Michael M David and Lee Fesperman
Advanced Data Access Technologies, Inc.
www.adatinc.com

SQLfX®
A Web 2.0 ANSI SQL Transparent Native XML Nonlinear Hierarchical *LCA Query* Processor

SQLfX®
Transparently Turns Your
ANSI SQL Processor
Into a Powerful
Web 2.0 and Enterprise 2.0
Nonlinear Hierarchical
XML Processor

With Interactive Multi-Leg Query Capability & Automatic Structured Formatted XML Output

Nonlinear Hierarchical Structures are full hierarchical data structures with multiple legs making the data on different legs nonlinearly related, semantically meaningful and useful.

Nonlinear Hierarchical Queries have the ability to reference and semantically process any single or combination of legs in a multiple leg nonlinear hierarchical data structure by using powerful nonlinear principled hierarchical processing. This greatly increases: query correctness; ease of use; provability; and value of the hierarchical data. Queries can also hierarchically combine nonlinear data structures into a hierarchical superstructure before being processed.

Nonlinear Hierarchical Processing has the ability to process single and multi-leg hierarchical queries nonprocedurally and without structure navigation. This is achieved by automatically analyzing the full nonlinear hierarchical structure and its structure semantics together with the nonlinear query requirements. This enables advanced hierarchical processing to be automatically performed for the user, at an optimal global hierarchical optimization level, and against the entire structure. This hierarchically principled semantic processing dynamically increases the value of the data and insures the hierarchical query results are logically correct. This nonlinear hierarchical data model allows for high level hierarchical conceptual operation.

Copyright © 2007 Advanced Data Access Technologies, Inc.

Contents

Abstract

1) Introduction to Nonlinear Hierarchical Processing

2) SQLfX®'s SQL Nonlinear Hierarchical Processing
2.1) Basic Hierarchical Foundation
2.1.1) Hierarchical Data Modeling Syntax
2.1.2) Hierarchical Data Modeling Semantics
2.1.3) Hierarchical Data Preservation
2.1.4) Variable Length Multi-leg Rowset
2.1.5) Basic Multi-leg LCA Cartesian product
2.2) Basic Hierarchical Structure Processing
2.2.1) Node Promotion and Node Collection
2.2.2) Mapping SQL Operation to Hierarchical Processing
2.2.3) SQL Specific LCA Cartesian product Processing
2.2.4) Hierarchical View Joining (Data Mashup)
2.2.5) Joining Hierarchical Structures Using Association Tables
2.2.6) Structure Fragment Control
2.3) Advanced Hierarchical Processing
2.3.1) Polymorphic Hierarchical Structure Transformation
2.3.2) Variable Structure Generation
2.3.3) XML Duplicate and Shared Nodes
2.3.4) Linking Below the Lower Structure's Root Node
2.3.5) Nonlinear Hierarchical Optimization
2.3.6) SQL Update Using XML Data
2.3.7) Nonlinear ORDER BY Operation
2.3.8) Global Views and Global Queries
2.3.9) Renaming, Replication, and Splitting of Nodes

3) SQLfX® Middleware XML Enabler for SQL
3.1) Pre Processing
3.2) Post Processing
3.3) Real-time Processing

4) SQLfX®'s Advanced Capabilities and Utilized Discoveries
4.1) Advanced Hierarchical Capabilities
4.2) Utilized SQL Technology Discoveries

5) SQLfX®'s ANSI SQL Transparent XML Support Examples
5.1) Relational Data Hierarchical Query Defined
5.2) XML Document Hierarchical Query Defined
5.3) Heterogeneous Data Mashup of Two Nonlinear Hierarchical Structures
5.4) Node Promotion Shown with Combined Logical View
5.4.1) Node Promotion Override Shown with XML Format Override
5.5) Linking Below the Lower Level Structure Root Node

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.6) Lowest Common Ancestor (LCA) Multi-leg Processing
5.7) Variable Structure Generation
5.8) Structure Transformations Using Restructuring and Reshaping
5.9) Global Queries Using Global Views
5.10) Hierarchical ORDER BY Operation
5.11) Renaming, Replication, and Splitting of Nodes and their Fields

Conclusion

Copyright © 2007 Advanced Data Access Technologies, Inc.

Abstract

SQL native XML integration has not taken off as expected. This is because all the current solutions are not 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 still linearly hierarchically restricted and are still 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 capabilities to solve these problems and significantly increase today's SQL processing capabilities to an advanced 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 look and feel. These new levels of ease of use, correctness, and hierarchical processing satisfy Web 2.0 and Enterprise 2.0 higher standards.

This paper is based on my company's research and development of SQLfX (SQL for XML), the first working ANSI SQL LCA (multi-leg) query prototype. It demonstrates how ANSI SQL can automatically and naturally operate at a valid and principled nonlinear hierarchical manner fully supporting the most complex multi-leg LCA queries against relational and XML data correctly and accurately. This ANSI SQL prototype transparently integrates native XML at a full nonlinear hierarchical level by utilizing and leveraging the inherent hierarchical processing capability in an unmodified commercial ANSI SQL processor as its hierarchical processing engine. With these capabilities proven, it is shown how this nonlinear hierarchical engine capability is seamlessly extended to support native XML from input, through full nonlinear hierarchical processing, to the automatic seamless generation of valid structured XML. This entire nonlinear hierarchical process is driven transparently and navigationless by ANSI SQL-92 syntax and semantics. Advanced hierarchical structuring joining and any-to-any structure transformation are also naturally supported in our prototype and actual examples are shown.

1) Introduction to Nonlinear Hierarchical Processing

Before discussing Lowest Common Ancestors (LCAs), nonlinear processing needs to be defined. Nonlinear hierarchical structures are hierarchical structures with multiple legs. Nonlinear hierarchical processing is the processing of multiple legs in a nonlinear hierarchical structure query. This processing requires that the semantics between the legs be utilized to accurately process this more powerful and internally complex hierarchical processing. An example of nonlinear query processing is selecting data from one leg of a hierarchical data structure based on a data value in another leg of the structure. The Lowest Common Ancestor (LCA) [1] node between the data node being selected and the data node being tested needs to be located to determine the range of influence to determine the meaningful data.

LCA processing implies nonlinear hierarchical processing and visa versa. ANSI SQL LCA processing also implies that this previously required navigational centric tree walking operation is performed nonprocedurally without any navigational instructions necessary. This means that SQL's coders and users do not need to know the structure of the data being processed. This locating and processing of the LCA by SQL will be examined further later.

Copyright © 2007 Advanced Data Access Technologies, Inc.

2) SQLfX®'s SQL Nonlinear Hierarchical Processing

Nonlinear hierarchical database processing has been around for at least three decades, since IDMS and IBM's IMS database systems in the 70's. Nonlinear hierarchical processing is based on solid principles and is proven. Unfortunately, with the advent of relational processing, hierarchical processing has been forgotten and overlooked. With the advent of hierarchical processing capabilities introduced in SQL-92 and now more recently hierarchical XML, it is time to utilize this SQL automatic and natural processing capability. After all, it is freely available and can be used automatically to better utilize XML and make its processing automatic and transparent [2].

To get SQL to naturally perform hierarchically, the data must be hierarchically modeled in SQL. This means that the SQL processing is limited to hierarchical operations. 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 assumptions sound and allowing for advanced hierarchical capabilities. Interestingly, XQuery's default join is the Inner Join which is not hierarchical; 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 [3, 4, 5, 6] were undertaken adding its automatic processing.

2.1. SQL Basic 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 2.1 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 legs. 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.

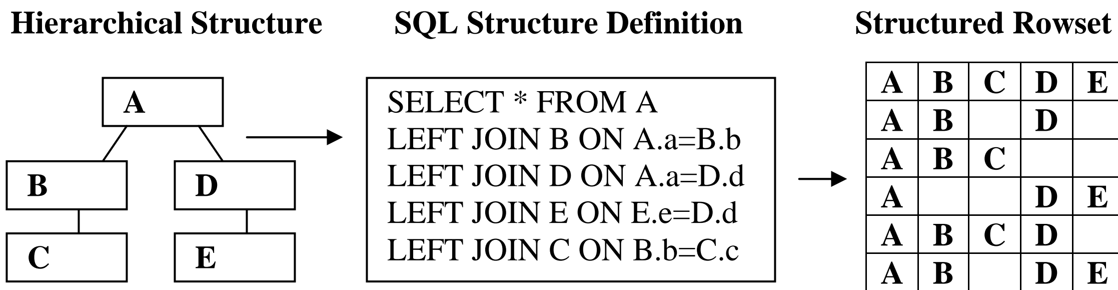


Figure 2.1 Hierarchical to Relational via Left Outer Join Mapping

For a more detailed explanation of nonlinear hierarchical processing principles and their internal operation within a 4GL than is presented in this document, see our nonlinear hierarchical processing tutorial at: http://www.adatinc.com/images/Hierarchical_Structures_and_4GLs.pdf.

