, we are required to use one of the schemaless approaches (quadrants E and F).

If some schema information is available at load-time (like quadrant D), we have the option of either using the information or using schemaless approaches. As a result, we might end in one of quadrants E, F or H in the second table. The choice at this point depends on the query workload to be satisfied. Structured approaches like [10] are good in answering structured SQL like queries while some of the unstructured approaches like [5] will start performing better than the structured approaches as wild-cards are introduced in the query. We show how the schema information can be used effectively to improve query response time for all techniques, irrespective of whether they used schema information during load-time (quadrants F and H).

There has been the general understanding that queries that specify the entire path expression for satisfying elements (SQL like queries) are *structured* queries while queries with a lot of wild-cards (regular path expression) are *unstructured* queries. We believe that the *structuredness* of a query cannot be estimated by just looking at the query. The *structuredness* of a query will depend on the schema of the source documents. Queries that look *unstructured* may actually have an equivalent *structured* query for a given schema. So, the *structure* of a query needs to be determined based on the schema, if present. Also, the nature of queries that can be asked on a given dataset is orthogonal to the decomposition techniques used. So, it is possible to have varying degrees of structure in the queries, while the decomposition techniques themselves may vary in *structure* based on the amount of schema information they use. So, we need a way to quantify the effectiveness of each decomposition technique for a given query workload. We propose a simple metric, $RelativeCost_m(q)$ that gives an estimate of the cost of executing query q for method m. This can be used to compare the performance of various decomposition techniques and choose a reasonable alternative. This technique can be used in two possible scenarios

[1] Note that quadrant C is unlikely to occur in practice, while quadrant G cannot occur as the decomposition strategy will have the schema associated with it

- A couple of relational decomposition strategies, using varying amount of schema information, can be built for the data. For a given query, the optimizer can compute the *RelativeCost* for each strategy and make a good choice.
- Given a schema and a sample query workload, the *RelativeCost* of the queries can be used to suggest a good relational decomposition strategy suited to the given query workload.

In Section 2, we define certain concepts related to the *structuredness* of queries. In Section 3 we will look at the alternative decomposition strategies for shredding XML documents into relational tables. We will then look at how the nature of queries can be orthogonal to the presence of schema at load-time in Section 4. We will then show the benefits obtained by using schema information, both at load-time and query-time, in Section 5. We then propose a simple metric to compare the alternatives in Section 6 and look at how it can be used to choose a good strategy. We also apply this metric to response times published in [6] and show how it works. We look at related work in Section 7 and then present our conclusions in Section 8.

2 Definitions

Let us define some of the concepts related to the *structure* of queries and *decomposition strategies* that store XML documents in relational databases. We consider path expression queries and use the notation developed by the *XPath* working group of the W3C[16]. A schema for information about a University database is given in Figure 2. We have shown only parts of the schema in the figure. The simple sub-elements[2] and attributes of *Department* (collectively referred to as the simple content) are shown in a box. Similar information about the other elements aren't shown for clarity. We use this schema to present sample queries to illustrate the definitions.

Definition **1** *A structured query is one that specifies the element name at each level of the query. It uses the parent and child XPath operators but does not use the descendant and ancestor operators.*

For example, the query "dept[name='computer science']/course/coursesection[RoomNo = '6']/@id" is a *structured* query.

Definition **2** *An unstructured query is one that contains ancestor-descendant operators and does not have any parent-child operators.*

[2] sub-elements with simple type that are also leaf nodes in the element graph

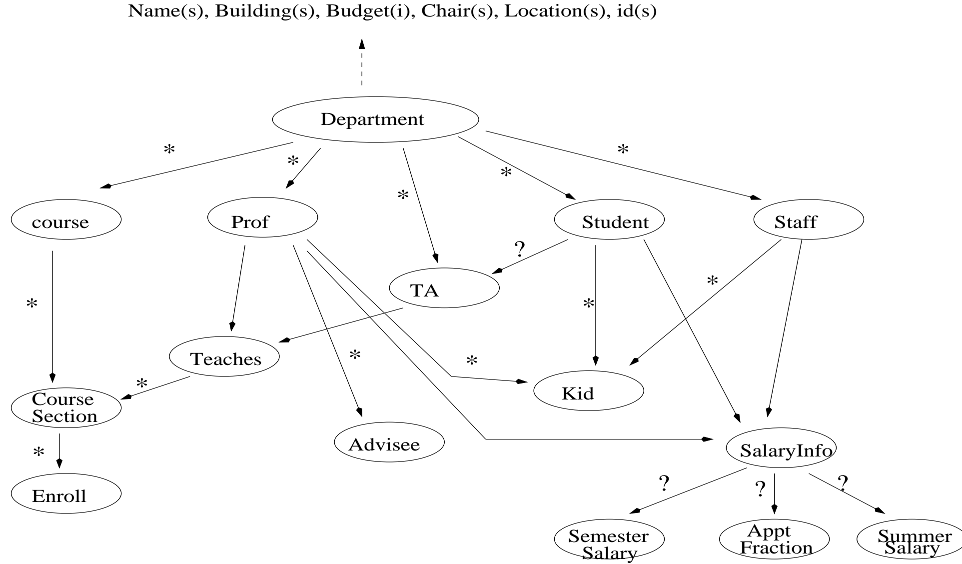

Figure 2: Sample graphical schema

For example, the query "//Student[//RoomNo = '67']//@id" is an unstructured path expression query.

DEFINITION **3** *A query that contains both ancestor-descendant and parent-child operators is called a semi-structured query.*

"dept[name='computerscience']//coursesection[RoomNo = '6']/@id" is an example *semi-structured* query.

