ANSI SQL Dynamic Schema-free XML Structured Data Multipath Hierarchical Processing

Michael M David
Advanced Data Access Technologies, Inc.
mike@adatinc.com

Lee Fesperman
FFE Software
lee@firstsql.com

ABSTRACT

This prototyped solution can be available for SQL users that do not desire or require semistructured markup processing at the cost of sacrificing navigationless XML access. SQL's natural processing is navigationless structured data processing and this can be naturally extended in SQL to the transparent hierarchical processing of XML structured data processing. By limiting SQL's XML data processing to only structured data hierarchical processing as described within, full multipath hierarchical processing can be supported naturally and correctly in SQL. The advantage of only supporting structured data and not supporting semistructured processing allows SQL to support XML's structured data at a full extremely powerful transparent multipath hierarchical processing level which supplies navigationless XML inherently and seamlessly. This inherent hierarchical processing also controls the dynamic definition of structures and their run-time joining controlled by the query user. In addition, the SQL specified hierarchical processing allows the processing to be structure-aware allowing automatic dynamic XML structure output.

Semistructured structures have an ambiguous data structure that allows duplicate named nodes that uses a fuzzy processing that is inexact. This goes against SQL's relationally correct results. Supporting semistructured processing would negatively impact SQL's relationally correct processing. For these reasons it just makes good sense to have an SQL/XML support of only structured data with its natural and transparent advanced XML support.

As proof of this inherent ANSI SQL advanced hierarchical processing capability, the construction of a transparent XML structured data nonlinear hierarchical processor prototype that utilizes an off-the-shelf ANSI SQL processor as its hierarchical processing engine was implemented. Actual examples from this prototype are then used to demonstrate the effectiveness of this solution and the reader is invited to submit the example queries from this paper made interactively using this available online ANSI SQL transparent XML nonlinear hierarchical processing prototype. A comparison of the prototype’s advances to today’s XQuery processor and to its current academic research will also be covered.

KEYWORDS

Hierarchical data processing, SQL/XML integration, XML query, LCA query, lowest common ancestor, schema-free query, XML keyword search, hierarchical proximity, XML transformation, semantic transformation, structure transformation, on demand XML publishing, twig queries, nearest common ancestor, least common ancestor, focused retrieval, result aggregation, relational/XML integration

1) INTRODUCTION

SQL native XML integration has not taken off as well as it could have. This is because all the current solutions are not entirely standard and are difficult to use. These solutions do not integrate relational and XML data without loss of information, nor do they integrate SQL and XML operations seamlessly because these problems have not been solved satisfactorily yet. In fact, current solutions are basically still

restricted to a linear single hierarchical path and are not based on any solid hierarchical principles. This means that XML hierarchical results can be incorrect and can go unnoticed. This paper will demonstrate and prove that ANSI SQL has the inherent hierarchical data processing capabilities to solve these problems and significantly increase today's SQL processing capabilities to an advanced multipath (nonlinear) principled hierarchical processing level automatically. This is naturally performed at a fully integrated and transparent level while taking advantage of XML's potential new hierarchical capabilities without sacrificing SQL's simple nonprocedural and navigationless look-and-feel and normal operation.

This paper is based on our research and development of the first working ANSI SQL transparent XML structured data nonlinear hierarchical processor prototype. It demonstrates how ANSI SQL can automatically and naturally operate at a valid and principled nonlinear hierarchical manner. It fully supports the most complex multipath queries against relational and XML structured data correctly and accurately without modifying SQL's well known operation. This ANSI SQL prototype referred to as the "Prototype" in this paper transparently integrates native XML data at a full nonlinear hierarchical processing level. This is accomplished by utilizing and leveraging the inherent hierarchical processing capability in an unmodified commercial ANSI SQL processor used as its hierarchical processing engine.

Keeping with SQL's friendly uncomplicated operation, no XML Schema is required for the Prototype's XML support. In the future, available schema or metadata information will be utilized to automatically build the prototype's SQL hierarchical views. The purpose of this Prototype is to transparently introduce and integrate native XML structured data processing and automatic dynamic XML structure output capability into SQL. This is done without limiting ANSI SQL's new and advanced XML hierarchical processing capabilities described in this paper. The Processor is fully interactive for easy use and proof that it can be used for powerful ad hoc, decision support, and on-demand dynamic XML publishing.

With this ANSI SQL hierarchical processing capability proven, it is shown how this automatic multipath hierarchical engine capability is seamlessly extended to support live native XML data from input, through full nonlinear hierarchical processing, to the flexible and dynamic generation of valid structured XML. This will support application use, decision support, and dynamic report publishing. This entire multipath hierarchical process is driven transparently and navigationlessly using only ANSI SQL-92 syntax and semantics. It will be explained and shown how advanced hierarchical processing capabilities such as multipath data ordering have been introduced seamlessly to utilize new nonlinear hierarchical capabilities.

2) BACKGROUND

ANSI SQL full database hierarchical processing is not being used today and there are a number of possible reasons for this, but none of these reasons is because full database hierarchical processing is not possible or not already proven. Nonlinear hierarchical database processing has been around for at least three decades, since IDMS and IBM's IMS database use in the 70's. Full multipath nonlinear hierarchical processing is based on solid principles that have been fully proven and time tested. Unfortunately, with the advent of relational processing, hierarchical processing has been forgotten and overlooked. With the introduction of more powerful hierarchical processing capabilities introduced in the SQL-92 standard and now more recently with XML, it is time to utilize this more useful and easier to use ANSI SQL automatic, natural, and inherent hierarchical processing capability. Hierarchical processing is now freely available in ANSI SQL and can be used automatically to better utilize XML (David, 2003).

2.1) Markup and Database XML Processing are Different

XML data can be structured or semistructured. Unstructured data as in email is not XML. When represented as XML it becomes semistructured data. On the other hand, hierarchical data when represented in XML becomes structured data. This structured XML data used in database type

applications has a very formal and fixed nature. SQL's inherent database hierarchical processing can operate on XML's hierarchical structure when its data has been shredded into tables or rowsets which are related by SQL hierarchical views configured to represent the hierarchical structure. The fixed unambiguous nature of the database hierarchical structure means that it can be processed nonprocedurally without procedural user navigation. On the other hand, the variable nature of hierarchical markup structures requiring semistructured XML is ambiguous for nonprocedural querying and requires user navigation. In addition this variable structure processing can produce operations that will produce invalid results for database applications. This means that a different processing logic must be used for database XML hierarchical processing than for semistructured markup use. These terms have been referred to as data centric (database) Vs document centric (markup).

One of the main reasons that the XML industry hasn't taken advantage of full multipath hierarchical processing was that XML processing was invented for markup processing and not database processing. Markup processing requires procedural navigation by the user and when database XML structured data use came along it was not realized that user navigation was no longer needed. This nonprocedural navigationless access of XML structured data also allows for full multipath processing because multipath processing is too complicated to be performed procedurally with user navigation which has caused this XML processing limitation today. This nonprocedural navigationless processing means that the database query user does not need to know the data structure. This is referred to as schema-free access and also means that the same query can be used to query different structures with the same named data items. The rest of this chapter, unless otherwise noted, will concentrate on ANSI SQL's transparent hierarchical processing of relational and XML structured data and its automatic XML formatted output.

3) INTRODUCTION TO MULTIPATH HIERARCHICAL DATA PROCESSING

This main section describes multipath hierarchical data processing also referred to as nonlinear hierarchical processing in this paper. Three decades ago nonlinear data processing was fairly popular and was used in commercial query products. The nonlinear processing was hidden from the user and the user was not required to know the structure being queried which means multiple paths were often referenced in queries and automatically processed correctly. Academically at that time, this processing was referred to as LCA Query. Today the term Schema-free query is used generally in nonlinear navigationless XML research. The difference today with hierarchical query processing is that it requires user navigation which requires knowledge of the structure. This section will cover most aspects of multipath processing, and there are a number of these. It is not a simple topic, there are many issues involved for correct nonlinear hierarchical data processing. In order to cover all the issues to get the big pictures, detailed examples are put off to a future section when the big picture is understood. But for those who wish to see the examples as they read through this section, the specific examples are cited so they can be located more easily.

3.1) Problems that Need Solving

Because no seamless and lossless solution to relational/XML data integration and SQL/XML system integration had been previously discovered, all solutions used by vendors have involved non ANSI standard and proprietary solutions even with the available SQL/XML standard. This makes them incompatible with each other. For these reasons, all of these solutions require complex procedural XML centric syntax which requires their own special training, and have not fully solved the XML and relational data integration problem. This has limited SQL/XML application development to static non dynamic solutions. There is also no specification for how to perform hierarchical processing, which means hierarchical results may be incorrect and go undetected. These problems have kept this XML industry from advancing as it could have. The Prototype offers a generic SQL/XML solution to these problems.

Very little has been achieved to solve the above problems. One possible reason for this is that the database SQL/XML vendors prefer to differentiate their solution from each other. Another solution that has been utilized by most SQL/XML vendors is to shoehorn XQuery into the SQL/XML mix to compensate for more SQL/XML processing. But this XQuery integration is actually an interoperation and is not seamless. It requires knowledge of XQuery and the data structure being processed making its use very difficult. Each vendor's implementation of XQuery is different and still does not solve the problems mentioned above. Its procedural and navigational operation limits its ability to perform complex hierarchical multipath processing or detect incorrect hierarchical results. ANSI SQL's inherent full multipath navigationless hierarchical processing solves these and all of the above identified problems.

3.2) ANSI SQL Multipath Hierarchical Processing

To get SQL to naturally perform fully hierarchically, the data must be hierarchically modeled in SQL. This means that the SQL processing is limited only to hierarchical operations using the one-sided Left Outer Join. This is the primary requirement for correct hierarchical processing. SQL Hierarchical processing is a case of less is more. By restricting processing to only hierarchical processing, the hierarchical principles are solely in control making hierarchical operations sound and allowing for advanced hierarchical capabilities such as Lowest Common Ancestor (LCA) processing required for powerful multipath processing. Interestingly, XQuery's default join is the Inner Join which is not hierarchical in nature; this means its results are not always hierarchical. It also does not inherently support LCA processing which is why a number of academic research projects (Chen, Ling & Lee, 2004; Chen, Ling & liu, 2004; Li, Yu & Jagadish, 2004) were undertaken to add LCA processing capability.

LCA processing requires that the semantics between the paths be utilized to accurately process this more powerful and internally complex multipath hierarchical processing. An example of multipath query processing is selecting data from one path of a hierarchical data structure based on a data value in another path of the structure. This LCA processing is applied to the entire hierarchical structure being processed which can be composed of multiple physical and logical structures so that the paths involved in LCA processing can be composed of different heterogeneous structures. The Lowest Common Ancestor (LCA) node between the data node being selected and the data node being tested needs to be located (Ullman, 1973) to determine the closest range of ancestry influence. This will limit the data processed to only the meaningful data for the query.

3.2.1) ANSI SQL Basic Relational Multipath Hierarchical Foundation

Relational hierarchical data modeling and XML which is inherently hierarchical can both be viewed as hierarchical structures which are composed of nodes hierarchically linked as shown in Figure 1 below. Relational tables are mapped to hierarchical structures as nodes, with the columns mapped as node fields. XML elements are mapped to hierarchical nodes with XML data strings and attributes mapped as node fields. Each different relational table or XML element represents its own specific node type. Each node type can have multiple child types (siblings) producing multiple pathways. Each node type can have multiple data occurrences (twins) under its parent node's data occurrence. A LCA's range of influence for meaningful data relationships is at this data occurrence level found in a hierarchically processed rowset or tree. This hierarchical model is a multiple data occurrence and multiple node type needed for complex structures like XML and should not be confused with simpler single node type external SQL hierarchical processing functions that are used procedurally.

The above description describes the input mapping through processing to converting XML data to internally hierarchically formatted relational working sets where the data is processed hierarchically by the SQL hierarchical structure syntax definition. The associated SQL semantics are hierarchical, causing the data in relational rowsets to be hierarchically processed and deposited in the structured rowset result.

This hierarchical result in the structured rowset result is then converted back to a hierarchical structure using the structured meta information represented in the SQL hierarchical input. The output structure definition which takes the dynamically variable SELECT list into consideration is then used to automatically format the output XML result to match the active query. This hierarchical metadata also allows for other structure-aware processing capabilities such as hierarchically optimizing the query based on its hierarchical structure and processing which is shown later in Figure 7.1.

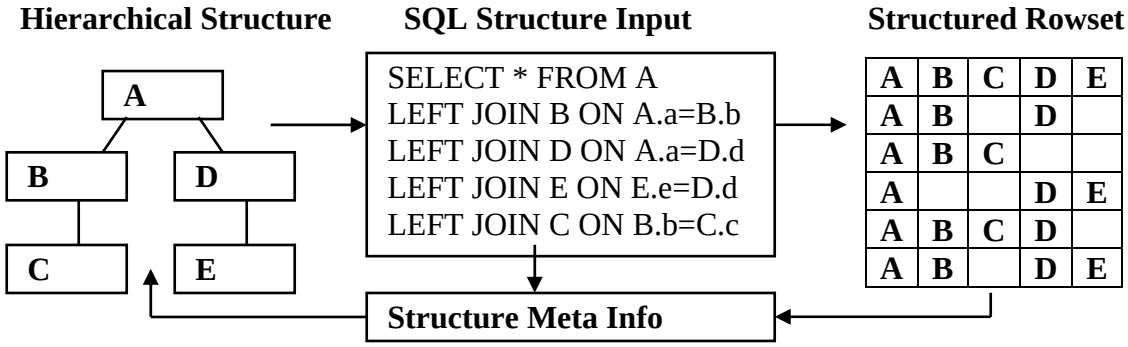


Figure 1: Hierarchical to and from Relational via Left Outer Join Mapping

The Hierarchical Structure in Figure 1 can represent a logical hierarchically modeled set of relational tables or a physically modeled XML document. Both are modeled using the SQL Left Outer Join structure definition. This allows seamless integration of heterogeneous relational and XML data. This heterogeneous hierarchical modeling and processing also solves the relational/XML integration problem completely and more than satisfactorily with no data or information loss and is shown later in Figure 15.

3.2.1.1) ANSI SQL Multipath Hierarchical Data Modeling Syntax

The SQL-92 standard introduced the Left Outer Join, known as a one sided join because it preserves the left side and not the right side. Preserving the left side over the right side is hierarchical. In SQL for example, *A Left JOIN B* places A over B hierarchically. The ON clause is used at each join point to specify the matching link point between nodes as in *A LEFT JOIN B ON A.a=B.b*. These Left Outer Join sequences can be strung together to generate any hierarchical structure (David. 1992). For example, *A LEFT JOIN B ON A.a=B.b LEFT JOIN C ON B.b=C.c* models the linear hierarchical structure A over B Over C while *A LEFT JOIN B ON A.a=B.b LEFT JOIN D ON A.a=D.d* models the nonlinear hierarchical structure A over both B and D a multipath structure because the second ON clause also references the A node. This is shown in Figure 1

The Outer Join ON clause has been available for over fifteen years and yet the WHERE clause is still being used in SQL/XML research and solutions today to specify relational joins which act as hierarchical link points. The ON clause is very different than the WHERE clause. First, it is specified at each join point allowing explicit unambiguous join information specified for each join point. Second, its domain and effect is limited to the join point similar to XPath. It is applied as the hierarchical structure is being constructed. On the other hand, the WHERE clause is applied logically after the full structure is built and it is applied hierarchically across the entire structure (row). These flexible capabilities allows for full hierarchical data modeling and processing. An example of past research in this area can be found at (Abiteboul & Bidoit, 1984; Leven & Loizou, 1993; Sengupta & Dalkilie, 2002). Figure 2 demonstrates the range of filtering the WHERE and ON operations have, notice that the ON filtering only affects its node type and those below it.

3.2.1.2) ANSI SQL Hierarchical Data Modeling Semantics

The SQL Left Outer Join data modeling syntax defines the hierarchical data structure to be processed. The associated Left Outer Join semantics defines the processing of the hierarchical structure which is being hierarchical processed. This means the SQL outer joined hierarchical structure definition and processing instructions are executed directly by SQL ensuring the most tightly coupled and accurate hierarchical modeling and processing possible.