2.1.1) SQL 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 specified 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. 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 multi-leg structure [7] because the second ON clause also references the A node. This is shown in Figure 2.1

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 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 and current research in this area can be found at [8, 9, 10]. Figure 2.1.1 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.

Figure 2.1.1: Range of Data Filtering

2.1.2) 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.

2.1.3) 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 2.1.

Copyright © 2007 Advanced Data Access Technologies, Inc.

2.1.4) SQL Variable Length Multi-leg Rowset

For SQL to perform nonlinear hierarchical processing, its standard fixed rowset must automatically accommodate multiple variable length legs. This is done automatically by the hierarchical data preservation being performed by the Left Outer Join operation which inserts NULL values in the rowset representing missing data which makes the length of the hierarchical legs variable and keeps the variable length legs aligned for proper SQL operation. This can also be seen in the structured rowset in Figure 2.1 where blank cells represent nulls.

2.1.5) SQL Basic Multi-leg 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 legs. This requires Lowest Common Ancestor (LCA) logic to coordinate the processing between legs. 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. The 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 are very similar to LCA nodes. They keep the relationships between leg occurrences meaningful. This multi-leg processing increases the value of the data by naturally utilizing the semantics between the legs.

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. Interestingly, LCA determination has been a difficult academic problem to solve efficiently and required hierarchical tree walking. SQL has managed an automatic way around these procedural problems [11]. This LCA automatic use in SQL will be discussed further. An SQLfX® example is shown in Section 5.6.

2.2) SQL Basic 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 views and their joining; and fragment control.

2.2.1) SQL Node Promotion and Node Collection

Hierarchical node promotion and node collection occurs when nodes are not selected for output and are removed. Actually they 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 roots are removed, multiple fragments can also be produced that also have different node types as their new roots. This is shown below in Figure 2.2.1

In Figure 2.2.1 below, 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 moves the fields listed for output. This

Copyright © 2007 Advanced Data Access Technologies, Inc.

operation removes fields and nodes without affecting other fields and nodes such that the structure stays intact except for the removed data. This causes node promotion and node collection to occur automatically. The SQL hierarchical query processor's processing of the SELECT operation recognizes the data structure modification and modifies the internal definition of the current hierarchical structure. An SQLfX® example of node promotion is shown later in Section 5.4.

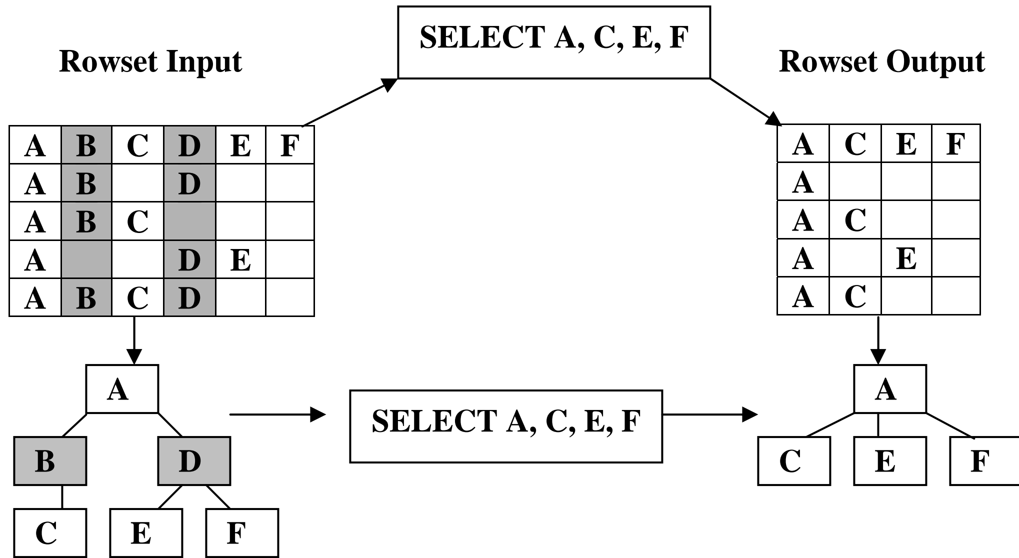


Figure 2.2.1: Demonstrating Node Promotion and Node Collection in SQL

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 and will be discussed further later. The basic hierarchical operation of the SELECT, FROM and WHERE operations are shown in Figure 2.2.2.1 below.

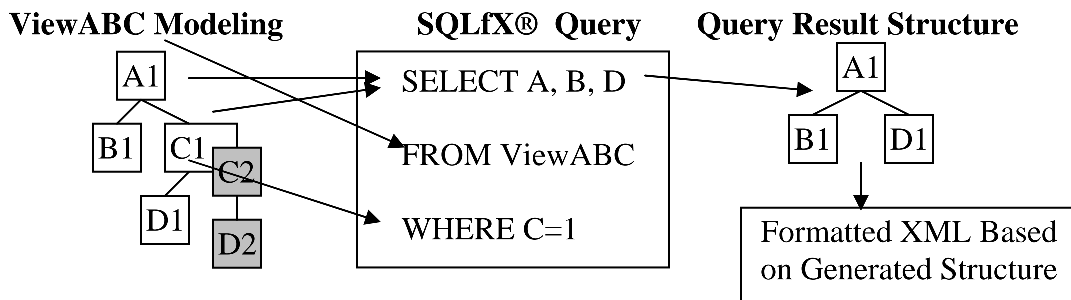
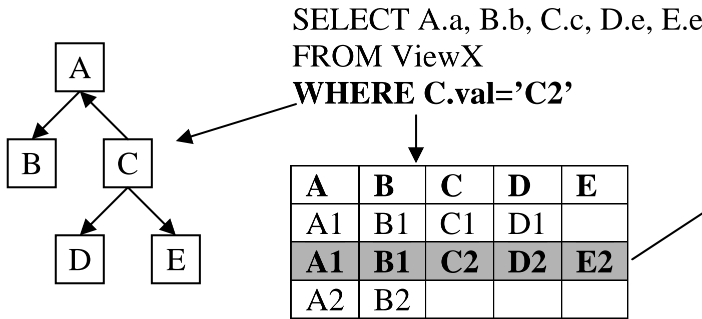


Figure 2.2.2.1: SQL hierarchical query specification and operation overview

Copyright © 2007 Advanced Data Access Technologies, Inc.

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 2.2.2.2. 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 which qualifies node A, and node A qualifies node B. The qualification of node B involves LCA processing and will be covered further later.

Data Qualification Flow **ViewX Data Result**



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

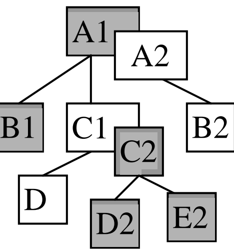


Figure 2.2.2.2 WHERE clause hierarchical data qualification flow

Additionally, a FOR XML clause 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 used, or it can override node promotion so that full paths to data are preserved even when nodes are not selected for output.

2.2.3) SQL Specific LCA Cartesian product Processing

The LCA processing is required in two locations in SQL processing and it occurs automatically thanks to the Cartesian product processing and its duplicate data generation described earlier. The duplicate values are necessary because the SQL processing occurs a row at a time, usually without memory across row occurrences, duplicate data provides this memory. For 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 legs, any correct combination of the multiple leg occurrences qualifies the condition. This simulates hierarchical tree walking and is very efficient.

The SQL SELECT operation also requires LCA processing when selecting data across legs from one leg of the structure based on data from another leg. 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 LCA processing in SQLfX® is shown later in Section 5.6.

2.2.4) SQL Hierarchical View Joining (Data Mashups)

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-leg nonlinear hierarchical structure. This operation naturally hierarchically combines the views between any link point in each structure using a simple single join operation performing this data mashup. This shows how powerful and flexible hierarchical views are. A SQLfX® example is shown in Section 5.3

SQL hierarchical view operation automatically adapts to fit the query and can be variable invoked to specify variable requirements. The SELECT list automatically and easily specifies the values to be output and processed. The Left Outer Join is more than just processing instructions, it represents the hierarchical structure metadata that defines the hierarchical structure and its semantics that has been modeled and is being processed hierarchically. This means this meta information can be utilized for value added enhancements described shortly. Meta information can also be used to influence internal operations. One of these is hierarchical optimization, allowing un-accessed legs to be dynamically removed from queries. This enables large global views with no penalties since they can always be optimally optimized.

2.2.5) Joining Hierarchical Structures Using Association Tables

Using SQL's data independence, it has the ability to join hierarchical structures using an externally introduced relationship in a single table where one does not exist previously in the structures being joined. In addition, the relationship being introduced can be a Many-to-Many relationship not previously possible with fixed structures. This many-to-many relationship also allows the supporting of intersecting data that can be very useful and also not possible without a many-to-many bi-directional relationship.