DEFINITION **4** *A schema is said to be complete if it does not have any wild-cards (like the* ANY *element or* ANYATTRIBUTE*) in it. It is said to be a partial schema otherwise.*

DEFINITION **5** *A query is said to be structured with respect to an XML schema if it can be rewritten as the union of a set of queries each of which is a structured path expression query. It is semi-structured with respect to the XML schema otherwise.*

For example, given a *complete* schema, the query "//Student[//RoomNo = '67']//@id" can be rewritten as "Department/Student[TA/Teaches/coursesection/RoomNo = '67']/@id". The same query will be *semi-structured* with respect to a *partial* schema that has an ANY element in it as the satisfying *Student* element can be in the fragment of the XML document corresponding to this ANY element.

DEFINITION **6** *Data that conforms to a complete schema is structured data while data that does not have an associated schema is called unstructured data. Data that conforms to a partial schema is called as semi-structured data.*

DEFINITION **7** *The structure of a relational decomposition strategy denotes the amount of schema information used by the strategy.*

A strategy that needs a *complete* schema, like the *binary* approach in [10], is *well-structured* while one that does not use any schema information, like the *edge* approach in [5], is an *unstructured* strategy. A strategy that works with a *partial* schema or only uses partial information from a *complete* schema is called as a *semi-structured* decomposition strategy.

DEFINITION **8** *The structure of a query Q with respect to a decomposition strategy S is the same as the structure of the query with respect to the schema information used by S to construct the relational tables.*

We will look at this in more detail in Section 4.

3 Alternative strategies for decomposing XML Documents into RDBMS

Storing XML documents in relational databases has been extensively researched and many commercial relational engines have varying degrees of support for XML data. These schemes vary from simple *unstructured* strategies like the **edge** approach [5] and the **binary association** approach [12] to *well-structured* strategies like the **hybrid** approach [10]. Some of the early results [5, 6] showed that reconstructing highly nested XML documents or elements may be expensive. The Xperanto[3, 11] project showed a way to use relational databases to efficiently store and query XML documents. In this paper, we assume a model where the documents are shredded into relational tables and they may also be optionally stored in a separate store (like a file system). The RDBMS is used for answering queries and reconstruction of results may be done outside the database. This corresponds to roughly saying that we are looking at the FOR, LET and WHERE clauses of XQuery[17] and are not considering the RESULT clause. Since, the FOR and LET are built using path expressions, we consider the class of path expression queries in this paper.

There has been a new W3C schema proposal for XML schema languages called **XML Schema Definition Language** (XSDL) [15]. This language is a superset of DTDs and has additional features like local namespaces, separation of element names from types, inheritance, integrity constraints like keys, enhanced basic datatypes, user defined datatypes etc. XSDL enables us to develop schemas that are more informative and one of the main goals of XSDL is to help in query formulation and optimization. We propose 3 decomposition strategies, *xschema, kxschema and kx-anyschema* that use different amount of *XSDL* schema information and discuss some of the benefits that can be leveraged from these additional features.

We shall now look briefly at the salient features of some of these decomposition strategies in this section. First, we will look at the **edge** approach[5] that does not use any schema information. We propose the **xschema** approach that uses simple features of *complete XSDL* schema to store XML documents. We extend it to the *kxschema* approach, which benefits from the presence of integrity constraints in the schema. We then propose an integrated approach, **kxanyschema** that is an extension of **kxschema**, which handles *partial schemas* . We will use the schema in Figure 2 to illustrate these methods.

3.1 The Edge Approach

In this scheme a single **Edge** table is used to store the entire XML repository. The XML document can be viewed as a collection of edges, each edge connecting an element to its sub-element or attribute. The **edge** table has 5 columns, *source, target, name, ordinal and value.* Each row represents an edge (sub-element or attribute) in the XML document and stores the *oids* of the source and target objects (element or attribute), the name of the edge, sibling position of this edge and value of the target object (if it has a simple type).

3.2 The Xschema Approach

This method uses a *complete XSDL* schema to store the XML documents in an RDBMS. From [10] it uses the concepts of in-lining sub-elements and attributes and extends the simplification of complex types and handling of recursive types to avoid the blow-up in number of relations due to big strongly connected components. It also extends the idea of binary associations in [12] to work in the presence of recursion. In the absence of recursion, it boils down to a bijective mapping between *structured* path expressions and relational columns. So, two elements will map to the same relational

column iff they have the same *structured* path expressions. The scheme also maps various simple datatypes to corresponding relational datatypes[3]. A brief description of the translation procedure is given below. The handling of inheritance and recursion are not explained as they are orthogonal to this discussion. This procedure when applied to the schema in Figure 2 creates 18 relations, one each for Department, Course, Prof, TA, Student, Staff and Advisee, 3 for Kid and 4 each for Course and Enroll.

- The element content of complex types are simplified by an extension to the method in [10].
- We create a type graph[4] to represent the relationship between the various complex and simple types and identify strongly connected components in this type graph.
- For each global element[5], E,
- We create an element graph, from the type graph, for all elements that are reachable from E.
  - Non-recursive types (T) occur as many times as there are unique path expressions from E corresponding to an element of type T.
- Relations are created for global elements[6].
- Attributes and sub-elements with a simple type are in-lined in the relations of corresponding parent elements.
- New relations are created for set-valued sub-elements and recursive sub-elements. An *id* column is added as a key for each relation and a *parentid* column is added as a foreign-key to identify the parent element.