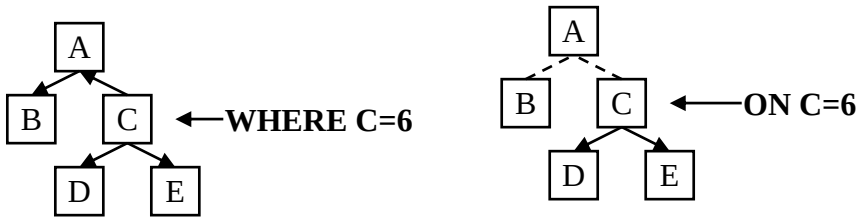


Figure 2: Range of Data Filtering

3.2.1.3) ANSI SQL Hierarchical Data Preservation

Hierarchical data preservation is a basic and important hierarchical processing operation. This process occurs automatically with the Left Outer Join's normal hierarchical operation of preserving the left table argument over the right table argument. With *A Left Join B ON A.a=B.b Left Join B.b=C.c*, if A does not exist, nether does B or C. If A exists but B does not, then B and C do not exist. If B exists then A must exist, and C may or may not exist. This can be seen in the structured rowset in Figure 1.

3.2.1.4) ANSI SQL Variable Length Multipath Rowset

For SQL to perform nonlinear hierarchical processing, its standard fixed rowset must automatically accommodate multiple variable length pathways. This is done automatically by the hierarchical data preservation being performed by the Left Outer Join operation. It inserts NULL values in the rowset representing missing data. This makes the length of the hierarchical pathways variable keeping the variable length paths aligned for proper SQL operation. This can also be seen in the structured rowset in Figure 1 where blank cells represent nulls. These have been called dangling tuples. Up until now, the hierarchical data modeling, its associated hierarchical data preservation semantics, and the representation and processing of multiple distinct pathways represent basic hierarchical processing. This basic hierarchical processing can be found piecemeal in the ANSI SQL specification. The technology behind processing multipath queries using LCA logic is much harder to explain in ANSI SQL which follows.

3.2.1.5) ANSI SQL Basic Multipath LCA Cartesian Processing

Linear hierarchical processing down a single path is well understood join processing. The real heart of nonlinear hierarchical processing is the processing of the semantics between the pathways. This requires LCA logic to coordinate and control the processing across the pathways. This is not well understood today, because SQL was not designed specifically to support nonlinear hierarchical processing. On the other hand, SQL was designed to be general purpose and we discovered LCA processing hidden in the powerful relational Cartesian product. processing. SQL data tables are joined using the relationships that model the data and then the Cartesian product generates the proper data replications necessary for the proper processing to follow the data structure modeled and constructed through joins. These data replications are based around key data join points which equate directly to LCA nodes for hierarchical processing. They keep the relationships between path data occurrences meaningful. This multipath processing increases the value of the data by naturally utilizing the semantics between the pathways.

When only hierarchical relationships are used in SQL, the Cartesian product data replications are automatically generated around the LCA nodes because they are also the join points. This means the LCAs are automatically determined and inherently control the nonlinear hierarchical processing by using the data replications as a memory mechanism a row at a time to simulating tree walking efficiently. LCA determination has been a difficult academic problem to solve efficiently (Czumaj, Kowaluk & Lingas, 2002) and required hierarchical tree walking. SQL has managed an efficient automatic way around these procedural problems. A Prototype example of LCA usage is shown in Figure 20 and further coverage can be found in (David, 2009). This LCA automatic use in SQL will be discussed further in Section 5.2.4.

3.2.2) ANSI SQL Basic Multipath Hierarchical Structure Processing

A set of basic hierarchical processing building blocks exists naturally that are naturally utilized and followed in standard hierarchical structure operations. These are: node promotion; node collection; linear hierarchical data preservation; WHERE Clause LCA hierarchical filtering processing; SELECT LCA hierarchical selection processing; SQL hierarchical views and their joining; and fragment control.

3.2.2.1) ANSI SQL Node Promotion and Node Collection

Hierarchical node promotion and node collection occurs when nodes are removed from the output. This automatically occurs when no data items are selected from a node for output. These nodes are sliced out of the structure preserving their dependent lower level nodes. This is precisely how the SELECT list's relational projection operates. When selecting data from the first table and third table joined, not selecting data from the second table joined, the rowset result has tables one and three logically next to each other. This will also cause multiple sibling nodes to collect under the next higher existing node. If the structure root is not selected and is removed, multiple fragments are produced that have different node types as their new roots. This is shown below in Figure 3.

In Figure 3, the input rowset has its B and D rows removed because the SELECT operation has not listed them for output. This is a projection operation and only outputs the fields listed for output. This operation eliminates fields, nodes, and processing without affecting other fields and nodes such that the structure stays intact except for the removed data. The Prototype's processing of the SELECT operation recognizes the data structure modification and modifies the internal definition of the current hierarchical output structure. This causes node promotion and collection to occur automatically. This operation dynamically modifies the output XML structure by removing unselected nodes. A Prototype example of all these capabilities is shown in Figure 16. This flexible and dynamic operation is not available in XQuery.

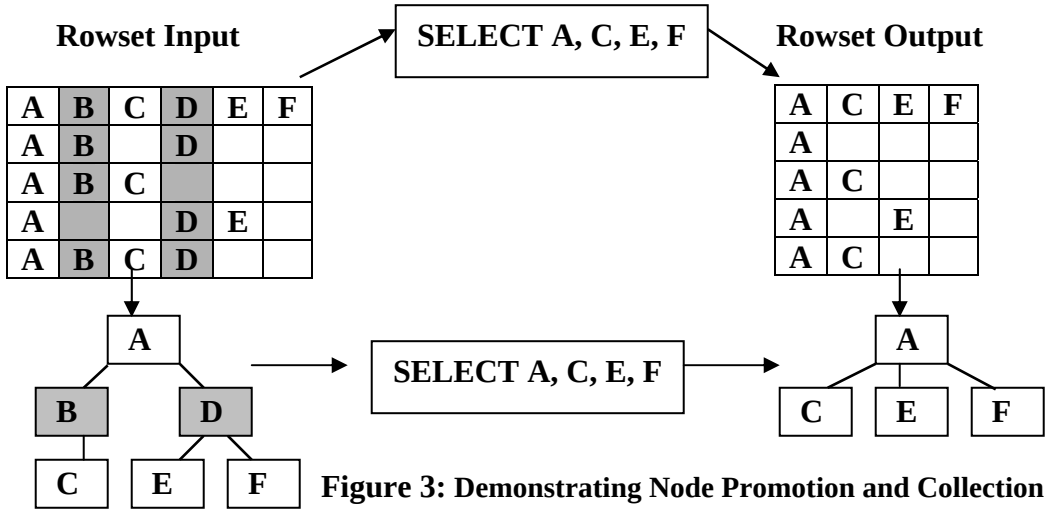


Figure 3: Demonstrating Node Promotion and Collection in SQL

3.2.2.2) Mapping SQL Operations to Hierarchical Processing

The SQL SELECT operation specifies the hierarchical processed data to be returned which will be hierarchically structured and formatted in XML. If nodes have no data selected for output, the node is removed and node promotion occurs. The FROM clause specifies the input data and how it is hierarchically modeled and constructed on input using Left Outer Joins to specify the hierarchical data modeling. The FROM clause ON keyword filters data down a path which controls hierarchical data preservation. The WHERE clause operating on the entire row, specifies the hierarchical filtering that takes place across the entire hierarchical result structure filtering data nonlinearly as shown in Figure 2 and shown in Figure 5. The basic hierarchical operations of the SELECT, FROM and WHERE SQL operations are shown in Figure 4 below and are very intuitive with hierarchical data structures. Notice that the C2 data occurrence and its D2 descendent occurrence was filtered out by the WHERE clause.

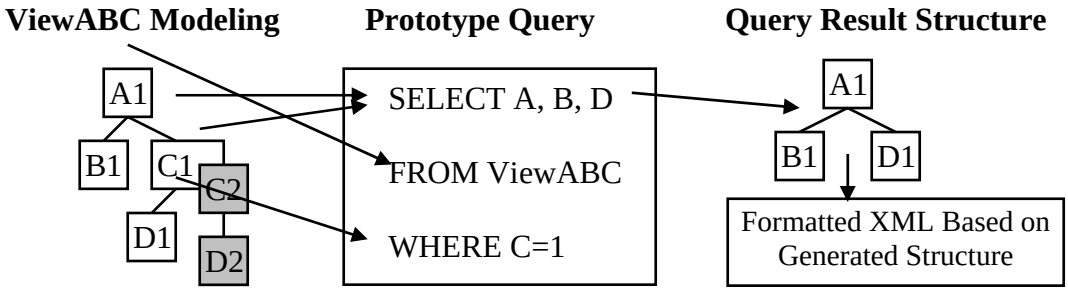
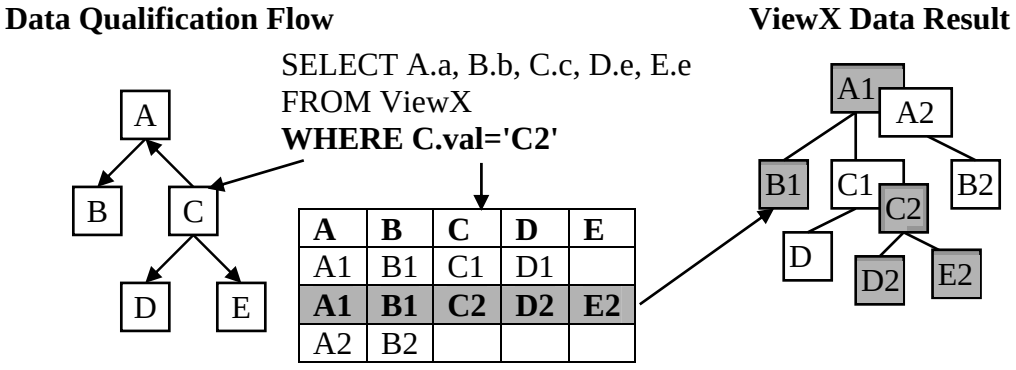


Figure 4: SQL hierarchical query specification and operation overview

It is important to realize the significance of hierarchical data filtering and its relationship to hierarchical processing. Hierarchical processing is occurring in relational processing, but can not be really understood or appreciated until it is seen in hierarchically formatted data to understand its complexity of operation and hierarchical semantic meaning. This is graphically demonstrated below in Figure 5. With hierarchical data filtering, it may be easier to think in terms of qualifying data rather than filtering it. In this example you will notice that nodes A and B can also be influenced by the WHERE operation which starts with Node C and still qualifies node A, and node A qualifies node B. The qualification of node B involves LCA processing and will be covered further below.



A	B	C	D	E
A1	B1	C1	D1	
A1	**B1**	**C2**	**D2**	**E2**
A2	B2			

Figure 5 WHERE clause hierarchical data qualification flow

Additionally, a FOR XML clause shown later in Figure 17 is also supported at the end of the SQL query to specify overrides for the automatic XML processing. For example, it can be used to specify a different type of XML formatting than the default attribute mode formatting, or to specify that no collection node is to be supplied, or it can override automatic node promotion so that full paths to data are preserved even when nodes are not selected for output.

3.2.2.3) SQL Specific LCA Cartesian Product Processing

The LCA processing is automatically performed in two locations in SQL processing and it occurs naturally thanks to the SQL Cartesian product processing and its replicated data generation. The replicated values are necessary because the SQL processing is performed a row at a time, usually without memory across row occurrences, replicated data provides this memory. For SQL WHERE processing, this allows all combinations of values that require testing to be generated under their appropriate LCA node type. When testing a WHERE condition across pathways, any correct combination of the multiple path occurrences qualifies the condition. This simulates hierarchical tree walking and is very efficient in SQL. An example of SQL WHERE LCA processing in the Prototype is shown in Figure 20.

The SQL SELECT operation also requires LCA processing when selecting data across pathways from one path of the structure based on data from another path. This is handled hierarchically by determining the LCA between the tested conditional node and the data node to relate the two nodes relationship across the nonlinear access path. A positive test condition will usually qualify a group of related data under the current LCA data node occurrence. This too, is handled automatically by the proper data replication under the LCA node even when the rows are tested a single row at a time, avoiding hierarchical tree walking. An example of SQL SELECT LCA processing in the Prototype is shown later in Figure 21. Both of these LCA types (WHERE and SELECT) can be used together in processing a query and this is shown in Figure 22. This also demonstrates why nonlinear logic which must use LCA processing can not practically support using user navigation because of the complexity involved.

3.2.2.4) SQL Hierarchical Joining of Hierarchical Views

Standard SQL views can be used for hierarchical modeled views. They enable very powerful hierarchical abstraction even supporting conceptual hierarchical structure processing. These hierarchical views can be hierarchically combined using the Left Outer Join in the same way the views were built themselves. They combine into a virtual unified view structure, such as: *DeptView Left Join EmpView ON DeptID=EmpDeptID Left Join DpndView ON EmpID=DpndEmpID.* Each of these views defines a multi-path nonlinear hierarchical structure. This operation naturally and hierarchically combines the views between any link points in each structure using simple single join operations. This is naturally supported by the SQL view expansion which is shown later in Figure 15. This shows how powerful and flexible hierarchical views are. A working prototype example is shown in Figure 15.

The SQL hierarchical view operation can be dynamically specified with varying specifications to fit the requirements which will adapt to fit the query. The SELECT list dynamically and easily specifies the values to be output and processed. The Left Outer Join is more than just processing instructions; it also represents the hierarchical structure metadata that defines the hierarchical structure and its associated semantics which defines its hierarchical processing. This means this meta information can be utilized for value added enhancements described shortly.

3.2.2.5) Structure Fragment Control

Structure fragments (isolated portions of the structure) can be created by which nodes are selected for output. Selecting portions of the structure that are not connected by a common selected node produces multiple different fragments. These fragments will be output under their collection node introduced by the Prototype to keep them contained. The user can still override this default operation and specify that no collection node is to be used, if desired for some special use. The separate fragments can also be separately manipulated and joined by using the SQL alias capability to separately identify and name the fragments so they can be separately manipulated in SQL operations. This capability is utilized in the structure transformation capability shown in Figure 24.

3.2.3) SQL Advanced Hierarchical Processing

The Prototype contains powerful and advanced hierarchical capabilities performed naturally in SQL. These are: hierarchical transformation; variable structures; XML duplicate and shared nodes; linking below lower structure's root; SQL nonlinear hierarchical optimization; and a nonlinear hierarchical ORDER BY.

3.2.3.1) Hierarchical Data Structure Transformation

It turns out that the Prototype's rowset manipulation through SQL's SELECT, FROM, WHERE and alias capability is quite powerful. This happens by using the SQL alias qualifier capability with the SELECT operation to isolate and group different nodes into fragments and rejoin them differently using the FROM clause's Join capability. Fragment groupings represent valid sub hierarchies. No matching values are required within the fragment grouping because of their contiguous data storage in the rowset. Rejoining fragments requires relationship values to join on which are not limited to their previous join criteria. This is shown in Figure 24.

ANSI SQL Reshaping is also possible; it can perform any-to-any structure transforms without using any available relationships. Reshaping operates by utilizing the semantics in the data structure and preserves the original semantics, while a restructuring operation deliberately changes the semantics by introducing different relationships. The transformed structure is recognized and utilized by SQL in the overall unified structure. Figures 25-27 show reshaping examples produced from the Prototype. Restructuring and reshaping can both be written to operate as a polymorphic operation where the structure of the source structure does not have to be known when the transformation is defined or performed. A Prototype example is shown later in Figure 28 of how polymorphic operation is performed. Other previous proposals to nonlinear structure transformations can be found at (Zhang & Dyreson, 2006).

3.2.3.2) Variable Structure Generation

It was mentioned earlier that the Outer Join's ON keyword operation is specifically applied to the join location it was specified on and that ON join operations take affect as the query's hierarchical structure is being built. These ON clauses can also contain filtering information to control if the particular node or view substructure occurrence is to be generated or not. This filtering can test a value anywhere on the active path upwards within its active working set. This allows a value in the upper path to control whether a node or view substructure occurrence is to be built or skipped along with all of its related lower level node occurrences. This can control structure generation at the node occurrence level. The same value can control which of a number of different node types or view substructures is selected. This operation is similar to COBOL's Depending On option and it is shown in the Prototype example in Figure 23.

