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

SQLfX®'s
ANSI SQL Transparent XML Hierarchical Processor Beta User Guide

SQLfX®
Transparently Turns Your
ANSI SQL Processor
Into a Powerful Enterprise 2.0
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 link nonlinear view structures into a hierarchical data 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.

Contents

16) Multi-type and Multiple Structure Transformations

- Multi-type Structure Transformations
- Multiple Structure Transformations
- Combining Multi-type and Multiple Structure Transformations

17) Global Hierarchical Queries

17.1) Simple Global Filtering
17.2) Complex Global Filtering

A) Powerful Automatic Features

A1) Powerful Hierarchical Access Optimization and Efficiency
A2) Full Ad hoc Interactive Capability
A3) Hierarchical View Capabilities
A4) Duplicated and Replicated Data Re-Normalization
A5) Reuse and Reusability
A6) Transparent Navigationless Processing
A7) Hierarchical Structure Aware Processing
A8) Automatic Distributed Hierarchical Processing

B) Capabilities to be Completed for the Initial SQLfX® Release

B1) Additional FOR XML Options
B2) View-to-view Explicit Transformation
B3) Using XML's Duplicate and Shared Node Capability
B4) Add Real-time EII and Integrate With Batch ETL XML
B5) SQL Update Using Native XML Data
B6) Automatic Generation of Hierarchical Views
B7) XML SQL View Definition Support for ON clause
B8) View.* Support

C) Future SQLfX® Capabilities

C1) Capturing Additional XML Document Information
C2) Data Grouping and Range Control
  C2.1) Hierarchical Data Driven GROUP BY and Summary Functions
  C2.2) Hierarchical Structure Driven Existential and Summary Functions
C3) Nonlinear Recursion Role-up
C4) Parallel Processing Capabilities
C5) Legacy Data Hierarchical Integration
C6) Denormalized Table Support
C7) Union Support
C8) Advanced Drill Down
C9) Occurrence Positional Operator
C10) XML Access Operations
C11) SQL Pass-through Capability
C12) XML and Hierarchical Processing Functions
C13) Keyword Hierarchical Search Function
C14) Support More Data Types

1.0) SQLfX® Introduction Overview

The fundamental belief that relational and hierarchical data and processing can not be integrated seamlessly and fully is fundamentally wrong. SQLfX® (SQL for XML) is a patented breakthrough ANSI SQL driven middleware product that finally solves the complete SQL/XML integration problem. It is also the first and only full nonlinear (multi-leg) hierarchical product for XML processing today. SQLfX® automatically enables your standard SQL processor to transparently access, integrate, and process relational and native XML data at a full hierarchical processing level. This totally seamless operation insures hierarchically accurate and correctly represented XML results. These are capabilities missing from XML processing today and significantly reduce risk and greatly increase ROI for the customer.

Without correct hierarchical processing, the hierarchical XML result would become invalid. This is corrected by processing the data hierarchically which SQLfX® does. Formatting or transforming XML data based on incorrect knowledge of the structure being processed also produces incorrect results. This is avoided by having correct knowledge of the SQL query's dynamically changing hierarchical structure to be automatically formatted for XML output. Intermixing standard non-hierarchical relational processing with hierarchical processing also invalidates the hierarchical results. These are all avoided in SQLfX®. Since SQLfX® with its SQL hierarchical processing is in control. XML Schema metadata is not required but can be utilized for automatic ANSI SQL hierarchical view generation for SQLfX®.

This document takes you through an actual working set of SQLfX® examples. These demonstrate and describe its patented control of inherent hierarchical operations and how they are naturally performed in the underlying customer's ANSI SQL processor. The customer's SQL processor relationship to SQLfX® is shown in Figure 1.0 below. The query execution path from beginning to end consists of hierarchically modeled data input with preprocessing, through hierarchical processing in the customer's SQL processor, to automatic XML structured output in post processing. The processing of relational and XML input data occurs naturally at a full hierarchical level. Batch data access occurs during preprocessing and real-time data access.

SQLfX® Processing Layout

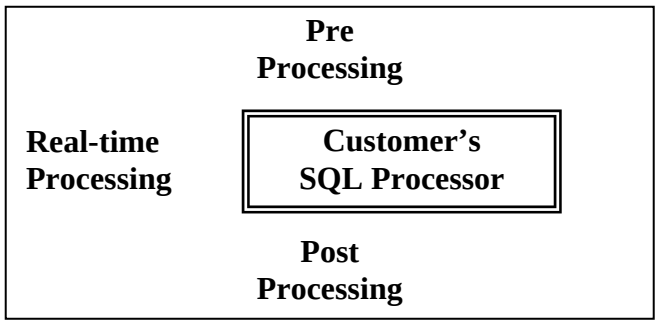


Figure 1.0 SQLfX® relationship to Customer's SQL Processor

The hierarchical processing capabilities are naturally and transparently performed in ANSI SQL and include: data modeling and processing of full multi-leg hierarchical structures; performing hierarchical joins of multi-leg data structures; full support and utilization of multi-leg hierarchical query semantics; basic hierarchical processing such as node promotion and fragment processing; structure transformations; and more. The user does not have to know XML or be aware of its processing. This allows SQL/XML development projects to begin immediately and require no additional: risk; design; development; training; debugging; or maintenance effort, while delivering efficient hierarchical accurate results consistently.

1.1) Who is the *SQLfX®* Customer

SQLfX® is for SQL shops and SQL applications that need to input, process, and/or output native XML. Our customers want standard SQL type capabilities and not unconventional features introduced with XML. They do not want modified SQL syntax and semantics, or procedural operations required for XML support. Total transparent operation is the best solution. They would be interested in new and value added capabilities if introduced from XML transparently. They want an XML solution that could be adapted easily and quickly without disturbing the status quo. They need SQL to continue operating consistently and accurately producing correct and reliable results for critical applications. *SQLfX®* satisfies these requirements and even delivers relational and hierarchical accuracy not found today in the XML industry.

In the XML Industry, it can also be used as an ad hoc nonprocedural nonlinear XQuery supplement supplying these characteristics not easily producible in XQuery. The capability to process entire hierarchical structures as a whole also allows for an unlimited range of query processing not feasible with user procedural navigation. Global, entire structure queries, with complex hierarchical data filtering can be applied to the entire structure, and processed correctly in *SQLfX®*.

1.2) Why the SQLfX® Solution is Better

SQLfX® is better because there is **No**: Database navigation; Basic XML training; Vendor XML training; Procedural coding required; Knowledge of the data structure necessary; Loss of ad hoc ability; Single leg processing limitation; Invalid hierarchical results; and Nonstandard XML centric syntax. SQLfX® solves all of these problem areas transparently. This eliminates: Risk; Produces a shorter time to market; Has a lower cost of ownership; and Significantly increases ROI for the customer.

In addition, SQLfX®'s nonlinear hierarchical processing provides a new Web 2.0 level of advanced multi-leg hierarchical processing capabilities beyond today's linear restricted processing. These new capabilities can also dynamically increase the value of the customer's data by naturally utilizing the semantics in the hierarchical structures being processed. Our discoveries include: hierarchical relational processing, multi-leg LCA queries, linking below root of structure, any-to-any structure transformations.

1.3) How the SQLfX® Beta Examples are Presented

This document's purpose is to demonstrate SQLfX®'s transparent XML hierarchical processing and its correct and accurate nonprocedural XML output. SQLfX®'s XML support is transparently, naturally, and fully integrated into SQL's natural processing. The user does not have to be aware that XML data and hierarchical processing is being performed. This hierarchical processing is transparent because it is actually a subset of relational processing and is naturally performed by ANSI SQL.

Hierarchical query processing has a set of operations that are naturally carried out hierarchically when operating on hierarchically modeled data. These are performed correctly by ANSI SQL's natural operations such as data selection and data filtering operation. These SQL natural operations will be pointed out and their resulting hierarchical XML formatted result will be shown. These capabilities will be shown in a building block fashion starting with a basic hierarchical foundation created by the FROM clause outer join data modeling operation, then by node data selection controlled by the SELECT list, followed by hierarchical data filtering specified by the WHERE clause, and finally the more hierarchically involved operations such as joining and transforming hierarchical structures. Some internal nonlinear processing explanation is given, though not necessary for operation. For a detailed explanation of nonlinear hierarchical processing principles and internal explanation, see our nonlinear hierarchical processing tutorial at http://www.adatinc.com/images/Hierarchical_Structures_and_4GLs.pdf.

1.4) SQLfX® Operational Overview

Figure 1.4.1 below demonstrates SQLfX®'s hierarchical processing using SQL's SELECT, FROM, WHERE syntax and their intuitive hierarchical structured operations: the input data and its hierarchical SQL modeled structure specified by the FROM clause; output data selection specified by the SELECT clause which indicates the hierarchically related nodes to be returned and the order of data items in each data node; and the data filtering specified in the WHERE clause which hierarchically filters the data following its hierarchical structure. These operations used together will hierarchically process the input data and automatically produce a hierarchical structured XML correctly representing the processed results. The FROM clause data modeling can also perform data driven transformations discussed later in Sections 14 and 15. This Beta version automatically displays the data result in structured XML format in an interactive ad hoc fashion. This new nonprocedural XML interactive processing is further enhanced by its nonlinear hierarchical data structure linking processing shown in Figure 1.4.2 in the next section.

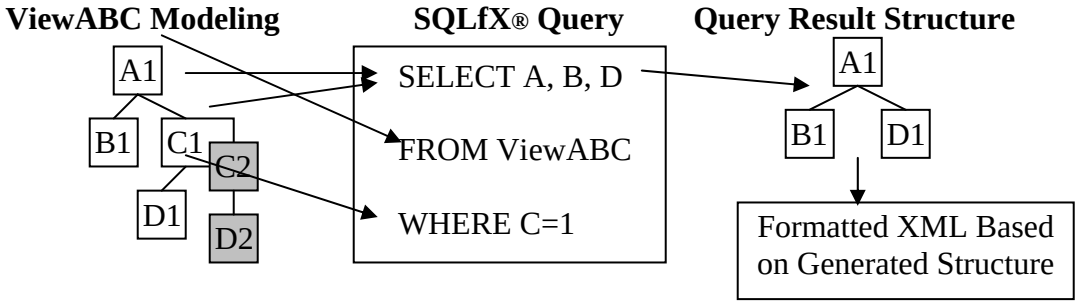


Figure 1.4.1: SQL hierarchical query specification and operation overview

1.4.1 XML Transparency and Hierarchical Conceptual Operation

In Figure 1.4.2 below, you can notice that the SQLfX® user can continue to use the same SQL but visualize the data structures as hierarchical and the operation as being performed hierarchically at a high conceptual level to produce hierarchical structures which will be automatically converted to its hierarchical XML structure unless overridden. The only difference between the current operation and SQLfX® operation is that the hierarchical structures are defined hierarchically using standard ANSI SQL Left Joins described shortly. This allows the structures defined to be visualized hierarchically which are automatically processed in a standard and valid hierarchical way. This is actually performed in the ANSI SQL processor when the structures are defined hierarchically since they are defined using standard SQL. Global structure processing is easily achieved, unlike XQuery which is limited to fewer data selections by its procedural structure navigation and fixed function code (acting as views). SQLfX®'s advanced nonprocedural hierarchical processing operation requires no XML user navigation, it is navigationless.

1.4.2 SQL Hierarchical Structure Joining and Transformation

It turns out that ANSI SQL nonlinear hierarchical processing has the ability to perform full nonlinear hierarchical structure joining and transformation, and at high conceptual fashion. This is naturally achieved by manipulating the hierarchical structure represented in the relational rowset and rearranging them using standard join operations. Linear and full nonlinear structure transformations are possible. In fact, a new breakthrough level of any-to-any structure transformation is possible using standard SQL. These structure transformations covered in more detail in Sections 14, 15, 16.

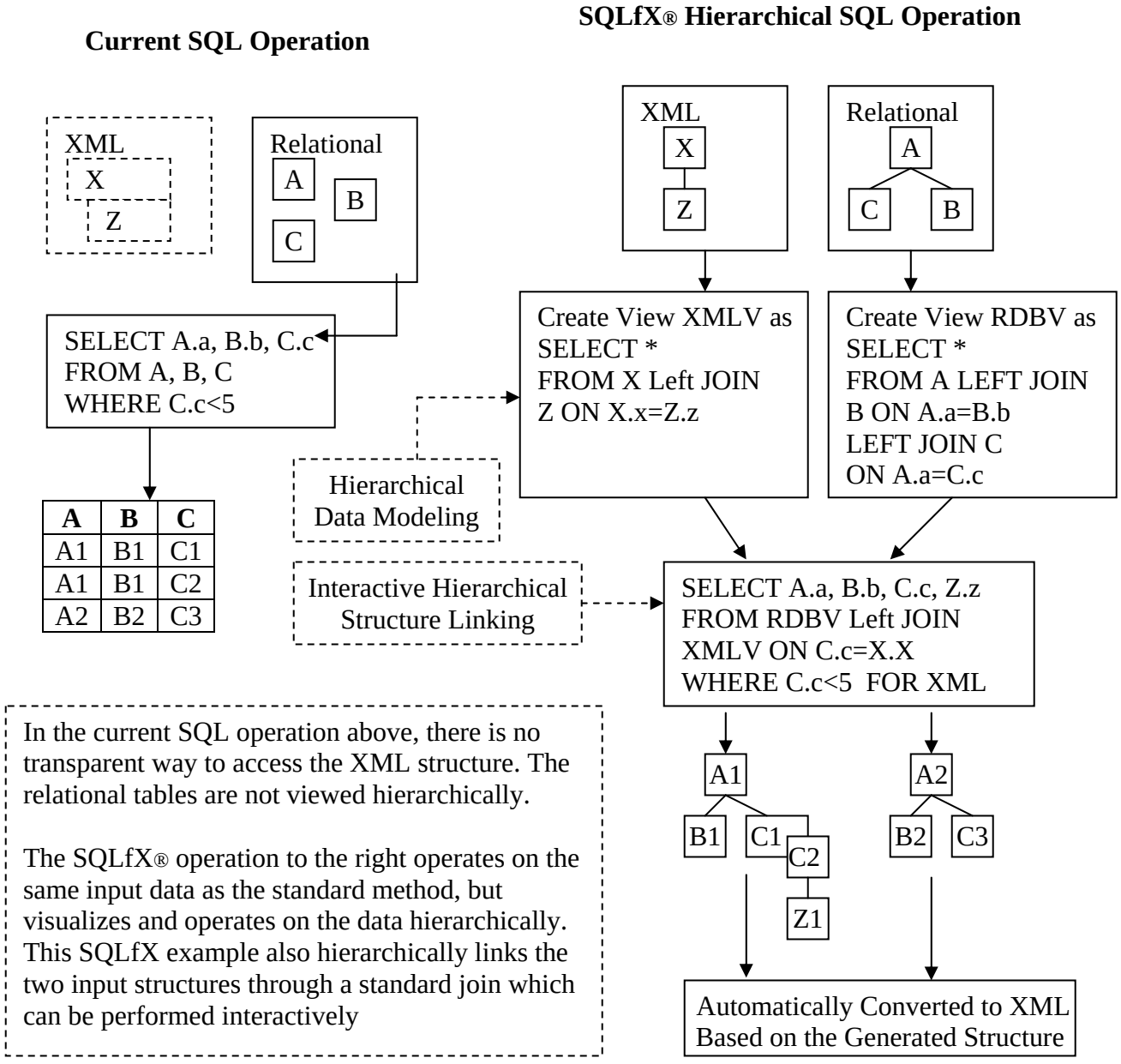


Figure 1.4.2: Comparison of Standard SQL to SQLfX®

1.4.3 Do Not be Mislead by the Examples' Technical Description

SQLfX®'s XML support and hierarchical processing support is transparent. In fact, its global view support with access optimization makes it even more user friendly than standard SQL. The user does not have to be aware that XML and hierarchical processing is being performed. Even though this is true, there are capabilities and complex processing that is occurring transparently. To be fully appreciated and validated, this process needs to be recognized and understood which is why it is described when each example is shown. This may distract from SQLfX® transparency and its apparent ease of use. Please keep this is mind. SQLfX® is externally simple and internally complex which is transparent.

1.5) SQLfX® Beta's Hierarchical Syntax Overview

The SQLfX® Beta uses a reduced instruction set to limit processing to hierarchical operations. It is basically SELECT, FROM, WHERE, ORDER BY operations with Left Outer Joins to specify hierarchical data modeling. Alias, prefix, and views are allowed. These are specified as normal. Other less used operations are not supported at this time and will be added. This limited set of SQL operations is very easy to use nonprocedurally and performs very complex internal hierarchical processing automatically. Except for structure transformations, SQL hierarchical processing does not require knowledge of the hierarchical structure. The user also does not need to know XML. Examples will describe the hierarchical processing occurring internally for those interested in nonlinear processing taking place, but is not necessary for the user to know, but helps to understand the results.

A "FOR XML" clause has been added to supply XML operation information and parameters to change XML hierarchical formatting operation. In our Beta version demonstrated in this document, the user has the option of having the SQL native XML result be formatted using the options of Element, Attribute, and Mixed style specified on the FOR XML clause. The default is Attribute style. There are also Node options for preserving empty nodes, removing empty fields, and renaming or removing the added collection node.

XML data ordering is maintained unless overridden by an ORDER BY clause which supports nonlinear, multi-leg hierarchical structure ordering. XML is processed transparently and is integrated fully into SQL the same as relational data. Relational tables and XML elements are treated exactly the same, as nodes in a hierarchical structure. Attribute, Element and Mixed mode input content is automatically supported.

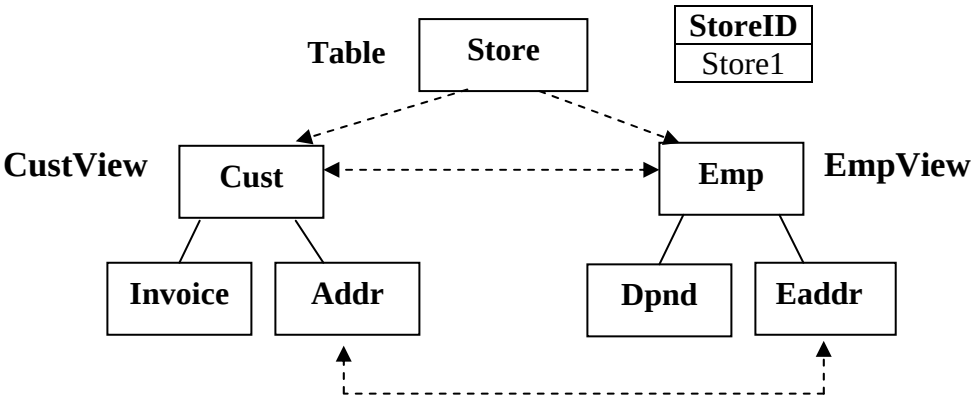
1.6) Example Data, Tables, Relationships, Views, and Structures

Our examples in this document are actual operational examples that can be performed along with the examples in this document. Only the SQL statements need to be entered. The data, tables, and views are already predefined. In most examples, only the key/ID fields are used in order to represent the structure more visibly by reducing clutter of other data fields in the nodes.