SELECT * FROM XMLView1
LEFT JOIN AssocTable ON XMLView1Key=AssocKey1
LEFT JOIN XMLView2 ON AssocKey2=XMLView2Key

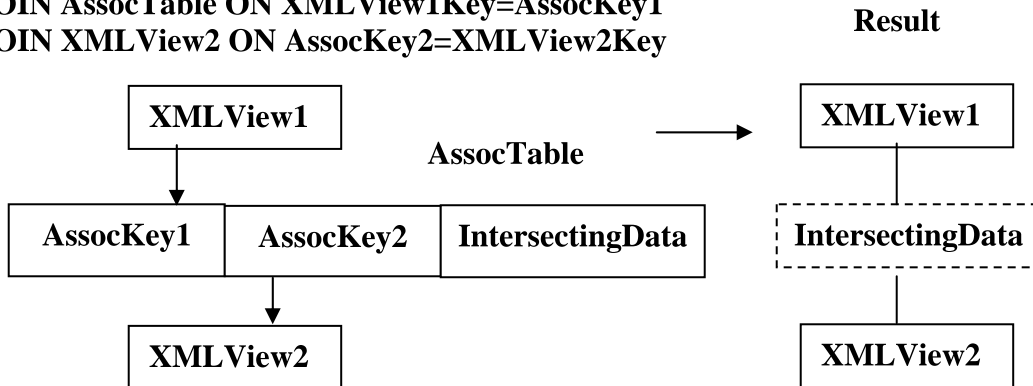


Figure 2.2.5 Association table with intersecting data

The structure diagram in Figure 2.2.5 demonstrates how two fixed XML structure document types can be joined using an external relationship in the association table and correctly produce the hierarchical result. If the user selects the intersecting data, it will appear as shown in the result, otherwise it is not including and node promotion will make it transparently disappear preserving the structure shown.

Copyright © 2007 Advanced Data Access Technologies, Inc.

2.2.6) 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 SQLfX® to keep them valid. The user can still override this standard 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 Section 5.8.

2.3) SQL Advanced Hierarchical Processing

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

2.3.1) Polymorphic Hierarchical Structure Transformation

It turns out that relational processor's rowset manipulation through SQL's SELECT, FROM, WHERE and alias capability is quite powerful. This happens by using the SQL alias 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 usually represent valid sub hierarchies. No matching values are required within the fragment grouping. Rejoining fragments requires relationship values to join on but are not limited to their previous join criteria. The transformed structure is recognized and used by SQL in the overall unified structure. Reshaping is also possible; it can perform any-to-any structure transforms without using any available relationships. Polymorphic operation [12] means the source structure does not have to be known when the transform is defined. A SQLfX® example is shown later in Section 5.8.

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 leg 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 will be shown in a SQLfX® example in Section 5.7.

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 to change the name of these nodes

for each path so that they can be referenced unambiguously as shown below in Figure 2.3.3.

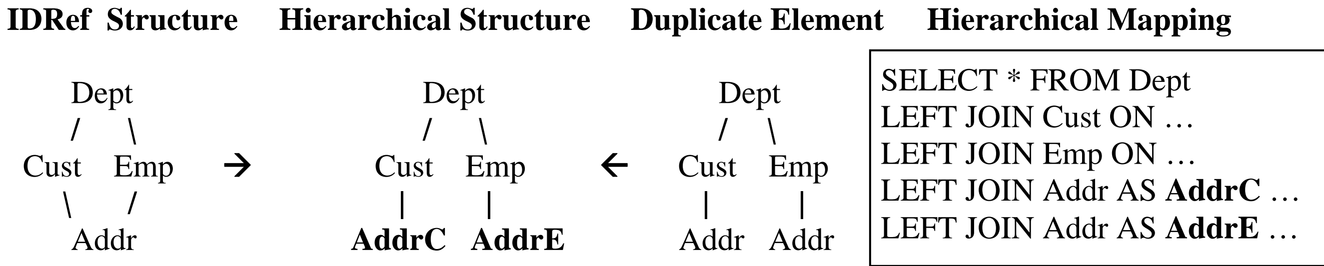


Figure 2.3.3: Hierarchical mapping solutions for network structures

It might be argued that this solution does not utilize duplicate element types usefulness and academic LCA logic solutions have taken into account duplicate element types by using the first one that currently exists. This does mean that an LCA node's position dynamically changes unpredictably. This is acceptable for variable semistructured data and searching it hierarchically, but for database data use and its hierarchical processing, you can not 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 capability will be added to the XML View definition in release 1.

2.3.4) Linking Below the Lower Structure's Root Node

There was previously an example of joining full hierarchical structures but no linking (joining) requirement was mentioned. Usually when hierarchically linking structures, the lower structure had to be linked at its root node otherwise the semantics caused problems. Since SQL hierarchical processing had handled all other hierarchical operations correctly automatically, we analyzed how SQL naturally links below the root and it made perfect semantic sense. This being the case, we can allow it so that the user still does not have to be concerned with the knowing the structure. The semantics of linking below the 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. A SQLfX® example will be shown later in Section 5.5.

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 views representing 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.

2.3.5) SQL Nonlinear Hierarchical Optimization

Our nonlinear hierarchical optimization works for ANSI SQL relational processing and all physical hierarchical structures. SQL 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

Copyright © 2007 Advanced Data Access Technologies, Inc.

required from the table since a missing match value anywhere in the view can cause the removal of data. For example a Department view comprised of the Department table inner joined to the Employee table will have a Department removed when the department has no employees. This view has to be maintained even when there is no reference to the Employee table to insure the integrity of the view.

The standard Inner Join logic is not used when hierarchical structures are being accessed modeled hierarchically by a Left Outer join so that the department's existence is not dependent on employee's existence [13]. 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. This optimization is easily applied to SQL by rewriting the query removing unnecessary joins as shown in Figure 2.3.5. The optimization occurs under the covers, so there is no actual example to show.

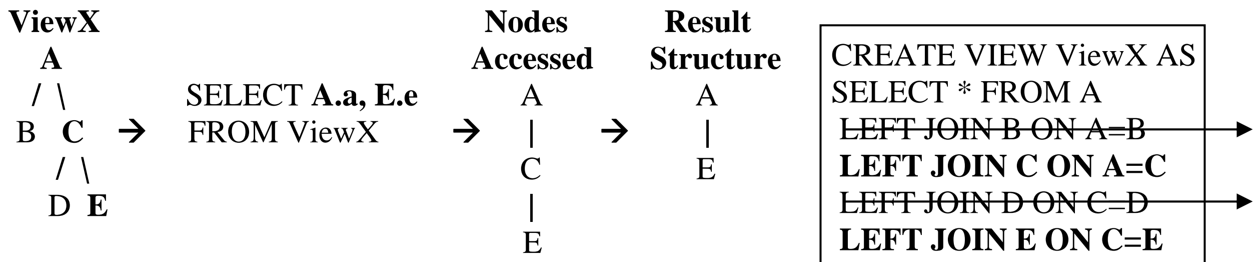


Figure 2.3.5: Hierarchical access optimization

2.3.6) SQL Update Using XML Data

Updating the SQL database using data from XML or a heterogeneous combination is easy to perform since the heterogeneous result is available from the processing described already. An SQL Update operation can access the data using a subquery. The required single row data can be isolated by walking down the structure from the root to desired node using a WHERE clause on the subquery. This capability will be added to release 1.

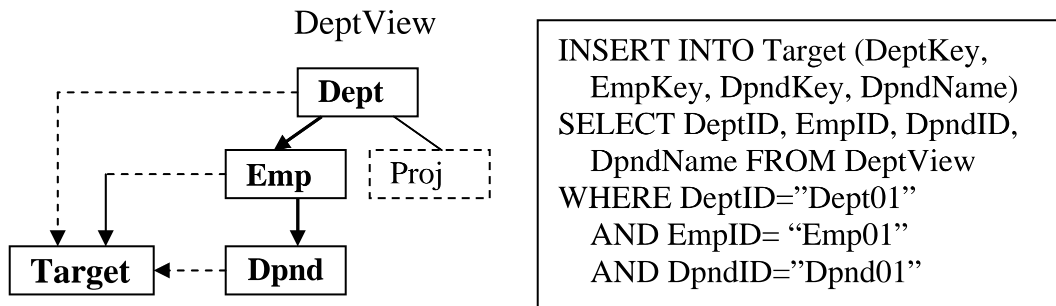


Figure 2.3.6: SQL Insert operation navigating a hierarchical data input source

2.3.7) Nonlinear ORDER BY Operation

SQL ordering of hierarchical structures does not relate 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 structure placing Employee over Department. This is demonstrated below in Figure 2.3.7. Second, in a nonlinear hierarchical structure, each hierarchical leg can be ordered separately. This was solved by changing the SQL ORDER BY semantics in the SQLfX® middleware to take the nonlinear hierarchical structure into account to

Copyright © 2007 Advanced Data Access Technologies, Inc.

arrange the sort fields down the paths of the hierarchical structure. This allows legs of the structure to be separately and independently ordered which will be shown in Section 5.10.