3.2.3.3) XML Duplicate and Shared Nodes

XML supports duplicate and shared (IDref) elements (nodes). Both of these XML capabilities model network structures because the node type has a multiple path choice to the same node type. This means they are ambiguous for nonprocedural navigationless database languages like SQL. These capabilities are usually included for markup use rather than for database. This does not mean that these ambiguous nodes must cause problems for SQL. SQL can use its alias rename ability shown by its AS syntax directly below to change the name of these nodes separately for each path so that they can be referenced unambiguously as shown in Figure 6.

```
SELECT * FROM Dept
LEFT JOIN Cust ON DeptID=CustDeptID
LEFT JOIN Emp ON DeptID=CustEmpID
LEFT JOIN Addr AS AddrC ON CustID=AddrC.AddrCustID
LEFT JOIN Addr AS AddrE ON EmpID=AddrE.AddrEmpID
```

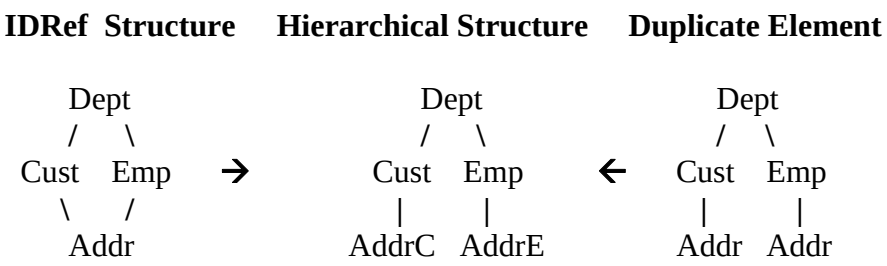


Figure 6: Hierarchical mapping solutions for network structures

It might be argued that this solution does not utilize XML's duplicate element type's usefulness and markup LCA logic solutions have taken into account duplicate element types by using the first one that currently exists in an XPath search. This means that an LCA node's position dynamically changes unpredictably. This is acceptable for variable semistructured markup data and searching it hierarchically, but for database data use and its variable hierarchical processing it is not acceptable to have an LCA that is moving around unpredictably because each different position of the LCA has a different semantics which means a different meaning. This is why multipath LCA logic must operate differently for database use than markup use. Database's unambiguous data structure enables its navigationless capability which also increases its hierarchical processing capabilities.

3.2.3.4) More Capability by Linking Below Lower Structure's Root

In a previous section there was an example of joining full hierarchical structures but no specific linking (joining) requirement was mentioned. Usually when hierarchically structures are linked together, the lower structure had to be linked at its root node otherwise the semantics caused problems. Since SQL hierarchical processing has handled all other hierarchical operations correctly automatically, we analyzed how SQL naturally handles links below the root and it made perfect semantic sense. This being the case, it can be allowed so that the user continues not having to be concerned with the knowing the structure. The semantics of linking below the root structure are that data filtering occurs at the lower link points, but for data modeling purposes, the lower level root still remains the data modeling link point. This makes sense, since the root and upper level nodes still have an influence on materializing the structure view making the operation very intuitive and semantically accurate. The unlimited flexibility of this joining of hierarchical structures qualifies this operation as a hierarchical data structure mashup. A Prototype example is shown later in Figure 18.

These same semantics and results are consistent with logical relational structures as well as physical XML structures. The perceived problem of referencing a logical relational structure below its root before the root node is accessed does not present a problem. Outer Join expanded views representing optimized structures materialize at their first reference because they push the ON clause to the right when they expand with their embedded ON clauses. For example: *View1 Left Join View2 ON V1=V2*, with View2 expanded: *View1 Left Join X Left Join Y ON X.x=Y.y ON V1=V2*. The ON clauses nesting causes View2's expansion to push its ON clause to the end of all the ON clauses expanded in View2. This causes View2 to materialize fully by joining its X and Y tables using their ON clause *ON X.x=Y.y* before being joined to

View1 which requires the last ON clause *ON V1=V2* which joins both fully materialized views. This makes all data available in View2 before it is joined to View1. This further demonstrates how the Left Outer Join syntax is a very powerful hierarchical data modeling language.

3.2.3.5) SQL Multipath Hierarchical Optimization

The Prototype adds support for nonlinear hierarchical optimization which works for ANSI SQL relational processing and can be applied to all physical hierarchical structures. This has not been possible before the addition of the hierarchical Outer Join because SQL's standard Inner Join view optimization has to insure that the integrity of the view is maintained. For this reason, all tables in an Inner or default Inner Join view are usually accessed even if no data is required from the table since a missing match value anywhere in the view can cause the removal of all data in the row. For example a Department view comprised of the Department table Inner Joined to the Employee table will also have the Employee table accessed even though the Employee table is not referenced.

The standard Inner Join logic is not used when hierarchical structures are being modeled and accessed hierarchically by a Left Outer join so that the department's existence is not dependent on employee's existence. In this case, the Employee table join can be dynamically removed from the Query at execution depending on the query request and as long as it does not exist on a path to another required node demonstrated in Figure 7.1. This optimization is easily applied to SQL by rewriting the query removing unnecessary joins as shown in Figure 7.2. This optimization was described in (Ullman, 1989) as a capability of the Universal Relation. It can reduce the number of tuples returned because of additional needless replications being eliminated which some consider an invalid optimization rather than an advantage, but this problem is negated because of the XML output which removes replications anyway. Since the Prototype is a hierarchical processor, it only accepts hierarchically modeled SQL views. This allows it to be hierarchically structure-aware at all times to always take advantage of hierarchical processing capabilities.

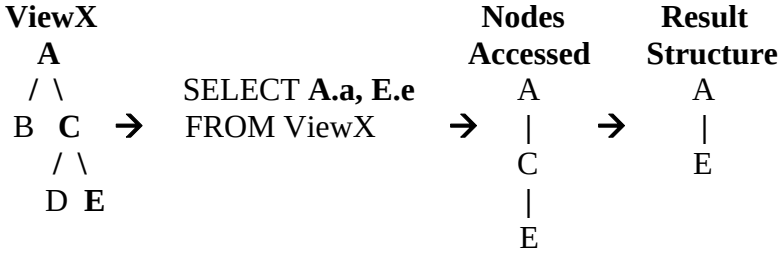


Figure 7.1: Hierarchical access optimization

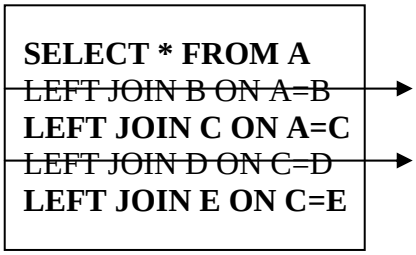


Figure 7.2: Optimized Rewrite

There are other approaches to join optimization in XML-to-SQL translation that are based on a relational algebra plan optimization. These techniques can and should be applied after the above hierarchical optimization. The SQL hierarchical XML processor prototype's relational processor performs this additional round of relational processing. In fact, the previously performed hierarchical optimization

should significantly aid the operation and gain efficiency of the following standard relational processing optimization. In addition, the hierarchical optimization with knowledge of the hierarchical structure can perform optimizations that may not be identified by relational optimization. These secondary level of possible hierarchical processing optimizations that can be applied afterward can be found at (Bao, Ling & lu, 2008; Mani, Wang & Dougherty, 2006).

3.2.3.6) Global Views and Global Queries

As mentioned already, the Prototype's hierarchical access is optimized so that each view executed is optimized at execution so that unneeded paths are not accessed. This means there is never overhead for using views that contain larger views than required at allow for unpredictable uses. This means that global views can be utilized for ease of use and reuse, and is further enhanced since the user does not need to know the structure or perform navigation. Combined with SQL's variable SELECT list this makes for a powerful dynamic structure transformation. This also opens the capability of Global Queries that can perform global operations on the entire global view and output all of its data. This is another area where today's procedural navigational XML access processors can not handle. But with SQL used by the Prototype all it takes is the use of a SELECT ALL, "SELECT *", to access all data in the global view and output it. Operations like hierarchical data filtering with a WHERE clause can easily filter the entire query range and output the results. A Prototype example of a global query is shown later in Figure 29.

3.2.3.7) Multipath SQL ORDER BY Operation

SQL ordering of nonlinear hierarchical structures does not relate fully logically to the SQL order operation unlike all the other SQL operations covered thus far. There are two reasons for this. First, ordering against a hierarchical data structure such as ordering Employee node before the Department node when the structure is Department over Employee will inadvertently alter the hierarchical structure placing Employee over Department. This is demonstrated below in Figure 8. Second, in a nonlinear hierarchical structure, each hierarchical path can be ordered separately. This was solved by changing the SQL ORDER BY semantics in the Prototype middleware to take the nonlinear hierarchical structure into account to arrange the sort fields down the separate paths of the hierarchical structure. This allows paths of the structure to be separately and independently ordered which will be demonstrated in Figure 31.

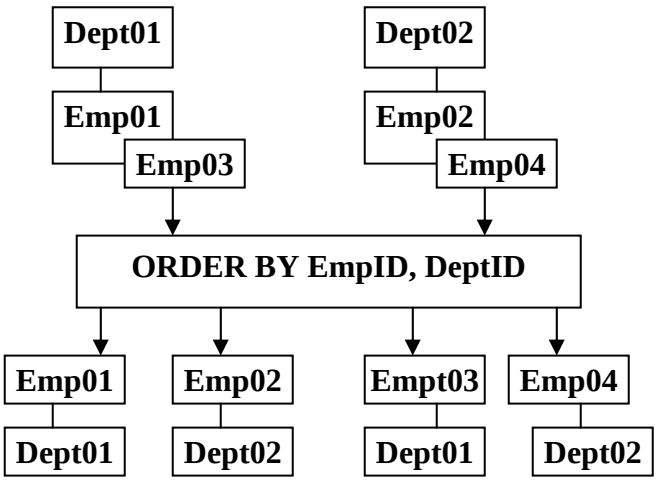


Figure 8: Ordering out of hierarchical order can inadvertently change the structure

The problem with performing multipath hierarchical ordering is that each path has to be separately ordered. Using the structure in Figure 9 it can be seen that ordering paths A/B and A/C would not be possible in SQL using its standard ORDER BY semantics. But it is necessary for XML hierarchical

output which can be nonlinear to be able to order each pathway separately in isolation. The Prototype can accomplish this as demonstrated in the example in Figure 31.

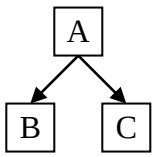


Figure 9: Separate Ordering

3.2.3.8) Preserving XML Natural Order and Removing Replicated Data

We differentiate the terms duplicate data and replicated data. Replicated data is spurious data that was introduced from performing relational operations such as joins and Cartesian product. Duplicate data is meaningful data that represents an actual meaningful data occurrence. The Prototype removes replicated data from XML results and preserves duplicate data. The same internal procedure used for tracking valid duplicate data is used to preserve XML data ordering unless it is reordered by an ORDER BY operation. This is achieved by attaching a value to each node data occurrence identifying its unique node type (name) and its occurrence value within the node type. Since our processing is hierarchically based, our solution can be a combination of node-identity and value based techniques. The issues involved can be found in (Krishnamurthy, Kaushik & Naughton, 2004). This is demonstrated in Figure 27.

3.2.3.9) Renaming, Replication, and Reorganizing of Nodes

SQL's aliasing allows views, tables, and columns to be renamed. This is seamlessly extended in the Prototype to enable renaming of nodes and their data items which is subsequently transferred to XML output. In addition, renaming allows the replicated of tables with the ability to separately control the data field inclusion in the tables so that different data items can exist in each causing it to be reorganized. These replicated renamed tables also correspond one-to-one to nodes in the hierarchical structure being built and processed with the result seamlessly output in XML. A Prototype example of these renaming capabilities is shown later in Figure 32.

3.2.3.10) Changing default Operations with FOR XML

The SQL "FOR XML" syntax is used by most SQL processors that support XML to override or specify runtime operating characteristics for XML processing. The Prototype also uses it to override operations of its advanced hierarchical processing.

Automatic node promotion around unselected nodes is very powerful, intuitive and is probably desired most of the time. If the original structure is desired with intervening unselected nodes represented by empty nodes so that it can be navigated by existing queries, this is possible by overriding node promotion with a FOR XML KEEP NODE specification. This FOR XML operation can be seen in Figure 17.

There are three formats for formatting XML. These are Attribute, Element and Mixed mode. Attribute uses XML attributes to store the data, Element mode uses XML Elements to store the data and Mixed mode is a combination of both. The default XML format output for the Prototype is the Attribute mode, the ELEMENT keyword is specified on the FOR XML clause to indicate that the Element formatting mode should be used for XML formatting. Currently the Mixed mode is not supported for output since there is no easy way to specify which data is to be stored in the Element data type for each node type. With XML input all three input XML format modes are supported automatically. This FOR XML operation can be seen in Figure 17.

Using the FOR XML syntax is optional. Using it automatically resets the default output document tag collector that is named “root”. Using FOR XML without specifying a new name for the document tag collector using the UNDER “tag-name” indicates that no document collector tag is used. In Figure 17, you can notice that no document collector tag name was specified on the FOR XML clause so it produced two separate documents in this example.

These FOR XML examples should demonstrate how in the case of automatically generated output can be additionally tailored and controlled easily.

4) ANSI SQL PROTOTYPE MIDDLEWARE XML ENABLER

There are three choices for how to build the ANSI SQL native XML hierarchical processor prototype. Build it from scratch, modifying an existing ANSI SQL processor, or build a middleware product that sits around an ANSI SQL processor to utilize its inherent hierarchical nonlinear processor as shown below in Figure 10. All of these implementation solutions can produce the same results with different advantages for each. The most economical for a prototype is the ANSI SQL driven middleware solution we chose. It also offers the best proof that the nonlinear hierarchical processing technology involved is ANSI SQL standard. It also offers the user the most user friendly, economical, and least disruptive solution requiring no XML training or SQL system conversion for the customer. The Prototype sits around the customer’s ANSI SQL processor and its already in place data and will automatically supply it with pre, post, and real-time processing to support native XML at a full hierarchical level. This middleware process is from input through hierarchical processing to output without modifying the SQL processor.

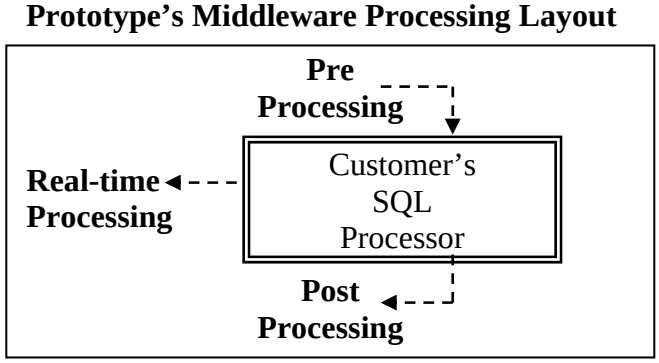


Figure 10 Middleware relationships to customer’s SQL processor

4.1) Prototype Pre Processing

The preprocessing involves accepting the ANSI SQL hierarchical request, determining the hierarchical data structure, building control blocks for its pre and post processing, and then optimizing, rewriting, and submitting the reworked SQL to the underlying SQL processor. The fully joined input data structure is determined by analyzing the hierarchically modeled input SQL. The knowledge of the active data structure allows structure-aware operation in the real-time and post processing phases. The optimization is determined by analyzing the hierarchical structure defined from the specified SQL to determine table or nodes that do not require access and removing them from the input SQL. This is possible because of hierarchical data preservation semantics supported by the Left Outer Join operation (allowing for shortened pathways or dangling tuples). The necessary setup operations required for preserving XML natural order and distinguishing replicated data from duplicate data described in Section 3.2.3.8 are

performed here. Tables and views are also defined to the preprocessor, and XML data documents can be optionally pre shredded into tables to be used by SQL hierarchically when accessed. At this point, the customer's SQL processor is invoked using the preprocessed hierarchical SQL.

The access of XML in this pre processing step is performed internally using a SAX parser. The Prototype has limited its access to SQL standard processing of only fixed hierarchical structures which require no user navigation. The DOM XML parser was designed for very intricate XML navigation and requires much overhead and memory usage which is why SAX was used. SAX under these fixed structure circumstances works out nicely and efficiently only retrieving the necessary data based on the access optimization. A specialized XML parser for structured data will be developed and used in the future.

4.2) Prototype Real-time Processing

The Prototype's real-time processing step is occurring asynchronously with the SQL processing and is structure-aware of the hierarchical structure being processed using the metadata derived from the pre processing step. It processes requests from the SQL processor to service EII real-time requests for native XML input. It returns a relational rowset that is naturally mapped by its associated view's hierarchical SQL metadata supplied from the preprocessing step. This maintains the XML's hierarchical input and processing transparency. The method of interfacing to the real-time processor for real-time XML input by the SQL processor is supported externally by using UDF or External table support for external XML input.

This real-time step was not implemented for the initial Prototype because it was not necessary in order to prove the ANSI SQL transparent XML nonlinear hierarchical processing capability. It can be proven by relying on the same XML shredding performed in ETL mode in the pre processing step which is run previously to load the XML once and accessed many times. When the Real-time processing is implemented, the XML dynamic access in the real-time step will use its own optimized XML access processor and parser. It is designed to take full advantage of the reduced fixed format structured XML used by SQL and its processing is tailored to the specific query. Only the active query's required data will be retrieved.

4.3) Prototype Post Processing

The Prototype's post processing phase involves automatically transforming the hierarchically processed relational result rowset into the structured formatted XML based on the resulting hierarchical output structure. The joined input hierarchical structures used during processing can change the structure being processed for output and so does the removal of nodes that are not selected for output resulting dynamic in node promotion. The Prototype's structure-aware capability allows for the managing of the data structure from input, through processing to output.