The ANSI SQL views used in examples can represent hierarchically structured relational data or hierarchically structured XML data. Both are hierarchically modeled internally which is performed using hierarchical Left Outer Joins that model complex nonlinear hierarchical structures. By limiting processing to hierarchical processing, hierarchical structure and processing principles can be utilized to produce higher levels of processing to increase data value and produce correct hierarchical results and XML structured formatting. A case of less is more.

The examples begin with two hierarchical structure views represented by the Emp and Cust views shown in Figure 1.6 below that can be either relational or XML structures. Relational structures are comprised of hierarchical modeled relational tables while physical XML structures are comprised of XML elements also hierarchically modeled. In hierarchical processing terms, relational tables and XML elements are both nodes of the same hierarchical structure. This allows the same SQL hierarchical data modeling Left Outer Joins to internally map both types of structures. This means the Cust and Emp views in Figure 1.6 can be seamlessly representing a relational or XML structure, and operate exactly the same. This also means that one view structure can be relational while the other is XML producing seamless heterogeneous processing naturally in SQLfX® in the same hierarchical fashion. The dashed lines between the Cust and Emp views and Store table represent relationships that can be used between the views or the Store table.

StoreView Data




```
CREATE VIEW CustView AS
SELECT * FROM Cust
LEFT JOIN Invoice
   ON CustID=InvCustID
LEFT JOIN Addr
  ON CustID=AddrCustID;
```

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

CustView Key Data

CustID	CustStoreID	InvID	InvCustID	AddrID	AddrCustID
Cust01	Store01	Inv01	Cust01	Addr01	Cust01
Cust01	Store01	Inv02	Cust01	Addr01	Cust01
Cust02	Store01	Inv03	Cust02	Addr02	Cust02
Cust02	Store01	Inv03	Cust02	Addr04	Cust02
Cust03	Store01			Addr03	Cust03

EmpView Key Data

EmpID	EmpStoreID	EmpCustID	DpndID	DpndEmpID	EaddrID	EaddrEmpID
Emp01	Store01	Cust01	Dpnd01	Emp01	Addr01	Emp01
Emp02	Store01	Cust03			Addr03	Emp02

Figure 1.6: Data, Tables, Relationships, Views, Nodes, and Structures

The two hierarchical structure views above in Figure 1.6 can be combined into a larger structure in more than one way. One way is under the Store table as shown below in Figure 2.1. Another way is to link the Employee (Emp) structure over the Customer (Cust) structure or the other way around. The structure below relates the CustView and EmpView sub views using the Store table as the common higher level reference to both of them. The relational Result Set shown in some of the examples in this document represents the intermediate hierarchically processed relational data of the underlying customer SQL processor. This is before additional hierarchical processing and XML formatted output is performed by SQLfX®. This is supplied to help demonstrate how ANSI SQL performs nonlinear hierarchical processing. Result set's null fields are represented as blank to help emphasize the embedded hierarchical data structure. The test examples used in this document use the ANSI SQL FirstSQL relational database.

1.7) Data Definition Language Requirements

The data, tables, and views shown in the examples in this document have been predefined and do not need to be created. They are stored in the library identified in the installation instructions. The invoking SQL for each example has also been stored their and can be retrieved for easy submittal using its SQL number as in SQLnn.nn, Views are stored under their view name preceded by Create View name. Tables are stored under their name preceded by Create Table name. Data is store under their table name preceded by Insert name.

Create views are performed at the SQLfX level, while Create Tables and Insert data is performed at the underlying customer SQL processor level. The definitions for the underlying database must be in place before the Create Views can be performed at the SQLfX level. Tables must be defined with unique keys, this is necessary to accurately remove duplicates for XML results.

Tables defined in the underlying SQL processor must also be identified to SQLfX at the SQLfX level with a simple Create Table name command with nothing else defined. The column and other information are automatically obtained from the underlying processor level. Unique primary keys are necessary for relational tables so that duplicate data can be removed for the hierarchical results. This means that primary keys must be defined for all tables.

The SQLfX® GUI is used to execute its query and view definition statements and access the SQL library for its SQL query and definition statements. The FirstSQL GUI is used to access and perform the underlying SQL processor table definitions and to access the FirstSQL database directly. Information on their location and operation is described in the SQLfX® installation instructions.

1.8) How is Database Navigationless XML Processing Possible

The “M” in XML stands for Markup and that was what XML was designed to handle. The additional use of XML for database data was an afterthought. Database data use is more fixed than Markup data. Markup data requires user navigation to get the full flexible use out of Markup data. Database data being more fixed with a more specific use does not need to be navigated by the user; it can be accessed transparently with no user navigation required, this is known as navigationless access. Unfortunately, this difference in use has not yet been recognized and utilized. Even more of a concern is that Database data processed as Markup data can produce incorrect results. Our SQLfX® product does make this important distinction and limits its processing to database hierarchical processing allowing it to operate navigationlessly. This also allows SQLfX® to perform many advanced full hierarchical processing capabilities not possible with user navigation.

2) Hierarchical Attribute Foundation and Principles Using FROM Clause

The FROM clause's main job in standard SQL is to define the input data sources. With SQL hierarchical processing, the FROM clause also defines the hierarchical structures of the input data sources which in turn specifies the operational semantics.

2.1) ANSI SQL Hierarchical Data Structure Modeling

The CustView and EmpView in Figure 1.6 above demonstrate hierarchical data modeling using the SQL-92 Left Outer Join which is performing a precise basic hierarchical operation. Its left data argument side is preserved if there is no matching right side data argument occurrence, while the right side is not preserved if there is no matching left data occurrence for it. Using this basic hierarchical capability and semantics, any hierarchical structure can be modeled and processed hierarchically by a relational engine when performing the hierarchical semantics associated with the SQL hierarchical modeling syntax. The hierarchical views in figure 1.6 are created by modeling them going down the structure and shows how sibling legs are formed when ON clauses specify the join criteria that links into a an existing leg. ON clauses can also specify a path going up the path connected by AND logical operators. In this case the lowest level node in the upper level structure is the link point Using OR logical operators will cause network structure errors which will be automatically detected. Multiple conditions can be tested in a single node connected by AND or OR logical operators.

This SQL hierarchical Left Outer Join will be generated automatically from existing metadata sources such as XML schemas in its commercial release. The data modeling can be easily specified by hand by thinking of the "LEFT JOIN" as "Over" and the "ON" as "Linked By" since this is literally describing what is happening in a top to bottom hierarchical order of the nodes. This is shown in Figure 2.1.1 below. These nodes can be specified depth or width first as they are built top to bottom. There is no semantic difference.

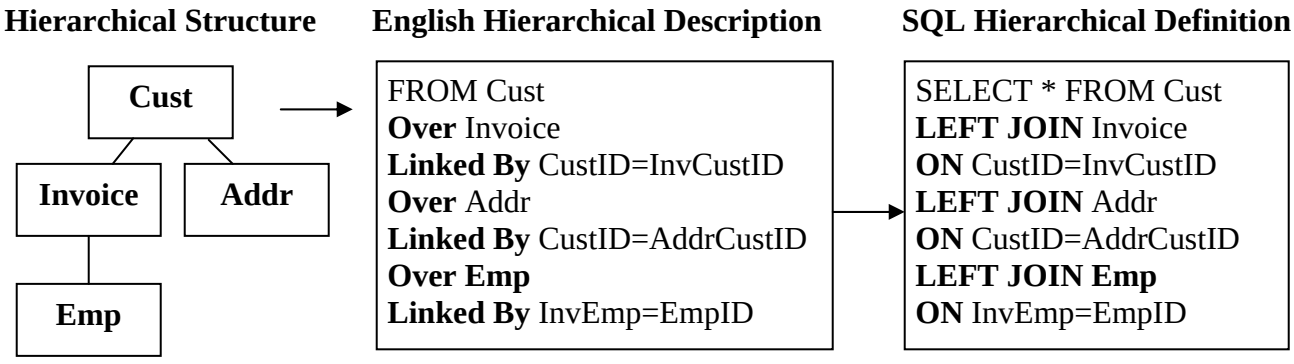


Figure 2.1.1: SQL Hierarchical Data Modeling is easy with visual thinking

In the same way that the hierarchical data modeling was performed a node at time using Left Outer Joins in Figure 1.6 above, hierarchical structures defined in SQL views can also be joined by Left Outer Joins deriving a hierarchical superstructure. The left structure is joined over the right structure linked by the ON clause join criteria at a high conceptual level as demonstrated in Figure 1.6 above. The ON clause takes on added importance by unambiguously specifying the link points in each structure being hierarchically joined.

The ON clause was introduced in SQL-92 to replace the WHERE clause's join use for a more precise join control geared more for local control at the join point. Notice that the EmpView has an ON clause data filter to remove Dependents that have no medical coverage (see "Dpnd02" in the Hierarchical Data Tree in Figure 2.1.2 below). This is an ANSI SQL hierarchical path filter that only affects the hierarchical path portion it is used on in the structure (similar to XPath data filtering). This is why only part of the row or leg is set to null instead of removing the entire row. The WHERE clause is more global in scope and affects the entire structure. This also demonstrates that the ON clause operates during structure creation while the WHERE clause operates after structure creation (at least logically). This is a huge difference between the WHERE and ON operation not generally realized and enables much of SQLfX®'s hierarchical capabilities.

The order that sibling legs are specified with ON clauses is assumed to be their physical order in the input structure and is the default output structure if specified for output. This can be seen in Figure 2.1.2 and 2.1.3 below in Figure 2.1.3 where the SQL structure definition demonstrates that the legs can be extended in any order once they are established with their first node. Other than physical input and output ordering of sibling legs, there is no semantic difference in their sibling order and will not affect the results. The order of items in the SELECT list does control the order of fields in each node output, but do not have to be specified contiguously for each node. They can be interspersed with fields from other nodes.

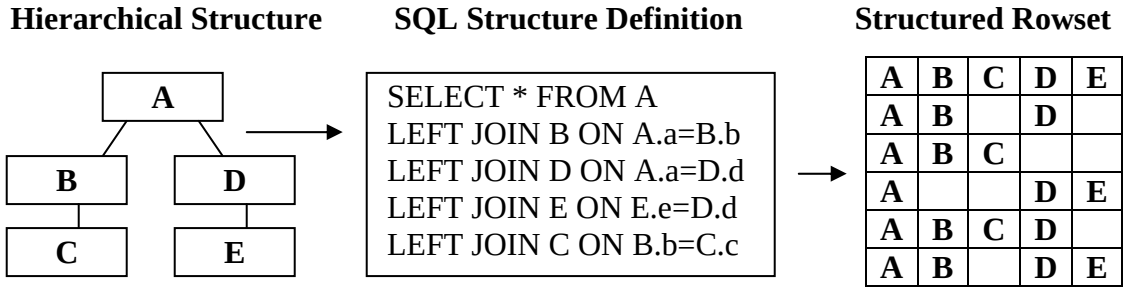


Figure 2.1.3 Hierarchical to Relational via Left Outer Join Mapping

2.2) From Hierarchical Data Modeling to Structured XML Processing

Figure 2.1.3 above demonstrates a number of the most basic hierarchical processing attributes produced in ANSI SQL and utilized by SQLfX®. These attributes are hierarchical data preservation from the left join and variable length leg creation with nulls inserted in the rowset to keep them aligned properly. This is shown in the structured rowset in figure 2.1.3 above where the two sibling legs BC and DE are independently represented at different variable lengths. The shortened null terminated legs show that hierarchical preservation is also working; otherwise they would be totally missing. These attributes do not produce hierarchical operations directly, but are basic hierarchical characteristics that are required for higher level hierarchical processing operations to occur which are covered shortly.

Our first real SQL example (SQL 2.1) in Figure 2.1.2 below also demonstrates these basic hierarchical processing attributes produced in ANSI SQL. For this reason, this example does not filter the data with a WHERE clause and does not exclude any columns with the SELECT clause to limit higher level hierarchical operations in order to emphasize these fundamental hierarchical characteristics. This means the entire hierarchical node structure is represented in the rowset in Figure 2.1.2 and its formatted XML result shown below in Figure 2.2.1. This full structure example and its hierarchical data tree are useful to refer to for verifying the result of other examples using these structures.

The customer's SQL processor will generate the relational rowset in Figure 2.1.2 below. The SQL hierarchical Left Outer Join is driving the processing so that full nonlinear hierarchical processing has produced the result. In other words, the full hierarchical result is represented in the relational result. The process of creating a hierarchical data structure from the relational result, as shown in Figure 2.1.2 below, is possible when the hierarchical structure represented in the relational result set is known. This is how SQLfX®'s patented technology enables it to dynamically derive the XML result structure.

2.3) Hierarchical SQL View Usage

Hierarchical structure definitions using multiple table left outer joins can be stored in a standard SQL view and used as a hierarchically structured data object which can be used in the same way a table can. These structured views can be joined to create larger hierarchical data structures such as StoreView in Figure 2.1.2. XML views described later in Section 12 are defined as an XML view which is a physical view describing the entire XML document structure. This XML data definition is still used as a standard SQL view and can be combined with other XML or relational data views to create heterogeneous views.

2.3.1 Hierarchical Global Views

Hierarchical queries are hierarchically optimized so that they can be used to process global (large) structure views which can be variably invoked using different SELECT lists with generate different queries and XML results without any global view overhead. There is no overhead for global views because each view is dynamically optimized for its specific structure usage. These global structure views can also be used in their entirety to perform a global query operation by specifying a SELECT * (Select All) operation when invoked. These global queries are demonstrated in Section 17 where the entire hierarchical view is hierarchically filtered. All examples in this document use global views.

2.3.2) Structure Independence

Because of the nonprocedural and navigationless access used by SQLfX®, the desired data to be output does not need to be specified in any specific order, or navigated. This means that the structure in an input hierarchical view does not need to be known to be accessed. This also means that the same named structure view and data content at different user locations can have a different structure and still be processed the same transparently.

2.3.3) Structure Aware Capabilities

SQLfX® gets its real power from being able to know the structure being processed at all times. This can dynamically change during processing by the join processing taking place. This structure-aware ability allows for hierarchical optimization to be applied, correct relational/XML mapping, and it controls the produced structured of the output hierarchically formatted XML. This assures that the automatically processed and produced hierarchical result is accurate.

2.4) Mapping Between XML and Relational Structures

Generally, relational tables and XML elements are treated as nodes. Relational columns, XML element strings and XML Element attributes are treated as fields in data nodes. Internal mapping between hierarchical node structures and relational rowsets has a straightforward exact mapping. Mapping from Relational to XML also has exact mapping with a choice of output XML formats. Mapping from XML to Relational structures does not always have an exact mapping. This is because of the semistructure

capability of XML which can allow for different mapping solutions which can produce difference results. Determining a hierarchical node structure in XML can also be difficult because of unnecessary nesting used by the element formatting mode. We do our best to match the Input hierarchical XML against its XML hierarchical structure definition in its SQLfX® View.

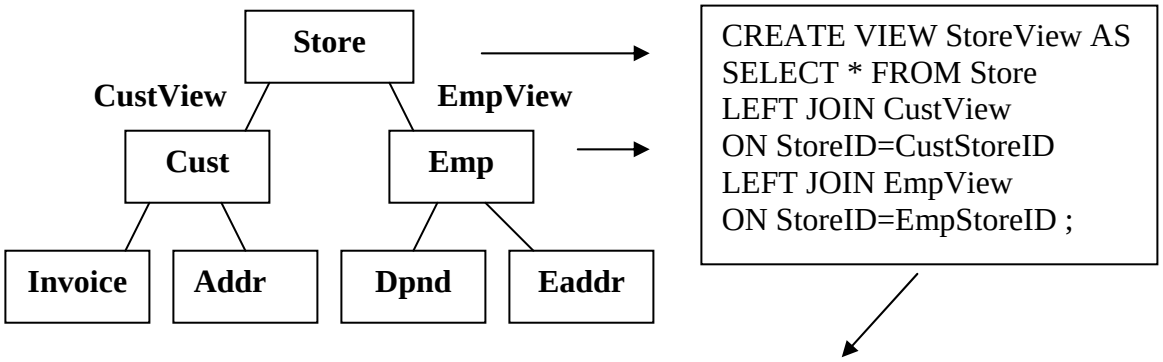
2.4.1) XML to Relational Mapping

Element, Attribute, or Mixed mode input is accepted. The hierarchical structure is automatically determined based on the input. This is demonstrated in Section 12.

2.4.2) Relational to XML Mapping

Element, Attribute, and Mixed mode output can be specified on the FOR XML clause. Attribute format is the default and if Mixed mode is specified, its string data output is determined by its string data input. These are demonstrated in Section 12.