3.3 The Kxschema Approach

This is an extension of the *xschema* approach, where the integrity constraints in the XSDL schema are mapped into equivalent relational integrity constraints. The possible integrity constraints in XSDL are *unique, key and keyref* that are translated into relational *unique, key and foreign key* constraints. The *key* constraints may eliminate the *id* and *parentid* columns in many cases. The

[3]Most simple datatypes can be mapped into SQL types and associated column conditions
[4]similar to the DTD graph in [10]
[5]potential root element for conforming documents
[6]potential root elements for conforming documents

XSDL integrity constraints are locally scoped[7]. For example, a constraint in the department element, which makes the name of all courses in the department unique prohibits two courses from the same department from having the same name. But it allows two courses in different departments to have the same name. So, this constraint gets translated into a unique constraint in the $Dept/Course$ table on the $deptid, courseName$ fields. Here, *deptid* is the key for the department element[8]. The relational schema obtained by the *kxschema* method for the schema in Figure 2, assuming key constraints, is given in Appendix A. We will show how the use of key information in this fashion can help in reducing query response times. It can also be used in situations where XML is used as an interchange format to transfer data between RDBMS's to ensure that the schemas match at the two ends. But, it has the drawback that the size of the keys in the relational tables may grow linear in the depth of the schema and integer key fields in *xschema* may be replaced with columns of more complex types.

3.4 The Kxanyschema Approach

XSDL schema allows the specification of partial schemas through the use of the ANY element. This allows any well-formed XML to occur as a sub-element[9]. For such a *partial* schema, we use the *edge*[10] approach for XML fragments corresponding to the *any* fragment. For example, if the *TA* sub-element of *Student* element is replaced with an *any* element, then an *edge* relation is created for that sub-element. Note that an *edge* relation is created for each occurrence of a wild-card. For example, if the *Coursesection* element had an ANY sub-element, then it would create 4 *edge* relations, one for each of the 4 *coursesection* relations.

4 Independence of data and query structuredness

As seen in the introduction, the schema availability may differ between load-time and query-time. The *structuredness* of a dataset depends on the schema availability at load-time, while that of queries depends on schema availability at query-time. It is possible that schema information that wasn't available initially becomes available at query time or that a partial schema was inferred

[7]This was a major motivation for the association-type mapping

[8]In general, the key constraint includes the key for the element within which it is scoped

[9]there are some restrictions here with respect to namespaces that are orthogonal to this discussion

[10]Notice that though we used a combination of the *edge* and *kxschema* approaches, we can replace them with a combination of a *unstructured* and a *structured* approach

based on the dataset. Let us look at some sample XML queries on the schema in Figure 2.

1. *structured* queries like dept/professor[Kid like 'mark'].
2. *semi-structured* queries like dept/student[//semestersalary > 18000].
3. *unstructured* queries like //Kid[Text like 'mark'].

These queries require differing amounts of schema information at query time. Query 1 requires *complete* schema information about the part of the document related to Professors in Departments, while Query 2 requires *partial* schema information about the dataset. Query 3 requires minimal schema knowledge (only the fact that Kid elements have the name as text content).

If *complete* schema information was available at load-time and the *xschema* approach was used to decompose the documents, then queries like Query 1 are *fully structured* with respect to *xschema*, while those like query 3 are *unstructured* with respect to *xschema*. Queries like Query 2 fall somewhere in between and are *partially structured* with respect to *xschema*.

If partial schema information was available at load-time and the *kxanyschema* approach was used, then queries 1 and 2 may be *fully* or *partially* structured with respect to *kxanyschema* depending on the actual schema information available. If the $Dept/Student/TA$ element in the schema in Figure 2 was replaced with an *any* element, then Query 1 is *fully* structured, 2 is *partially* structured and 3 is *unstructured* with respect to *kxanyschema*.

If the *edge* approach was used to shred documents, either because of lack of schema information at load-time or because it was preferred over other approaches due to its simplicity, then all the queries are *unstructured* with respect to this scheme.

The *well-structured* decomposition strategies can be expected to efficiently answer queries that are *structured* with respect to the strategy while *unstructured* strategies can be expected to work well for queries that need to be rewritten as the union of a number of *structured* queries over a *complete* schema. We have shown that the *structure* of the query is orthogonal to the *structure* of the decomposition strategy. What is important is the *structure* of the query with respect to various decomposition strategies and its structure with respect to the schema that is available at query time which will play a major part in choosing the decomposition strategy.

Let us now look at how schema information can be used effectively at load time and query time to reduce query response time. This will play a major part in the way we choose our cost metric to estimate the cost of evaluating a query with a decomposition strategy.

5 Benefits of using schema information

In this section, we shall see how schema information can be used at load-time to choose effective decomposition strategies and at query time, for all decomposition strategies, to simplify the XML queries and to eliminate wild-cards.

5.1 Experimental Setup

We conducted our experiments on an XML dataset conforming to the schema given in Figure 2. The size of the dataset was 115MB. We ran the experiments on an Intel 800 MHZ Pentium processor with 256MB of main memory running Linux. The data generator for the BUCKY benchmark[2] was modified to generate data synthetically. For the *edge* approach, the relation was clustered on the *name* column and indices were built on the *source, target, name and value* columns and on the *source,value* pair. For the approaches using *XML Schema*, indices were built on the key and foreign key columns for each relation. The queries that we use for our experiments are given below

1. Select all departments with budget > 7 million.
   dept[*budget* > 7000000]/@id
2. Select all course sections in *computer science* department that are in room number 6.
   dept[name='computer science']/course/coursesection[RoomNo = '6']/@id
3. Select all course sections, within any department, which are in room number 6.
   dept/course/coursesection[RoomNo = '6']/@id
4. Select all course sections that are in room number 6.
   //coursesection[RoomNo = '6']/@id
5. Select all course sections, within any department, which have a course number like 102.
   dept/course/coursesection[csNo like '102']/@id
6. Select parents of kids with name containing "mark".
   Kid[Text like 'mark']/../@id
7. Select course number and grades for student with id P19144 from student records and department records.
   dept/student[@id='P19144']/enroll/(csNo,Grade) ∪
   dept/course/coursesection/enroll[@id='P19144']/(csNo,Grade)
8. Select all students who teach in room number 67.
   //Student[//RoomNo = '67']/@id

9. Select all students who are TAs.
   //Student[TA]/@id

10. Select all students who teach a course.
    //Student[//coursesection]/@id

11. Select all students with semester salary > 18000.
    //Student[//semestersalary > 18000]/@id

Let us next look at how schema information at load-time can be used to efficiently answer queries.