Data replications are recognized in this phase and eliminated from the XML result and valid data duplicates are preserved and output for XML and relational data. The hierarchical optimization performed in the preprocessing stage should significantly help keep these data explosions smaller. Nonlinear hierarchical ordering is also performed in the post processing phase. The setup for these operations was described in the preprocessing in Section 4.1.

5) ANSI SQL TRANSPARENT XML SUPPORT EXAMPLES

The examples shown in this section are taken from a functioning ANSI SQL transparent full nonlinear hierarchical XML processor prototype. The examples start with the hierarchical modeling basics continuing through hierarchical processing to advanced transformations examples. This online interactive prototype can be invoked now at **http://www.adatinc.com/prototype.html** to dynamically test the examples that follow. Instructions on its use will be made available online.

5.1) Example Structure and Data

This section contains the data and data structures used in the SQL hierarchical processing examples. The examples use diagrams demonstrating the data flow and structure processing demonstrating the entire operation which are explained here. Usually the input and output structures are shown as separate structures demonstrated below in Figure 11.

```
SELECT EmpID, DpndID, InvID, AddrID
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID
```

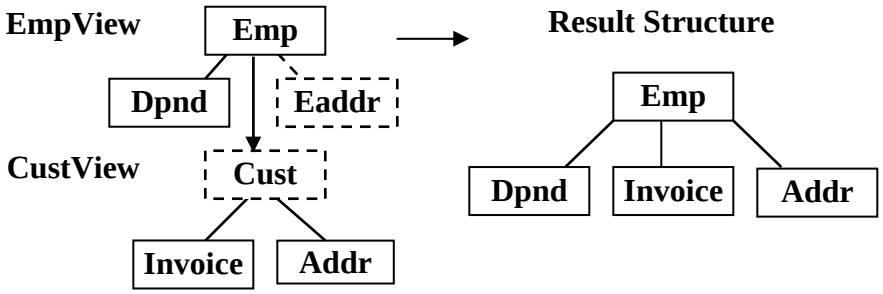


Figure 11: Sample diagram structure

In Figure 11 above, solid boxes connected by solid lines are used to identify logical and physical hierarchical structures. Solid boxes represent nodes that have been selected for output. Solid lines connect selected boxes into the active structure. These structures can be combined with SQL joins and solid arrows show the connection points where the structures are combined to form a new hierarchical structure. Usually structures are combined at the same point as their linkage relationship, but this can be different when linking below the lower level root. In this case, this linkage relationship is indicated by a dashed arrow. Dashed boxes represent nodes that were not selected for output. If unselected nodes are not referenced or on a path to referenced node it is not accessed and this is indicated by being connected by a dashed line. This is summarized in Table 1 below.

In Figure 11 the solid boxes are selected so they are transferred to the result structure. The Cust node is not selected but is required for input since it is referenced for structure joining purposes. The Eaddr node is not referenced and is not selected so it not accessed at all indicated by being connected by a dashed line.

Hierarchical Structure Processing Diagram Definitions
Solid Box: Node is selected for output Dashed Box: Node is not selected for output Solid Line: Connects nodes into active structure Dashed Line: Connects nodes not in active structure Dashed arrow: Qualification flow or a note pointer Solid arrow: Structure Connector and default qualification flow (unless overridden)

Table 1

The ANSI SQL transparent XML hierarchical processing examples shown in this section utilize the hierarchical structure and data in Figure 12.

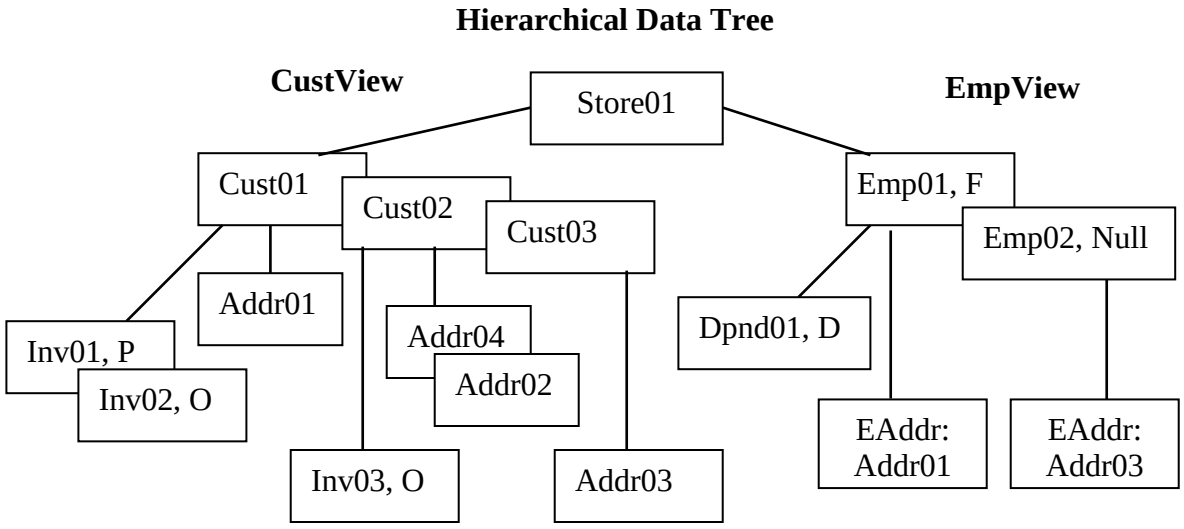


Figure 12: Data structure and data used in examples

5.2) Start of Processing Examples

5.2.1) Views Defined and Joined

In this section, two hierarchical views are created and then joined hierarchically into a larger hierarchical structure at execution time for processing. The first view defines relational data and models it hierarchically while the second view defines a physical hierarchical structure which includes its hierarchical structure. The joining of these two views demonstrates the capability to seamlessly join heterogeneous structures in ANSI SQL.

5.2.1.1) Relational Data Hierarchical Query View Defined

The Prototype example in Figure 13 demonstrates the creation of the relational Employee view EmpView. This includes the Left Outer Join that models the logical data structure to be processed. EmpView is invoked and it automatically determines and produces structured XML in its default attribute format. Notice that the view can contain ON clause filtering, *AND DpndCode='D'*, for the Dpnd node. This is very similar to XPath filtering.

The SQL syntax models the hierarchical structure and its associated hierarchical semantics defines how the SQL engine processes the data hierarchically. The Prototype dynamically determines the hierarchical structure based on its SQL data modeling of the input data. This allows it to perform hierarchically structure-aware which enables it to automatically format the hierarchical result in the relational rowset to XML for output.

```
CREATE VIEW EmpView AS
SELECT * FROM Emp
LEFT JOIN Dpnd
   ON EmpID=DpndEmpID AND DpndCode='D'
LEFT JOIN Eaddr
   ON EmpCustID=EaddrCustID
                ↓
SELECT EmpID, DpndID, EaddrID
FROM EmpView
```

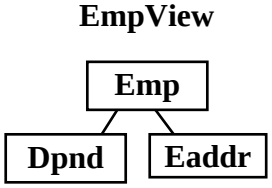


Figure 13: EmpView structure

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <eaddr eaddrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
  <eaddr eaddrid="Addr03"/>
 </emp>
</root>
```

5.2.1.2) XML Document Hierarchical Query Defined

This Prototype example demonstrates the creation of the XML Customer view CustView. This includes the XML create view which uses a hierarchical syntax for defining physical hierarchical structures. It will automatically and transparently create the hierarchical Left Outer Join view that models it in Figure 14 below. The SQL view has the same name as the XML view name for transparency. The SQL CustView is executed and automatically produces the structured XML following it. It preserves its XML hierarchical order described previously in Section 3.2.3.8 unless overridden by an ORDER BY statement.

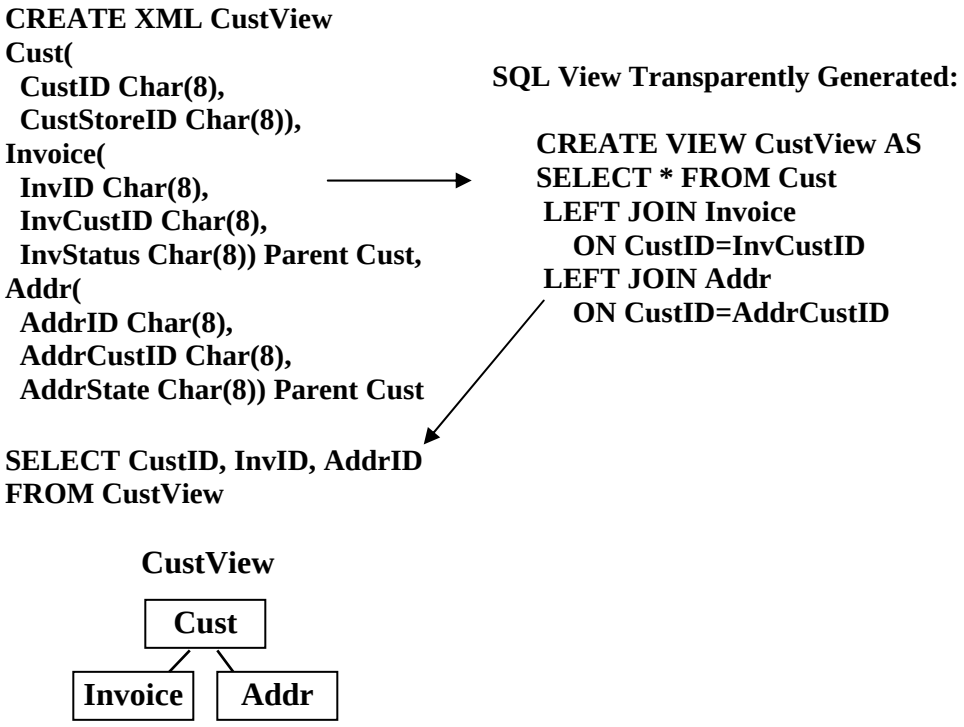


Figure 14: CustView structure

```
<root>
  <cust custid="Cust01">
   <invoice invid="Inv01"/>
   <invoice invid="Inv02"/>
   <addr addrid="Addr01"/>
  </cust>
  <cust custid="Cust02">
   <invoice invid="Inv03"/>
   <addr addrid="Addr02"/>
   <addr addrid="Addr04"/>
  </cust>
  <cust custid="Cust03">
   <addr addrid="Addr03"/>
  </cust>
</root>
```

5.2.1.3) Heterogeneous Join of Two Multipath Hierarchical Structures

In the example in Figure 15, the Employee relational view, EmpView, is joined over the Customer XML view, CustView, under its Emp node as determined by the ON join condition. For this to work in all situations, CustView must be fully materialized before being joined to EmpView. This is the first example of a join of hierarchical structures; notice how simple it was, a simple single join of the conceptual SQL views. Below this view joining SQL example is the automatic view expansion to create a single sequence of SQL's Left Outer joins as shown below. This expansion of the views causes the previous views to still be separately materialized because of the nesting of the ON clauses between views. Joins are not performed until their associated ON clause is encountered. Stacking is allowed. This

example also demonstrates a heterogeneous join of relational and XML data which is seamless and transparent. All views operate the same regardless of their structure type which can be logical or physical.

You can notice in the result structure that the CustView root node Cust linked under the Emp node was added after Employee' s Eaddr node because it was the last sibling node under Emp. This is the default because sibling nodes are added to the structure being built in a left to right order. Sibling order in hierarchical structures has no semantic difference, except that when navigating the structure order this could be important. This example, as all others in this document can be interactively performed. Most XML access products today require procedural navigation which limits their ability to be interactive.

SELECT EmpID, DpndID, EaddrID, CustID, InvID, AddrID
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID

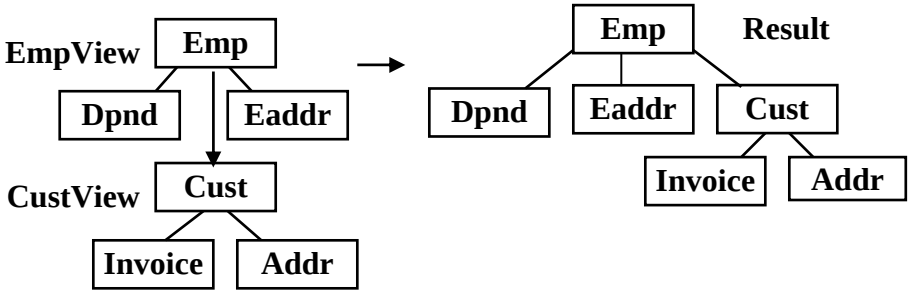


Figure 15: Heterogeneous structure join of relational and XML data

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <eaddr eaddrid="Addr01"/>
  <cust custid="Cust01">
   <invoice invid="Inv01"/>
   <invoice invid="Inv02"/>
   <addr addrid="Addr01"/>
  </cust>
 </emp>
 <emp empid="Emp02">
  <eaddr eaddrid="Addr03"/>
  <cust custid="Cust03">
   <addr addrid="Addr03"/>
  </cust>
 </emp>
</root>
```

5.2.2) Node Promotion Processing

This section demonstrates the standard hierarchical operation of node promotion occurring around nodes that were not selected for output. There are times when node promotion is to be replaced by including empty nodes to be used as placeholders to preserve the original structure which is also demonstrated in the second example in this section. Earlier Section 3.2.2.1 showed why hierarchical node promotion is actually a natural operation of the Relational SELECT operation making it a natural mapping for relational to hierarchical processing.

5.2.2.1) Node Promotion, Logical View Join and Modified View Invocation

In this example, the Employee view, EmpView, is joined over the Customer view, CustView and the result is stored in a SQL view, EmpCust, used in this example. This is an example of a logical view comprised of two embedded views. Logical views can be comprised of physical and logical views. They operate the same as physical views, there is no difference.

When this EmpCust view is invoked in this example, the Eaddr and Cust nodes are not selected for output. This causes the Eaddr node not to be accessed and node promotion to occur around the Cust node which also includes node collection because more than one node was added to the node above. This dynamically changes the structure being processed controlled by the data items not selected in the SQL SELECT list. These unselected nodes are indicated in Figure 16 by the dotted boxes. The result structure is shown in below with the result XML. This demonstrates the flexibility and power of SQL Hierarchical views to dynamically modify the view based on the SELECT operation. This is automatically supported in SQL, but is missing in other XML processors. The Result structure below in Figure 16 also demonstrates an example of Data Collection caused by the Node Promotion where multiple nodes (Invoice and Addr) are appended to the next higher level existing node increasing its number of direct child nodes. Excluding portions of the structure including the root will create structure fragments which can be useful on their own and used in hierarchical operations such as transformations shown later.