Store View Structure **Store View**



SQL 2.1: **SELECT StoreID, CustID, InvID, AddrID, EmpID, DpndID, EaddrID FROM StoreView**

Intermediate Relational Result Set

StoreID	CustID	InvID	AddrID	EmpID	DpndID	EaddrID
Store01	Cust01	Inv01	Addr01	Emp01	Dpnd01	Addr01
Store01	Cust01	Inv01	Addr01	Emp02		Addr03
Store01	Cust01	Inv02	Addr01	Emp01	Dpnd01	Addr01
Store01	Cust01	Inv02	Addr01	Emp02		Addr03
Store01	Cust02	Inv03	Addr02	Emp01	Dpnd01	Addr01
Store01	Cust02	Inv03	Addr02	Emp02		Addr03
Store01	Cust02	Inv03	Addr04	Emp01	Dpnd01	Addr01
Store01	Cust02	Inv03	Addr04	Emp02		Addr03
Store01	Cust03		Addr03	Emp01	Dpnd01	Addr01
Store01	Cust03		Addr03	Emp02		Addr03

Hierarchical Data Tree

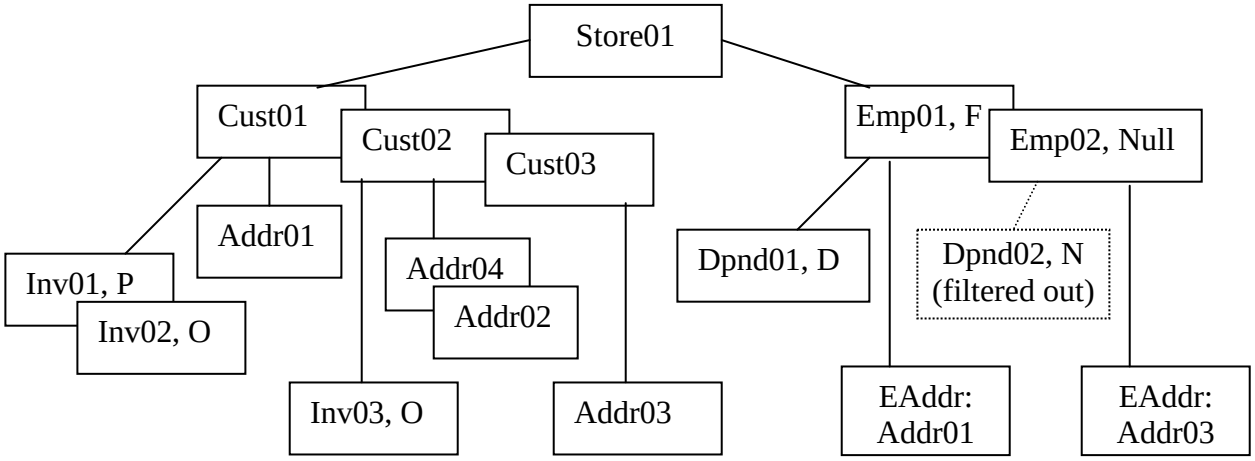


Figure 2.1.2: StoreView and data (Print out for future reference)

Lets display the store view's Key Fields in both of its two XML formats, this will produce all nodes:

SQL 2.1: **SELECT StoreID, CustID, InvID, AddrID, EmpID, DpndID, EaddrID FROM StoreView**

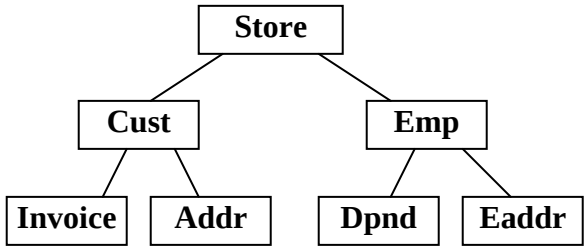
FOR XML Element

```
<store>
 <storeid>Store01</storeid>
 <cust>
  <custid>Cust01</custid>
  <invoice>
   <invid>Inv01</invid>
  </invoice>
  <invoice>
   <invid>Inv02</invid>
  </invoice>
  <addr>
   <addrid>Addr01</addrid>
  </addr>
 </cust>
 <cust>
  <custid>Cust02</custid>
  <invoice>
   <invid>Inv03</invid>
  </invoice>
  <addr>
   <addrid>Addr02</addrid>
  </addr>
  <addr>
   <addrid>Addr04</addrid>
  </addr>
 </cust>
 <cust>
  <custid>Cust03</custid>
  <addr>
   <addrid>Addr03</addrid>
  </addr>
 </cust>
 <emp>
  <empid>Emp01</empid>
  <dpnd>
   <dpndid>Dpnd01</dpndid>
  </dpnd>
  <eaddr>
   <eaddrid>Addr01</eaddrid>
  </eaddr>
 </emp>
 <emp>
  <empid>Emp02</empid>
  <eaddr>
   <eaddrid>Addr03</eaddrid>
  </eaddr>
 </emp>
</store>
```

FOR XML Attribute

```
<store storeid="Store01">
 <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>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <eaddr eaddrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
  <eaddr eaddrid="Addr03"/>
 </emp>
</store>
```

StoreView



Hierarchical Structure Processing 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)

Figure 2.2.1: StoreView key fields returned as XML in either of two standard formats

Lets display the store view's complete data structure in default attribute format

SQL 2.2: **SELECT * FROM StoreView**

```
<store storeid="Store01">
 <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>
 <cust custid="Cust02" custstoreid="Store01">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O"/>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"/>
 </cust>
 <cust custid="Cust03" custstoreid="Store01">
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV"/>
 </cust>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
  <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
  <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
 </emp>
</store>
```

Figure 2.2.2: Complete StoreView data in XML attribute format (Print out for future reference)

Lets display a partial full structure portion of the Store View data structure.

SQL 2.3: **SELECT * FROM storeview WHERE custid='Cust01' AND Empid='Emp01' FOR XML element**

```
<store>
 <storeid>Store01</storeid>
 <cust>
  <custid>Cust01</custid>
  <custstoreid>Store01</custstoreid>
  <invoice>
   <invid>Inv01</invid>
   <invcustid>Cust01</invcustid>
   <invstatus>P</invstatus>
  </invoice>
  <invoice>
   <invid>Inv02</invid>
   <invcustid>Cust01</invcustid>
   <invstatus>O</invstatus>
  </invoice>
  <addr>
   <addrid>Addr01</addrid>
   <addrcustid>Cust01</addrcustid>
   <addrstate>CA</addrstate>
  </addr>
 </cust>
```

Continued:

```
<emp>
  <empid>Emp01</empid>
 <empstoreid>Store01</empstoreid>
  <empcustid>Cust01</empcustid>
  <empstatus>F</empstatus>
  <dpnd>
   <dpndid>Dpnd01</dpndid>
   <dpndempid>Emp01</dpndempid>
   <dpndcode>D</dpndcode>
  </dpnd>
  <eaddr>
   <eaddrid>Addr01</eaddrid>
   <eaddrcustid>Cust01</eaddrcustid>
   <eaddrstate>CA</eaddrstate>
  </eaddr>
 </emp>
</store>
```

Figure 2.2.3: StoreView and data (Print out for future reference)

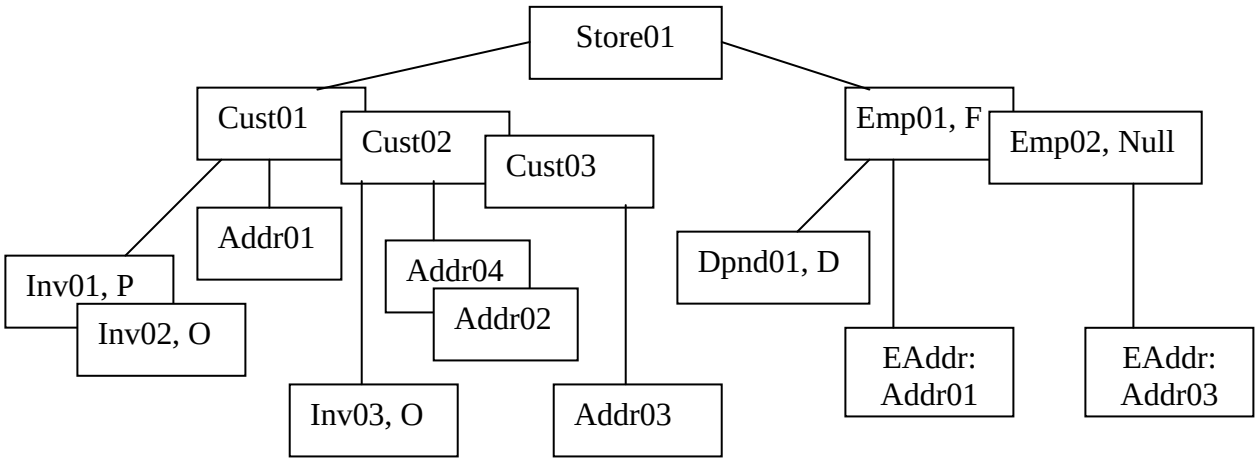
The above SQL 2.1 statement's XML result in Figure 2.2.1 reflects the Relational Result Set in Figure 2.1.2. Blank boxes are null values. SQLfX® knows how to interpret the hierarchically processed relational results to enable it to automatically produce the correct hierarchical XML formatted results. The SQLfX® XML result shown in Figures 2.2.1, 2.2.2 and 2.2.3, proves the following hierarchical attributes in the SQL and XML processing have been empirically demonstrated:

Basic Hierarchical Attributes Recap Based on Figure 2.2 Correct SQL and XML Results:

SQL Hierarchical Data Modeling	The SQL Left Outer Join performs hierarchical data modeling of full multi-leg structures as shown by results.
Data Preserved Hierarchically With Multiple Variable Length Legs	Each leg's data occurrences are preserved until each path ends or a missing node data occurrence is encountered.
Hierarchical Legs in Rowset Aligned Hierarchically	Each leg remains aligned in rowset, even if they dynamically vary in length. Null values act as placeholders.
ON Clause Path Filtering Used to Support Data Mode Rules	Supports XPath type data filtering, "Dpnd02" is filtered out without removing the parent or affecting other legs.
ANSI SQL Views and Embedded Views Hierarchical Expansion	Data modeling views have expanded correctly into a unified hierarchical view to produce the correct hierarchical results.
XML Structured Output Built Automatically and Selectable	SQLfX® technology correctly determined the final hierarchical structure to map relational rowset to XML.
Sibling Leg Order	Sibling Legs are added left to right as joined. Sibling leg order have no hierarchical significance.
All Capabilities Work Together	As proven by valid XML generated in examples

Table 2: Proven hierarchical attributes for SQL 2.1 example

The following Hierarchical Data Tree below is a copy of Figure 2.1.2. It is repeated here because it can be used to compare the SQLfX® results in Section 3 to fully understand each query's full meaning and verifying its XML result to the input data and structure. Only key fields are usually used to keep examples and hierarchical output simple to visualize in its structure.



Copy of Figure 2.1.2: StoreView and data used in examples to follow:

3) Node Selection With SQL SELECT List Operation

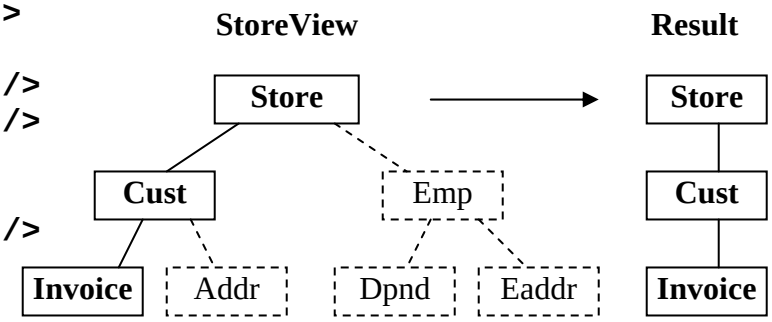
Output node selection is controlled by which nodes are SELECTed on the invoking SQL statement. A node is selected for output if at least one field is SELECTed for output from it. This can be overridden by a FOR XML option (see Section 3.7) specified at the end of the query. Node promotion, node collection and fragment operation are also shown in this section, and other FOR XML options will be demonstrated.

3.1) Selecting a Single Linear Leg

One of the main purposes of SQLfX® is that the user does not need to know the hierarchical structure, so accessing data along a single leg could just be a coincidence. Even more of a coincidence is selecting the data in node order. The linear query directly below is actually specified out of node order (CustID before the root StoreID) in the SELECT list but, it still operates correctly returning the nodes in correct node order. This is because user navigation or knowledge of the structure is not necessary with SQLfX®. Notice in the following XML result that Cust03 was hierarchically preserved with no lower level Invoice child. Examining the hierarchical data tree in Figure 2.1.2, you can see this in the correct SQL 3.1 result.

SQL 3.1: **SELECT CustID, StoreID, InvID FROM StoreView**

```
<root>
 <store storeid="Store01">
  <cust custid="Cust01">
   <invoice invid="Inv01"/>
   <invoice invid="Inv02"/>
  </cust>
  <cust custid="Cust02">
   <invoice invid="Inv03"/>
  </cust>
  <cust custid="Cust03">
  </cust>
 </store>
</root>
```

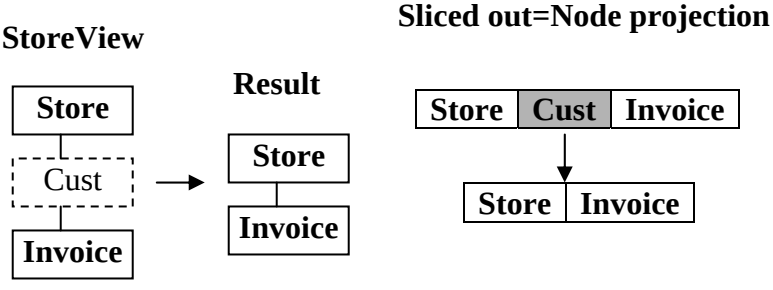


3.2) Node Promotion With Single Leg

This SQL 3.2 query below is similar to the last query, except the Cust node is not SELECTed. The user does not include any fields from the Cust node in the SELECT list. Normal hierarchical operation is to simply skip over the Cust node and keep accessing other fields down the path. This is the same operation with the relational processor which also slices out unSELECTed columns which can equate to nodes.

SQL 3.2: **SELECT StoreID, InvID FROM StoreView**

```
<root>
 <store storeid="Store01">
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <invoice invid="Inv03"/>
 </store>
<root>
```

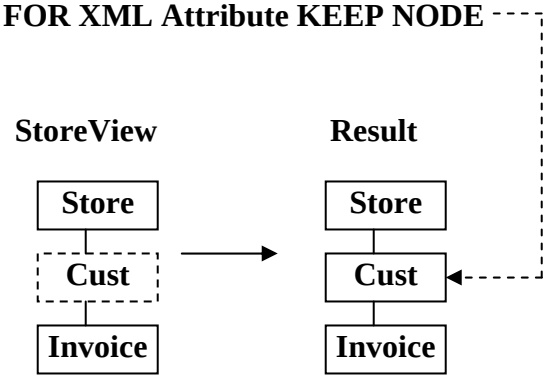


3.2.1) Overriding Node Promotion

This is the same query as the previous one except the user does not want intervening empty unselected nodes such as Cust in this query to be sliced out of the result. This can be specified using the KEEP NODE option in the FOR XML clause at the end of the Query as in SQL 3.2.1. This is useful for XML navigation purposes where the same navigation logic can be used as that defined for the original structure.

SQL 3.2.1: **SELECT StoreID, InvID FROM StoreView FOR XML Attribute KEEP NODE**

```
<store storeid="Store01">
  <cust>
   <invoice invid="Inv01"/>
   <invoice invid="Inv02"/>
  </cust>
  <cust>
   <invoice invid="Inv03"/>
  </cust>
 </store>
```



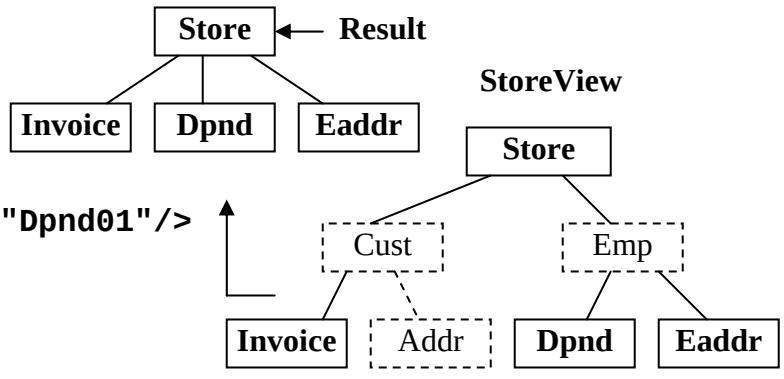
3.3) Node Collection With Multi-legs

As mention above, the user does not have to know the structure or perform navigation. So the processing and result may involve multiple legs. This does not present a problem for the user, since there are no special requirements for specifying multiple legs on the SELECT request. This is shown in the following request which SELECTs fields from Dpnd and Invoice nodes which are located on separate legs. As an additional test, we will also select two fields from the Dpnd node and reference DpndCode first which is out of node order and separate from its other Dpnd node field DpndID. This demonstrates that there are no order requirements at all for users to worry about.

To make this query more interesting, we have excluded SELECT references for the intervening Cust and Emp nodes causing node promotion on both legs to occur in SQL 3.3. This causes the Store node to collect the node promotion from both legs. This Node Collection processing is demonstrated with the following generated XML. It also shows the additional capability of changing the default Collection node of "root" to "collection" using the FOR XML option of "UNDER" to specify a different collection name.

SQL 3.3: **SELECT DpndCode, StoreID, InvID, DpndId, EaddrID FROM StoreView FOR XML attribute UNDER collection**

```
<collection>
 <store storeid="Store01">
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <invoice invid="Inv03"/>
  <dpnd dpndcode="D" dpndid="Dpnd01"/>
  <eaddrid="Addr01"/>
  <eaddrid="Addr03"/>
 </store>
</collection>
```

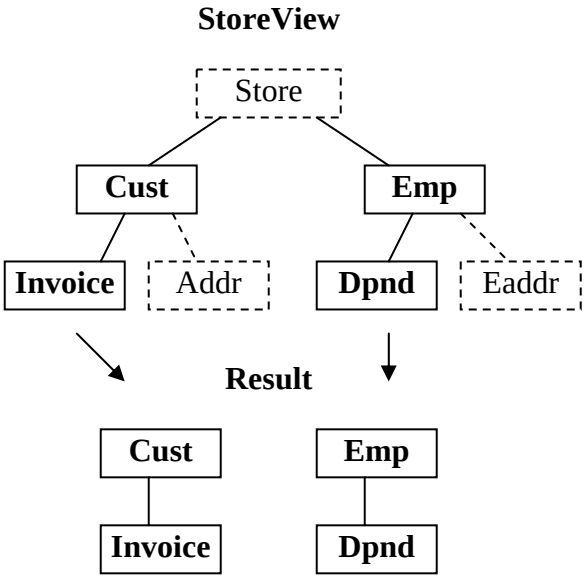


3.4) Selecting Structure Fragments

Structure fragments are similar to node promotion in that they are partial node structures isolated by the SELECT operation that also excludes the original Root node making separate different fragments possible. FOR XML without UNDER is used to avoid specifying a collection node. As you can see in the generated XML, there are multiple occurrences of the Cust and Emp fragments returned.

SQL 3.4: **SELECT CustID, InvID, EmpID, DpndID, EmpStatus FROM StoreView FOR XML Attribute**

```
<cust custid="Cust01">
 <invoice invid="Inv01"/>
 <invoice invid="Inv02"/>
</cust>
<cust custid="Cust02">
 <invoice invid="Inv03"/>
</cust>
<cust custid="Cust03">
</cust>
<emp empid="Emp01" empstatus="F">
 <dpnd dpndid="Dpnd01"/>
</emp>
<emp empid="Emp02" empstatus="">
</emp>
```



3.5) Removal of Null Values in XML Result

By adding Skip Null to the FOR XML of the previous SQL example SQL 3.4 as done below in SQL 3.5, then you can see that the null EmpStatus Attribute value of the Emp02 node has been removed.

SQL 3.5: **SELECT CustID, InvID, EmpID, DpndID, EmpStatus FROM StoreView FOR XML Attribute SKIP NULL**

```
<cust custid="Cust01">
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
 </cust>
 <cust custid="Cust02">
  <invoice invid="Inv03"/>
 </cust>
 <cust custid="Cust03">
 </cust>
 <emp empid="Emp01" empstatus="F">
  <dpnd dpndid="Dpnd01"/>
 </emp>
 <emp empid="Emp02">
 </emp>
```

empstatus="" is removed

3.6) Node Field Order

The SQLfX® Beta displays the result in XML in the hierarchical structure processed. This means the node order is fixed into the structure of the hierarchical result. But the order of the fields for each node can be specified. This is conveyed by the relative order the fields are specified. This means they can still be intermixed with other nodes to be user friendly. If all fields are listed with a SELECT * then the fields are listed in the order they are defined in their defining DDL.

3.6.1) Controlling Node Field Default Order

We will perform a SELECT * (All) in SQL 3.6.1 below to display all the fields in default order which is the order the fields were defined in. This result can be used to compare to the following query's different order explicitly specified.