5.2 Load-time benefits of schema

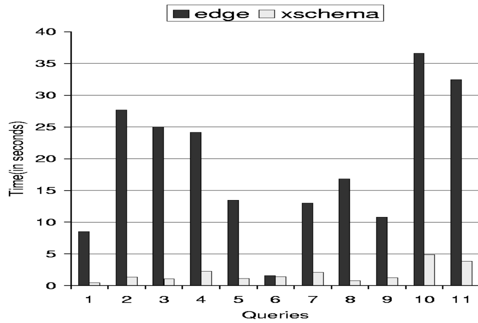

Figure 3: Using schema information during decomposition

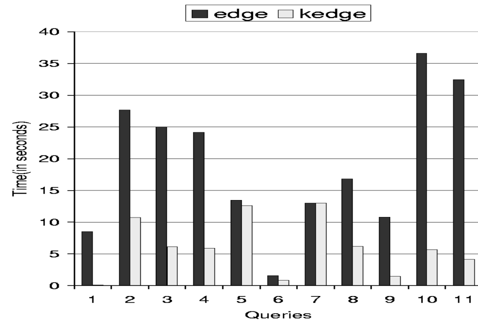

Figure 4: Using key information in edge approach

Schema information at load-time can be used in the decomposition strategy like the *xschema* approach. The time required for query execution for the queries given in Section 5.1 is given in Figure 3. The figure compares the time required for the *edge* and the *xschema* approaches. Notice how the time taken by the *xschema* approach is an order of magnitude less for most queries. For SQL like queries, like queries 1- 3, the use of multiple relations is beneficial. The main reason for the better performance of the *xschema* approach is the mapping of elements occurring in different path expression to different relational columns. As a result, the queries are on relations containing only data relevant to the query. On the other hand, the *edge* approach operates on the huge single edge relation. Also, *inlining* in *xschema* approach leads to better performance. The main drawback is that there are 18 relations in the xschema approach as against the single relation in the edge approach, which gives an indication of the potential blowup in the number of relations that is possible with the xschema approach.

Schema information can also be used in the edge approach to improve query response time. Note that the edge approach uses *id*s for elements in the edge relation. In this example, there is an *id* attribute for each element in the actual documents itself. This also serves as the key that identifies the element. If this key information is used in the edge approach (*kedge* approach), then the performance gain is shown in Figure 4. The improvement comes mainly because the join to get the *id* sub-element and its value is avoided. Since the edge relation is clustered on tag name, this value lookup would otherwise be unclustered. The clustering is done on the tag name as most queries access elements based on this field. This is an example of using schema in an unstructured approach at load-time.

Similarly, type information present in the schema can be used to map content of leaf elements and attributes to the appropriate types. This can be helpful in query processing. For example, query 1 can be processed easily if the type is known to be integer or float. In the absence of any type information about the *Budget* element, we are forced to either pull up the selection condition to the application level or use a user-defined function to check this condition.

So, we have seen how the use of schema information at document load-time can help in better performance either by the use of decomposition strategies that extensively use the schema information or by using key information in unstructured approaches like the edge approach. We shall next look at some of the benefits of using schema information at query time.

5.3 Query-time benefits of schema

At query time, schema information can be effectively used to improve query response times and also to answer certain queries that cannot be otherwise answered completely by the relational engine for certain decomposition strategies. Some of the possible optimizations include use of integrity constraints and query rewriting.

5.3.1 Using integrity constraints to simplify queries

The use of integrity constraints in the schema for XML documents and their translation to equivalent constraints on the relational decompositions can help in query evaluation. Let us look at how *kxschema* can use this information to simplify query 7. Consider the part of Query 7 that looks up the student's records. This involves a join between the STUDENT and the corresponding

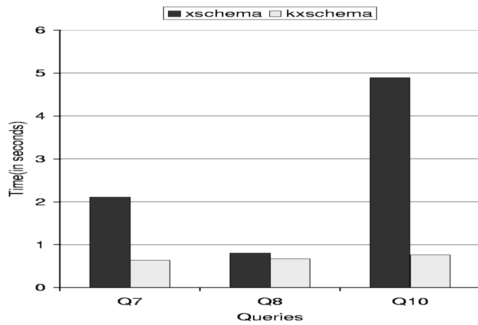

Figure 5: Using integrity constraints during query processing

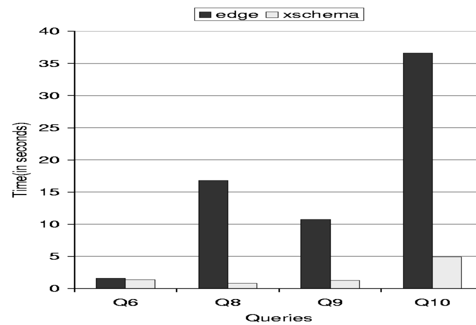

Figure 6: Query rewriting benefits

ENROLL relations. The join condition is on the *id* field of the student[11] and the *parentid* field of the enrollment. Note that in the presence of key constraints, the *id* field of the student is specified as a key for the *student* relation. As a result, the *parentid* field of *enroll* relation corresponds to the actual *studentid* and the query becomes a selection on the *enroll* relation. The query runs faster on *kxschema* than *xschema* as shown in Figure 5.

Similarly, for Query 8, *xschema* generates a SQL query involving a join between STUDENT and COURSESECTION relations while *kxschema* generates a simple selection on the COURSESECTION relation. Notice that Query 10 shows a similar phenomenon too. The additional join operation increases the cost for *xschema* as seen in Figure 5.