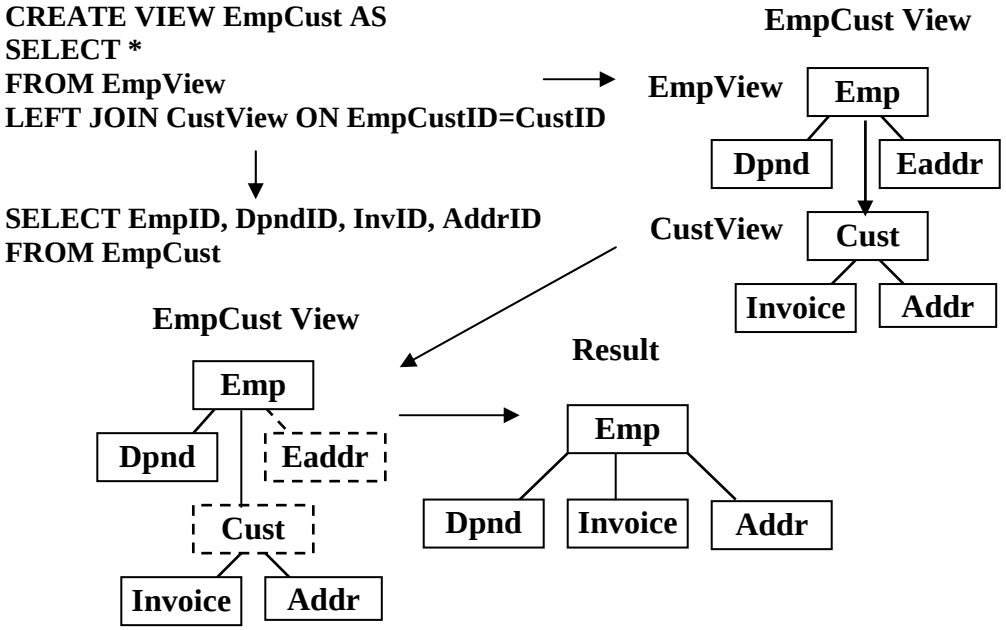


Figure 16: EmpCust node promotion and collection result structure

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <addr addrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
  <addr addrid="Addr03"/>
 </emp>
</root>
```

5.2.2.2) Node Promotion Override Shown with XML Format Override

This is basically the same query as in the previous example in Figure 16 except with ELEMENT mode format specified for XML in the FOR XML query syntax and with empty intervening nodes preserved in the output by also specifying KEEP NODE on the FOR XML query syntax. Notice that the empty unselected Cust node is included in the XML Element mode formatted result. When FOR XML is specified, the default UNDER Root keyword specification is not automatically supplied and if not supplied by the user, no collection node is used. This is the case in this example below in Figure 17.

SELECT EmpID, DpndID, InvID, AddrID
FROM EmpCust FOR XML ELEMENT KEEP NODE

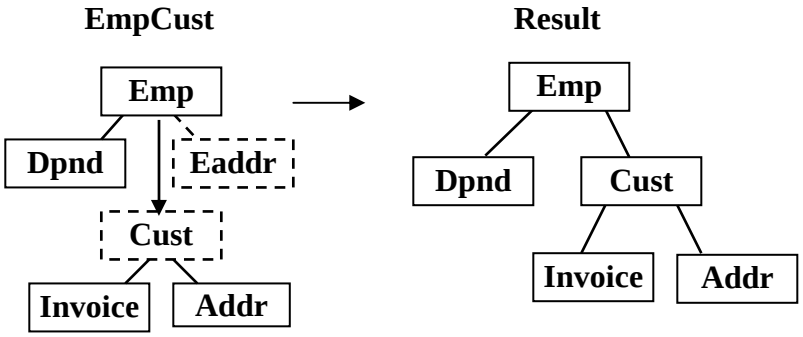


Figure 17: EmpCust node promotion overridden and element format used

```
<emp>
<empid>Emp01</empid>
<dpnd>
 <dpndid>Dpnd01</dpndid>
</dpnd>
<cust>
 <invoice>
  <invid>Inv01</invid>
 </invoice>
 <invoice>
  <invid>Inv02</invid>
 </invoice>
 <addr>
```

Continued:

```
   <addrid>Addr01</addrid>
  </addr>
 </cust>
</emp>
<emp>
 <empid>Emp02</empid>
 <cust>
  <addr>
   <addrid>Addr03</addrid>
  </addr>
 </cust>
</emp>
```


5.2.3) Advanced Data Structure Mashups

In the past, linking below the lower level hierarchical structure's root node was not allowed for physical structure reasons or more importantly because of hierarchical data modeling semantic issues. Just what are the semantics? It turns out that inherent ANSI SQL hierarchical processing had solved the solution naturally in its inherent nonlinear hierarchical processing. This capability enables hierarchical structures to be joined in any way to support data structure mashups that are hierarchically correct and meaningful.

5.2.3.1) How to Link Below the Lower Level Structure's Root Node

In this example in Figure 18, the lower level structure is linked to below its root to the Addr node as specified in the ON condition. Filtering of the lower level will occur at this lower link (join) point of the lower structure.

The hierarchical data modeling is not affected by the lower link point; the lower level root is still used as the hierarchical connection point. The reason for this is that the lower level view root and its descendents still have their same data modeling responsibility and semantics for materializing the view before it is joined. This occurs naturally because of the ON clause nesting in lower level views which was described previously. This also works perfectly for physical structures such as XML. This allows linking the two hierarchical structures from any point to any point in each structure qualifying this capability as a hierarchical data mashup.

```
SELECT EmpID, DpndID, EaddrID, CustID, InvID, AddrID
FROM EmpView LEFT JOIN CustView
        ON EaddrID=AddrID
```

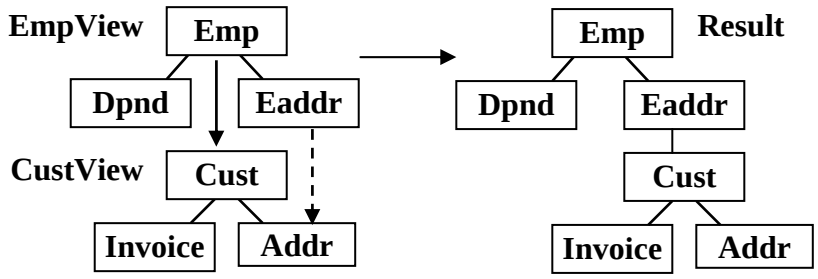


Figure 18: Linking below lower level structure root

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <eaddr eaddrid="Addr01">
   <cust custid="Cust01">
    <invoice invid="Inv01"/>
    <invoice invid="Inv02"/>
    <addr addrid="Addr01"/>
   </cust>
  </eaddr>
 </emp>
 <emp empid="Emp02">
  <eaddr eaddrid="Addr03">
   <cust custid="Cust03">
```

```
    <addr addrid="Addr03"/>
   </cust>
  </eaddr>
 </emp>
</root>
```

5.2.3.2) Linking Below Root of Lower Level Structure with Node Promotion

This is the same query as the previous query in Figure 18 except the Cust and Eaddr nodes are not selected for output. Unselected nodes can still be referenced such as the Eaddr node is. The Cust node is also not selected for output, this causes the Invoice and Addr nodes to be promotion around the unselected Cust node. These unselected nodes are indicated in Figure 19 by the dotted boxes. This feature will be utilized later in examples to support extremely powerful structure transformations.

```
SELECT EmpID, DpndID, InvID, AddrID
FROM EmpView LEFT JOIN CustView
        ON EaddrID=AddrID
```

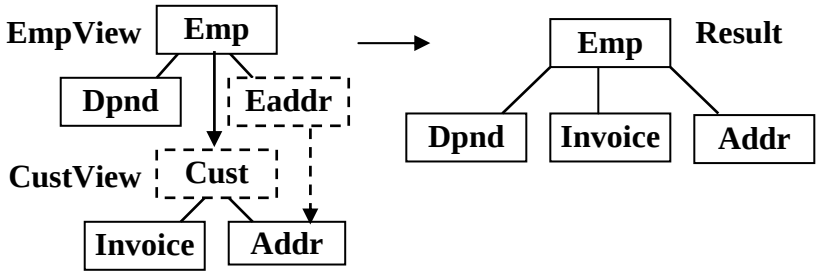


Figure 19: Linking below lower level root with node promotion

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <addr addrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
  <addr addrid="Addr03"/>
 </emp>
</root>
```

5.2.4) Multipath Lowest Common Ancestor (LCA) Processing

As described previously in a number of locations, LCA processing is required to process hierarchical processing involving more than a single linear leg. There are two different types of hierarchical Lowest Common Ancestor (LCA) processing that must take place to process SQL multipath nonlinear hierarchical queries. One type occurs for the WHERE clause and another one for SELECT list. LCA processing coordinates the processing between multiple paths referenced in the query to keep the processing of the paths under a common ancestor coordinated so the results remain meaningful. This use of multiple paths and its utilization of additional semantics between the related paths also dynamically increase the value of the data and significantly increase the number of possible queries. Increasing the number of queries possible also further increases the value of the data because of the data's use with the additional queries possible.

5.2.4.1) LCA WHERE Clause Semantics Example

This is an example of how the WHERE clause uses LCA logic in processing multipath queries. It uses the EmpCust view defined previously in Figure 16. The LCA logic coordinates the complex WHERE decision logic occurring across the Invoice and Addr nodes. The LCA for this coordination is the Cust node as shown below in Figure 20. All combination of values tested is performed under the LCA, it is the control point. In addition, the processing is performed separately under each occurrence to keep it within its same ancestor family. The ANSI SQL relational processor's standard Cartesian processor is controlling this process naturally as part of its normal operation. Normally, hierarchical processors would have to be performing this operation performing a good deal of hierarchical tree walking.

This operation also provides the basic solution for the XML Keyword Search problem by keeping the search result meaningful. These are still being researched in academic projects.

```
SELECT EmpID, CustID
FROM EmpCust
WHERE InvID='Inv01' AND AddrID='Addr01'
```

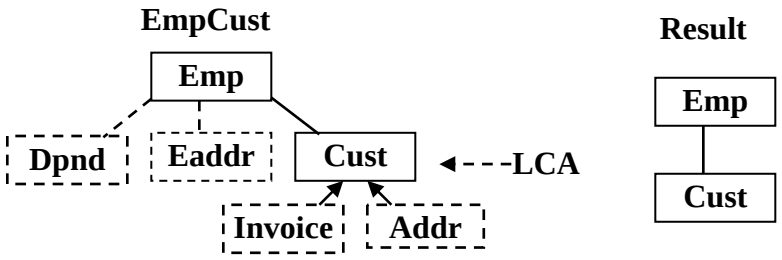


Figure 20: WHERE clause LCA semantics

```
<root>
 <emp empid="Emp01">
  <cust custid="Cust01"/>
 </emp>
</root>
```

5.2.4.2) LCA SELECT List Semantics Example

This is an example of how the SELECT list operates hierarchically using LCA logic in processing a multipath query. This example uses the EmpCust view defined in Figure 16. By qualifying on the AddrID value of 'Addr01' value with the selected InvID data in a different path, causes all of the qualified Invoice nodes under the selected LCA Cust node occurrence to be included as shown below in Figure 21. This isolates the most meaningful range of Invoice node return values. In addition, its data value is dynamically increased because of the additional semantics associated with locating it in this specific query instance.

On closer inspection, you will also notice that there is a second data field, DpndID, selected for output also based on the same AddrID value. This is on a different pathway from the qualification based on AddrID. This produces another LCA, Emp, to control the range of qualified values for DpndID. This produces a fine grained accurate result for each specific selected data item that would be difficult to process procedurally. This is a very powerful easy to use capability of SQL's SELECT operation.

SELECT InvID, DpndID
FROM EmpCust
WHERE AddrID='Addr01'

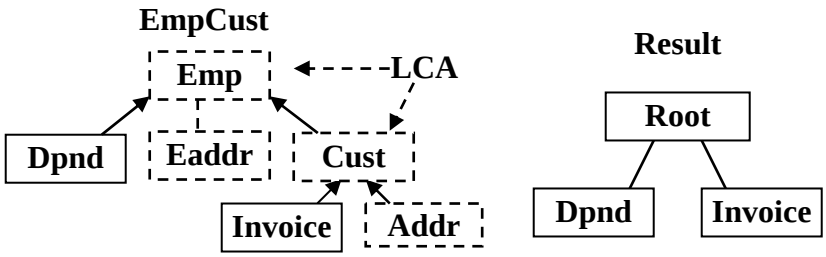


Figure 21: SELECT clause LCA semantics

```
<root>
 <dpnd dpndid="Dpnd01"/>
 <invoice invid="Inv01"/>
 <invoice invid="Inv02"/>
 </root>
```

5.2.4.3) LCA SELECT and WHERE Combination Semantics Example

This is an example of how the SELECT list and WHERE clauses both use their LCA logic together in processing multipath queries. It combines the LCA processing from the previous two LCA examples as shown below in Figure 22. The WHERE LCA is nested under the SELECT LCA which occurs automatically. Even this example is a fairly simple LCA example. A query can have many SELECT and WHERE LCAs that require complex nested LCA processing. This example demonstrates the completeness of ANSI SQL's inherent LCA processing.

SELECT DpndID
FROM EmpCust
WHERE InvID='Inv01' AND AddrID='Addr01'

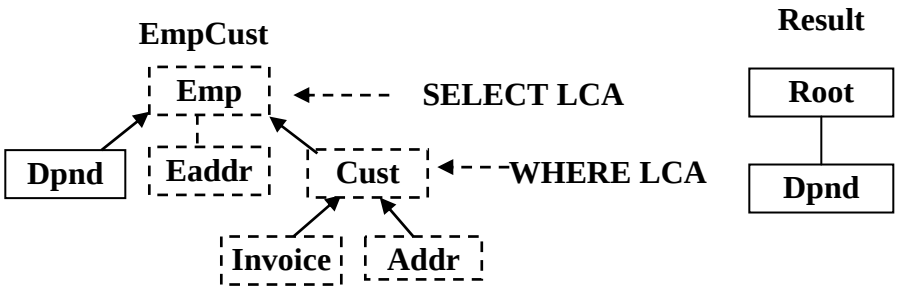


Figure 22: WHERE and SELECT clause LCA semantics combined

```
<root>
 <dpnd dpndid="Dpnd01"/>
</root>
```

5.2.4.4) WHERE Clause OR Condition Requires a Closer Look

As shown above, LCA processing can get quite complex and the LCA examples shown are generated from simple multipath queries. Another interesting LCA situation is a WHERE LCA that occurs when a complex OR condition is used with the WHERE operation which is demonstrated in Figure 22-2:

```
SELECT InvID, AddrId
FROM EmpCust
WHERE InvID='Inv01' OR AddrID='Addr01'
```

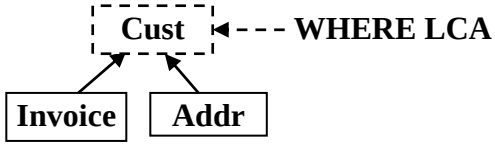


Figure 22-2: WHERE clause OR condition operation

This brings up the question: if *InvID='Inv01'* is true, does the second predicate test, *AddrID='Addr01'*, need to be executed since this is an OR operation and the first side of the predicate tests has already tested true? The logical answer for this is no, but not in this case.

The answer to the above question is that both sides of the predicate OR condition need to be tested in hierarchical processing. When the Invoice test is true, all Addr values under the same Cust LCA node are qualified and when the Addr test is true all Invoice values under the Cust LCA are qualified for output. In the XML result below you will notice that the first OR condition tested true matching *'Inv01'* selecting "Inv01" and the single address value "Addr01", but the second OR condition test is still performed testing *'Addr01'* and qualifying both Invoice values to be output. The proof of this is that Invoice value of "Inv02" was output (shown below) and it was not qualified on the first OR condition. This proves that the second OR condition needs to be tested in order to consistently get the correct hierarchical result.

```
<root>
<invoice invid="Inv01"/>
<invoice invid="Inv02"/>
<addr addrid="Addr01"/>
</root>
```

Interestingly, the SQL standard processing carried out a row at a time with its Cartesian product data replications in place will do this hierarchical OR processing naturally and correctly. If you still do not think this will produce the correct result naturally, imagine duplicating this query with each WHERE clause side tested separately and the result UNIONed together. Ignoring the replicated data of this query, the result will be the same proving that both sides of this predicate's OR (disjunction) is correct.

5.2.5) Variable Structure Generation

Variable structure generation is the dynamic control of whether certain pieces of the structure generate or not. In processing a hierarchical multipath query variable structure generation is still based on fixed structures where the existence of a portion of the structure can exist or not. This is implemented by a data value further up or further down the current path which controls whether a node or a view is to generate.

5.2.5.1) Variable Structure Generation Using Views

This is an example of a variable structure. It uses a variable view generation that is controlled by a data value up the current path by the addition of the test: *AND EmpStatus='F'* appended to the ON clause as shown in Figure 23 below. Its EmpStatus value controls whether the CustView is generated or not as also shown in the XML output below. Once when it is *EmpStatus='F'* and the other time it is left out when EmpStatus is empty. This is a simple example, but proves its use.

SELECT EmpID, DpndID, CustID, InvID, AddrID, EmpStatus
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID AND EmpStatus='F'

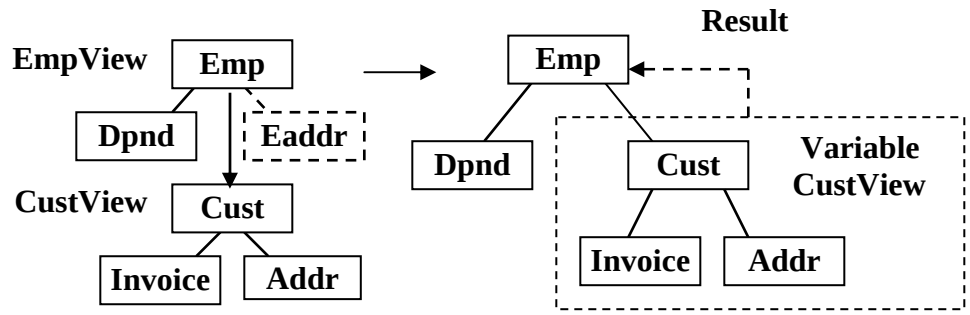


Figure 23: Variable structure example looking back