SQL 3.6.1: **SELECT * FROM EmpView**

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
  <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
  <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
 </emp>
</root>
```

3.6.2) Node Field User Specified Order

The following SQL 3.6.2 query displays the fields for the Emp and Eaddr nodes. The desired order is specified, but intermixed between the two nodes being displayed. The fields are still listed in the order specified for each node.

SQL 3.6.2: **SELECT EmpCustID, EmpStatus, EaddrState, EaddrCustID, EmpStoreID, EmpID, EaddrID FROM EmpView**

```
<root>
 <emp empcustid="Cust01" empstatus="F" empstoreid="Store01" empid="Emp01">
  <eaddr eaddrstate="CA" eaddrcustid="Cust01" eaddrid="Addr01"/>
 </emp>
 <emp empcustid="Cust03" empstatus="" empstoreid="Store01" empid="Emp02">
  <eaddr eaddrstate="NV" eaddrcustid="Cust03" eaddrid="Addr03"/>
 </emp>
</root>
```


3.7) FOR XML Syntax and Operation

The FOR XML clause is added to the end of the SQLfX® Query to specify XML options and capabilities that can alter the XML processing of the query. The current usage of the FOR XML syntax is currently a little confusing, but it will be cleaned up and new capabilities will be added to it for the first release of SQLfX®.

Here is the current FOR XML clause format and options:

FOR XML [Format] {ATTRIBUTE | ELEMENT | MIXED}
[UNDER Nodename] [SKIP NULL] [KEEP NODE[

Here are the meanings and use of the FOR XML options:

ATTRIBUTE | ELEMENT | MIXED: Is the choice of Output XML format type desired.

UNDER Nodename: Nodename specifies the collection node name that surrounds XML results.

SKIP NULL: Will cause null data values not to be specified in output nodes.

KEEP NODE: Will keep unselected intervening nodes in output structure.

FOR XML Operation:

If the FOR XML clause is specified, the output format type of Attribute, Element or Mixed must be specified which can be optionally followed by any of the other keyword options, which if specified must be in order indicated above. If the FOR XML clause is specified and the UNDER Nodename keyword option is not specified, no Collection node is added.

If FOR XML is not supplied, its default options in effect are:

FOR XML ATTRIBUTE UNDER root

This means that Attribute XML output format mode is used by default, and a collection node named "root" is automatically added to the XML formatted output.

Recap of Node Selection Operation

Select Clause Controls Node Promotion	Unselected nodes are normally sliced out of structure, but the dependent segments are not lost.
Select Clause Controls Node Collection	Due to node promotion, multiple legs can be collected under next node which is the same node type.
Select Clause Controls Fragment Generation	By not Selecting the root node, fragments are created and can be optionally left being unconnected.
FOR XML Can Control Node Promotion	Some times node promotion is not desired between existing nodes because of navigation concerns and this option includes the unselect node as an empty node.
FOR XML Can Specify, Remove, or Change Container Root Node.	Remove container node needed for fragments. Change container name to another.
FOR XML Skip NULL Specifies that NULL Values Are to be Removed	Remove null values and empty fields.
SELECT List Items Can be Placed in Any Order.	User does not need to know data structure. There is no output logic or navigation to specify.
Order of Data Items in Node is Controllable.	The order of node data items is their SELECT list relative order even when they are intermixed with other nodes. If SELECT * is used, their physical defined order is used.

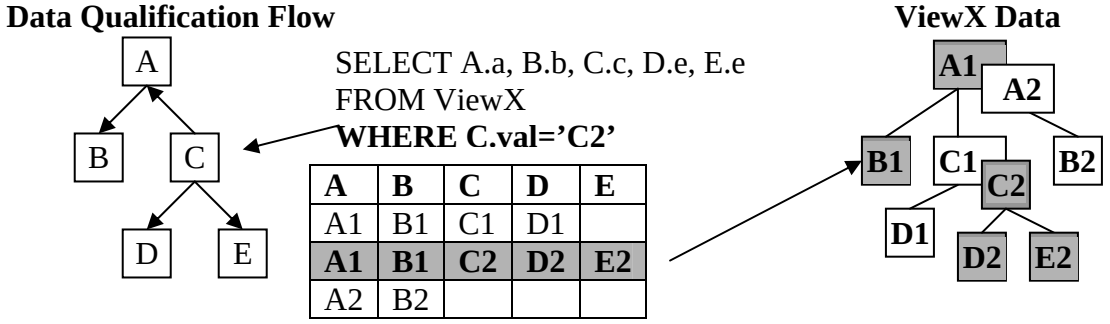
Table 3: Node Selection Operations

4) Multi-Path Hierarchical Data Filtering Using WHERE Clause

The SQL WHERE clause filters the entire structure's data by excluding it or including it based on your view of the operation. If the WHERE clause is not specified, all the data is included which means when specified it must be excluding data. On the other hand, when the WHERE clause is used, it is specified as if you are specifying what data to include. It is easier to comprehend the positive so we will describe the WHERE clause as specifying what data to include. This also is not that easy, simply specifying a given value to include also qualifies all other related data occurrences for preserving. Nonlinear multi-leg qualification can become complex to understand, but is extremely powerful and being performed automatically, the user does not need to be concerned with how it is performed, since it is performed automatically. It is also naturally intuitive for the coder. We will describe it in the examples anyway, so you can understand the power of it.

The ON clause is not covered here; it is a linear single leg qualification that is used during structure creation where it can control only the path it is on. A comparison of ON clause and WHERE clause data filtering is presented later in Section 10. The WHERE clause qualification (data filtering) is logically applied after the entire structure has been created and can affect the entire structure such that qualification based on one leg can affect data qualification on another leg as mentioned above. This full nonlinear hierarchical qualification is not designed or made up. It is standard processing for nonlinear hierarchical processing based on naturally occurring hierarchical principles. These principles have been utilized as far back as the original hierarchical databases. And now with relational databases automatically performing them naturally when the data is modeled hierarchically proves their inherent technology out. XML database products today are lacking this level of principled hierarchical processing.

WHERE clause data filtering affects the entire structure as a whole. This is a new capability for the XML industry. It also follows standard nonlinear hierarchical principles. Figure 4.0 is quick overview on how it works. You can notice how the rowset and its hierarchical view are related. The WHERE clause points to a node data field (located in the relational rowset) to base filtering on. If this node is selected for output, all of its associated node occurrences are also qualified, up, down, and around as shown in Figure 4.0. This happens automatically because the entire row is qualified. Notice how the unqualified row is not output. The darker cells are qualified and output.



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

Figure 4.0: WHERE clause data qualification flow

The following Hierarchical Data Tree below in Figure 4.1 is repeated here because it can be used to compare the SQLfX® results in Section 4 to fully understand each query's full meaning and verify its XML result to the input data and structure.

Hierarchical Data Tree

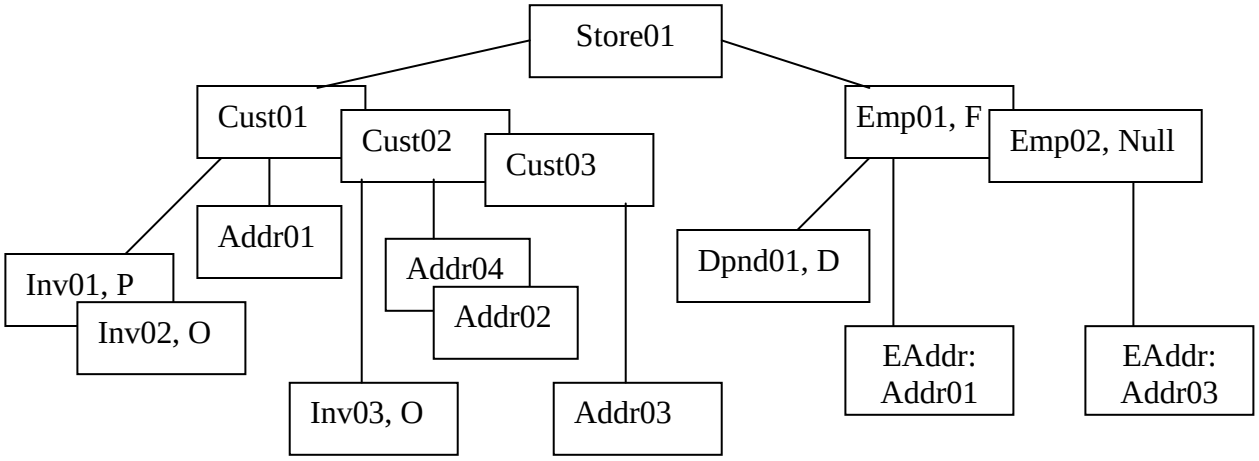


Figure 4.1: StoreView and data used in examples in Section 4 to follow:

4.1) Single Leg Linear Data Qualification

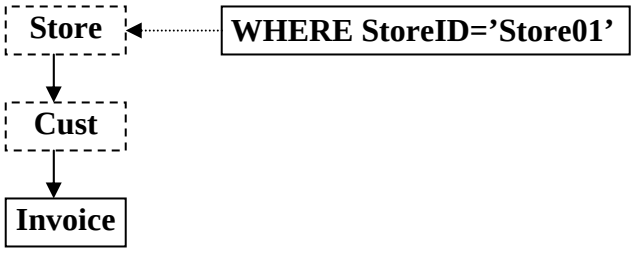
The following example queries demonstrate how data nodes and all their data occurrences located down all paths from a qualified node data occurrence are also qualified, and how the single path data occurrence up the path from a qualified data occurrence is also qualified. Compare the output of the following queries to the Hierarchical Node Set in Figure 4.1 above. This logic is similar to XPath logic.

4.1.1) Downward Path Data Qualification

The SQL 4.1.1 query below returns all invoice IDs (Invoice IDs 1,2,3) under the Store occurrence "Store01". All invoice IDs under the qualified ancestor data occurrence "Store01" are returned.

SQL 4.1.1: **SELECT InvID FROM StoreView WHERE StoreID='Store01'**

```
<root>
 <invoice invid="Inv01"/>
 <invoice invid="Inv02"/>
 <invoice invid="Inv03"/>
</root>
```



4.1.2) Qualification at the Point of Direct Data Qualification

The SQL 4.1.2 query below returns all invoice IDs (Invoice IDs 1,2) under the customer occurrence "Cust01" and the Cust ID being tested positive. All invoice IDs under the qualified ancestor data occurrence "Cust01" are returned. If "Cust01" occurred in other stores, their data would also be included. Also note that Inv03 did not qualify because its parent Cust occurrence Cust02 did not qualify.

SQL 4.1.2: **SELECT CustID, InvID FROM StoreView WHERE CustID='Cust01'**

```
<root>
 <cust custid="Cust01">
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
 </cust>
</root>
```

4.1.3) Upward Path Data Qualification

The SQL 4.1.3 query below returns only the Customer ID ("Cust01") associated with the qualifying invoice "Inv01" located below it. This is because only the single path occurrence up from the path of the qualified invoice data is qualified. Also note that invoice Inv02 would have qualified Cust01 also because it is a twin occurrence of Cust01. Twins have the same node type and same parent occurrence. Children have their own node type under their parent, they are located on different sibling paths while twins are on the same path.

SQL 4.1.3: **SELECT CustID FROM StoreView WHERE InvID='Inv01'**

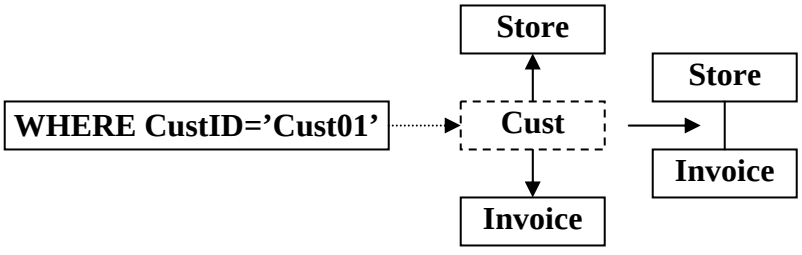
```
<root>
 <cust custid="Cust01"/>
</root>
```

4.1.4) Bi-directional Data Qualification

The SQL 4.1.4 query below returns all invoice IDs (Invoice IDs 1,2) under the customer occurrence "Cust01" being tested positive and only the Store ID located on the single qualified path above. This is a combination of the above single leg queries which qualifies down and up.

SQL 4.1.4: **SELECT StoreID, InvID FROM StoreView WHERE CustID='Cust01'**

```
<root>
 <store storeid="Store01">
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
 </store>
</root>
```



4.2) Simple Multi-leg Nonlinear Data Qualification.

Simple Multi-leg data qualification involves SELECTing data from one leg of a hierarchical structure, based on data in another leg of the structure. This is simple multi-leg hierarchical processing but involves complex hierarchical logic involving relating the two legs by their Lowest Common Ancestor (LCA) node data occurrence. All SELECTed data under a common ancestor node qualifies (similar to qualifying down a structure as demonstrated earlier). In academic terms, these are known as LCA queries. XQuery does not handle LCA queries automatically because different paths are navigated separately. The XQuery user must specify the procedural logic for multi-leg processing.

LCAs used for hierarchical structure processing is an important concept, but is not well known. In physical hierarchical databases it is performed by much tree walking back and forth across the legs. In relational databases this process is actually performed a single row at a time because of the relational engine's Cartesian product generation of the data. This Cartesian product is influenced by the LCA node naturally. What this means is that multi-leg queries are automatically and correctly processed for the user.

Multi-leg Structures enforces the fact that SQLfX® users do not need to know or be concerned with the structure of the data being processed. Even if multi-leg processing is not necessary, it still allows the capability to query any single-leg query from a single hierarchically optimized view.

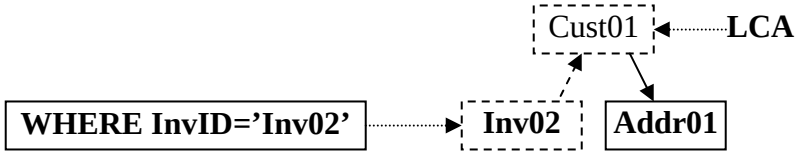
Selecting data from one leg based on data from another leg of the structure are cousin relationships. This is how all node types in the structure are related to each other across legs. The actual node data relationships depend on node (data) occurrence relationships in addition to the node type relationships.

4.2.1) LCA Many to One Result Data Qualification

The SQL 4.2.1 query below returns "Addr01" related through "Cust01" its common ancestor for value "Inv02" on another leg. Sibling legs across a query are related by their common ancestor data occurrence. Note that Inv01 also qualifies Addr01 (many to one) because Inv01 and Inv02 are twin occurrences.

SQL 4.2.1: **SELECT AddrID FROM StoreView WHERE InvID='Inv02'**

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

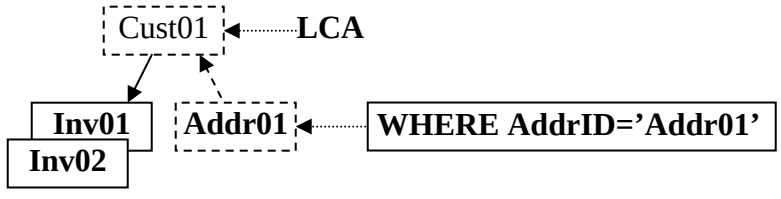


4.2.2) LCA One to Many Result Data Qualification

The SQL 4.2.2 query below returns invoices ("Inv01" and Inv02") under the common ancestor data occurrence "Cust01" for value "Addr01" (one to many). ALL occurrences are selected under the common ancestor node occurrence as in downward path qualification covered previously.

SQL 4.2.2: **SELECT InvID FROM StoreView WHERE AddrID='Addr01'**

```
<root>
 <invoice invid="Inv01"/>
 <invoice invid="Inv02"/>
</root>
```

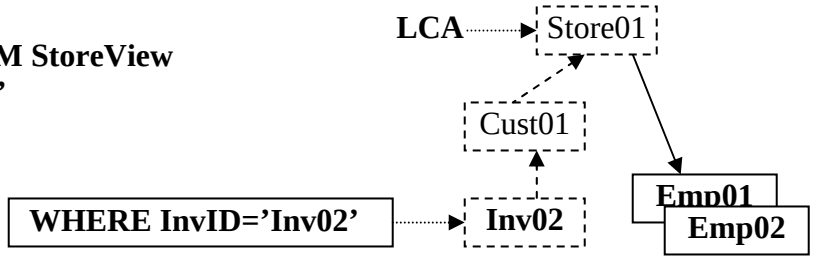


4.2.3) LCA Can be Located Higher Than Parent

The SQL 4.2.3 query below returns all employees ("Emp01" and "Emp02") under the common ancestor data occurrence "Store01". This shows that the common ancestor can be anywhere up the structure as long as it is the Lowest Common Ancestor (LCA).

SQL 4.2.3: **SELECT EmpID FROM StoreView**
WHERE InvID='Inv02'

```
<root>
 <emp empid="Emp01"/>
 <emp empid="Emp02"/>
</root>
```

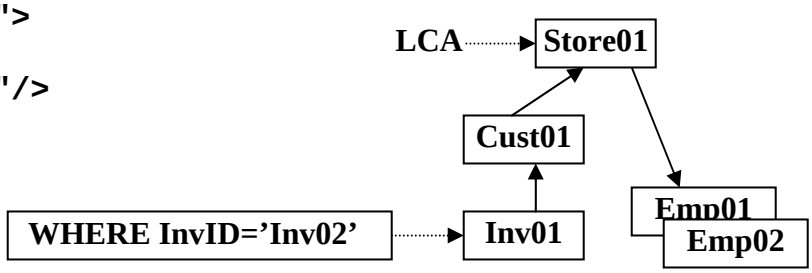


4.2.4) LCA Data from Below and Above

In the SQL 4.2.4 query below, "Inv02" is selected along with "Cust01", "Store01" because they are on a selected path up the structure, and on the downside of LCA occurrence of "Store01", all Employees ("Emp01" and "Emp02") are selected. The downward path can qualify multiple occurrences.

SQL 4.2.4: **SELECT InvID, CustID, StoreID, EmpID FROM StoreView WHERE InvID='Inv02'**

```
<root>
 <store storeid="Store01">
  <cust custid="Cust01">
   <invoice invid="Inv02"/>
  </cust>
  <emp empid="Emp01"/>
  <emp empid="Emp02"/>
 </store>
</root>
```

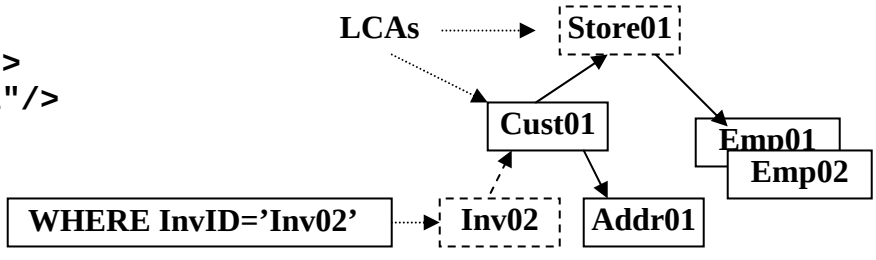


4.2.5) Multiple LCAs

The SQL 4.2.5 query below is similar to the previous query, but removes InvID and StoreID from the Select list, and adds AddrID. With this query, "Addr01", "Cust01", and Employees ("Emp01 and Emp02") are selected. The big difference with this query is that there are two LCAs, "Store01" same as last time, but by also selecting AddrID, Cust01 is a second (nested) LCA and selects all AddrID's under qualified Cust01. In this case there is only one "Addr01". Internally more complex, but not for the coder.

SQL 4.2.5: **SELECT AddrID, CustID, EmpID FROM StoreView WHERE InvID='Inv02'**