5.3.2 Query Rewriting

Query rewriting can be used to answer *unstructured* queries with *structured* and *semi-structured* decomposition strategies. It can also be used to avoid recursive queries in *unstructured* approaches like the edge approach.

Answering wild-card queries in structured approaches

One of the main drawbacks of using a structured approach like *xschema* is that it is expensive to answer *unstructured* queries in this approach. But, in the presence of schema information, a single query with wild-cards can be rewritten as a set of queries that do not have any wild-cards and the result is the union of these queries. For example, using schema information, Query 10 can be rewritten to $Dept/Student[TA/Teaches/coursesection]$. Similarly,

[11] a column created as key for this relation by the *xschema* translation scheme

Query 8 can be rewritten to $Dept/Student[TA/Teaches/coursesection/RoomNo =' 67']$. Some queries get translated into union queries, when the wild-card leads to more than one match in the schema. For example, Query 6 gets translated into a union query as $Dept/Staff$[Kid like 'Mark']$\cup$ $Dept/Professor$[Kid like 'Mark']. Some queries, like 4, become more expensive as they get translated into a union query involving several SQL queries, since COURSESECTION occurs 4 times in the schema. Rewriting makes *xschema* perform better than *edge* as shown in Figure 6.

Eliminating recursive queries

Some unstructured decomposition strategies like the edge approach can handle queries with a leading wild-card effectively. But, the presence of wild-cards in the middle of the query needs recursive query processing to get the results. For example, while query 9 gets translated into a simple SQL query in the *edge* approach, Query 8 and Query 10 need some sort of recursive query processing. This need can be eliminated by using schema information and removing the wild-cards that appear between *student* and *roomno* in Query 8 or *student* and *coursesection* in Query 10. The importance of this optimization, if relevant schema information is available, can be understood by the response time figures in Figure 7. We can see that the recursive are an order of magnitude costlier than queries that use schema information. Also, the increase in response time is more for the *edge* approach as compared to the *kxanyschema* approach. This indicates that the size of the relation on which the recursive query operates plays a role in the magnitude of this increase.

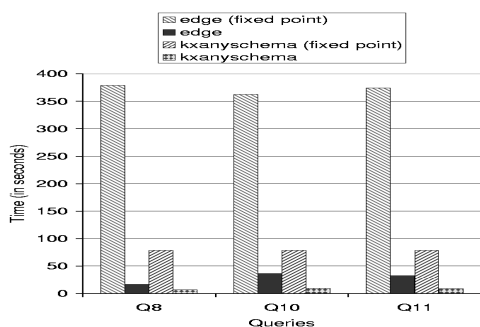

Figure 7: Cost of recursive queries

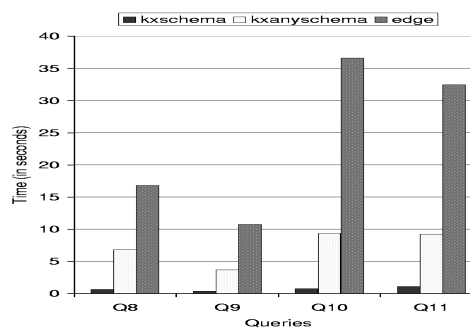

Figure 8: Query response time as available schema information varies

5.4 Varying the amount of schema information

Let us now examine how the same set of queries perform, when the amount of schema information available at document load time is varied. We consider the cases when there was no schema (*edge*),

complete schema (*kxschema*) and *partial* schema (*kxanyschema*), where the *TA* element was replaced with an ANY element.

There was no significant difference for queries 1-7 between *kxschema* and *kxanyschema* as the structure of the queries with respect to the two schemes were the same.

Query 9 involves a single path expression query on a schemaless part of the document (TA in this case) for kxanyschema and it becomes *partially* structured with respect to kxanyschema while it remains *structured* with respect to kxschema. Note how the execution time for kxschema and kxanyschema differ in Figure 8. Similarly, Query 10 involves 3 additional path expressions on schemaless data (after the rewriting using schema information) and this translates into a more expensive lookup using kxanyschema. Note that the relative increase in costs is roughly proportional to the number of levels in the path expression that do not have schema information.

Similarly, for Query 8, the query involves 3 levels through the *any* relation corresponding to TA information and an additional selection condition. This translates into an increasing gap between the two decompositions. A similar behavior can be noted for Query 11 as well.

6 Choosing among various decomposing strategies

In the previous section, we noticed how schema information can be used at load-time and query-time to answer queries. Notice that if schema is not available at load-time, we have a choice of *unstructured* decomposition strategies like the *edge* and the *binary association* approaches. In the presence of schema information, we have more options that use varying amounts of schema information. Also, we saw how in the presence of *partial* or *complete* schema information, queries that look unstructured, may have equivalent well structured query rewritings. So, the *structure* of a query depends on the interaction between the decomposition scheme and the query and the presence of schema information at query time.

6.1 Necessity for decomposition strategy specific cost mechanism

In this section, we show how the performance of various queries can vary depending on the finer details of the decomposition strategy.

Let us look at the performance of queries 2, 3 and 4. They represent similar queries that fetch the course sections that are held in room no 6. While query 2 restricts the result set to the course sections in the computer science department, Query 3 allows course sections of any department.

Query 4 generalizes it to any course section, i.e. course sections that may occur anywhere in the document.

The expectation will be that since Query 2 and query 3 are *structured* queries, the *xschema* method will perform efficiently for them, with Query 2 having a smaller response time due to the additional selection condition. But, due to the special property of the decomposition scheme that each path expression can be uniquely mapped to a relational column, we notice that Query 3 is faster than query 2. Notice that Query 2 requires 2 joins in the equivalent SQL query. On the other hand, since $dept/course/coursesection$ gets mapped to a unique relation, query 3 gets translated into a selection query and is faster than Query 2. This fact can be noticed in Figure 3.