```
<root>
 <emp empid="Emp01" empstatus="F">
  <dpnd dpndid="Dpnd01"/>
  <cust custid="Cust01">
   <invoice invid="Inv01"/>
   <invoice invid="Inv02"/>
   <addr addrid="Addr01"/>
  </cust>
 </emp>
 <emp empid="Emp02" empstatus="">
 </emp>
</root>
```

5.2.5.2) Variable Structure Generation Using Look Ahead

Views also offer another powerful capability for variable structure generation. As was shown and explained in the previous Section 5.2.3.2, views can be referenced below their root node. This means that a view can be excluded or included in the structure being constructed based on a value anywhere in the view being tested for inclusion. This means the control value being tested does not have to be in sequential range. If not matched, an entire view structure can be excluded. The reason that this “look ahead” capability is possible is because the lower CustView view is

naturally expanded and materialize before it is joined making all of its contents available before it is referenced. From the SQL in Figure 23-2 you will notice that the ON clause that joins the EmpView and CustView follows CustView. This means that both of these views have to be expanded and processed before they are joined. This allows any value in a document or structure to control whether it qualifies to be used in the query. This is an extremely powerful, useful and internally complicated operation specified very easily. This operation also naturally supports the processing of lower level linking for fixed hierarchical structures keeping logical and physical structures operating the same and consistently.

SELECT EmpID, DpndID, CustID, InvID, AddrID, InvStatus
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID AND InvStatus='P'

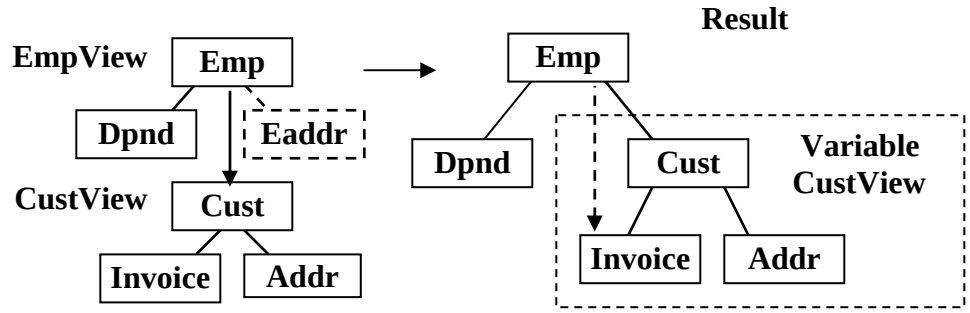


Figure 23-2: Variable structure example looking ahead

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <cust custid="Cust01">
   <invoice invid="Inv01" invstatus="P"/>
   <addr addrid="Addr01"/>
  </cust>
 </emp>
</root>
```

5.2.5.3) More Complex Variable Structures are possible

More complex uses of variable structures can use multiple variable sub structure views that can even be nested or on separate pathways and can be invoked many times in a given document occurrence. Another powerful use of variable structure generation is to use a dynamic choice of sub structure replacement. This is performed by having multiple Left Joins where only one in a group qualifies by a data value test made up the structure. An example would be:

SELECT * FROM EmpView
LEFT JOIN DpndViewF ON EmpID=DpndEmpID AND EmpStatus='F'
LEFT JOIN DpndViewP ON EmpID=DpndEmpID AND Empstatus='P'

It is not possible for both of these Left Joins to test true. If the first join is false, the second will still be tested. These different techniques can produce very dynamic structures that match the data needed to be processed and displayed.

5.2.6) Structure Transformation Using Restructuring and Reshaping

In Figure 16 we saw the join of two hierarchical structures with only a portion (fragment) of each structure joined. This can be considered a form of structure transformation. A real structure transform is taking a single structure and transforming its structure by pulling it apart and reassembling it differently. These structure transformations can be performed in two different ways, restructuring or reshaping. Restructuring uses existing relationships in the structure to rejoin the structure differently. Reshaping does not rely on existing relationship data; it utilizes the structure semantics in the structure to restructure the structure in any way. Each has their own outcomes and purposes. Restructuring is usually used to match an application, while reshaping is used to match a desired structure.

Restructuring and reshaping operate by making logical duplicate copies of the structures using SQL and then isolates the different fragments in each copy so they can be moved, copied and rejoined independently. Interestingly, the relational rowset that was previously shown to handle multipath variable length structures is also very flexible at handling the movement of hierarchical fragments in the rowset.

5.2.6.1) Restructuring Structures

This example transforms the EmpCust structure in Figure 24, by moving Invoice over the Emp node. It uses SQL's alias capability to create two structure fragments out of EmpCust (X and Y prefixes). It then rejoins them with Invoice on top using existing relationships in the data. The resulting structure is shown in Figure 24. It also shows that the Invoice fragment data properly replicated and moved as a single contiguous group.

Notice that EmpCust Y is hierarchically joined over EmpCust X. This places Y.InvID over X.Emp, based on the relationship established by ON Y.InvCustID=X.EmpCustID.

SELECT X.EmpID, X.DpndID, Y.InvID, X.AddrID
FROM EmpCust Y LEFT JOIN EmpCust X ON Y.InvCustID=X.EmpCustID

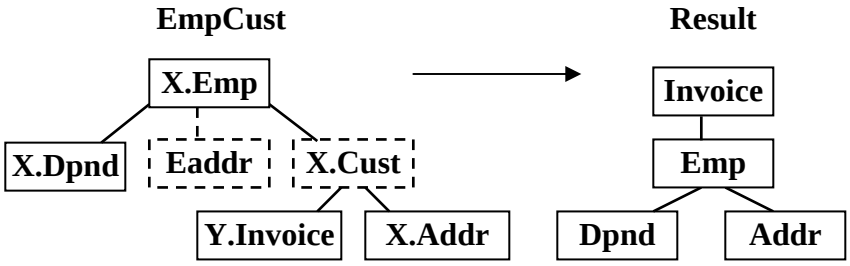


Figure 24: Restructured EmpCust structure

```
<root>
 <invoice invid="Inv01"/>
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
   <addr addrid="Addr01"/>
  </emp>
 </Invoice>
 <invoice invid="Inv02"/>
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
```

```
   <addr addrid="Addr01"/>
  </emp>
 </root>
```

5.2.6.2) Simple Reshaping Nonlinear Structure to Linear Structure

This first reshaping example reshapes the EmpView structure into its most closely resembling linear structure. It basically bends the Dpnd leg up and over the Emp root making Dpnd the new root and transforming the nonlinear EmpView structure into a linear structure preserving the data to fit the new semantics.

Reshaping does not rely on relationships in the data, this ability can often be necessary since XML does not need to rely on foreign keys because of its contiguous storage natural format. Without relationships in the data we use the reshaping technique of creating our own instant relationships by comparing the same data field to itself in the copies of the structures to synchronize them. The key fields chosen to synchronize the structure copies are taken from their nodes and used in the order desired to build the required new structure top-to-bottom. This can be seen below in Figure 25 where DpndID is used to synchronize the two copies. This allows X.DpndID to be retrieved from the top structure and Y.EmpID from the lower level. Instead of having to use a third lower copy of the data to access Eaddr, it turns out that it is already in position under Emp and can also be accessed from the second copy using Y.EaddrID. The nodes in the solid boxes are moved over to the output structures being built.

```
SELECT X.DpndID, Y.EmpID, Y.EaddrID
FROM EmpView X LEFT JOIN EmpView Y
        ON X.DpndID=Y.DpndID
```

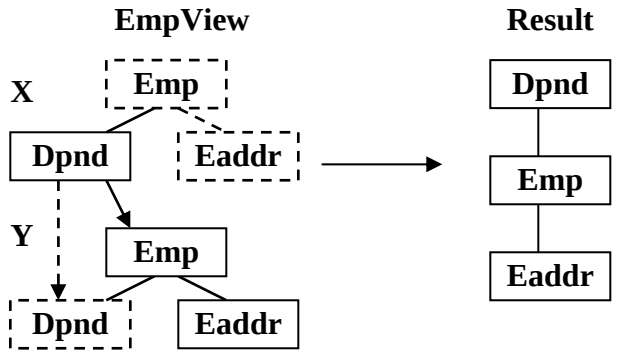


Figure 25: Simple reshaping nonlinear to linear structure

```
<root>
 <dpnd dpndid="Dpnd01">
  <emp empid="Emp01">
   <eaddr eaddrid="Addr01"/>
  </emp>
 </dpnd>
</root>
```

5.2.6.3) Complex Reshaping Nonlinear Structure to Linear Structure

This example is similar to the previous reshaping example in Figure 25 except the structure transformation is more complex. In this transform, EmpView still bends the Dpnd node back up making it the new root, but this time the Emp and Eaddr nodes are reversed producing the structure shown below in Figure 26 under the title Result. This is more complex and interesting because the new structure Dpnd and Eaddr are directly related where they were not previously. So how can they be directly related without any prior relationships in place? This is possible because every node in a nonlinear hierarchical structure is related to ever other node.

In the example below, you will notice in structure copy Y, we use the Dpnd node reference to indirectly reference and retrieve Eaddr going through the Emp node which we do not need until the final step since Emp is needed at the bottom. Since the Eaddr node is the only node selected from step Y it is located directly under Dpnd in the result as shown. The final step Z needs to retrieve the Emp node which is to be located under Eaddr. This is why step Z uses Eaddr as the synchronizing relationship.

```
SELECT X.DpndID, Y.EaddrID, Z.EmpID
FROM EmpView X
LEFT JOIN EmpView Y ON X.DpndID=Y.DpndID
LEFT JOIN EmpView Z ON Y.EaddrID=Z.EaddrID
```

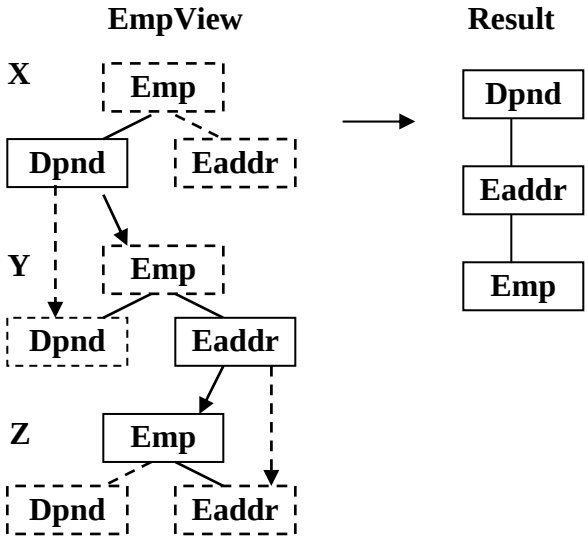


Figure 26: Complex reshaping nonlinear to linear structure

```
<root>
 <dpnd dpndid="Dpnd01">
  <eaddr eaddrid="Addr01">
   <emp empid="Emp01"/>
  </eaddr>
 </dpnd>
</root>
```

5.2.6.4) Polymorphic Reshaping

Polymorphic reshaping does not rely on the structure of the input structure. The advanced reshaping capability shown previously does support polymorphic reshaping when only one node is moved per join. The example in Figure 25 is not polymorphic. This is because it takes advantage of the structure by moving two related nodes at one time. On the other hand, the same basic reshaping performed in Figure 26 is polymorphic because it does not rely on the structure of the source structure by locating and moving one node at a time without user navigation. The choice of using reduced steps when possible or a polymorphic single node at a time solution is up to the user.

Since polymorphic reshaping does not depend on the source structure's hierarchical structure to operate by operating navigationless (schema-free), this allows either version of ViewX shown below in Figure 28 to be used as source input in a polymorphic reshaping. It produces the same transformed target structure in either case. This is demonstrated below in Figure 28.

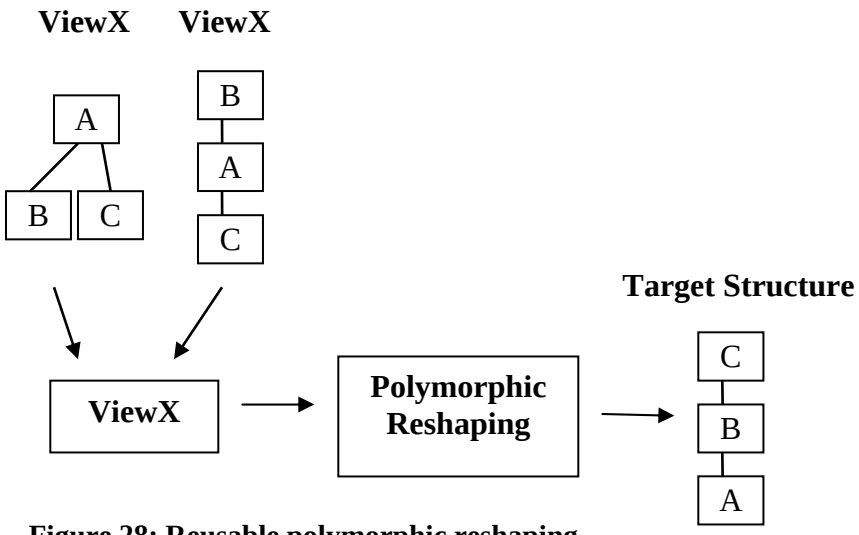


Figure 28: Reusable polymorphic reshaping

5.2.6.5) Combining Restructuring and Reshaping Operations

Restructuring and reshaping can be combined in a single structure transformation. The modeling of the target structure remains the same for both of the transformation methods as does the coordination; the two types of ON clause linking using relationships or just matching on the same value can be used interchangeably. When there is a choice, you need to decide which method correctly models the target structure you desire. This is important because their semantics are different as described earlier. This level of nonlinear hierarchical structure transformation is not available elsewhere today and it is also being automatically performed semantically correct.

Current specialized advanced nonprocedural and polymorphic reshaping solutions have been proposed in (Pankpwski, 2004; Dyreson, 2009). Being specialized they seem easier to use than SQL for this purpose, but they are untested proposals and are not standard.

5.2.7) Global Queries

As described previously in Section 3.2.3.5, the Prototype's hierarchical access is optimized so that each view executed is optimized at execution so that unneeded pathways are not accessed. This means there is never overhead for using views that contain larger views than required allowing for Global Queries. This means that global views can be utilized for ease of use and reuse, and is further enhanced since the user does not need to know the structure or perform navigation. The query examples so far have demonstrated this Global View capability by dynamically specifying queries that identified only a portion of the structure defined by the global view. This feature uses the variable SELECT list to dynamically represent any portion of the structure without incurring additional overhead for this significant increase in flexibility. This also opens up the capability of Global Queries that can perform queries involving the entire global view and output any or all of its data.

5.2.7.1) Global Queries Using Global Views

Global Queries are another area where today's procedural navigational XML processors can not handle. But with SQL and the prototype, all it takes is the use of a SELECT * to access all data in the global view and output it. Operations like hierarchical data filtering with a WHERE clause can easily filter the entire query range and output the results as shown in Figure 30. But, first we will use no query processing and just print out the entire Global Query so we have something to compare against in Figure 29.

SELECT * FROM EmpCust

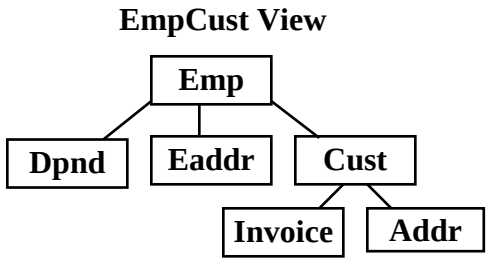


Figure 29: Global Query with no modifications applied

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01"
      empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
  <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  <cust custid="Cust01" custstoreid="Store01">
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P"/>
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O"/>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03"
      empstatus="">
  <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
  <cust custid="Cust03" custstoreid="Store01">
   <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV"/>
  </cust>
 </emp> </root>
```

5.2.7.2) Global Query Filtering

In Figure 30 below, the Global Query (SELECT *) filters the entire global structure based on *InvStatus='O'*. You can notice the global hierarchical filtering taking place by comparing this result to the previous one in Figure 29 which had no filtering or modifications taking place. All of the data below is related only to invoices of status "O". All of the available and related fields are output. All of the data not hierarchically filtered out is output preserving most of the hierarchical structure. The WHERE clause filtering has no limitation and can reference multiple fields from anywhere in the structure. This global type of hierarchical operation is not feasible using procedural navigation used today. The Global Filtering of the data below in Figure 30 is performed by the hierarchical filtering flow as indicated by the arrows in the hierarchical structure.

SELECT * FROM EmpCust WHERE InvStatus='O'

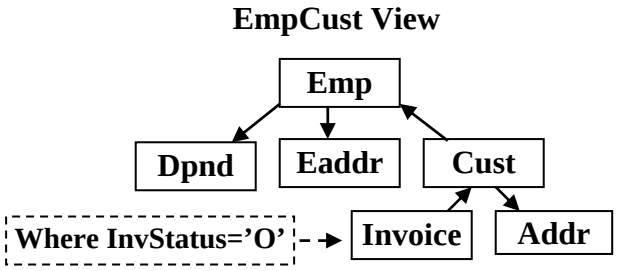


Figure 30: Global Query with data filtering applied

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01"
      empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
  <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  <cust custid="Cust01" custstoreid="Store01">
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O"/>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
 </emp>
</root>
```

5.2.8) Miscellaneous Capabilities

5.2.8.1) Hierarchical Multipath ORDER BY Operation

The concept of hierarchical multipath nonlinear ordering where each hierarchical pathway can be ordered separately in isolation was described in Section 3.2.3.7. This example in Figure 31 demonstrates hierarchical ordering for nonlinear hierarchical structures. The sort fields can be entered in any order as demonstrated below in Figure 31; they will be reordered to fit the hierarchical structure. Multiple fields in nodes will be left in the logical order specified for each node. In the produced hierarchical result, you will notice that the separate pathways have each been ordered separately in a descending fashion, which is different from its input default order. Standard SQL's linear ordering can not do this, so the Prototype has helped it along by modifying the ORDER BY operation's semantics to fit nonlinear hierarchical structures. This solution has kept SQL's look-and-feel the same and has satisfied the need for nonlinear ordering when producing hierarchical structures. There are no limitations for the hierarchical ORDER BY's nonlinear hierarchical operation, multiple fields per node can be specified and their logical sequence per node controls their ordering significance for each node.