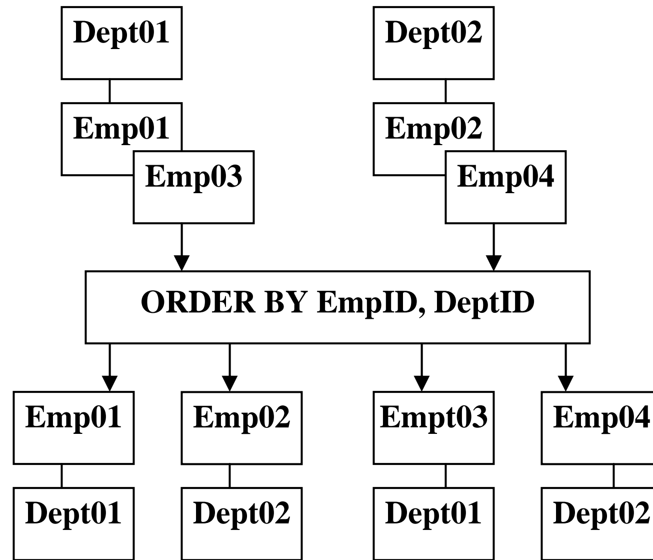


Figure 2.3.7: Ordering out of hierarchical order can change the structure inadvertently

The problem with performing multi-leg hierarchical ordering is that each leg has to be separately ordered. Using the structure in Figure 2.3.8 on the right it can be seen that ordering leg AB and AC would not be possible in SQL using its Order By operation. But it is necessary for XML hierarchical output which can be nonlinear. SQLfX® can accomplish this as shown in the example in Section 5.10.

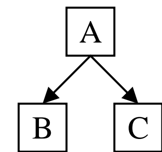


Figure 2.3.8

2.3.8) Global Views and Global Queries

As mentioned already, SQLfX®'s hierarchical access is optimized so that each view executed is optimized at execution so that un-accessed paths are not accessed. This means there is never overhead for using views that contain larger views than required. 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. This 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 and SQLfX® 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 SQLfX® example of a global query is shown later in Section 5.9.

2.3.9) Renaming, Replication, and Splitting of Nodes and Fields

SQL's aliasing allows views, tables, and columns to be renamed. This can be seamlessly extended in SQLfX® to enable renaming of nodes and their data items which is subsequently transferred to XML output. In addition, renaming allows the duplication of tables with the ability to separately control the data field inclusion in the tables so that different data items could exist in each causing it to be split up. These duplicate renamed tables also correspond one-to-one to nodes in the hierarchical structure being built and processed with the result seamless output in XML. A SQLfX® example of these renaming capabilities is shown later in Section 5.11

3) SQLfX® Middleware XML Enabler for SQL

There were three choices for how to build our SQLfX® ANSI SQL native XML processor prototype. Build it from scratch, modifying an existing ANSI SQL processor, or build a middleware product that sits around any ANSI SQL processor to utilize its inherent hierarchical nonlinear processor as shown below in Figure 3.0. The most economical is the ANSI SQL driven middleware solution we chose. It also offers the best proof that the nonlinear 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. SQLfX® sits around the customer's ANSI SQL processor and supplies pre, post, and asynchronous processing to support native XML at a full hierarchical level, from input to output.

SQLfX® Middleware Processing Layout

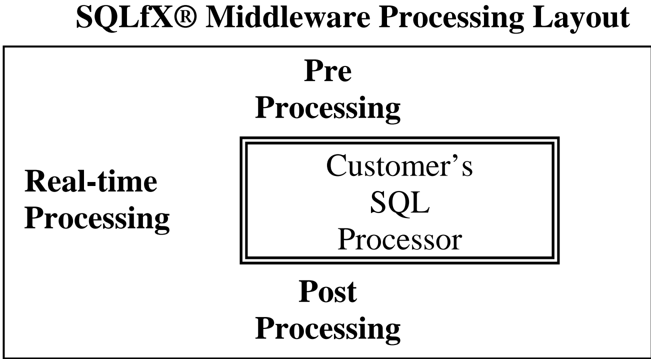


Figure 3.0 Middleware relationship to customer's SQL processor

3.1) 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 data structure is determined by analyzing the hierarchically specified SQL. The optimization is determined by also analyzing the hierarchical structure defined from the specified SQL to determine table or nodes that do not require access. This is possible because of hierarchical preservation semantics. Tables and views are also defined to the preprocessor and XML data documents can be optionally pre loaded in ETL batch mode into tables to be used hierarchically when accessed.

3.2) Post Processing

Post processing involves automatically transforming the hierarchically processed relational result rowset to the structured formatted XML based on the resulting hierarchical structure. The input hierarchical structures can change their structure by the natural hierarchical processing taking place. Data replications are recognized and not included in the XML result. The hierarchical optimization performed in the preprocessing stage should significantly help keep these data explosions smaller. Nonlinear hierarchical ordering is assisted in the post processing phase also.

3.3) Real-time Processing

The real-time processing is occurring simultaneously with the SQL processing. It processes requests from the SQL processor to service EII real-time requests for native XML input. It returns a rowset that is naturally mapped by its associated view's hierarchical SQL. This maintains the XML's hierarchical input and processing transparency.

Copyright © 2007 Advanced Data Access Technologies, Inc.

4) SQLfX® Advanced Capabilities and Utilized Discoveries

Our research into SQL hierarchical processing produced many new and advanced hierarchical capabilities possible in ANSI SQL. Many of these were based on our turning up previously unknown hierarchically principled operations in ANSI SQL. Some of these discoveries were searched for after we found the new capabilities by determining the processes behind the capability, such as how and where LCA processing was operating. Other new hierarchical capabilities became possible based on our earlier new capabilities.

4.1) Advanced Hierarchical Capabilities

Dynamic operation, transparent XML, and complex LCA queries make a powerful combination of features supported by ANSI SQL. The SQL hierarchical views enable a very high level of conceptual query specification. The XML integration and hierarchical processing is performed transparently without the need of new syntax and is integrated naturally into all aspects of SQL processing. All of SQL's operations and XML support are still operationally supported dynamically and the nonprocedural and navigationless access makes interactive processing very practical and feasible. Complex multi-leg LCA Queries involving many legs is impractical for procedural queries. SQL can handle the most complex multi-leg query automatically and accurately. Supporting global queries is a current problem because procedural navigation is necessary.

Our ANSI SQL nonlinear native XML transparent integration prototype inherently supports nonlinear hierarchical processing and supports multi-leg LCA processing naturally. It has proven that these capabilities can be naturally and seamlessly extended to support ANSI SQL native XML integration fully and transparently at a full hierarchically processing level. By automatically determining the hierarchical structure being processed, it can also output the hierarchical formatted XML automatically and correctly.

4.2) Utilized SQL Technology Discoveries

Our SQLfX® technology is unique in that we have uncovered new capabilities in ANSI SQL processing that are not being utilized today. We are utilizing this new technology to transparently support high levels of hierarchical integration and processing of relational and XML data directly in ANSI SQL.

4.2.1) SQL Linear Hierarchical Processing

The SQL-92 Outer join additions to ANSI SQL enable ANSI SQL to inherently operate hierarchically. Our research showed that these SQL additions also enable: hierarchical preservation; the separate storage of variable length hierarchical legs in rowset; and the new Syntax of the Left Join can define and model hierarchical structures while its semantics define its hierarchical processing. SQL is not aware of this hierarchical processing occurring.

4.2.2) SQL Nonlinear LCA Hierarchical Processing

Multi-leg nonlinear hierarchical processing requires a complex processing known as Lowest Common Ancestor (LCA) processing to coordinate the processing between legs of the structure. Nonlinear hierarchical processing has always used hierarchical tree walking to do this processing. Our research discovered that this LCA processing was occurring naturally in the relational Cartesian product processing of ANSI SQL. This inherent LCA processing enables ANSI SQL hierarchical processing to operate at a full nonlinear level. Even more remarkable is that the LCA processing occurs correctly regardless of the number of separate and nested LCAs that are naturally produced by multi-leg queries.

Copyright © 2007 Advanced Data Access Technologies, Inc.

4.2.3) Valid Hierarchical Modeling of Linking Below the Root

Hierarchical data modeling has always had problems applying correct semantics to the linking of the upper level structure to the lower level structure to any other node than its root. During our research of SQL's natural hierarchical processing we discovered SQL was naturally handling this powerful new capability correctly in its natural joining of hierarchical structures and the derived semantics made sense. So we allow this very useful and user friendly data modeling and hierarchical processing capability and know how to interpret the data modeling semantics which SQLfX® must keep current track of.