```
<root>
 <cust custid="Cust01">
  <addr addrid="Addr01"/>
 </cust>
 <emp empid="Emp01"/>
 <emp empid="Emp02"/>
</root>
```



4.3) Complex Multi-leg Nonlinear Data Qualification

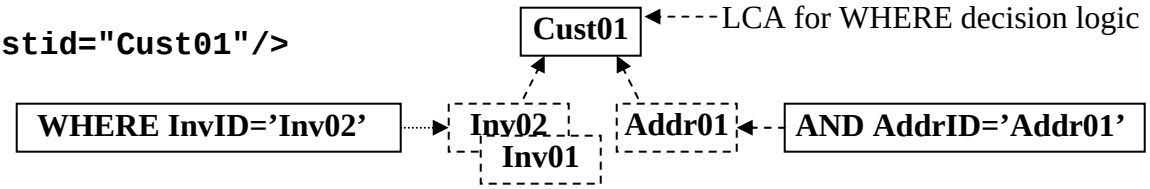
So far we have seen simple multi-leg data qualification involving simple single sided qualification tests producing static Lowest Common Ancestor (LCA). But more complex qualification tests are possible that will produce more complex hierarchical decision logic where qualification is based on values in multiple legs. This uses the natural Cartesian product operation of producing all data combinations to test all combinations of qualification test across legs. When hierarchical structures are defined in SQL the Cartesian product data replications formed around Join points are also the LCA points. So the control of data duplication is naturally centered on the LCAs and their hierarchical processing logic. In this way the correct combination of tests performed is geared to the LCA.

4.3.1) LCA Determines Range of Combinations for Decision Logic

The SQL 4.3.1 query below tests the two sibling legs under the LCA node Cust where one of the data combinations tested does match the test for "Inv02" and "Addr01" selecting "Cust01". This demonstrates how the relational Cartesian product can perform these complex multi-leg hierarchical LCA tests one row at a time (avoiding tree traversal logic). Note that Cust02 and Cust03 node occurrences where not qualified because their node occurrences did not have matching Invoice and Addr node occurrences.

SQL 4.3.1: **SELECT CustID FROM StoreView WHERE InvID='Inv02' AND AddrID='Addr01'**

```
<root>
 <cust custid="Cust01"/>
</root>
```

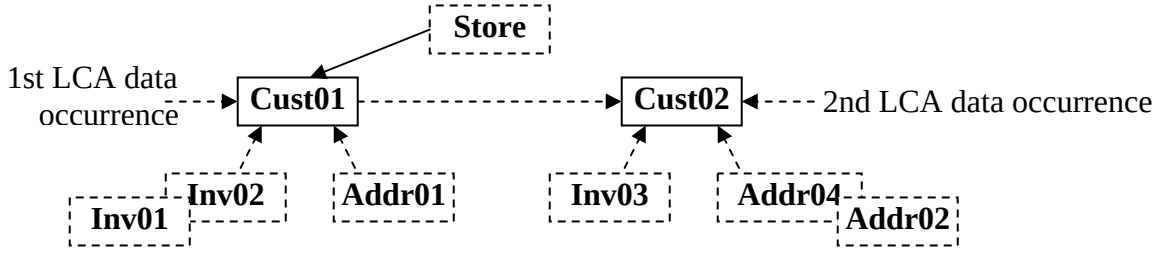


4.3.2) LCA Data Combinations Controlled by Data Occurrence

The SQL 4.3.2 query below returns a null result because no data is qualified because while there is an "Inv02" and "Addr02" data occurrence, they are not related by the same lowest common ancestor data occurrence. In the above example 4.3.1, the LCA data occurrence was Cust01 which was the same LCA occurrence of both. In example 4.3.2, Inv02 and Addr02 under the Cust node are in different parent occurrences and are not considered meaningfully related. These are standard nonlinear hierarchical processing rules and SQL processing naturally follows it with its Cartesian product controlled data duplication. Adding StoreID to the SELECT list does not select StoreID at the higher level either because the LCA and the Cartesian product duplicate data generation remains the same for the Invoice and Addr nodes.

SQL 4.3.2: **SELECT CustID FROM StoreView WHERE InvID='Inv02' AND AddrID='Addr02'**

```
<root>
</root>
```

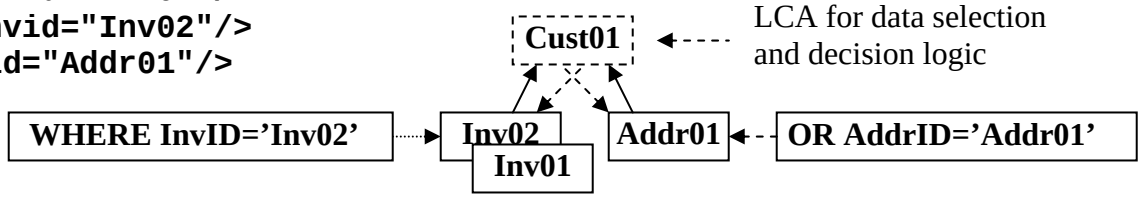


4.3.3) Variable LCAs With OR Decision Logic

Hierarchical level OR decision logic can get tricky. The following SQL query returns both invoices ("Inv01" and "Inv02") and "Addr01". Normally with OR operations, if the first condition is true, the second condition on the right side does not require testing. This is not the case for hierarchical processing semantics, if it was, then "Inv01" would not be selected. You might think it should not be selected because the left side does test true with InvID="inv02", but it is selected because the other sibling (right) side test where AddrID="Addr01" was also tested and selected. It qualified both invoices under the common ancestor "Cust01". This means that both sides of the OR condition always needs to be tested at the hierarchical query level. This result and hierarchical logic can be proven by breaking the query into two queries each with one side of the WHERE clause and unioning the results together. This logic also results in LCA qualification logic being dynamically switched between the left and right OR condition depending on which side is true, which is tested below in SQL 4.3.3. This double sided testing of OR conditions on the relational WHERE clause is naturally performed by the Cartesian product building all combinations so that both sides of the WHERE clause are eventually tested over multiple rows containing replicated data so that the following query operated corrected in relational hierarchical processing.

SQL 4.3.3: **SELECT InvID, AddrID FROM StoreView**
WHERE InvID='Inv02' OR AddrID='Addr01'

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

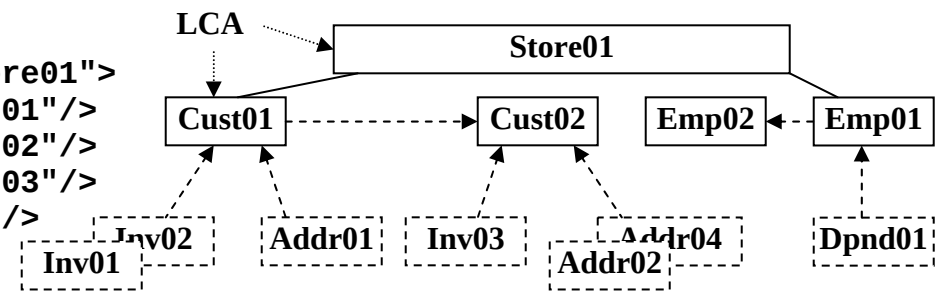


4.3.4) Complex Multi-leg LCA Decision Logic

The SQL 4.3.4 query below demonstrates that complex decision logic can involve more than one instance of common ancestor decision logic. This query first evaluates the AND condition which is false (no common ancestor combination found under a Customer node for "Inv02" and "Addr02") and then evaluates this false condition with the right side of the OR condition under the common ancestor node Store which is true (DpndID="Dpnd01"). Based on this, "Store01" is selected, all customers ("Cust01", "Cust02", and "Cust03") are selected (because their opposite side qualified), and only employee "Emp01" was selected (because its opposite side didn't qualify; but only its side with "Dpnd01", only qualifying "Emp01" above it). This is all performed under the covers automatically and accurately.

SQL 4.3.4: **SELECT StoreID, CustID, EmpID FROM StoreView**
WHERE InvID='Inv02' AND AddrID='Addr02' OR DpndID='Dpnd01'

```
<root>
 <store storeid="Store01">
  <cust custid="Cust01"/>
  <cust custid="Cust02"/>
  <cust custid="Cust03"/>
  <emp empid="Emp01"/>
 </store>
</root>
```



4.4) Focused Retrieval with Result Aggregation

Focused Retrieval with Result Aggregation is an IR (Information Retrieval) term used to mean that XML documents can be dynamically searched and only correctly identified XML documents are identified and then only the correctly associated data is returned condensed into a meaningful result. It also implies that this is done in an ad hoc or interactive manner without requiring pre established query logic. The previous example of SQL 4.3.4 is a good example of focused retrieval and result aggregation. Focused retrieval is met by the hierarchical filtering of the WHERE clause isolating the qualification within single documents and result aggregation is then met by the SELECT list operation only outputting the desired data in a structured XML format preserving the semantics. This example like all the others in this document are produced dynamically, this satisfies the last requirement for focused retrieval with result aggregation.

4.5) Recap of Hierarchical Query Qualification

Hierarchical query qualification must take the hierarchical data and hierarchical structure into consideration when performing data selection even though the user or developer does not usually need to be aware of the exact data structure when specifying the SQL query. The following capabilities that comprise this hierarchical query qualification were tested and proven.

Single Path Data Qualification Up the Path	All data up the current qualified path data occurrence qualifies
Single Path Data Qualification Down the Path	All related data occurrences down the path qualify (This involves multiple occurrences)
Multiple Path Data Qualification Down a Path	Can break off to multiple paths producing one or more Lowest Common Ancestors (LCAs)
Common Parent Single Leg (one-sided) Data Qualification	All data under the common parent occurrence qualifies WHERE test on the other leg
Common Parent "AND" Multi-leg Decision logic	All data conditional combinations under common parent tested and both must be true.
Common Parent "OR" Multi-leg Decision Logic	It was proven that OR logic requires both sides tested with common parent hierarchical semantics
Multiple Nested Lowest Common Ancestors	Each handled in turn automatically, combining to produce the correct hierarchical result

Table 4: Hierarchical Query Qualification Operation

It is quite amazing that SQL can perform multi-leg qualification because it requires very complex hierarchical processing semantics and SQL's processing is basically performed a row at a time, but it does work no matter how complex the hierarchical query may be. Using XQuery which is not nonprocedural and requires navigation to do this processing, the level of required programming is extremely difficult, tricky, and requires user knowledge on nonlinear LCA hierarchical processing.

5) Conceptual Hierarchical Structure Linking (Combine Structures)

Conceptual hierarchical joins are key to SQLfX®'s easy and powerful operation that distinctly sets its external operation apart from all other SQL/XML products. The combined hierarchical structures are unified into a hierarchical superstructure with all of its hierarchical semantics combined into an even more powerful structure with its additional semantics that control its nonprocedural processing. This is a powerful data mashup capability.

5.01) Full Hierarchical Structure Linking

Full hierarchical structure linking operates on entire hierarchical structures combining them automatically into a hierarchical combined result structure. It uses the same Left Outer Joins that modeled the structures being linked. The hierarchical structures being linked are stored in SQL views and are easily linked conceptually into a hierarchically unified structure as can be seen below in SQL 5.0.

SQL 5.0: **SELECT * FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID**

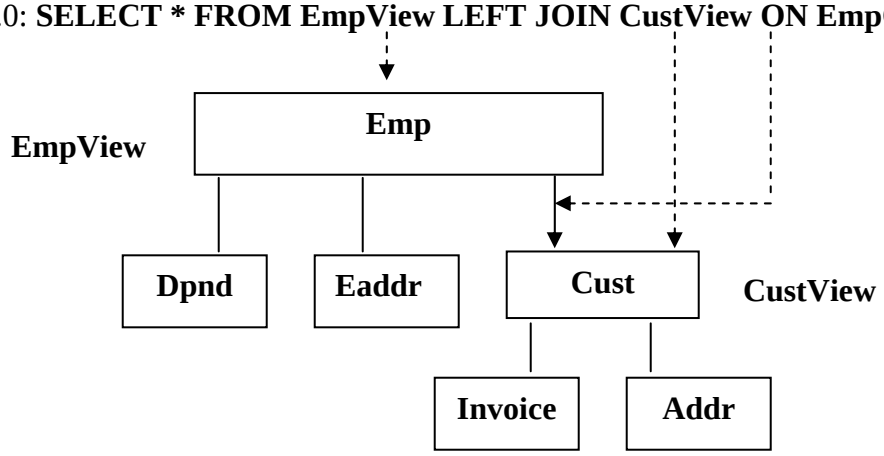
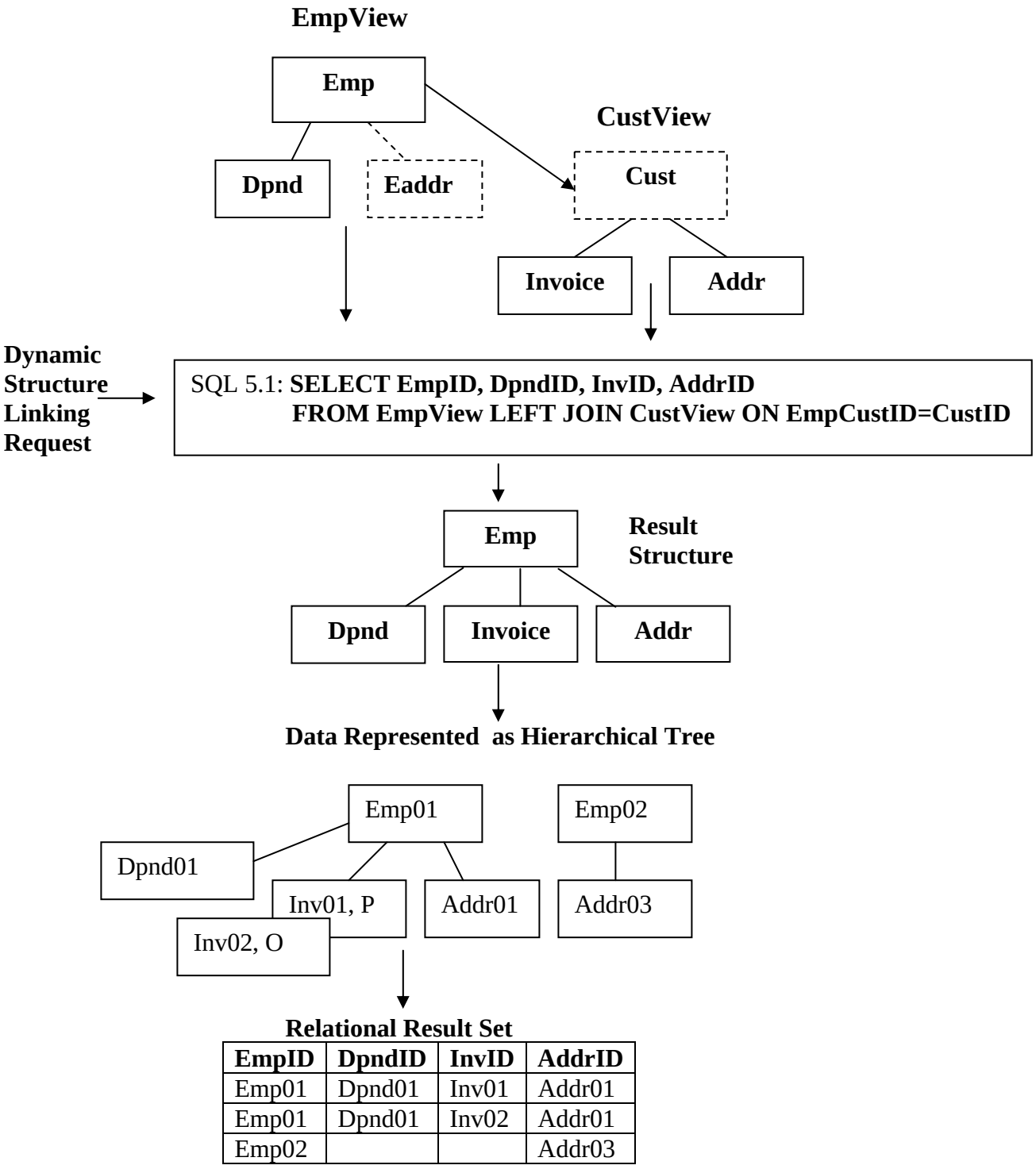


Figure 5.0 Hierarchical join process

```
<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>
```


5.1) Dynamic Conceptual Hierarchical Modeling and Structure Linking

SQL hierarchical views are incredibly powerful and useful for hierarchical processing. They represent an entire structure as a single abstract object and allow simple hierarchical operations like linking (left joins) to be easily applied to entire complex hierarchical structures as shown below in Figure 5.1 below.



EmpID	DpndID	InvID	AddrID
Emp01	Dpnd01	Inv01	Addr01
Emp01	Dpnd01	Inv02	Addr01
Emp02			Addr03

Figure 5.1: Dynamic join and its data

The dynamic join in Figure 5.1 above demonstrates that hierarchical views can be used to very easily construct new combined structures at a high conceptual hierarchical level. The above materialized Relational Result Set in Figure 5.1 can be produced from the ad hoc query modeled above with its SQL query and XML results show below in SQL 5.1.

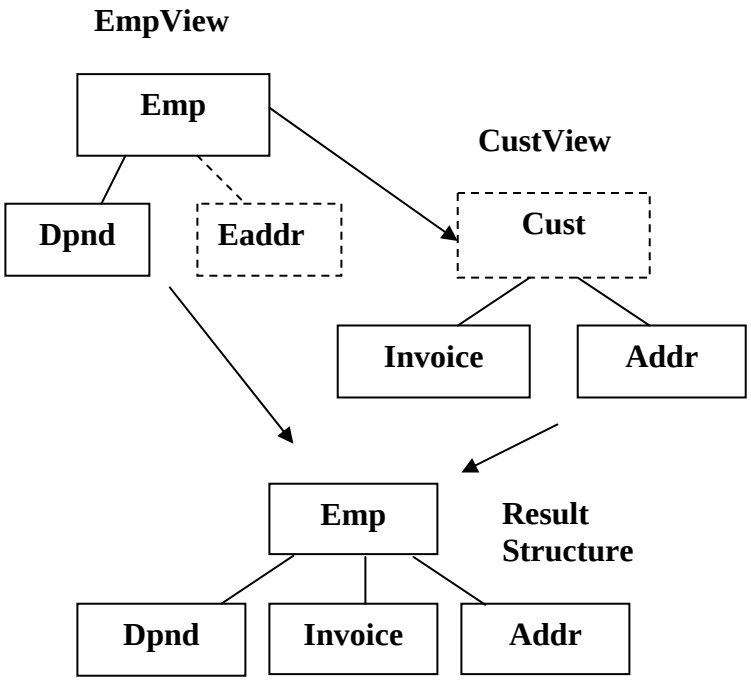
Since the below query only needs one address node type, the EAddr table in the EmpView is not referenced in the query's SELECT list. Since we do not need or want customer information to be retrieved with this view, Customer data is also excluded from the SELECT list. This has the effect of not ever having a Eaddr or Cust node in the result structure. This causes the Invoice and Address nodes to always be promoted up and around the missing Customer node in the result to take its position, preserving the integrity of the result structure. This can be inferred from the Data Represented as a Hierarchical Tree and the Relational Result in Figure 5.1 above.

This query is also an example of joining hierarchical structures hierarchically and conceptually at a very high level. It is very powerful for interactive use. The Customer view is linked under the Employee view. The ON clause allows this join to be linked unambiguously and precisely as required. The result of this structure joining is supported in the query that externalizes the materialized view above.

This combined view is also an example where an unselected (non output) node can supply required information. In fact in this example, Cust is an unselected node and is the lower level link point node.

SQL 5.1: **SELECT EmpID, DpndID, InvID, AddrID**
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID

```
<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>
```

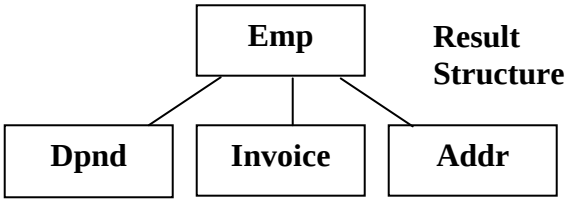


5.2) Global Hierarchical Filtering Operation

Using the same query as the previous dynamic query, SQL 5.2, to demonstrate that dynamic queries can accept global WHERE clause filtering, a WHERE clause is added qualifying the query based on the value "INV02". This will remove the entire Emp02 record since it does not have an Inv02. The Emp01 record does qualify with its Inv02 keeping the record, but the Inv01 node occurrence is removed because it does not qualify. This hierarchical filtering demonstrates a level of filtering not utilized fully with relational filtering

SQL 5.2: **SELECT EmpID, DpndID, InvID, AddrID**
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID
WHERE InvID='Inv02'

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



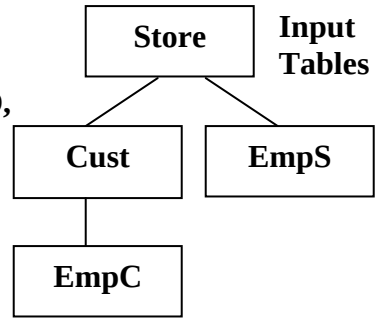
5.3) Replicating, Renaming, and Splitting Nodes

Let's look at the capability of SQL and SQLfX® to handle duplicate node types (relational table and XML elements) in the hierarchical structure. This is done in SQL using the table rename capability to introduce the same node type into the structure multiple times. A renamed node is used in the SQLfX® generated XML. The following SQL 5.3 example demonstrates this replicated structure with the Emp node renamed to EmpC and EmpS. Additionally the Emp node data was split across the two renamed versions of the Emp nodes. Both versions retained the EmpId value with the EmpC node taking the EmpStatus, and the EmpS node taking EmpCust. To make these renamed data fields easier to specify and to accommodate XML usage the Emp fields have been renamed in SQL to EmpcID and EmpcStatus for the EmpC node, and EmpsID and EmpsCustID for the EmpS node.

These renamed column names are picked up in the XML generated names as shown below. The optional AS alias keyword is used emphasize the renaming. The below SQL can be stored in a view, not shown. Additionally renaming capability will be provided in the initial commercial release using a view to view reshaping and renaming explicit capability.

SQL 5.3:

```
SELECT StoreID, CustID, Empc.EmpID AS EmpcID,
  EmpC.EmpStatus AS EmpcStatus, EmpS.EmpID AS EmpsID,
  EmpS.EmpCustID AS EmpsCustID
FROM Store
 LEFT JOIN Cust ON CustStoreID=StoreID
 LEFT JOIN Emp AS EmpC ON CustID=EmpC.EmpCustID
 LEFT JOIN Emp AS EmpS ON StoreID=EmpS.EmpStoreID
```



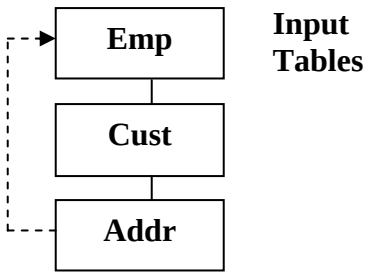

```
<root>
 <store storid=”Store01”>
  <cust custid=”Cust01”>
   <empc empcid=”Emp01” empcstatus=”F”/>
  </cust>
  <cust custid=”Cust02”>
  </cust>
  <cust custid=”Cust03”>
   <empc empcid=”Emp02” empcstatus=””/>
  </cust>
  <emps empsid=”Emp01” empcustid=”Cust01”/>
  <emps empsid=”Emp02” empcustid=”Cust03”/>
 </store>
</root>
```

5.4) Backward Path Data Filtering (Static Value)

When joining tables or views the ON clause can also specify additional join criteria with the AND operator that acts as an additional data filter that takes affect at this join location. What gives this capability exceptional power is that the filter criteria can be supplied from anywhere up the active path. In the SQL 5.4 example below the Addr node ON clause uses the EmpStatus value from the Emp node above to determine the filtering for Addr node. In this example you will notice that only the Addr node is present for Employees who are full time (have an “F” status Code). This filtering does not affect the Emp and Cust nodes.

SQL 5.4:

```
SELECT CustID, EmpID, AddrID, EmpStatus
FROM Emp
 LEFT JOIN Cust ON CustID= EmpCustID
 LEFT JOIN Addr ON CustID=AddrCustID
                AND EmpStatus='F'
```




```
<root>
 <emp empid="Emp01" empstatus="F">
  <cust custid="Cust01">
   <addr addrid="Addr01"/>
  </cust>
 </emp>
 <emp empid="Emp02" empstatus="">
  <cust custid="Cust03">
  </cust>
 </emp>
</root>
```

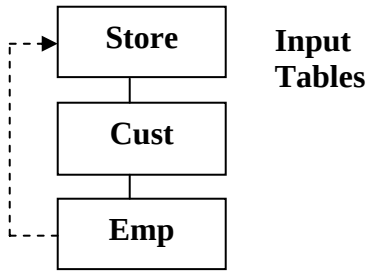
“Addr03” filtered out

5.5) Backward Path Qualification (Dynamic Value)

When joining tables or views the ON clause can also specify additional join criteria with the AND operator that acts as an additional path qualification from values up the active path. This capability additional qualifies the path at the point its ON clause is at. In the SQL 5.5 below example below the Emp node ON clause uses the StoreID value above it from the Emp node to restrict Emp to their assigned Store. In this example it is possible that the Emp under Cust belongs to a different Store than Cust. This filtering does not affect the Emp and Cust nodes. The Emp node is attached directly to the lowest upper node referenced which is Cust and not the Store node.

SQL 5.5:

```
SELECT StoreID, CustID, EmpID
FROM Store
 LEFT JOIN Cust ON StoreID=CustStoreID
 LEFT JOIN Emp ON CustID= EmpCustID
                AND StoreID=EmpStoreID
```




```
<root>
 <store storeid="Store01">
  <cust custid="Cust01">
   <emp empid="Emp01"/>
  </cust>
  <cust custid="Cust02">
  </cust>
  <cust custid="Cust03">
   <emp empid="Emp02"/>
  </cust>
 </store>
</root>
```

5.6) Sibling Leg Join Order and View Expansion

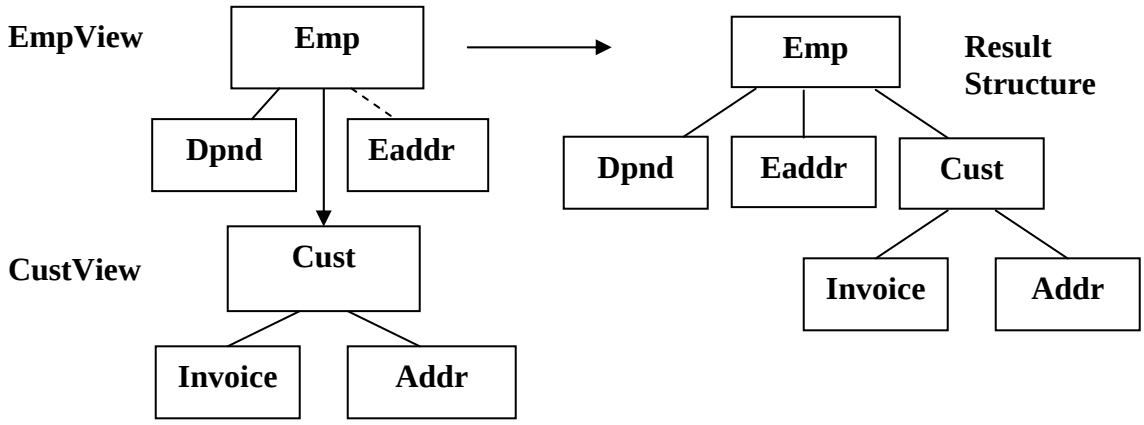
This example shows the default order that sibling nodes are added left to right in the structure being built from the supplied SQL. The same is true for siblings added from a view. This is shown in the expansion SQL of SQL 5.6 below with its Emp and Cust views underlined, they are expanded in place. Notice that the Cust node is naturally added after the Dpnd and Eaddr nodes.

SQL 5.6: **SELECT EmpID, DpndID, CustID, InvID, AddrID, EaddrID**
FROM EmpView LEFT JOIN CustView ON EmpCustID=CustID

Expanded SQL:

SELECT EmpID, DpndID, CustID, InvID, AddrID, Eaddr
FROM
Emp LEFT JOIN Dpnd ON EmpID=DpndEmpID AND DpndCode = 'D'
LEFT JOIN Eaddr ON EmpCustID=EaddrCustID ← **EmpView**
LEFT JOIN
Cust LEFT JOIN Invoice ON CustID=InvCustID
LEFT JOIN Addr ON CustID=AddrCustID ← **CustView**
ON EmpCustID=CustID

The expanded SQL directly above demonstrates the default sequential sibling node order. This can be changed by structure transformation performed after the structure is built. In the future a new syntax keyword can be added to the ON clause to specify where the lower level root of the lower view is to be inserted in the existing sibling legs. This will occur during structure building and will be more efficient.




```
<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>
```

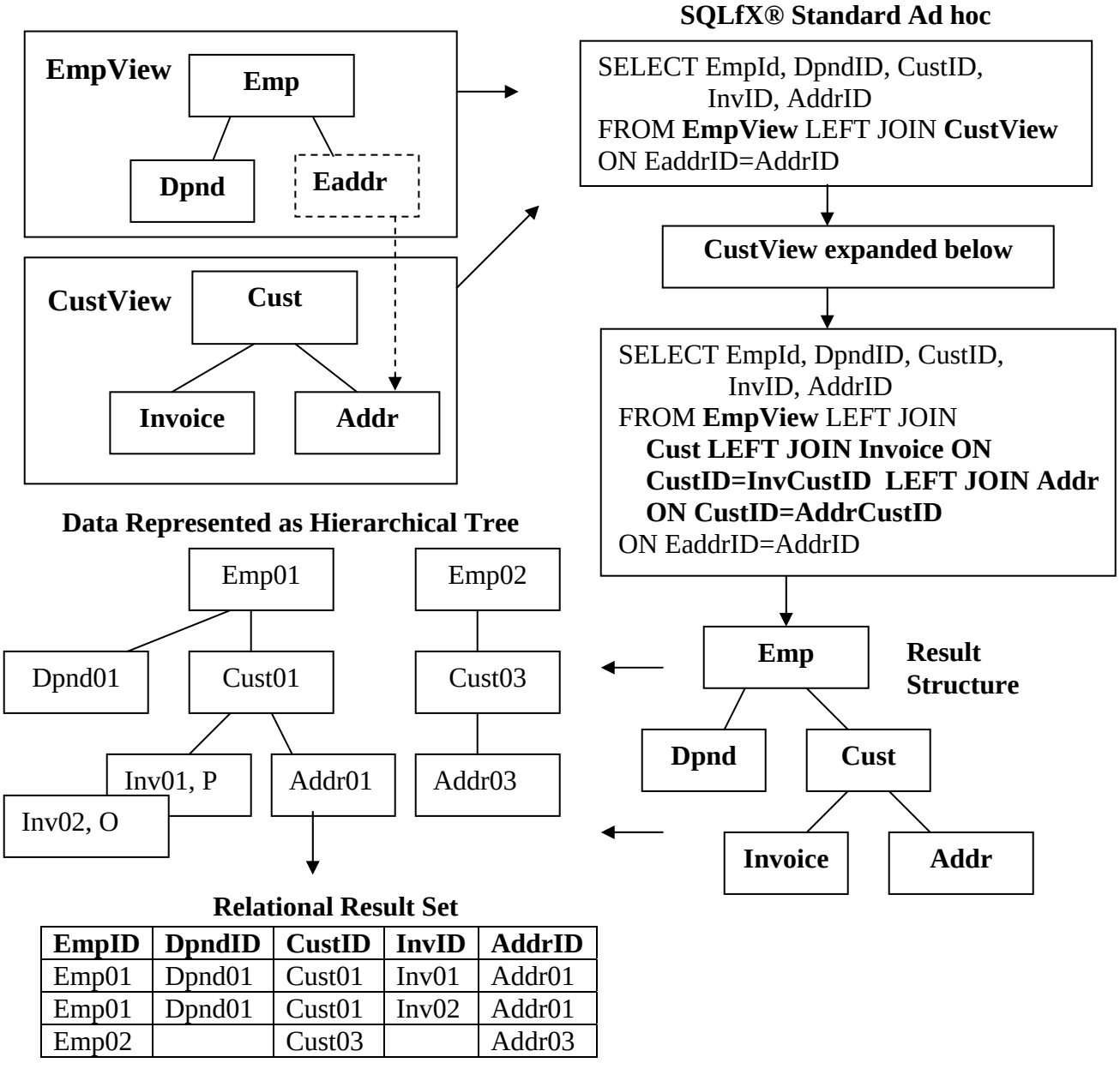

5) Recap of Conceptual Hierarchical Modeling and Joining

Joining Hierarchical Structures	Hierarchical structures defined in SQL views can be easily and exactly combined into a larger hierarchical data structure
Unselected Nodes Can be Referenced	Node promotion automatically occurs when input nodes are not selected for output. This also causes node collection.
Conceptual Hierarchical Join	Because of the powerful SQL views, extremely powerful and automatic combining of structures becomes simple and user friendly
WHERE Clause Filtering Allowed	SQL's full operation including WHERE clause filtering is available for dynamic operation, even when XML is being accessed.
SELECT List Allows Interactivity	A select list is not over-defined with XML formatting or embedded in a looping procedural structure remains practical for ad hoc interactive op
Replicating, Renaming and Splitting Nodes	The capability of replicating and renaming nodes is possible. A further capability of splitting the node into different node is also possible.
Backward Path Filtering and Qualification	Filtering and/or qualifying the hierarchical path on the ON clause can reference data items up the active path. It offers more filtering flexibility than XPath.

Table 5: Conceptual Hierarchical Modeling and Joining

6) Advanced Structure Linking With Look Ahead (Unrestricted Data Mashups)

SQLfX® has introduced a new hierarchical capability. When linking structures, linking below the root of the lower level structure is now possible. It is extremely flexible, avoids having restrictions, and is very useful allowing unrestricted data mashups. This will cause the lower level structure to be filtered based on the link point's value and will be covered shortly. Lower level linking is possible in views because SQL views cause their structure to be materialized before being joined to the upper level structure. This is also necessary for logical views which require their original root to materialize the structure. This also means that the original root remains the root for data modeling purposes since the original root in logical and physical structures still has control over what data is in the structure. This capability is also required in other valuable capabilities to be discussed later making it a very significant capability. An example of linking below the root in a lower level structure is shown below in Figure 6.0 below.



Relational Result Set

EmpID	DpndID	CustID	InvID	AddrID
Emp01	Dpnd01	Cust01	Inv01	Addr01
Emp01	Dpnd01	Cust01	Inv02	Addr01
Emp02		Cust03		Addr03

Figure 6.0: Linking below root example 1, root is selected

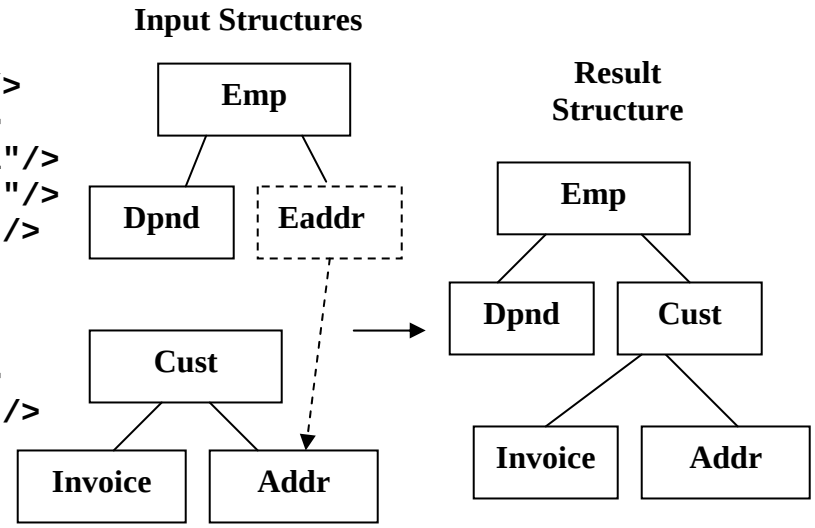
6.1) Linking Below Root of Lower Structure With Root SELECTed

The relational result set above in Figure 6.0 demonstrates that that linking below the root feature works as described above and dynamically. The lower level view is materialized before being linked to the above structure. This occurs because right sided nesting is performed with SQL views because of the way views expand pushing the current ON clause to the right to expand the lower level structure view with its own ON clauses. This is shown in Figure 6 above. The right sided nesting places the current structure being built in suspension and starts building the new right sided structure.

The lower level structure's original root node remains the root node even if the link point is below the root. This is because the original root node still affects what tables (or nodes) are in the structure. For example, in this example, if "Cust01" data occurrence did not exist then this lower level structure occurrence would not exist. This is demonstrated in SQL 6.1 example below.

SQL 6.1: **SELECT EmpID, DpndID, CustID, InvID, AddrID**
FROM EmpView LEFT JOIN CustView ON EAddrID=AddrID

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



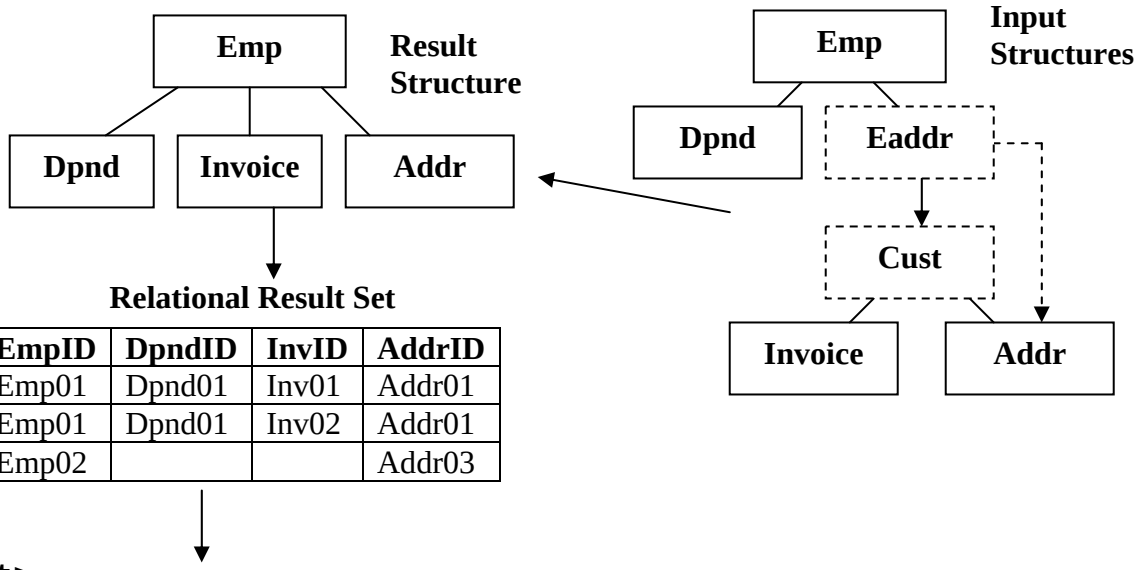
The filtering of the linking below the root needs to be pointed out in the example directly above. The lower level ON clause reference was to the Addr node in the Cust structure. The matching Eaddr higher level node values from the Emp structure are "Addr01" and "Addr03". These will match the "Addr01" and "Addr03"of the Lower level structure qualifying them up, down and across qualified legs (Inv01, Inv02). Downward, there are no lower level nodes under Addr01 and Addr03, but if so they would be qualified. Upward, Cust01 and Cust03 qualify. But notice that Addr02 and Addr04 under Cust02 that did not match do not qualify, this means that Cust02 does not qualify either because it is not qualified from any other qualified node occurrence. For this reason they are filtered out by the join operation. This is very sophisticated lower level structure processing carried out easily and automatically. It applies to logical or physical structures since they are both in the rowset at this point. This is shown in Figure 6.0 above and can be verified with Figure 2.1.

This process is performed automatically in SQL and is the semantically correct way to perform linking below the root. This has been an operation that has usually not been attempted because of its great implementation difficulty and lack of a semantically valid solution that SQLfX® has solved and implemented. SQL's rowset allows the materialized lower level view to be filtered easily.

6.2) Linking Below Root of Lower Structure Without the Root SELECTed

The original root node also logically and semantically holds the lower structure together so that if the root Cust is not selected for output, all of the selected lower level items including those that are not directly connected (on the same path) to the link point (Addr) are still processed correctly. This is shown in the example below in Figure 6.2, the Cust root is not selected, but the Addr and Invoice nodes are. The Addr and Invoice nodes are not connected directly to the link point but are related indirectly through the Cust node, a common ancestor. This does not present a problem, because the Cust node is still logically and semantically the root, even if it is not SELECTed as the example below demonstrates. This holds the record together in the Working Structure (set) as shown below in Figure 6.2.

SQL 6.2: **SELECT EmpID, DpndID, InvID, AddrID FROM EmpView LEFT JOIN CustView ON EAddrID=AddrID**



Relational Result Set

EmpID	DpndID	InvID	AddrID
Emp01	Dpnd01	Inv01	Addr01
Emp01	Dpnd01	Inv02	Addr01
Emp02			Addr03