SELECT CustID, InvID, AddrID
FROM CustView
ORDER BY AddrID DESC, InvID DESC, CustID DESC

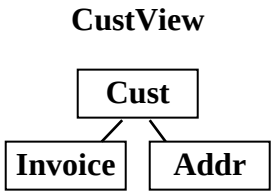


Figure 31: Nonlinear Order BY

```
<root>
 <cust custid="Cust03">
  <addr addrid="Addr03"/>
 </cust>
 <cust custid="Cust02">
  <invoice invid="Inv03"/>
  <addr addrid="Addr04"/>
  <addr addrid="Addr02"/>
 </cust>
 <cust custid="Cust01">
  <invoice invid="Inv02"/>
  <invoice invid="Inv01"/>
  <addr addrid="Addr01"/>
 </cust>
</root>
```

5.2.8.2) Renaming, Replicating and Redistributing Nodes and their Fields

This query in Figure 32 renames the nodes and the fields using SQL aliases. The aliases have been specified using the optional AS keyword to emphasize their use. You can see the names have been renamed in the resulting XML.

In addition, the aliasing capability has allowed the Eaddr node to be replicated and the data to be broken up between the two redistributing the data. The Eaddr table was broken apart into two tables named Addr1 and Addr2. Addr1 contained the field AddrName, and Addr2 contained the field AddrState. To demonstrate the flexibility of aliasing with the prototype data modeling, Addr2 has been linked underneath Addr1. This can be seen in the XML output below.

When the Eaddr table is duplicated and renamed, its column names are no longer unique. This is why their column names are prefixed by their new table names to distinguish them. This is all standard SQL. This query can be placed in a view that will reflect the new structure and names. The specifying of which column values are placed in their associated nodes occurs on the SELECT list. In addition, filtering can be applied to the redistribution process to divide the data.

```
SELECT EmpID EmpName, DpndID AS DpndName,
        Addr1.EaddrID AS AddrName,  Addr2.Eaddrstate AS AddrState
FROM Emp AS Employee
LEFT JOIN Dpnd AS Dependent ON EmpID=DpndEmpID and DpndCode='D'
LEFT JOIN EAddr AS Addr1 ON EmpCustID=Addr1.EAddrCustID
LEFT JOIN EAddr AS Addr2 ON Addr1.EaddrCustID=Addr2.EAddrCustID
```

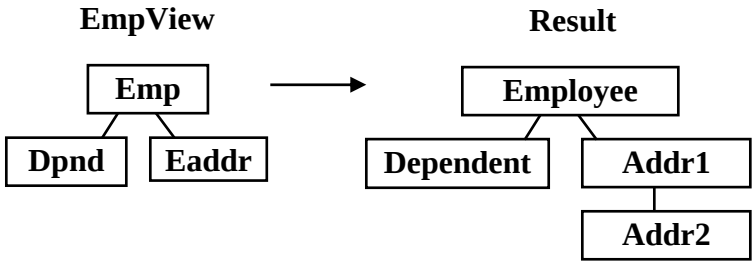


Figure 32: Renaming nodes and their fields

```
<root>
 <employee empname="Emp01">
  <dependent dpndname="Dpnd01"/>
  <addr1 addrname="Addr01">
   <addr2 addrstate="CA"/>
  </addr1>
 </employee>
 <employee empname="Emp02">
  <addr1 addrname="Addr03">
   <addr2 addrstate="NV"/>
  </addr1>
 </employee>
</root>
```

6) PROTOTYPE COMPARISON TO TODAY'S XML PROCESSORS

There are four categories of prototype-to-query comparisons to be examined. These are new ANSI SQL Prototype comparisons to: the ANSI/SQL standard; the current XQuery standard: and the two basic areas of academic research. These research areas are XQuery research and general query research not necessarily based on XQuery. These topics are covered in this same order listed here because they build upon each other and reduce having to cover the same material multiple times. New and missing capabilities will be covered as well as memory and efficiency issues. These are not meant to be a description of these XML processors, only a brief comparison to the ANSI SQL Prototype which can also familiarize the reader with these and other query XML processors. Structured XML processing is only covered here since SQL is a structured data processor. Semistructured processing is not considered

6.1) Prototype Vs SQL/XML Standard

The SQL/XML Standard (SQL/XML Standard, 2005) uses a series of XML centric functions placed in SQL's SELECT list to output relational data as structured XML. The structure is specified by using multiple nesting of SQL SELECT lists to conform to the data structure output. The requirements for control of the output XML are knowledge of XML and the output data structure, and how to code the SQL to model the structure. This will require a programming professional with XML experience. The development of the output query is static and fixed. Changes to SQL/XML output can not be automated. The adding to or changing of the structure being modeled is not a simple process and the use of these SQL/XML functions eliminates the flexibility of SQL's dynamic use of the SELECT list and its ease of use when invoking a query.

The ANSI SQL Prototype was developed to support structured XML data processing transparently so that no XML centric syntax or functions are necessary. In addition, it was designed to take maximum advantage of ANSI SQL's inherent full nonlinear hierarchical processing as described in this document. The output XML structured format processing is performed automatically based on its hierarchical structure specified in the input SQL query hierarchical views and structure combining specified dynamically during query invocation. This can be further controlled naturally by the user adding to or removing data references from the SELECT list. This automatic processing can support any multipath queries which increases the value of the data and can produces the correctly formatted XML result for dynamic operations such as On Demand Publishing.

The SQL/XML standard does not support or enforce hierarchical processing but does allow its optimization to reduce replicated data generated between the different hierarchical pathways when joined. A popular solution is to use the outer union to store the different pathways by performing their joins separately for each pathway. This works nicely for inputting XML and outputting the data formatted as XML, but does prevent the operation of the WHERE clause testing and filtering across the pathways. This means multipath queries are not supported, but the storage optimization can be considerable.

The Prototype does support full nonlinear hierarchical processing and requires the standard relational hierarchical processing to perform the full Cartesian product and its introduction of data replications to be able to operate hierarchically correct. It also supports a powerful hierarchical dynamic optimization that does eliminate join operations that are not necessary to the query. This also eliminates unnecessary data and the replicated data it causes and does not prevent the inherent multipath processing. This optimization may also produce better CPU performance operation. Comparing memory and CPU performance between the prototype and the SQL/XML standard is not an equal comparison. The Prototype has many advantages and advancements over the SQL/XML standard mentioned above and in this paper that makes this efficiency comparison not straight forward.

6.2) Prototype VS Current XQuery Standard

The XQuery standard (W3C XQuery, 2007) and its processing does resemble the SQL/XML standard's processing. XQuery uses its FLWOR expression and FOR loops that control the structure processing much like the SQL/XML standard's use of SQL in nested SELECT list. It was always the desire to keep these two products similar when they were being designed. But XQuery being designed from the ground up to support XML and relational processing does have more flexibility and features. Under pressure to support a migration path from SQL to XQuery, XQuery was required to support relational processing and hierarchical processing weakening both. And even more serious poor decision was making the non hierarchical Inner Join in XQuery its primary join. This maybe one of the reasons XQuery does not enforce full and correct hierarchical processing, so its results may not always be hierarchically correct as is also the case with the SQL/XML standard. These decisions also prevent XQuery from being Structure-aware so that it could make automatic decisions and processing of hierarchical processing.

While XQuery designers claim that SQL's SELECT, FROM and WHERE operations are represented clearly in XQuery, this is not the case. The WHERE clause is supported and the FROM clause functionality is closely supported, but the power of SQL's dynamic SELECT clause is not supported. Similarly to the SQL/XML standard, the location of the output data in XQuery is scattered throughout the Flower expression looping structure where their placement is critical to the processing. Adding new output XML requires adding new data variables correctly in the looping structure of the FLWOR expression. This could also require the addition of new looping constructs in the control structure. This requires additional programming and testing by an XQuery professional. SQL's dynamic data SELECT list operation dynamically controls the structured data output and can support dynamic output formatting which can be used for decision support and Dynamic Publishing for structured data reports. This is a new level of structured data reporting.

The Prototype's SQL SELECT list is simply a list of what data items are to be output. To add a new data item, it is simply added to the list. There is no placement or looping structures to be concerned with. This data specification can be done dynamically at query execution by modifying the SELECT list. With the Prototype's automatic XML output, the dynamic SELECT list dynamically controls what data and what portions of the structure are processed and output. This SQL dynamic flexibility and control f the processing is missing in XQuery. An XQuery query can be coded to operate with this type of dynamic control, but it would require programming it with the code that specifically supports the required flexibility which must be known ahead of time.

Two issues prevent XQuery from supporting full nonlinear multipath processing basically limiting its processing to linear single path processing. The first reason is that XQuery is fully driven by user navigation. This severely limits the complexity of its processing possible because multipath processing is too complex to do by procedural user navigation. The second reason is that multipath query processing requires Lowest Common Ancestor (LCA) logic that requires extremely advanced processing that should be processed automatically using internal knowledge of the structure and XQuery is not structure-aware.

The Prototype's XML operation is transparent and navigationless allowing non technical users to specify powerful multipath queries. Its ability to automatically optimize full multipath hierarchical structures at a high conceptual level allows it to efficiently process large documents and make powerful multipath queries practical while its ability to dynamically increase data value more than compensates for any additional internal processing it may have to perform. But this has always been expected and has always been a tradeoff enabled by ever increasing more powerful processor chips. Even today with the need to go to multicore processors, hierarchical processors offer a further automatic solution to automatic parallel processing because they are naturally parallelizable by following the independent pathways of their hierarchical structures. With XQuery limited to linear processing efficiency comparisons with the Prototype's nonlinear processing is not possible.

6.3) Prototype VS XQuery Research

XQuery's burden of a high level of required knowledge use which consists of its complex operational use, and required knowledge of XML and its hierarchical structure have not gone unnoticed at the academic research community. They have already launched projects to enable XQuery to perform schema-free access and processing. This enables XQuery to query XML data automatically and efficiently without requiring precise knowledge of the document structure. This action helps establish that there is a need for a more advanced automatic navigationless processing of XML and there is some movement to offer solutions that move the industry in this direction.

The academic community working on XQuery has decided that the solution to schema-free XQuery processing is through the external addition of Lowest Common Ancestor (LCA) procedural function support that is currently being utilized internally and automatically in the Prototype. This processing is not being utilized currently by standard XQuery. The XQuery LCA academic research solution has had some success introducing LCA processing onto XQuery through the introduction of an external LCA XQuery function usually placed on the WHERE operation. This approach is used in (Li & Yu, 2008) which also includes an additional helper function that assists the LCA function.

From our experience with the Prototype which transparently performs the LCA logic internally as needed, we have seen that there are more than one use for LCA's and these uses require different logic to be applied to each use. There is the WHERE clause use and the SELECT operation use covered previously each concerned with proper ancestor range control. So there should be more than one type of LCA function and its use. The SQL SELECT operation also allows easy control over which data is desired. This type of operation needs to be supported by a dynamic output formatting capability that is still missing in XQuery. Our Prototype supports this automatic output formatting.

More importantly, the Prototype demonstrated that the LCA process can be needed in many locations which is made more complicated by the number of paths accessed and the location of the data types. This requires nesting of many different LCAs. External solutions may not succeed. Nesting of LCA functions may not be possible; this will limit the complexity of the multipath query possible. In addition, the specifying of the LCA function is not transparent; it will still require its use by an XQuery professional. In fact, it will require a professional who is also knowledgeable of these more complicated LCA processing and programming issues. The use of the LCA function can also affect the way the XQuery query is coded requiring a different design strategy that may be much less efficient.

At the beginning of this paper in the Background section, it was mentioned that the Prototype's processing is limited to database (data centric structured data) data and ignores the processing of Markup (document centric semistructured data) data in order to obtain the most power and accuracy for SQL's processing of XML's structured data. But it should be pointed out that XQuery research is not limiting its access to database data and is also processing semistructured markup data. This means it has to process duplicate named node types common in markup which means LCA node types are no longer fixed and their location can dynamically change their position. This is necessary for markup searching where the results can be fuzzy by continually reducing the meaningfulness of search results until a match is found. This complicates the LCA determination logic for markup processing.

6.4) Prototype VS Generic Query Research

Generic query research into schema-free access is generally referred to as "XML Keyword Search". It does not necessarily use XQuery and is not limited by it. This enables that LCA processing to be implemented internally with no limitations or implementation problems based on XQuery. This means

that multiple types of LCAs can be used as needed. Using and nesting of LCA nodes can occur as needed. The processing of the formatted XML output can also be performed automatically to support meaningful aggregated results as shown in (Liu & Yi, 2007). This is also more like the SQL Prototype with its automatic output that uses LCA processing to automatically determine the meaningful data.

But the generic query research does mean that the language, syntax and semantics used is not standardized. The ANSI SQL prototype is performing the navigationless LCA nonlinear processing naturally and automatically utilizing the natural hierarchical processing in ANSI SQL operating hierarchically. It does not require hierarchical processing code to be supported which can be very difficult to get working for all cases. The fact that there are still generic query research projects trying to support schema free navigationless processing means that a perfect solution to this problem has not been found. So far ANSI SQL natural processing performed today works perfectly under all processing conditions and examples. There would be no advantage of this generic query research over the Prototype for structured data hierarchical processing.

7) PROTOTYPE FEATURES AND OPERATIONS IN DEVELOPMENT

7.1) XML Update in SQL

The Prototype is foremost an SQL product operating hierarchically, but is still strongly SQL centric. SQL's internal data is still relational. For this reason, XML update support in SQL is interpreted and implemented seamlessly in SQL as enabling relational data updating from XML data. Actually this means any data that is defined hierarchically. The same ANSI SQL Update statement is utilized and the internal data source is already hierarchically data, the only operation required is to isolate the hierarchical input path into a single occurrence using a WHERE clause to isolate the path being updated in standard SQL.

7.2) Hierarchical Legacy Access in SQL

Legacy data such IMS and structured VSAM present no more of a problem to support hierarchical processing than XML does. This means they will be accessed and processed fully hierarchically. VSAM is similar to XML in its makeup since it is also a contiguous data structure. IMS like XML requires navigation, but not being contiguous structure, multiple legs can be navigated concurrently significantly optimizing its access.

7.3) Flat Data Legacy Access in SQL

Legacy data such flat fixed files and spreadsheets sheets present no more of a problem to support hierarchical processing than the flat relational data does. This means they will be accessed and processed fully hierarchically. Most flat data is easily modeled hierarchically.

8) FUTURE RESEARCH DIRECTIONS

8.1) Nonlinear Aggregates

The current level of support in the Prototype more than serves its purpose to prove that ANSI SQL supports full multipath hierarchical processing complete with automatic LCA support and advanced hierarchical processing capabilities that have been shown in this chapter. The adapting of the ORDER BY operation to take advantage of the hierarchical structures and processing by separately ordering each

different pathway as would be naturally expected for hierarchical structures serves to demonstrate that this advanced capability is possible. In this regard, data aggregations should also take advantage of the hierarchical structure in the same way as ordering so that the separate parallel pathways could be separately aggregated in isolation. Another possible advanced capability is the addition of hierarchical syntax additions to override the default hierarchical processing range control such as the "HAS" syntax. An example would be "…*WHERE Department HAS ANY Dependent…*".

8.2) Automatic Parallel Processing

Another particularly interesting research project is to transparently replace the relational engine of the SQL processor with a true hierarchical engine. This is possible when ANSI SQL processing is restricted to hierarchical processing. This would serve to greatly improve the operating and memory usage efficiency of hierarchical processing. With this in place, or during its implementation, a further research project would be to make the full query processor utilize parallel processing. The significant advantages here is that the hierarchical structures being hierarchically processed act as roadmap for the required parallel processing. Since the Prototype is structure-aware, each query can be parallel processed automatically without requiring any human guidance. With the hierarchical engine and its hierarchical structure processing and storage in place, a significantly larger portion of the query application can be parallelized than is normal for parallel processing. This has been explained in (David, 2009-2).