Similarly, in the edge approach, each additional selection condition adds to the cost of the query response time. Also, each additional element in the path expression adds to the cost. So, Query 4 has the least cost, as it has wild-cards and the decomposition strategy is unstructured, while Query 3 has a higher cost due to the additional $dept/course$ path expression and Query 2 has even higher cost due to the $Name =' ComputerScience'$ selection condition. This can be noted in Figure 3.

Notice that if we used a variation of *xschema*, where multiple occurrences of the same element in different parts of the schema were mapped to the same relation (in this case, multiple $coursesections$ were mapped to the same relation), then the cost of query 2 will be lesser than that of Query 3 for this scheme.

This shows that the cost metric that we associate to estimate the *structuredness* of a (*query*, *decomposition pair*) has to take into account the finer details of the strategy.

6.2 Simple metric for analyzing query costs

We propose a simple decomposition strategy specific metric for analyzing the cost of executing queries.

6.2.1 Metric for the Xschema Approach

Let us look at how the execution time varied for some of the queries in 5.1 for this method.

- We notice that, in general, traversing one level of the path expression translates into a relational join. For example, query 2 gets translated into a SQL query with 2 joins.
- Consider queries 2 and 3. We notice that, as discussed in 6.1, that Query 3 is answered faster than Query 2. Since, path expressions have a bijective mapping with relational columns, we

notice that initial elements in the query that do not have any selection conditions do not add any cost to the query.

- For Query 11, we notice that even though the query has 4 levels in the path expression, due to the inlining technique followed by the decomposition scheme, effectively there is no join in the equivalent SQL query.
- A query involving a wild-card can be translated into a union of multiple SQL queries, in the presence of complete schema information. So, the cost of evaluating the query is equal to the sum of the cost of evaluating each of the SQL queries.
- Substring operations are costlier than normal integer operations and string equality operations.

So, we see that the cost of evaluating a query on *xschema* can be estimated as

$$BasicCost_{xschema}(q) = \sum_{i=1}^{i=n} BasicCost_{xschema}(p_i)$$

where the query q, after query rewriting, resulted in n queries and $BasicCost_{xschema}(p_i)$ represents the cost assigned to the query p_i that does not have any wild-cards in it. Let us define an edge in the query corresponding to a schema entry where the child element can occur either exactly once or at most once as a *simple* edge. If the child can occur multiple times, we call the edge a *set* edge. The BasicCost can be defined as

$$BasicCost_{xschema}(q) = max(f_1(n_1 - n_2 - n_3), \gamma) + f_2(n_4)$$

where n_1 is the number of edges in q , n_2 represents the number of edges in prefix of q before the first selection condition in q, n_3 is the number of *simple* edges after the first selection condition in q, γ is the relative cost of a scan with respect to to a join operation (to account for the leading scan operation), n_4 is the number of *like* operations, f_1 is a function to account for non-linear cost of joins and f_2 is a function that assigns relative cost of a text operation with respect to a join operation.

Notice that for *kxschema*, *BasicCost* needs to be adjusted for the fact that selection conditions on the key of an element may be pushed down to its descendants, if the descendants have a locally scoped key. These conditions are *non-essential*, while the other conditions are *essential*. So, we

have

$$BasicCost_{kxschema}(q) = max(f_1(n_1 - n_2 - n_3), \gamma) + f_2(n_4)$$

where n_2 and n_3 are defined based on *essential* selection conditions and the other variables represent the same thing as given above.

6.2.2 Metric for the Edge Approach

Let us now look at the interaction between the edge approach and the variations in the query. Let us first consider the case where wild-cards may occur only in the beginning of the query.

- Each edge traversal in the query adds one relational join to the SQL query.
- Each selection condition in the query adds a relational join to the SQL query.
- The initial wild-card does not add anything to the cost.
- Projections and selection conditions on unclustered fields that are not join conditions increase the response time.

So, we have

$$BasicCost_{edge}(q) = max(f_1(n_1 + n_2), \gamma) + f_2(n_3) + f_3(n_4)$$

where n_1 is the number of edges in q(excluding leading edge if it involves a wild-card), n_2 is the number of selection conditions in q, n_3 is the number of *like* operations, n_4 is the number of unclustered columns as mentioned in 6.2.2 and f_3 is the corresponding scaling factor. f_1, f_2 and γ are scaling factors as mentioned for *xschema*.

A query having arbitrary wild-cards in it can be rewritten, with schema information, into a set of queries having wild-cards only in the beginning of the query and the cost is the sum of the individual costs. If complete schema information for the data region involving wild-cards in the query is unavailable, the cost can be given as

$$BasicCost_{edge}(q) = (n_1 + n_2 + \alpha n_5 - 1, \gamma) + f_2(n_3) + f_3(n_4)$$

where n_1 is the number of elements in q whose name is specified, n_2 is the number of selection conditions in q, n_3 is the number of wild-cards in the query(excluding the leading wild-card, if

present), α is the relative cost of executing a recursive query on the particular system as compared to a simple join query and the other variables are as defined above.

6.2.3 Metric for the Kxanyschema Approach

Let us now look at the cost executing queries using the *kxanyschema*. This scheme is a combination of the *kxschema* and *edge* method. A query q can query across relations corresponding to elements or *edge* relations. A single query can be broken up into k queries, $(q_1,\ q_2,\ \ldots q_k)$, q_1 queries the structured (*kxschema*) part and the remaining $k-1$ queries are on different *edge* relations. Note that the k queries can be combined as a single SQL query. The cost of a query q on this scheme can be given by

$$BasicCost_{kxanyschema}(q) = BasicCost_{kxschema}(q_1) + \sum_{i=2}^{i=k} BasicCost_{edge}(q_i)$$

The value of α (if recursive query processing is used) will vary depending on the relation size as we saw in Figure 7. So, we can either use an average value for α or we can estimate α_i for each relation in the relational decomposition and replace αn_3 by $\sum \alpha_i$.