```
<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>
```

Figure 6.2 Linking below root example 2, root not selected

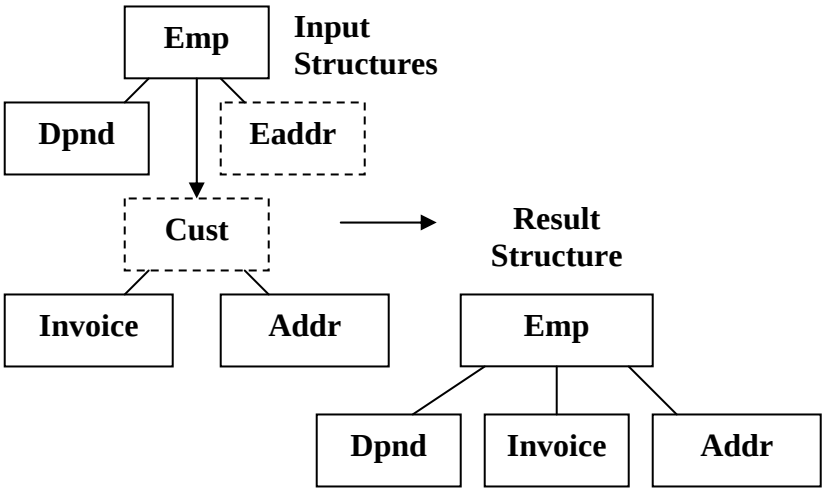
When you submit the query above in SQL 6.1, you should get the same result as in the Relational Result Set above in Figure 6.2. The Invoice node along with the Address node is dynamically promoted up under the Employee node. This makes sense logically and semantically because if there was no match for the Cust data occurrence "Cust01", both the Invoice and Address nodes would not have been located. This means its output hierarchical structure is still consistent with the previous example in Figure 6.1.

6.3) Filtering Below Root of Lower View

The same view capability that enables linking below the root of the lower level view allows lower level data to be used as ON clause filtering criteria. This allows either filtering out (removing) or accepting the full view. The following SQL statement and result establishes the full data before filtering is applied, Addr and Inv from the lower level CustView is shown. The following two examples (SQL 6.3 and SQL 6.3.1) will filter the lower level CustView first accepting the entire view, the second filtering it.

SQL 6.3: **SELECT EmpID, DpndID, InvID, AddrID FROM EmpView LEFT JOIN CustView ON EmpCustID=Custid**

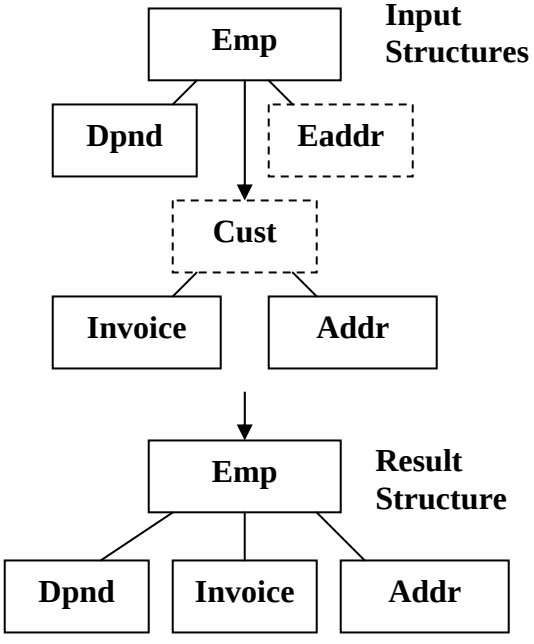
```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <addr addrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
  <addr addrid="Addr03"/>
 </emp>
</root>
```



The following SQL 6.3.1 query qualifies the customer view with Addr01 present (existing). This removes the view with Addr03.The EmpView is always preserved. This ability to reference ahead to qualify data is quite powerful and useful. Notice how the full view for Addr01 is included and Addr03 is not.

SQL 6.3.1: **SELECT EmpID, DpndID, InvID, AddrID FROM EmpView LEFT JOIN CustView ON EmpCustID=Custid AND AddrID='Addr01'**

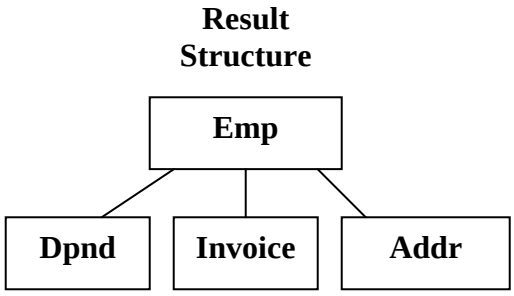
```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <addr addrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
 </emp>
</root>
```



The above example is the same as the last example but excludes Addr01 to demonstrate a larger lower level view occurrence being excluded. The following SQL 6.3.2 statement qualifies the customer view with Addr03 present. This removes the view with addr01 which also included Dpnd01, Inv01 and Inv02.The EmpView is always preserved.

SQL 6.3.2: **SELECT EmpID, DpndID, InvID, AddrID FROM EmpView LEFT JOIN CustView ON EmpCustID=Custid AND AddrID='Addr03'**

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
 </emp>
 <emp empid="Emp02">
  <addr addrid="Addr03"/>
 </emp>
</root>
```



6.4) Qualifying Multiple Legs with "AND" Condition

ON conditions with AND conditions along a single path in the upper structure is valid and qualifies the path to its lowest level. An AND condition along a single path in the upper structure referencing two different nodes in the lower level structure is OK too since the root in the lower level structured remains the root regardless of where it was joined. The derived structure from SQL 6.4 is shown.below. This is the same example as SQL 6.1 with an addition ON condition added that references another link point in the lower level structure.

SQL 6.4: **SELECT EmpID, DpndID, CustID, InvID, AddrID**
FROM EmpView LEFT JOIN CustView
ON EAddrID=AddrID AND EmpCustID=CustID

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

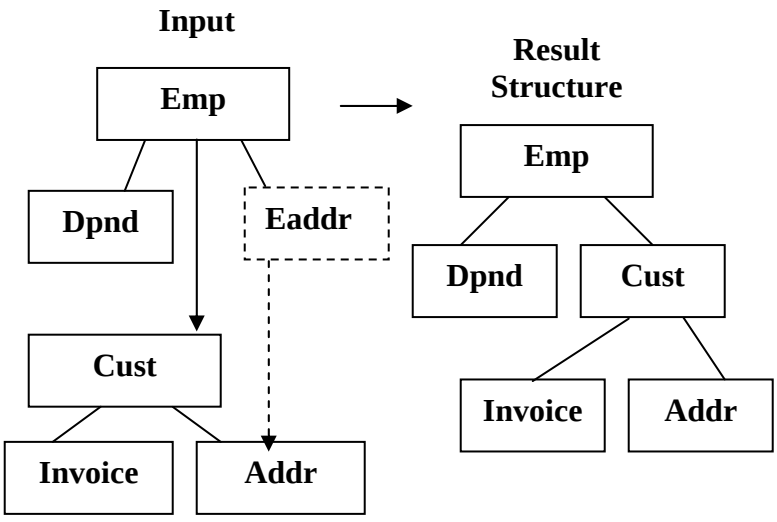


Figure 6.4 AND operations can qualify multiple legs

Both of the qualifying conditions, EAddrID=AddrID and CustID=EmpCustID, used in SQL 6.4 above have been tested separately in this section, and they produce the same result as each other. This example shown above in Figure 6.4 using the AND operation with both condition also produces the same result as each condition taken separately. This is proof that it is working correctly and is permitted.

6.4.1) Qualifying Multiple Legs with "AND" Condition Additional Test

If the Eaddr node had been SELECTed in Figure 6.4 above, then the Cust node would have been attached directly to it since it is the lowest qualified reference in the upper structure. This is shown below and proves that the lowest upper reference on the path is taken. This also demonstrates why the previous SQL 6.4 was correct because of the node promotion around the unselected Eaddr node.

SQL 6.4.1: **SELECT EmpID, DpndID, CustID, InvID, AddrID, EAddrID**
FROM EmpView LEFT JOIN CustView
ON EAddrID=AddrID AND CustID=EmpCustID

```
<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>
```

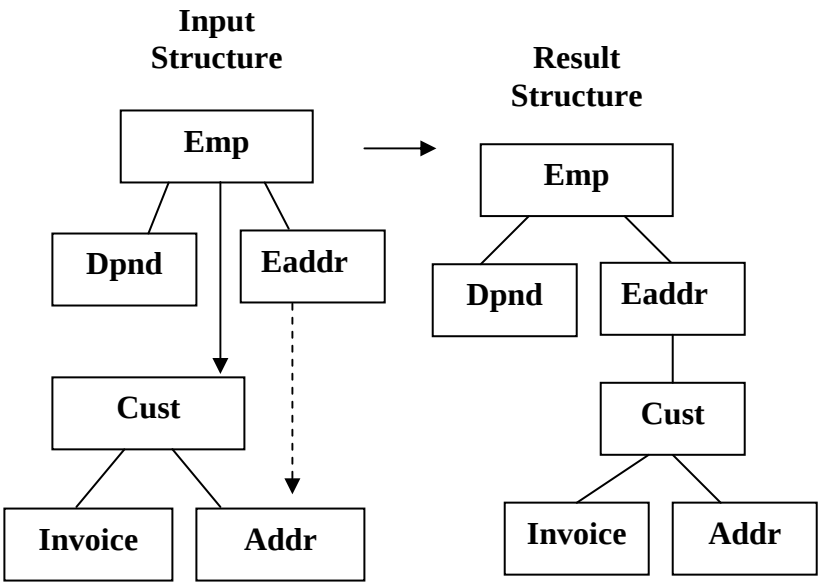


Figure 6.4.1 AND operations can qualify multiple legs

Recap of Linking and Filtering Below the Root

This section has demonstrated that linking below the root operates logically and hierarchically correctly while following SQL ANSI semantics. It has also demonstrated that the SQL view expansion works for embedded views along with the dynamic joining of hierarchical structures defined in views.

Linking Below the Root	This ability is possible because SQL structure views materialize before being joined. Their SQL semantics also makes sense.
Root not Selected	There is no requirement to select Root for input
Filters at Link Point	The below root link point is the data filtering point. Data qualification is processed up and down from this link point.
User Friendly	Users still do not need to know structure of data being processed.
More Join Capabilities	More capabilities to form associations otherwise not previous possible. Also eliminates restrictions, more freedom for the user.
Filter Referencing Below the Lower Level View	This ability is possible because SQL structure views materialize before being joined.
ON Condition AND Operations OK	ON condition AND operations can reference and filter multiple legs in the underlying structure

Table 6: Linking and Referencing Below the Lower Level Root

7.0) Dynamic Variable Structure Generation Control

SQL's Outer Join ON clause that is used to specify the hierarchical join criteria and structure link points can also be used to specify conditional join criteria as described and shown in Section 1.6 and Section 5.4. This can be used to dynamically control how structures can dynamically generate differently based on data values in the structure being created. This can be done at the node or view level and can use criteria values above and below the lower link point.

7.1) Variable Structure Generation Controlled at the Node Level

Lets look at an example of data path filtering based on a data value further up the current data structure path which can dynamically control the building of variable structures. The SQL 7.1 query below is built either with the Dpnd node or the Eaddr node based on a value from the Emp node higher up in the structure. When the current EmpStatus value is "F" the Dpnd node is included, when EmpStatus value is NULL the Eaddr node is included, but not both. This is similar in capability to COBOL's DEPENDING ON clause for those familiar with COBOL. This Variable Structure Control allows pieces of the data structure to be excluded or included by a control value further up its path. Depending where this control value is located up the path, its current value can change many times even in a single document (record) occurrence. There is no limit to how often this method can be used throughout the structure view.

Notice in the SQL 7.1 result below that employee Emp01 with a status of 'F' Contains a Dpnd node and not an Eaddr node, while employee Emp02 with no status has an Eaddr node but no Dpnd node.

SQL 7.1 **SELECT * FROM Emp**
LEFT JOIN Dpnd ON EmpID=DpndEmpID AND EmpStatus = 'F'
LEFT JOIN Eaddr ON EmpCustID=EaddrCustID AND EmpStatus Is NULL

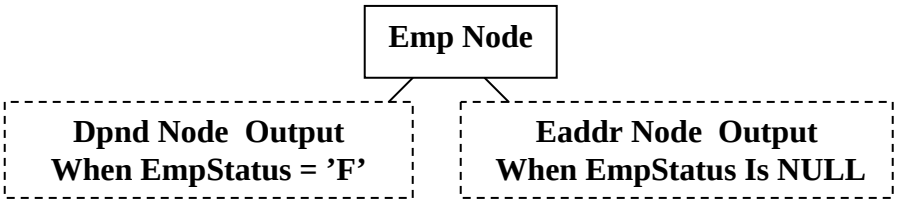


Figure 7.1 Variable structures built at node level

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
  <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
 </emp>
</root>
```


7.2) Variable Structure Generation Controlled at the View Level

Views can also be used with building variable structures. In this case the entire view structure is either included or excluded. In the SQL 7.2 example below, the EmpStatus field in the root above is used again, but this time it controls whether the full CustView structure is included in the result or not. The XML result shows that it is included in one case and not the other.

SQL 7.2 **SELECT * FROM Emp**
LEFT JOIN CustView ON EmpCustID=CustID AND EmpStatus = 'F'
LEFT JOIN Dpnd ON EmpID=DpndEmpID AND EmpStatus Is NULL

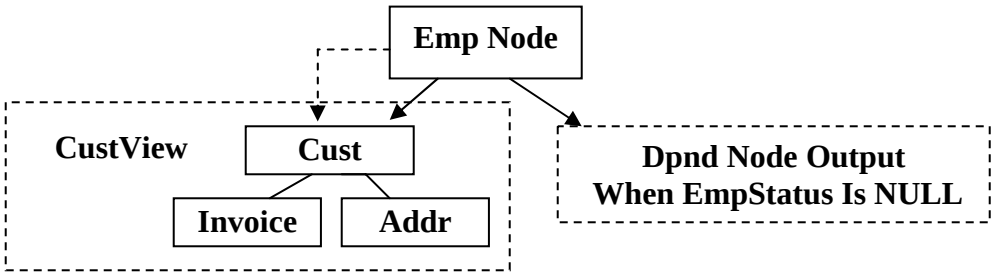


Figure 7.2 Variable structures built using views

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <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="">
  <dpnd dpndid="Dpnd02" dpndempid="Emp02" dpndcode="N"/>
 </emp>
</root>
```

7.3) Variable Structure Generation Using View Look Ahead

Views also offer another powerful capability for variable structure generation. As was shown and explained in the previous Section 6, 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 already be in the structure and if not selected the view structure will not be included. The reason that this "look ahead capability" is possible is because the view, EmpView in this case, naturally expand and materialize before it is joined as explained in Section 6, making all of its contents available.

In the SQL 7.3 example below, the value controlling the variable generation for the CustView (InvStatus) is not up the path from it, but is contained within it below its root. This is valid and produces the desired result. This result is similar to SQL 7.2 but you will notice that Invoice node Inv02 is excluded since its InvStatus is not 'P'. This filtering below the root is semantically correct because as explained in Section 6, it keeps only the proper matching occurrences qualified to keep the semantics meaningful. This is a very powerful intricate operation properly carried out automatically.