8.3) Automatic Distributed Hierarchical Processing

ANSI SQL hierarchical processing is unique because the SQL hierarchical data modeling input statements control the hierarchical processing. This means that the query continues to operate hierarchically when distributed so the hierarchical result remains accurate and this technique can be utilized in parallel processing. Parallel processing can also utilize distributed hierarchical processing as an additional way to perform concurrent processing using parallel processing by distributing the query. Distributing SQL queries by breaking them apart to distribute their processing and then resembling the results is a known technology.

9) CONCLUSION

This paper has explained, shown, and proven how ANSI SQL with its inherent full hierarchical processing capability can be used to transparently support native XML and relational data integration. In addition, it was shown and demonstrated how the full hierarchical processing greatly enhanced the natural ANSI SQL processing by seamlessly elevating it to a full hierarchical level allowing for relational/XML integration and advanced hierarchical processing operations to be automatically carried out.

This paper has also shown how the Prototype's SQL/XML solution described operates so easily, powerfully and naturally on hierarchical structures using standard SQL. The result of processing the example SQL statements in this document have proven and shown how ANSI SQL can perform full hierarchical processing automatically and transparently by hierarchically modeling the data structures. It was shown how hierarchical structures could be modeled in SQL views and how the hierarchical modeling ANSI SQL semantics processes the data hierarchically. This involved and demonstrated how the relational rowset can contain hierarchical multipath structures. It was also shown and demonstrated how the relational engine processes complex hierarchical multipath queries using the additional naturally occurring multipath semantics available increasing data value.

In addition, it was shown how query data filtering and selection logic naturally followed global hierarchical processing principles for meaningful hierarchical results. And finally, it was also shown that

SQL can naturally and automatically perform very advanced hierarchical capabilities being introduced by XML, such as: node promotion, fragment processing and structure transformations. These capabilities allow for the following capabilities.

9.1) SQL Nonlinear Hierarchical Processing Capabilities:

1) ANSI SQL standard and mathematically sound requiring no nonstandard SQL or functions
2) Nonprocedural, navigationless ease of use by nontechnical users, no XML centric syntax
3) Hierarchically correct XML results using only principled nonlinear hierarchical processing
4) Greater efficiency (powerful inherent hierarchical processing with limiting optimization)
5) Fully interactive operation dynamically processes powerful multipath XML hierarchically
6) Conceptual hierarchical processing easily and semantic join full hierarchical structures
7) SQL queries operate across entire heterogeneous relational/XML hierarchical structure
8) Dynamically increases value of customers' data by utilizing semantics across paths
9) Enables full nonlinear hierarchical processing with advanced hierarchical data filtering
A) Can be used for dynamic decision support and On Demand dynamic publishing

What makes the above SQL capabilities particularly valuable is that they are performed simply using powerful nonprocedural ANSI standard SQL. With SQL naturally performing hierarchically, it should be clear that interfacing to XML can now be easily performed at a seamless transparent high hierarchical level fully accessible by non technical users.

REFERENCES

Abiteboul, Serge., & Bidoit, Nicole. (1984). *Non First Normal Form Relations to Represent Hierarchically Organized Data*. Proceeding of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. Waterloo, Ontario, Canada.

Bao, Zhifeng., Ling, Zok. Wang., & Lu, Jiaheng. (2008). *SemanticTwig: A Semantic Approach to Optimize XML Query Processing.* In J. R. Haritsa et al. (Eds), DASFAA, LNCS 4947, (pp.282-298) Springer-Verlag Berlin Heidelberg.

Chen, Ya. Bing., Ling, Tok. Wang., & Lee, Mong. Li. (2003). *Automatic Generation of XQuery View Definitions from ORA-SS Views*. In L.Y. Song et al. (Eds), LNCS 2813, (pp, 158-171) Springer-Verlag Berlin Heidelberg.

Chen, Zhuo., Ling, Tok. Wang., & Liu, Mengchi. (2004). *XTree for Declarative XML Quering*. In Y. Lee et al. (Eds), DASFAA, LNCS 2973, (pp, 100-112) Springer-Verlag Berlin Heidelberg.

Czumaj, Artur., Kowaluk, Miroslaw., & Lingas, Andrej. (2006). *Faster algorithms for finding lowest common ancestors in directed acyclic graphs.* Electronic Colloquium on Computational Complexity, Revision 2 of Report No. 111.

David, Michael. M. (1992). *Advanced Capabilities of the Outer Join.* ACM SIGMOD Record. March.

David, Michael. M. (2003). *ANSI SQL Hierarchical Processing Can Fully Integrate Native XML*. ACM SIGMOD Record. March.

David, Michael. M. (2009). *The Ghost in the Machine.* TDAN, The Data Administration Newsletter August 9, 2009. http://www.tdan.com/view-articles/11069

David, Michael. M. (2009-2). *Automatic Full Parallel Processing of Hierarchical SQL Queries*, DevX , Retrieved February 26, 2009, from http://www.devx.com/SpecialReports/Article/40939/1954.

Dyreson, Curtis., Bhowmick. Sourav. & Jannu, Aswani. Rao. (2009 January). *Morph: A (Shape) Polymorphic XML Query Language*. Plan-X Workshop 08, Savannah, GA.

Krishnamurthy, Rajasekar., Kaushik, Raghav., & Naughton, Jeffrey. F. (2004). *Unraveling the Duplicate-Elimination Problem in XML-to-SQL Translation*. Seventh International Workshop on the Web and Databases (WebDB).

Leven, Mark., & Loizou, George. (1993). *Semantics for Null Extended Nested Relations*. ACM Transactions on Database Systems (TODS).

Li, Yunyao., Yu, Cong., & Jagadish, H. V. (2004). *Schema-Free XQuery*. Proceedings of the 30th VLDB Conference, Toronto, Canada.

Li, Yunyao., Yu, Cong., & Jagadish, H. V. (2008). *Enabling Schema-Free XQuery with Meaningful Query Focus*. The International Journal on Very Large Data Bases, Volume 17, issue 3,May.

Liu, Ziyang., & Yi, Chen. (2007). *Identifying Meaningful Return Information for XML Keyword Search*. Proceedings of the 2007 ACM SIGMOD International Conference on Management of data, Beijinging, China.

Mani, Murali. Wang, Song., & Dougherty, Daniel. J. (2006). *Join Minimization in XML-to-SQL Translation: An Algebraic Approach*. ACM SIGMOD Record. March.

Pankowski, Tadcusz. (2004). *A high-level for specifying XML data transformation.* In A. Benczur. J., Demetrovics. G. Gottlab (Eds), ADBIS, LNCS 3255, (pp.159-172) Springer-Verlag Berlin Heidelberg.

Sengupta, Arijit., & Dalkilic, Mehmet. (2002). *DSQL – An SQL for Structured Documents*. In A. Banks Pidduck et al. (Eds), CAISE, LNCS 2348, (pp, 757-760) Springer-Verlag Berlin Heidelberg.

SQL/XML Standard. (2005). *SQL Standard-SQL/XML Functionality*. from: http://www.wiscorp.com/H2-2005-197-SC32N1293-WG3_Presentation_for_SC32_20050418.pdf

Ullman, J. D., & Aho A. V., Hopcroft J. E. (1973). *On Finding Lowest Common Ancestors in Trees*. Annual ACM Symposium on Theory of Computing.

Ullman, J. D. (1989). *Principles of Database and Knowledge- Base Systems. Volume II, Chapter 17, The Universal Relation*. Computer Science Press, page 1050.

W3C XQuery. (2007). *XQuery 1.0: An XML Query Language*, from: http://www.w3.org/TR/xquery/

Zhang, Shuohao., & Dyreson, Curtis. (2006). *Polymorphic XML Restructuring*. WWW workshop 06, May 23-26. Edinburg, Scotland.

ADDITIONAL READING

Aghili, Alireza. S., Li, Hua-Gang., & Agrawal, Divyakant. (2006). *TWIX: Twig Structure and Content Matching of Selective Queries using Binary Labeling*. ACM International Conference Proceeding Series, Vol 152, Proceedings of the 1st International Conference on Scalable Information Systems, Hong Kong.

Bao, Zhifeng., Wu, Huayu., & Chen, Bo. (2008). *Using semantics in XML query processing*. Proceedings of the 2nd international conference on Ubiquitous information management and communication, Suwon, Korea.

Cohen, Sara., Kanza, Yaron., & Kimelfield, Benny. (2005). *Interconnection Semantics for Keyword Search in XML*. Proceedings of the 14th ACM International Conference on Information and Knowledge Management, Bremen, Germany.

Cohen, Sara., Mamou, Jonathan., & Kanza, Yaron. (2003). *XSEarch: A Semantic Search Engine for XML*. Proceedings of the 29th VLDB Conference. Berlin, Germany.

David, Michael. M. (2008). *Navigationless Database XML: Hierarchical Data Processing*, DevX, Retrieved January, 2009 from: http://www.devx.com/xml/Article/39183/1954.

David, Michael. M. (2009). *Creating Hierarchical Structure Data Mashups*, DevX, Retrieved January, 2009, from: http://www.devx.com/xml/Article/40550/1954.

David, Michael. M. (2009). *Performing Hierarchical Restructuring Using ANSI SQL*, DevX , Retrieved April 15, 2009, from: http://www.devx.com/xml/Article/41464/1954.

David Michael M. (2009-3) *ANSI SQL Semantically Controlled Any-to-Any Data Structure Reshaping*, Semantic Universe web site, Feb 20, 2009, from: http://semanticuniverse.com/articles-semantically-controlled-any-any-data-structure-reshaping.html

Guo, Lin., Shao, Feng. & Botev, Chavdar. (2003). *XRANK: Ranked Keyword Search over XML Documents*. Proceedings of the 2003 ACM SIGMOD International Conference on Management of data, San Diego, CA.

Histidis, Vagelis., Koudas. Nick., & Papakonstantiniu, Yannis. (2004). *Keyword Proximity Search in XML Trees*. IEEE Transactions ON Knowledge AND Data Engineering, Vol. 18, NO. 4.

Li, Guoliang., Feng, Jianhua., & Wang, Jianyong. (2007, November). *Effective Keyword Search for Valuable LCAs over XML Documents*. Proceedings of the Sixteenth ACM Conference on Information Knowledge Management, Lisboa, Portugal.

Li, Quanzhong, & Moon, Bongki. (2001). *Indexing and Querying XML Data for Regular Path Expressions.* Proceedings of the 27th VLDB Conference, Roma, Italy.

Liu, Ziyang., Walker, Jeffrey., & Yi, Chen. (2007). XSeek: *A Semantic XML Search Engine Using Keywords*. Proceedings of the 29th VLDB Conference, Vienna, Austria.

Pal, Shankar., Cseri, Istvan., & Seeliger, Oliver. (2005). *XQuery Implementation in a Relational Database System*. Proceedings of the 31st VLDB Conference, Trondheim, Norway.

Shanmugasundaram, Jayavel., Kierman, Jerry., & Shekita, Eugene. (2001). *Querying XML Views of Relational Data*. Proceedings of the 27th VLDB Conference, Roma, Italy.

Shanmugasundaram, Jayavel., Krishnamurthy, Rajasekar., & Tatarinov, Igor. (2001). *A General Technique for Querying XML Documents using a Relational Database System*. SIGMOD Record, Vol. 30, No. 3. September.

Shanmugasundaram, Jayavel., Tufte, Kristin., & He, Gang. (1999). *Relational Databases for Querying XML Documents Limitations and Opportunities*. Proceedings of the 25th VLDB Conference, Edinburgh, Scotland.

Sun, Chong., Chan, Chee-Yong., & Goenka, Amit. K. (2007). *Multiway SLCA-based Keyword Search in XML Data*. International World Wide Web Conference, Proceedings of the 16th International Conference on World Wide Web, Banff, Alberta, Canada.

Vagena, Zografoula., Mora, Moro. M., & Tsotras, Vsassilis. J. (2004). *Twig Query Processing over Graph-Structured XML Data*. Seventh International Workshop on the Web and Databases, Paris France.

Xy,Yu., & Papakonstantinnou, Yannis. (2008,). *Efficient LCA based Keyword Search in XML Data*. Proceedings of the 11th ACM International Conference Series on Extending database Technology, Nantes, France.

Zhang, Shuohao., Dyreson, Curtis. (2006), *Symmetrically Exploiting XML*. Proceedings of the 15h International World Wide Conference on Extending database Technology, Edinburgh, Scotland.

KEY TERMS AND DEFINITIONS

Cartesian product: The Cartesian product is a relational process that is used in relational join operations to produce all combinations of join criteria matches producing replicated data in the rowset. Since relational processing uses flat rowsets and usually processes one row at a time the process of testing for a certain combination of different values requires generating all combinations of values which uses multiple rows where each different combination is represented in a single row.

Dynamic XML Publishing: XML Publishing has become popular starting from the beginning of XML. More recently dynamic XML publishing has become more important in order to publish more up-to-date dynamic information. This dynamic XML publishing has not previously been fully utilized for XML structured data dynamic report publishing as it is performed in the Prototype demonstrated in this paper which supports multipath processing.

Hierarchical Data Filtering: Hierarchical data filtering requires filtering data not just down the path from the filtering point but also going up the path too. The act of filtering data going up the path has the effect of filtering other related pathways by removing common ancestors of other pathways. So filtering one spot in a hierarchical structure can affect the entire structure. It becomes a complex operation.

Hierarchical Fragment: A fragment is a hierarchical sub portion of a hierarchical structure. In the prototype it is very flexible and can be joined easily to other portions of the structure.

LCA: Stands for Lowest Common Ancestor. It is also known as Least Common Ancestor and Nearest Common Ancestor. Finding the LCA is necessary for nonlinear hierarchical data processing. When

processing multiple paths of a hierarchical structure, the LCA must be determined to coordinate and limit the processing between the multiple paths to make the result meaningful.

LCA Query: This is an academic term that means the query can automatically process multiple hierarchical paths. Other similar terms are *twig queries* and *bushy queries*, but these could still be required to be performed nonprocedurally using navigation.

Navigationless: Is a term borrowed from SQL processing, which means that the query coder does not need to specify and database navigation to control processing of the query. It can also be applied to XML processing to mean the same thing.

Node type Vs Node Data occurrence: Node type applies to the definition of a specific node and its name. This is different than *node data occurrence* which applies to a specific node data occurrence which is an important distinction to the hierarchical processing of multiple node type databases because they usually require multiple data occurrences at the node level.

Nonlinear Hierarchical Processing: Nonlinear hierarchical processing is the multipath query processing of full multipath hierarchical structure data. This could still apply to the nonprocedural (navigationless) or procedural (user navigation) processing of these queries.

Nonlinear Structure Transformation: Nonlinear structure transformation involves the complex task of transforming one structure into another structure. If at least one of the structures is a multipath structure involving the access or construction of multiple pathways the transformation is a nonlinear structure transform. The types of nonlinear transforms are nonlinear structure to linear structure, linear structure to nonlinear structure, and nonlinear structure to nonlinear structure.

Semantic Structure Transformation: Semantic Structure Transformations implies that the transformation logic utilizes only the natural semantics in the structure to control the transform thereby requiring only nonprocedural guidance from the user. This natural semantic reshaping operation also automatically preserves the natural semantics in the result structure to reflect the new data structure. Data relationships are not used. The process is a reshaping where the original structure guides the movement of data as it is moved into the desired new location reflecting the semantics of the new structure shape.

Schema-free Query Processing: Schema-free query processing means that the processing of the query is navigationless; it does not require any navigation by the query coder or user. This also implies that the coder or user does not have to know the hierarchical structure of the data being accessed; this is the key to user friendly access. It also implies that LCA multipath processing is automatically supported. This opens the capability to support more powerful hierarchical processing capabilities not possible otherwise. Since schema-free operation does not require exact structure knowledge, it can also operate in a polymorphic manner handling different input structures.

Structure-Aware: Structure-aware processing implies that the operational query is aware of the query structure being constructed and processed and can actively utilize this meta information for the benefit of the query in any way. Dynamic operations such as joining hierarchical structures can dynamically change the data structure being processed. Dynamic structure-aware processing will detect if the structure dynamically changes.

Structured Data: The meaning of structured data and its processing when it includes XML structured data implies a well behaved hierarchical structured data and its hierarchically principled processing that follows rigid hierarchical processing principles.

View Expansion: Before SQL can parse an SQL statement, the views on the statement need to be replaced with their source SQL. This is referred to as a view expansion.

View Materialization: The act of accessing and processing a view's representative data to produce a rowset is called view materialization.