6.3 Comparing different decomposition strategies

We have proposed a simple cost model for path expression queries based on the decomposition scheme. The costs assigned to a query are the basic costs for that *query,decomposition strategy* pair. In practice, we need to compare the costs across decomposition strategies for a given query workload. The *BasicCost* for a query cannot be compared across schemes because operations like joins will have different costs for various schemes. Notice that all entries in the BasicCost metric are normalized to the cost of a join operation. For example, the *edge* approach will have a higher join cost than the *xschema* approach due to larger relations. So, we compute relative costs as

$$RelativeCost_m(q) = BasicCost_{method}(q) * \beta_m$$

where β_m represents the cost of performing a join operation in the method m. Now we can compare the relative costs of queries across decomposition strategies and choose a good method. Note that this technique will identify potentially bad schemes that should be avoided. The comparisons can be made for each query and we can identify methods that are very expensive for certain queries.

The *RelativeCost* numbers cannot be used to compare across queries for the same method, unless the queries are on the same fragment of the XML documents. So, given a query workload, we can compare various decomposition strategies by looking at how each strategy performs for each query relative to the other strategies. We can avoid strategies that perform very badly for some queries and choose one that has minimum deviation from the optimal strategy for each query.

6.4 Verifying effectiveness of the metric

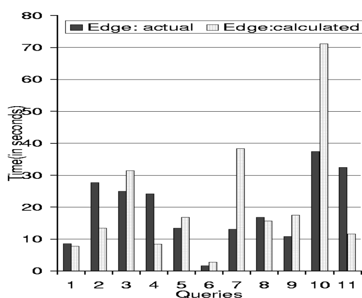

Figure 9: Application of cost metric to estimate query response time

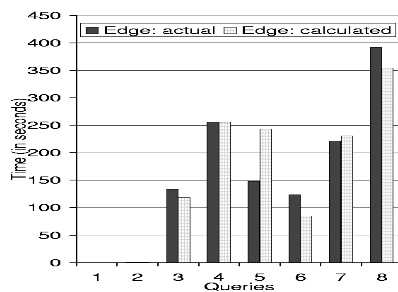

Figure 10: Application of cost metric to results in [6]

We can check how the effectiveness of the cost metric by taking the actual query response times for one approach and then estimating the response times for another method with it. We start with the values for the *xschema* method and predict the time taken by the *edge* approach. We assumed that $\beta_{edge}/\beta_{xschema}$ was 3. This was the average value for this ratio obtained by using the actual response times for the queries. Also, f_2 and f_3 were assumed to be factors of 0.25 and 0.2 and $\gamma_{xschema}$ was taken to be 0.5. We assumed that each additional join operation costs 10 percent more than the previous one. With these assumptions, we calculated the response time for the *edge* approach. These values are compared with the actual values in Figure 9. Notice that the predicted value are within a factor of two of the actual values for most queries.

We then tried to see how the metric works for the results published in [6]. We chose the *edge* and *attribute* approach values for the 8 queries presented in the paper and starting from the *attribute* values, predicted the values for the *edge* approach. The cost metric for the *attribute* approach was similar to that of the *edge* approach and accounted for the number of joins, recursive queries and unclustered column lookups. The comparisons between the predicted values and the actual values

are presented in Figure 10. Notice that 5 queries had an error of less than 10 percent, while the others were almost within a factor of 2. The response times for the first two queries were under a second in both cases and are not visible in the graph. They had an error percentage of -60 and 5 respectively[12]. The queries ranged from simple selections to regular path expression queries that require recursive SQL queries. From Figure 7, we can see that recursive queries are 10 times more expensive than normal join queries for the *edge* approach. Also, the *attribute* approach has smaller relations making recursive queries less expensive. So, we assumed that recursive queries were 10 and 3 times more expensive for the *edge* and *attribute* approaches with respect to normal joins. Since, the relations in the *attribute* are expected to be smaller than those in the *xschema* approach which are in turn smaller than those in the *edge* approach, the value of $\beta_{edge}/\beta_{attribute}$ was taken to be 5.

7 Related Work

There has been a lot of work on using relational databases to store and query XML documents. One of the earliest studies was done by Florescu and Kossman [5] where they compared the performance of a number of different mapping techniques that do not require any schema information. The *Stored* [4] system stored XML documents in relational databases into structured and semi-structured components using data mining. It did not require the presence of a schema but used the query workload to identify frequently occurring path prefixes. But, it did not handle recursion efficiently. Shanmugasundaram et al.[10] developed methods to use DTD information to obtain the relational mapping. Here, we propose a combination of the two strategies and the show how to use both schema information and query workload to identify good decomposition strategies. In [1], the authors define graph schemas for unstructured data and discuss about how it can be used for query optimization and query decomposition.

8 Conclusions and Future Work

In this paper, we compared various relational decomposition strategies for XML documents and showed how the choice of a good strategy depends on presence (absence) of schema information

[12] query 1 is a simple selection based on object id and the response times are very small causing higher error in predicted values

at data load time and query time and the query workload. We also showed how the *structure* of a query is relative to the decomposition strategy under consideration and the presence of schema information. We proposed decomposition schemes that used *XML Schema* information effectively. We showed how schema information can be used effectively at both load time and query time for both *structured* and *unstructured* decomposition techniques to improve query response time. We finally proposed a simple metric that can be used to choose a good relational decomposition strategy based on a query workload or to choose from alternative decompositions during query optimization. This metric can be used to choose between multiple unstructured decomposition strategies, if schema is not available at load time, or between all possible strategies in the presence of schema information. We also showed how our metric gives reasonable estimates on the relative costs of the various strategies for a given workload.