4.2.4) Powerful Hierarchical Access Optimization

ANSI SQL does not inherently support hierarchical access optimization since SQL has no concept of hierarchical structures or their processing. By limiting SQL processing to only hierarchical processing we found that we could apply powerful hierarchical access optimization and do it in a preprocessing step. In this way, un-accessed hierarchical pathways could be eliminated from the SQL query before being submitted to the SQL processor where it will be further relationally optimized.

4.2.5) SQL Hierarchical Structure Transformation

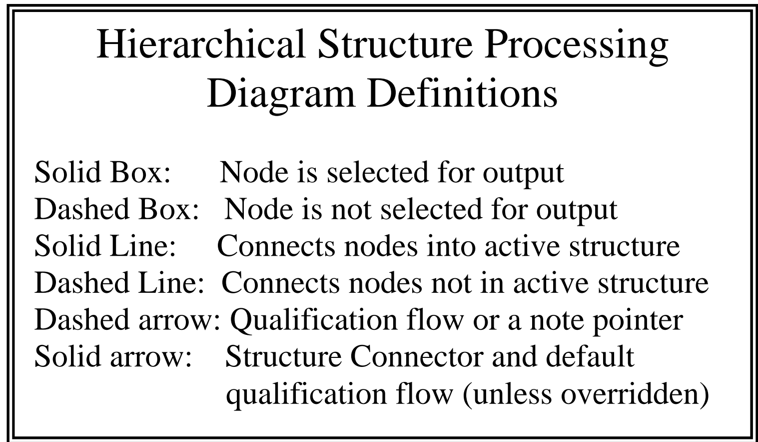
Using the logical and/or physical contiguous storage nature of relational rowsets combined with their flexible position manipulation capability, our research turned up new uses for it. These uses were exploited in a new method for performing hierarchical structure transformations. These include Restructuring structures using existing data relationships and using Reshaping to perform any-to-any structure transformations without the use of existing relationships in the data structure.

Copyright © 2007 Advanced Data Access Technologies, Inc.

5) SQLfX®'s ANSI SQL Transparent XML Support Examples

This section contains working and dynamic examples from our SQLfX® prototype starting with the hierarchical modeling basics continuing through advanced unique transformations examples. This data modeling and structure processing is also shown through box diagrams demonstrating the entire operation. Sometimes the input and output structures are shown as separate structures.

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 below.


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)

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

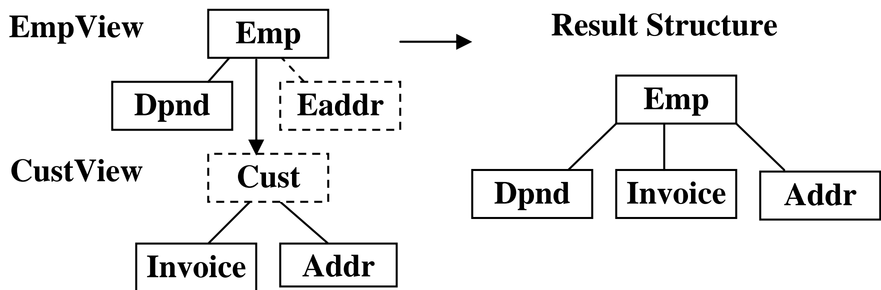


Figure 5.0: Sample diagram structure

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

5.1) Relational Data Hierarchical Query Defined

This prototype example demonstrates the creation of the relational Employee view EmpView. This includes the Left Outer Join that models the data structure to be processed which is shown in Figure 5.1. EmpView is executed 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.

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

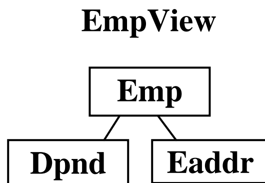


Figure 5.1: EmpView structure

```
SELECT EmpID, DpndID, EaddrID
FROM EmpView
```

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

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.2) 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 hierarchical structures. It will automatically create a Left Outer Join associated with the view that models the data structure shown in Figure 5.2 below. For internal processing, the CustView is executed and automatically produces structured XML following it. It preserves its XML hierarchical order unless overridden by an ORDER BY (see section 5.10).

```
CREATE XML CustView
Cust(
  CustID Char(8),
  CustStoreID Char(8)),
Invoice(
  InvID Char(8),
  InvCustID Char(8),
  InvStatus Char(8)) Parent Cust,
Addr(
  AddrID Char(8),
  AddrCustID Char(8),
  AddrState Char(8)) Parent Cust
```

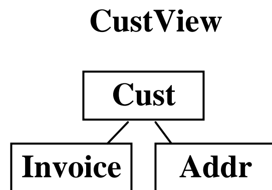


Figure 5.2: CustView structure

```
SELECT CustID, InvID, AddrID
FROM CustView
```

```
 <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>
```

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.3) Heterogeneous Data Mashup of Two Nonlinear Hierarchical Structures

In this example in Figure 5.3, the Employee relational view, EmpView, is joined over the Customer XML view, CustView, under its Emp node as determined by the ON join condition. This is the first example of a join of hierarchical structures; notice how simple it was, a simple single join using SQL views. This was performed dynamically and produced the combined hierarchical structure. This can be considered a powerful data mashup operation. It also demonstrates a heterogeneous join which is transparent; all views operate the same regardless of their data type. 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 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

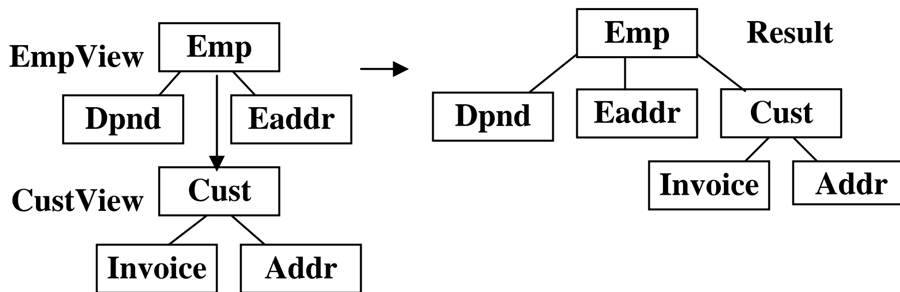


Figure 5.3: Heterogeneous data mashup join

```
<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>
```

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.4) Node Promotion Shown with Combined Logical View

In this example, the Employee view, EmpView, is joined over the Customer view, CustView and is stored in a SQL view, EmpCust, used in this example. This is an example of a logical view comprised of embedded physical 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, causing the Eaddr node not to be accessed and node promotion to occur around the Cust node. This is indicated in Figure 5.4 by the dotted boxes. The result structure is shown in Figure 5.4 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.

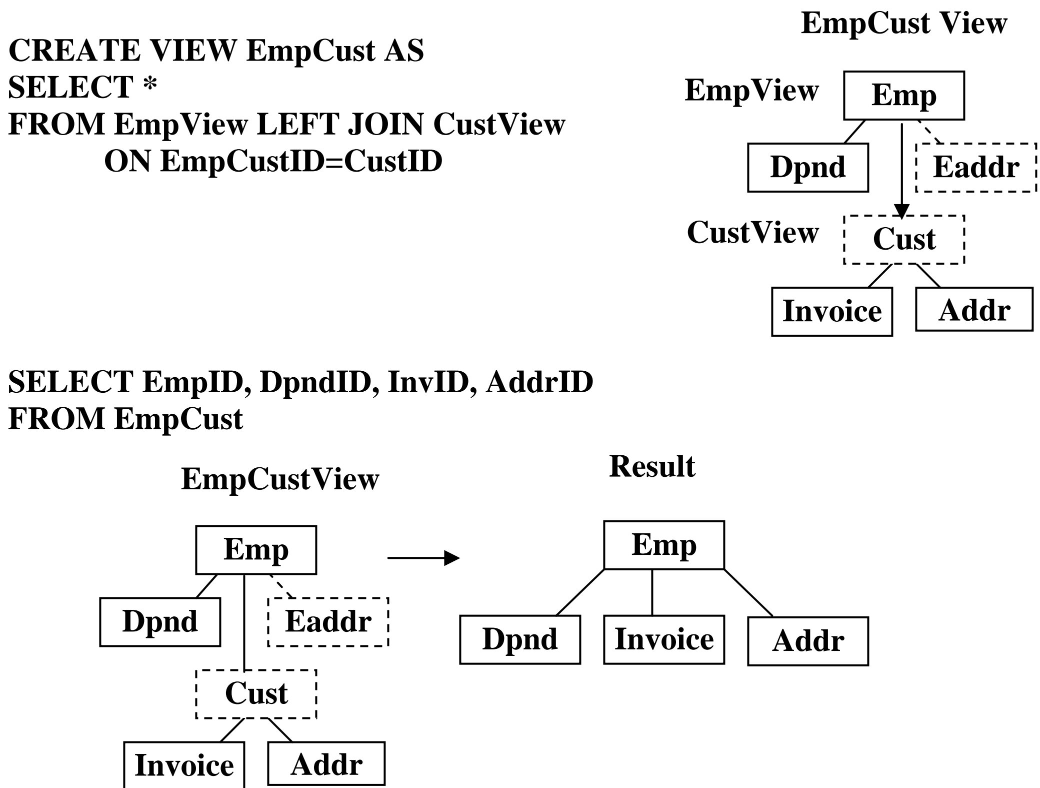


Figure 5.4: EmpCust node promotion result structure

```
<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>
```

5.4.1) Node Promotion Override Shown with XML Format Override

This is basically the same query as in the previous example in Figure 5.4 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.