We have presented some analysis about the interaction between the structure of XML data and queries in the context of relational databases. It will be interesting to look at the general problem where the data could be in a native store and indexes could be value indices, inverted lists etc. We can then come up with a cost metric for these strategies and compare among all possible strategies. The metric proposed requires very little information about the dataset, while the ideal metric will need complete information about actual data distribution. So, identifying the crucial statistics that can help in obtaining a more accurate metric that can be used to choose the *best* (instead of *good*) strategy will be a good problem. We can also extend this study of the interaction between data, queries and schema to identify an integrated decomposition strategy that starts with a partial schema and query workload to identifies path expressions where schema information can be used and regions where an unstructured approach will be better.

References

[1] Peter Buneman, Susan Davidson, Mary Fernandez, Dan Suciu, "Adding Structure to Unstructured Data", ICDT 6th International Conference, January 1997.

[2] M. Carey. D. Dewitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gerke, D. N. Shah, "The bucky object-relational benchmark", Proc. of the ACM SIGMOD Conference on Management of Data, 1997.

[3] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, S. Subramanian, "XPERANTO: Publishing Object-Relational Data as XML," Workshop on the Web and Databases (Informal Proceedings), May 2000.

[4] Alin Deutsch, Mary F. Fernandex, Dan Suciu, "Storing Semistructured data with STORED", Proceedings of SIGMOD International Conference on Management of Data, June 1999.

[5] D. Florescu, D. Kossman, "Storing and Querying XML Data using an RDBMS" , Data Engineering Bulletin, 22(3), 1999.

[6] D. Florescu, D. Kossman, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database", Technical Report, INRIA, May 1999.

[7] Zachary G. Ives, "Efficient Evaluation of Regular Path Expressions in XML", Ph. D. General Examination.

[8] M. Klettke, H. Meyer, "XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics", WebDB Workshop, May 2000.

[9] Dongwon Lee, Wesley W. Chu, "Constraint-preserving Transformation from XML Document Type Definition to Relational Schema", Proc. of 19th Int'l Conf. on Conceptual Modeling (ER), October 2000.

[10] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities," VLDB Conference, September 1999.

[11] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, "Efficiently Publishing Relational Data as XML Documents", Proceedings of the VLDB Conference, Egypt, September 2000.

[12] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas, "Efficient Relational Storage and Retrieval of XML Documents", WebDB Workshop, May 2000.

[13] T. Shimura, M.Yoshikawa, S. Uemura, "Storage and Retrieval of XML Documents using Object-Relational Databases", Database and Expert Systems Applications, pages 206-217, Springer, 1999.

[14] B. Surjanto, N. Ritter, H. Loeser, " XML Content Management based on Object-Relational Database Technology", Proc. of the 1st Int'l Conf. on Web Information Systems Engineering, HongKong, June 2000.

[15] "XML Schema", W3C Candidate Recommendation, "www.w3.org/XML/Schema.html"

[16] "XML Path Language: XPath", W3C Recommendation, "www.w3.org/TR/xpath"

[17] "XQuery: A Query Language for XML", W3C Working Draft, "www.w3.org/TR/xquery"

A Relational Schema for decomposition using kxschema approach

DEPARTMENT(Chair, Longitude, Budget, Latitude, Building, Name, id, *@id*)

COURSE(Credit, Name, *cNo*, *@id*)

COURSESECTION(instructor, TextBook, NoStudents, Building, *csNo*, RoomNo, Semester, *@id*, *cNo*)

ENROLL(Grade, StudentId, csNo, Name, id, *csNocolumn0*, *@id*, *cNo*)

PROFESSOR(DateHired, Street, ApptFraction, Longitude, Zipcode, BirthDate, Latitude, Name, AYSalary, Status, *id*, City, AnnualSalary, Picture, State, MonthSummer, SemesterSalary, *@id*)

COURSESECTIONTABLE0(instructor, TextBook, NoStudents, Building, *csNo*, RoomNo, Semester, *id*, *@id*)

ENROLLTABLE1(Grade, StudentId, csNo, Name, *id*, *csNocolumn1*, *id column0*, *@id*)

KID(*Kid*, *id @id*)

TA(DateHired, StudentId, EmployDept, ApptFraction, Name, AYSalary, Status, id, AnnualSalary, *@id*, MonthSummer, SemesterSalary, *@idcolumn0*)

COURSESECTIONTABLE3(instructor, TextBook, NoStudents, Building, *csNo*, RoomNo, Semester, *@idcolumn0*, *@id*)

ENROLLTABLE4(Grade, StudentId, csNo, Name, *id*, *csNocolumn1*, *@idcolumn0*, *@id*)

STUDENT(DateHired, Street, StudentId, EmployDept, *@idcolumn4*, @idcolumn3, ApptFraction, Name, StudentIdcolumn1, Longitude, Zipcode, BirthDate, Latitude, Namecolumn2, AYSalary, Status, id, City, AnnualSalary, Namecolumn0, @id, Picture, State, MonthSummer, SemesterSalary, *@idcolumn5*)

COURSESECTIONTABLE6(instructor, TextBook, NoStudents, Building, *csNo*, RoomNo, Semester, *@idcolumn4*, *@id*)

ENROLLTABLE7(Grade, StudentId, csNo, Name, *id*, *csNocolumn0*, *@idcolumn4*, *@id*)

KIDTABLE5(*Kid*, *@idcolumn4*, *@id*)

STAFF(DateHired, Street, ApptFraction, Longitude, Zipcode, BirthDate, Latitude, Name, AYSalary, Status, *id*, City, AnnualSalary, Picture, State, MonthSummer, SemesterSalary, *@id*)

KIDTABLE2(*Kid*, *id*, *@id*)