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

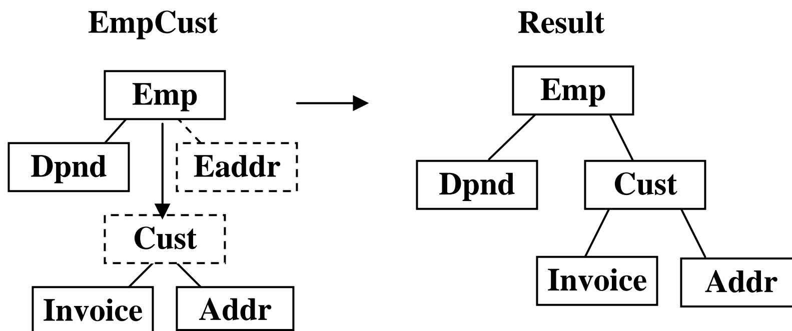


Figure 5.4.1: EmpCust node promotion overridden and element format used

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

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.5) Linking Below the Lower Level Structure's Root Node

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. In this example in Figure 5.5, 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 root and its descendents still have their same responsibility for materialize the view. This works for physical structures such as XML too.

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

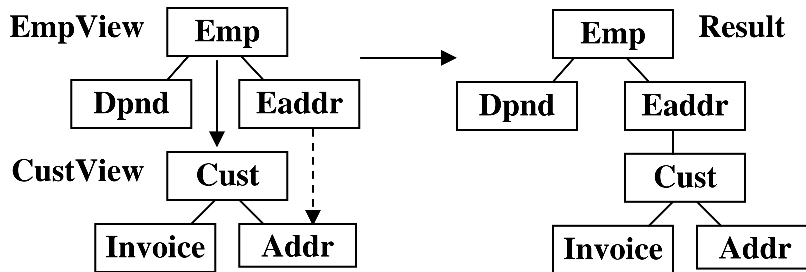


Figure 5.5: Linking below lower level structure root

```
<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>
```

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.5.1) Linking Below Lower Level Structure Root with Node Promotion

This is the same query as the previous query in Section 5.5 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 5.5.1 by the dotted boxes. The result structure is also shown in Figure 5.5.1 with its resulting XML.

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

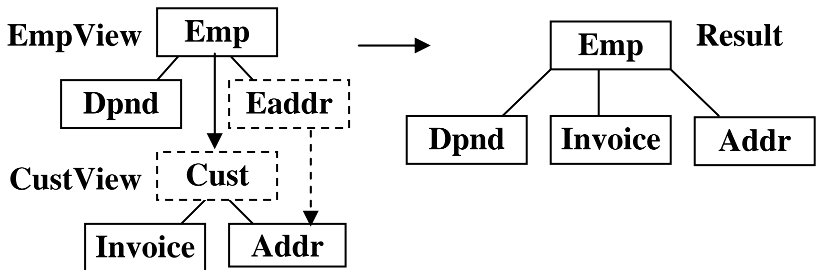


Figure 5.5.1: Linking below lower level root with node promotion

```
<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>
```

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.6) Lowest Common Ancestor (LCA) Multi-leg Processing

There are two different types of hierarchical Lowest Common Ancestor (LCA) processing that must take place for SQL multi-leg nonlinear hierarchical queries. One type occurs for WHERE clauses and another one for SELECT clauses. LCA processing coordinates the processing between multiple legs referenced in the query to keep the processing of the legs under a common ancestor so the results remains meaningful. This participation of multiple legs and its utilization of additional semantics between the related legs also dynamically increases the value of the data.

5.6.1) LCA WHERE Clause Semantics Example

This is an example of how the WHERE clause uses LCA logic in processing multi-leg queries. It uses the EmpCust view defined in Section 5.4. 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 5.6.1. All combination of values tested is performed under the LCA, it is the control point. The 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.

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

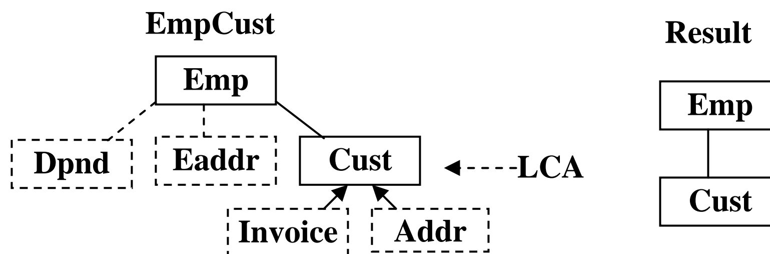


Figure 5.6.1: WHERE clause semantics

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

5.6.2) LCA SELECT Clause Semantics Example

This is an example of how the SELECT list operates hierarchically using LCA logic in processing a multi-leg query. This example uses the EmpCust view defined in Section 5.4. By qualifying on the 'Inv01' value, all the Dpnd and Cust nodes under the selected LCA Emp01 are included as shown below in Figure 5.6.2. The two fragments nodes are placed un the optional Root collection node.

SELECT DpndID, CustID
FROM EmpCust
WHERE InvID='Inv01'

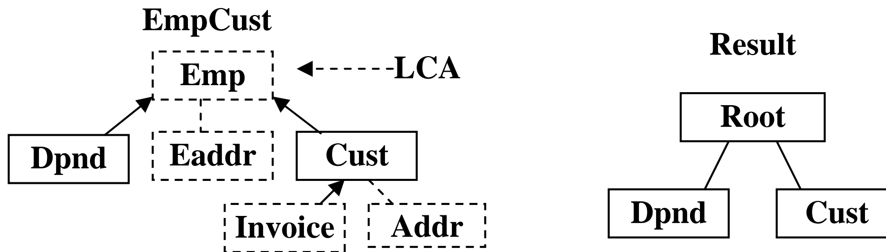


Figure 5.6.2: SELECT clause semantics

```
<root>
 <dpnd dpndid="Dpnd01"/>
 <cust custid="Cust01"/>
</root>
```

5.6.3) LCA SELECT and WHERE Combination Semantics Example

This is an example of how the SELECT and WHERE clauses both use their LCA logic together in processing a multi-leg queries. It combines the LCA processing from the previous two LCA examples as shown below in Figure 5.6.3. The WHERE LCA is nested under the SELECT LCA, but this occurs automatically. Even this example is a fairly simple LCA example. A given query can have many SELECT and WHERE LCAs that require processing and interoperation.

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

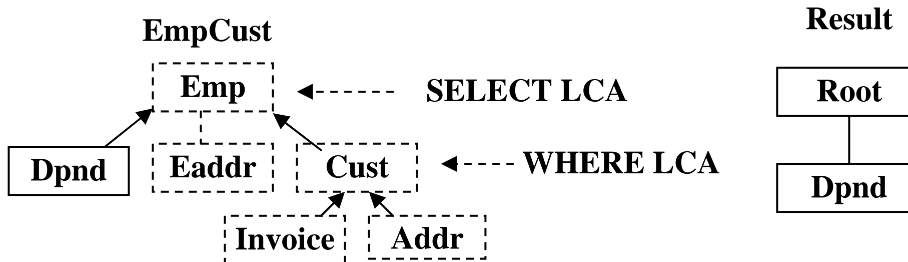


Figure 5.6.3: WHERE clause semantics

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

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.7) Variable Structure Generation

This is an example of a variable structure. It uses a variable view generation that is controlled by a data value up the path by the addition of *AND EmpStatus=”F”* appended to the ON clause as shown in Figure 5.7 below. Its EmpStatu*s* value controls whether the CustView is generated as also shown below. Once it is when EmpStatus=”F” and the other time it is left out when EmpStatus is empty. This is a very simple example, but proves its use. More complex uses can use multiple variable sub structures that can even be nested and can be invoked many times in a given document occurrence.

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

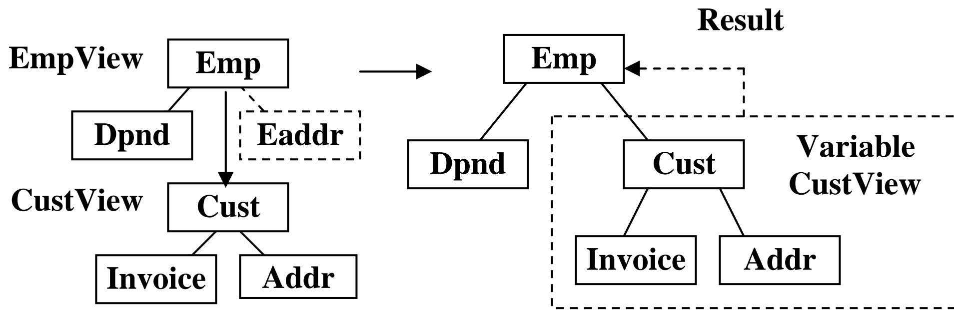


Figure 5.7: Variable structure example

```
<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>
```

5.8) Structure Transformation Using Restructuring and Reshaping

In Figure 5.4 we saw the join of two hierarchical structures with only a portion (fragment) of each structure joined. This is 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 and 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 multi-leg variable length structures, it also very flexibly automatically handles the movement of hierarchical fragments in the rowset.

5.8.1) Restructuring EmpCust Structure

This example transforms the EmpCust structure in Figure 5.8.1, by moving Invoice over the Emp node. It uses SQL's alias capability to create two structures out of EmpCust (X and Y). It then rejoins them with Invoice on top using existing relationships in the data. The resulting structure is shown in Figure 5.8.1 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

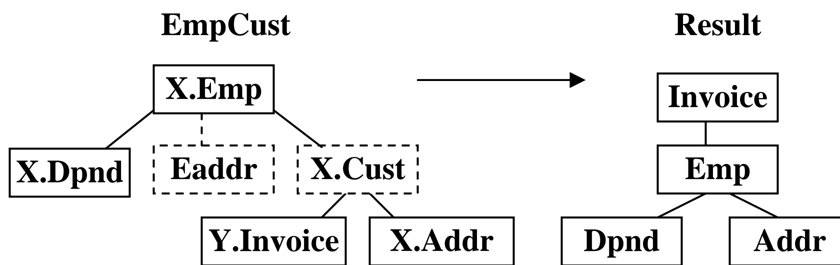


Figure 5.8.1: Restructured EmpCust structure

```
<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>
```

5.8.2) 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 can often be necessary since XML does not need to rely on foreign keys because of its contiguous 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 to build the desired new structure top-to-bottom. This can be seen below in Figure 5.8.2 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.

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

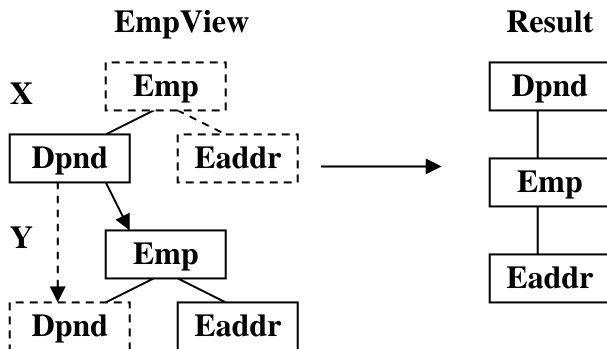


Figure 5.8.2: Simple Reshaping nonlinear to linear structure

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

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.8.3) Complex Reshaping nonlinear Structure to Linear Structure

This example is similar to the previous reshaping example in Section 5.8.2, except the structure transform is more complex. In this transform, EmpView still bend 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 5.8.3 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 this 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
```

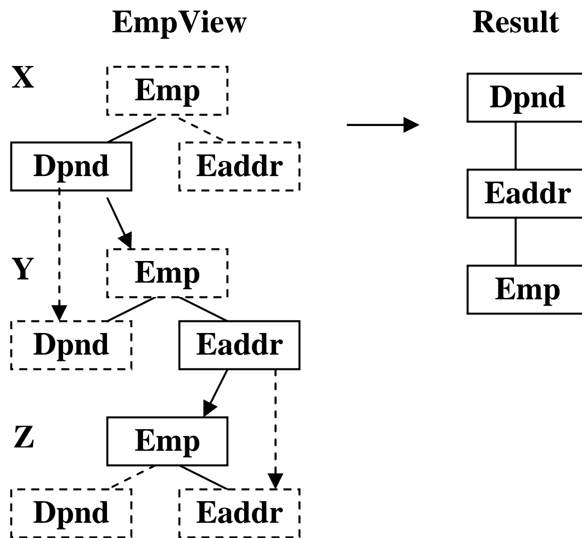


Figure 5.8.3: Complex Reshaping nonlinear to linear structure

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

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.8.4) Reshaping Previous Linear Result Back to Nonlinear

This example reverses the EmpView reshaping from the previous example in Section 5.8.3.back to the EmpView structure. Since transforms can cause data loss, there may be data that is not restored. To recreate the previous result we will take the reshaping SQL from the previous reshaping and store it in a view named EmpReshape and then we will reshape this reshaping view back the original EmpView. This will also test out the ability to place a transformation into a view. Notice on the final ON clause that the new Z.EmpID reference is also to the first structure occurrence at X.EmpID. This places Eaddr under the EMP node in the result also as shown in the result in 5.8.4. Otherwise if it was Y.EmpID, Eaddr would be placed under the Dpnd node. This happens because of the data modeling when linking below the root.

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

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

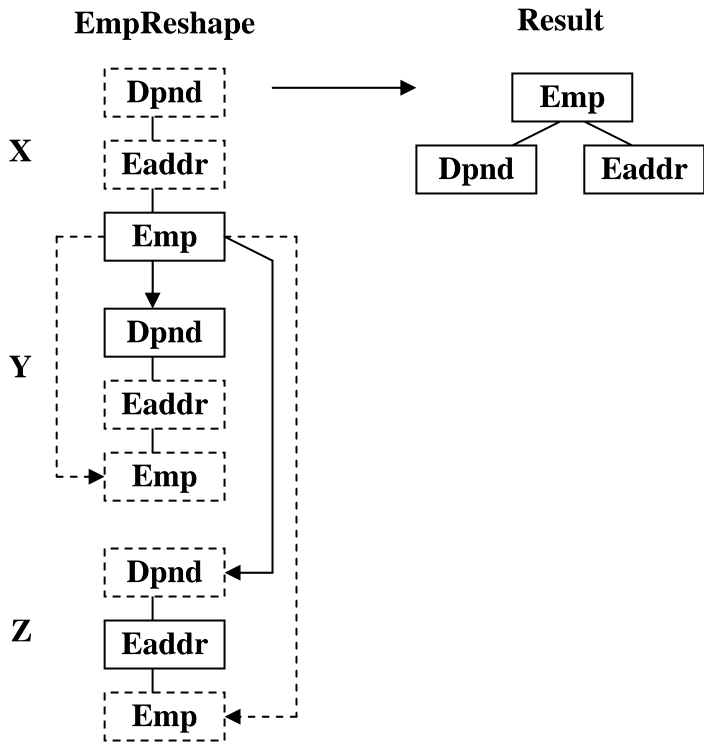


Figure 5.8.4) Reshaping previous linear result back to its nonlinear shape

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

5.8.5) 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 Section 5.8.2 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 Section 5.8.3 is polymorphic because it does not rely on the structure of the source structure by locating and moving one node at a time. The choice of using reduced steps or a polymorphic solution is up to the user.

Since polymorphic reshaping does not depend on the source structure's hierarchical structure to operate, this means that either version of ViewX shown below could be used as source input in a polymorphic reshaping. It would produce the same transformed target structure in either case. This is demonstrated below in Figure 5.8.5.

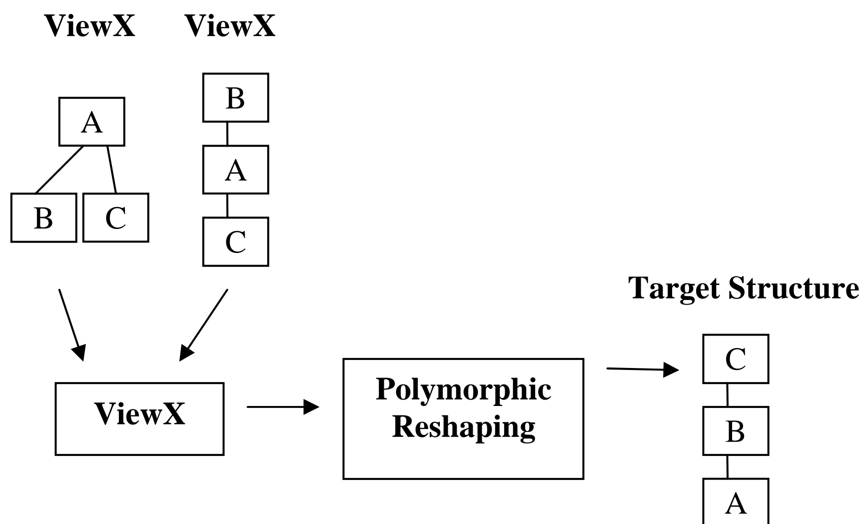


Figure 5.8.5: Reusable polymorphic reshaping

5.8.6) 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 one correctly models the target structure you desire. This is important because their semantics are different as described earlier.

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.9) Global Queries Using Global Views

As mentioned already, SQLfX®'s hierarchical access is optimized so that each view executed is optimized at execution so that un-accessed paths are not accessed. This means there is never overhead for using views that contain larger views than required. 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. Most of the query examples so far have demonstrated this Global View capability using specific queries that identified the required output data explicitly in their Select list. This 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 and SQLfX® 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 such as in the example in Figure 5.9.1, but first we will use no query modification and just print out the entire Global Query so we have something to compare against.

SELECT * FROM EmpCust

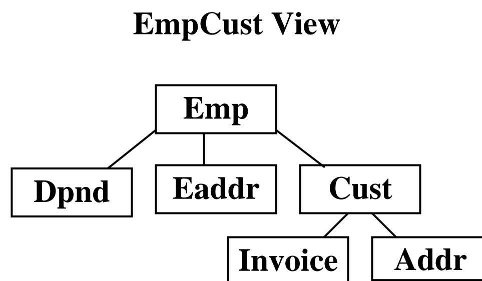


Figure 5.9.0: Global Query with no modifications applied

```
<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>
```

5.9.1) Global Query Filtering

In Figure 5.9.1 example below, The Global Query 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 5.9.0 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 the entire hierarchical structure. This global type of hierarchical operation is not really feasible using procedural navigation.

SELECT * FROM EmpCust WHERE InvStatus='O'

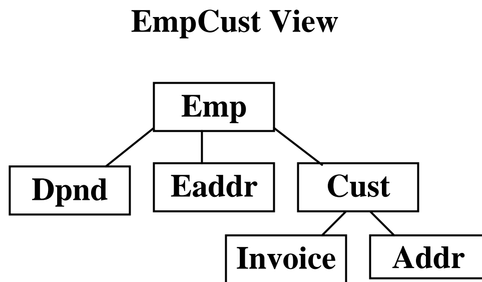


Figure 5.9.1: Global Query with data filtering applied

```
<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>
```

Copyright © 2007 Advanced Data Access Technologies, Inc.

5.10) Hierarchical ORDER BY Operation

This example demonstrates hierarchical ordering for nonlinear hierarchical structures. The sort fields can be entered in any order as demonstrated below in example 5.10; they will be reordered to fit the hierarchical structure. Multiple fields for a given node type will be left in the logical order specified. In the produced hierarchical result, you will notice that the root and both siblings have each been ordered separately descendingly, which is different from its input default order shown in example 5.2 of the CustView. Standard SQL's linear ordering can not do this, so SQLfX® has helped it along by modifying the ORDER BY operation's semantics. 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

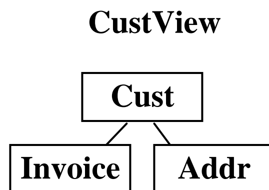


Figure 5.10.1: CustView structure

```
<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>
```

5.11) Renaming, Replicating and Splitting Nodes and their Fields

This query is based on the view EmpView used often previously. This query 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 duplicated and the data to be broken up between the two. 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 SQLfX® data modeling, Addr2 has been linked underneath Addr2. 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.

```
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
```

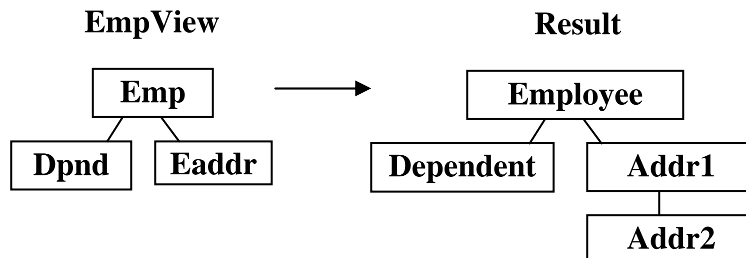


Figure 5.11: Renaming Nodes and Their Fields

```
<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>
```

SQLfX® Beta Conclusion

No other SQL/XML solution operates so easily, powerfully or naturally on hierarchical structures as SQL natural hierarchical processing does. The result of processing the example SQL statements in this document should 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. How the hierarchical modeling ANSI SQL semantics processed the data hierarchically. This involved and showed how the relational rowset can contain hierarchical multi-leg structures and how the relational engine processes them and complex hierarchical decision support queries. It was shown how query data filtering and selection logic followed hierarchical processing principles. 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.

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 hierarchical level.

With these nonlinear hierarchical processing capabilities already in place with the customer's own SQL processor and data, SQLfX® middleware fully leverages these powerful capabilities not currently being utilized. It runs on top of the customer's SQL processor accepting ANSI SQL hierarchical requests and utilizes the hierarchical metadata naturally contained in it. Using this metadata, SQLfX® submits the needed SQL to the customer's SQL processor making it operate hierarchically and producing hierarchically accurate relational rowset data. SQLfX® then transforms this relational result to the proper structured XML automatically because it knows what the output hierarchical structure is. The whole XML integration operation is performed hierarchically and transparently solving all of the current SQL/XML integration problems today.

SQLfX® supports these advanced new SQL/XML capabilities:

1) ANSI SQL standard and mathematically sound (uses no nonstandard SQL or functions)
2) Ease of use (nonprocedural, navigationless, no XML centric syntax)
3) Hierarchically correct results (uses only principled hierarchical processing)
4) Greater efficiency (powerful hierarchical access limiting optimization)
5) Fully interactive operation (can dynamically process XML hierarchically)
6) Conceptual hierarchical processing (can join full hierarchical structures)
7) SQL queries operate naturally across the entire virtual hierarchical structure
8) Enables full nonlinear hierarchical processing and structure transformations

SQLfX® solves the following problems producing no additional customer:

1) Preparation (changing SQL processors)
2) Coding (requiring procedural coding)
3) Training (XML and vendor specific training)
4) Manpower (transparent XML support means no additional coders)
5) Time to market (with none of the above, there is no additional time)
6) Risk (with none of the above, there is significantly reduced risk)
7) Maintenance (no need, XML is supported transparently)

Copyright © 2007 Advanced Data Access Technologies, Inc.

References

[1] J. D. Ullman, A. V. Aho, J. E. Hopcroft; *On Finding Lowest Common Ancestors in Trees*; Annual ACM Symposium on Theory of Computing, 1973.

[2] Michael M David; *ANSI SQL Hierarchical Processing Can Fully Integrate Native XML*; ACM SIGMOD Record; March 2003.

[3] Ya Bing Chen, Tok Wang Ling, and Mong Li Lee; *Automatic Generation of XQuery View Definitions from ORA-SS Views*; School of Computing National University of Singapore, 2004.

[4] Zhuo Chen, Tok Wang Ling, Mengchi Liu, and Gillian Dobbie; *XTree Declaritive XML Quering*, School of Computing National University of Singapore; 2004

[5] Sara Cohen, Jonathan Mamou, Yaron Kanza, Yehoshua Sagiv; XSEarch: *A Semantic Search Engine for XML*; Proceedings of the 29th VLDB Conference, 2003.

[6]; Yunyao Li, Cong Yu, and H. V. Jagadish; *Schema-Free XQuery*; 30th VLDB Conference, Toronto, Canada, 2004.

[7] Michael M David; *Advanced Capabilities of the Outer Join*; ACM SIGMOD Record; March 1992.

[8] Serge Abiteboul and Nicole Bidoit; *Non First Normal Form Relations to Represent Hierarchically Organized Data*; PODS; 1984.

[9] Mark Leven and George Loizou; *Semantics for Null Extended Nested Relations*; ACM Transactions on Database Systems; Sept 1993.

[10] Arijit Sengupta and Melmet Dalkilie; *DSQL – An SQL for Structured Documents*; Department of Accounting and Information Systems Kelley School of Business Indiana University; 2002.

[11] Michael M David; *Using ANSI SQL as a Conceptual Hierarchical Data Modeling and Processing Language for XML*, The Journal of Conceptual Data Modeling, May 2005

[12] Shuohao Zhang and Curtis Dyreson; *Polymorphic XML Restructuring*; WWW 2006, May 23-26; Edinburg, Scotland; 2006

[13] J. D. Ullman; *Principles of Database and Knowledge- Base Systems, Volume II, Chapter 17, The Universal Relation*; Computer Science Press, 1989.

Copyright © 2007 Advanced Data Access Technologies, Inc.