SQL 7.3 **SELECT * FROM Emp**
LEFT JOIN CustView ON EmpCustID=CustID AND InvStatus = 'P'
LEFT JOIN Dpnd ON EmpID=DpndEmpID AND EmpStatus Is NULL

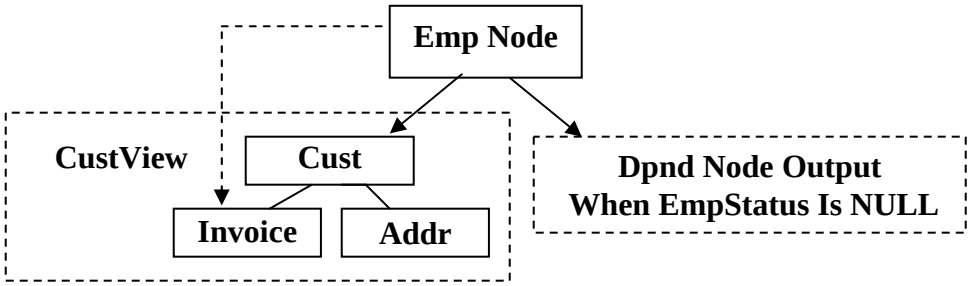


Figure 7.3 Variable structures built using views

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <cust custid="Cust01" coststoreid="Store01">
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P"/>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
  <dpnd dpndid="Dpnd02" dpndempid="Emp02" dpndcode="N"/>
 </emp>
</root>
```

7.4) Variable Structure Generation Using Embedded View

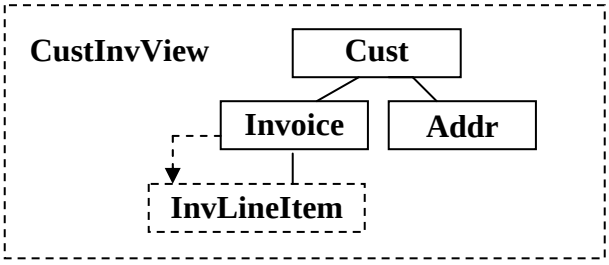
It is important to note that views can also contain this same variable structure control logic in a self contained form. This also means that variable structure fragments can also contain variable substructures. Suppose that the CustView view had an invoice line item table named InvItem under the Invoice node as shown below in Figure 7.4 and this node is a variable node controlled by InvStatus data field. This InvItem table contains multiple inventory line items for each invoice. This variable controlled node and its control data field are self contained in a view. This means that this view can be variably selected and its generation can be variably controlled internally within the view making this example a multi-level variable structure generation. Lets first test the self contained variable CustInvView to see its use and XML output in SQL 7.4 below. In that result you will notice that there are no line items for invoices that do not have an InvStatus of 'P'.

SELECT * FROM InvItem

```
<root>
 <invitem invitemid="ITem01" inviteminvid="Inv01" invitemname="Coat" invitemcost="20"/>
 <invitem invitemid="ITem02" inviteminvid="Inv01" invitemname="Shirt" invitemcost="10"/>
 <invitem invitemid="ITem03" inviteminvid="Inv01" invitemname="Pants" invitemcost="35"/>
 <invitem invitemid="ITem04" inviteminvid="Inv02" invitemname="Socks" invitemcost="5"/>
 <invitem invitemid="ITem05" inviteminvid="Inv02" invitemname="Tie" invitemcost="7"/>
 <invitem invitemid="ITem06" inviteminvid="Inv02" invitemname="Gloves" invitemcost="12"/>
 <invitem invitemid="ITem07" inviteminvid="Inv03" invitemname="Coat" invitemcost="25"/>
 <invitem invitemid="ITem08" inviteminvid="Inv03" invitemname="Shoes" invitemcost="28"/>
</root>
```


CREATE VIEW CustInvView AS
SELECT * FROM CustView LEFT JOIN InvItem ON InvID=InvItemInvID AND InvStatus='P'

SQL 7.4: **SELECT * FROM CustInvView**




```
<root>
 <cust custid="Cust01" custstoreid="Store01">
  <invoice invid="Inv01" invcustid="Cust01" invstatus="P">
   <invitem invitemid="ITem01" inviteminvid="Inv01" invitemname="Coat"
    invitemcost="20"/>
   <invitem invitemid="ITem02" inviteminvid="Inv01" invitemname="Shirt"
    invitemcost="10"/>
   <invitem invitemid="ITem03" inviteminvid="Inv01" invitemname="Pants"
    invitemcost="35"/>
  </invoice>
  <invoice invid="Inv02" invcustid="Cust01" invstatus="O">
  </invoice>
  <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
 </cust>
 <cust custid="Cust02" custstoreid="Store01">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O">
  </invoice>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"/>
 </cust>
 <cust custid="Cust03" custstoreid="Store01">
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV"/>
 </cust>
</root>
```

7.5) Multi-level Variable Structure Generation Using Embedded View

Variable structure generating views can themselves be invoked variably. This means that this view can be variably selected and its generation can be variably controlled internally within the view making this example a multi-level variable structure generation. This is a multi-level variable view because Invoice is variable and it is under Emp which is also variable. This is demonstrated in SQL 7.5 below.

SQL 7.5 **SELECT * FROM Emp**
LEFT JOIN CustInvView ON EmpCustID=CustID AND EmpStatus ='F'

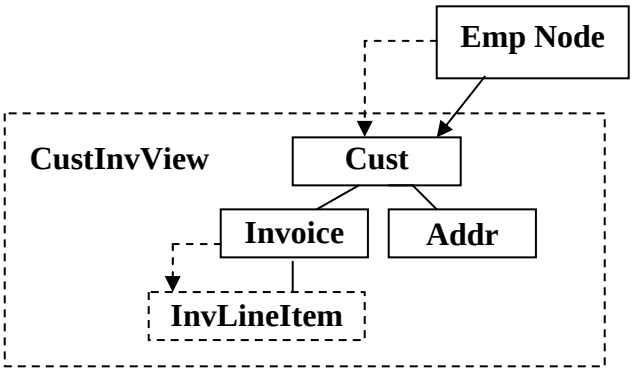


Figure 7.5 Variable structures built using view

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <cust custid="Cust01" custstoreid="Store01">
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P">
    <invitem invitemid="ITem01" inviteminvid="Inv01" invitemname="Coat"
     invitemcost="20"/>
    <invitem invitemid="ITem02" inviteminvid="Inv01" invitemname="Shirt"
     invitemcost="10"/>
    <invitem invitemid="ITem03" inviteminvid="Inv01" invitemname="Pants"
     invitemcost="35"/>
   </invoice>
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O">
   </invoice>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
 </emp>
```

7.6) Multi-level Variable Structure Generation Externally Specified

Embedded variable views shown above in Section 7.4 above are good for reuse, ease of use, and data abstraction, but their capability for multi-level variable structure creation can still be created at the top level SQL processing. The SQL shown in SQL 7.5 below demonstrates this by replacing the variable structure view (CustInvView) used in SQL 7.5 above with the original none variable view (CustView) and add the variable capability is replaced externally as shown below in SQL 7.6. This is a multi-level variable view because Invoice is variable and it is under Emp which is also variable. Notice that the XML results in SQL 7.6 match SQL 7.5.

SQL 7.6 **SELECT * FROM Emp**
LEFT JOIN CustView ON EmpCustID=CustID AND EmpStatus = 'F'
LEFT JOIN InvItem ON InvID=InvItemInvID AND InvStatus = 'P'

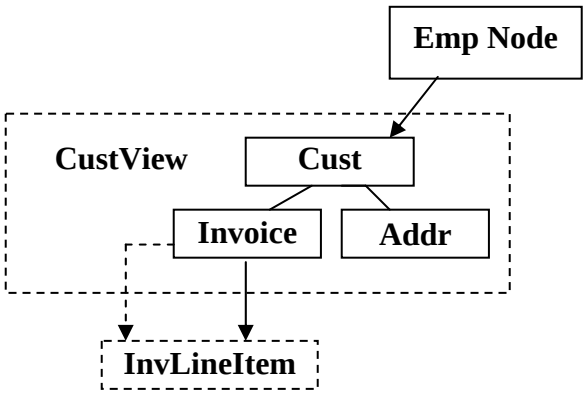


Figure 7.6 Multi-level Variable Structure Specified Externally

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <cust custid="Cust01" coststoreid="Store01">
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P">
    <invitem invitemid="ITem01" inviteminvid="Inv01" invitemname="Coat"
     invitemcost="20"/>
    <invitem invitemid="ITem02" inviteminvid="Inv01" invitemname="Shirt"
     invitemcost="10"/>
    <invitem invitemid="ITem03" inviteminvid="Inv01" invitemname="Pants"
     invitemcost="35"/>
   </invoice>
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O">
   </invoice>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
 </emp>
</root>
```

Recap of Variable Structure Control

Basic Variable Structure Control With Nodes	Any number of nodes can be separately or jointly controlled by one or more data items up their path
Views are Supported	Views can be variably controlled making their entire structure subject to all or nothing variable selection.
Lower Level View Reference Below Root Supported	This is a very powerful look ahead option without committing the addition of the lower level structure after being accessed. It can test data anywhere in the lower level view before being added.
Embedded Variable Sub Views are Supported	Views can contain variable structure logic embedded and operational within the view. This also means that variable substructures can contain variable substructures.
Multiple Level Variable Structures Can be Performed Externally	Embedded variable views are good for reuse, ease of use, and data abstraction, but their capability can still be created at the top level SQL processing if needed.

Table 7: Variable Structure Generation Features

8) Composite Keys Support

Let's start by creating table EmpMKey directly below. Its FirstName followed by LastName are defined as composite keys that specify a composite primary key. This table has two examples of duplicate fields, Mike which is in two Custs and Tim which occurs twice in the same Cust02. The LastName field will be used along with the FirstName field to specify a unique key. Unique keys are important for protecting and removing replicated data values without losing any information when building the XML result. We will be testing this out in this section.

Table EmpMKey

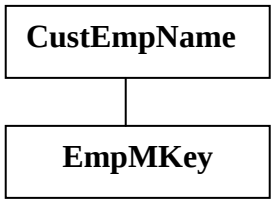
FirstName	LastName	EmpCustID
Mike	I	Cust01
Mike	II	Cust03
Tim	I	Cust02
Tim	II	Cust02

8.1) Preserve Correct Data

We will join the above EmpMKey table under the following CustEmpName table using only the FirstName to join on to see if the result has properly kept track of the duplicate named data in example SQL 8.1. The result XML for SQL 8.1 is accurate, each occurrence of Mike and Tim is meaningful.

Table CustEmpName

CustID	CustFirstName
Cust02	Tim
Cust01	Mike
Cust03	Mike



SQL 8.1: **SELECT CustID, CustFirstName, FirstName, LastName, EmpCustID**
FROM CustEmpName LEFT JOIN EmpMKey ON CustFirstName=FirstName

```
<root>
 <custempname custid="Cust01" custfirstname="Mike">
  <empmkey firstname="Mike" lastname="I" empcustid="Cust01"/>
  <empmkey firstname="Mike" lastname="II" empcustid="Cust03"/>
 </custempname>
 <custempname custid="Cust02" custfirstname="Tim">
  <empmkey firstname="Tim" lastname="I" empcustid="Cust02"/>
  <empmkey firstname="Tim" lastname="II" empcustid="Cust02"/>
 </custempname>
 <custempname custid="Cust03" custfirstname="Mike">
  <empmkey firstname="Mike" lastname="I" empcustid="Cust01"/>
  <empmkey firstname="Mike" lastname="II" empcustid="Cust03"/>
 </custempname>
</root>
```


8.2) Remove Correct Replicated Data

Now the real test is to remove the upper CustEmpName level from the XML output above and see if the duplicate data this creates is removed without removing too much info. This is performed correctly in SQL 8.2 below by removing the CustID and CustFirstName from the SELECT list triggering node promotion which introduces replicated data (Mike). The replicated Mike occurrences from SQL 8.1 are correctly detected and removed from the result XML leaving only the information necessary (two different Mikes). This makes the XML result value correct and semantically correct.

SQL 8.2: **SELECT FirstName, LastName, EmpCustID**
FROM CustEmpName LEFT JOIN EmpMKey ON CustFirstName=Firstname

```
<root>
 <empmkey firstname="Mike" lastname="I" empcustid="Cust01"/>
 <empmkey firstname="Mike" lastname="II" empcustid="Cust03"/>
 <empmkey firstname="Tim" lastname="I" empcustid="Cust02"/>
 <empmkey firstname="Tim" lastname="II" empcustid="Cust02"/>
</root>
```

8.3) Multi-Level Ordering with Composite Keys

There are no special requirements or restrictions with composite keys. The following SQL 8.3 example Orders By LastName and then Firstname. This reverses the natural precedence of the Firstname/Lastname compound key with out any problem.

SQL 8.3: **SELECT CustID, LastName, FirstName, EmpID**
FROM Cust Left Join EmpMKey ON CustID=EmpCustID
ORDER BY LastName, CustID Desc, FirstName

```
<root>
 <cust custid="Cust03">
  <empmkey lastname="I" firstname="Marry" empid="Emp04"/>
  <empmkey lastname="I" firstname="Ralph" empid="Emp06"/>
  <empmkey lastname="II" firstname="Mike" empid="Emp05"/>
 </cust>
 <cust custid="Cust02">
  <empmkey lastname="I" firstname="Steph" empid="Emp10"/>
  <empmkey lastname="I" firstname="Sue" empid="Emp08"/>
  <empmkey lastname="I" firstname="Tim" empid="Emp07"/>
  <empmkey lastname="II" firstname="Tim" empid="Emp09"/>
 </cust>
 <cust custid="Cust01">
  <empmkey lastname="I" firstname="John" empid="Emp03"/>
  <empmkey lastname="I" firstname="Mark" empid="Emp02"/>
  <empmkey lastname="I" firstname="Mike" empid="Emp01"/>
 </cust>
</root>
```


9.0) Nonlinear Hierarchical ORDER BY Operation

SQL ordering is linear. Hierarchical ordering of nonlinear multi-leg structures presents a number of problems because each leg needs to be independently ordered. In relational processing with rowsets, the ordering of one leg can make other legs already ordered unordered. A problem with separate leg ordering is that legs share the same lowest common ancestor node. Another problem is that ordering can easily inadvertently change your hierarchically modeled structures and then your result will not reflect your input data structure. For example, ordering the Emp node before the Cust node will inadvertently change the data structure and make the result incorrect if Cust over Emp is the desired structure.

9.1) Nonlinear Hierarchical Ordering Follows Hierarchical Structure

The problem of ordering out of hierarchical order can be seen below in Figure 9. First, it can be seen that the data replications can not be properly maintained while remaining ordered properly. Second, ordering Emp before Cust naturally gives it a higher significance level.

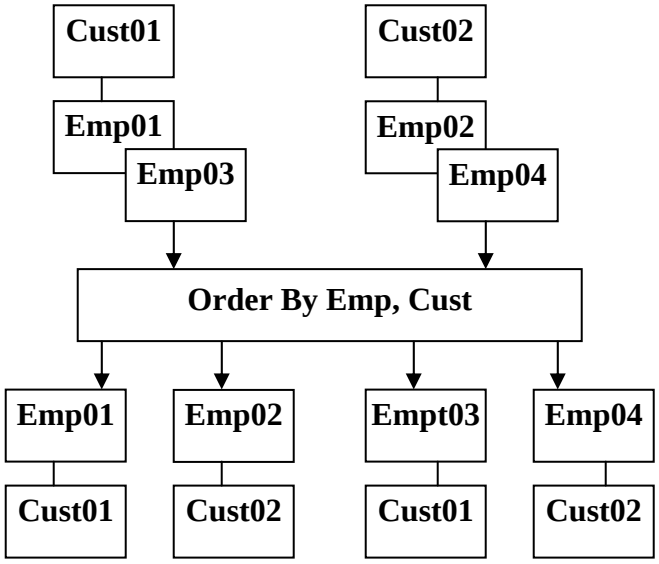


Figure 9.1: Ordering out of hierarchical order can change the structure inadvertently

9.2) SQLfX® Solution to Nonlinear Hierarchical ORDER BY

Special value added processing added by SQLfX® enables the ORDER BY process to operate hierarchically. Nonlinear hierarchical structures must be processed hierarchically. SQLfX® makes sure this happens. So the order that ORDER BY arguments are entered, do not make any difference, they are rearranged to fit the resulting hierarchical structure. The following SQL 9.2 query below with its three sort arguments will sort them correctly even if they are on separate legs. This is very difficult to accomplish with ANSI SQL/XML functions or XQuery which must be carried out procedurally.

You will notice that the three data items, Cust, Invoice and Addr, on different legs are sorted correctly using the single ORDER BY statement. Compare these results to the unsorted result in Figure 2.1.There are no special usage requirement as to the order of their placement in the in the ORDER BY statement. Try changing the order of their placement and you will see that the XML result does not change.

SQL 9.2: **SELECT CustID, InvID, AddrID FROM StoreView**
ORDER BY InvID DESC, AddrID Desc, CustID Desc

```
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>
```

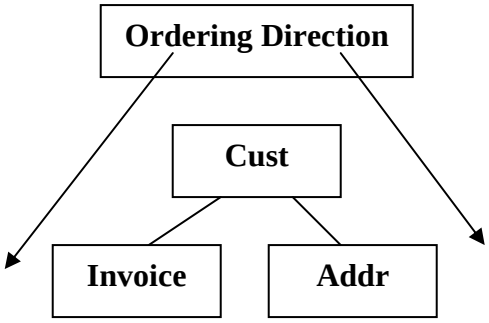


Figure 9.2 Nonlinear hierarchical ordering flow

This hierarchical ordering is intuitive, but is internally problematic. The Invoice and Addr nodes are both being ordered directly under the Cust node and this is not normally possible. In addition, if it was possible to sort them both at the second level, they are going to conflict with each other's orderings. Our value-added hierarchical ordering fixes this problem nonprocedurally retaining the ANSI SQL syntax.

9.3) Multi-level Hierarchical ORDER BY

It is not a requirement to order the key fields, nor is ordering limited to a single field in a node. This is demonstrated in SQL 9.3 below which uses multi-level node ordering in the Inventory and Addr nodes and also assigns the standard fields, InvStatus and AddrState, at a higher node ordering level than their key value. This can be seen in the Cust01 sub structure in the SQL 9.3 result below where InvStatus's ascending order has overridden the ascending Order of its key field InvId.

SQL 9.3: **SELECT * FROM StoreView**
ORDER BY InvStatus, InvID, AddrState, AddrID Desc, CustID Desc

```
<root>
 <store storeid="Store01">
  <cust custid="Cust03" coststoreid="Store01">
   <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV"/>
  </cust>
  <cust custid="Cust02" coststoreid="Store01">
   <invoice invid="Inv03" invcustid="Cust02" invstatus="O"/>
   <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"/>
   <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"/>
  </cust>
  <cust custid="Cust01" coststoreid="Store01">
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O"/>
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P"/>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
  <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
   <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
   <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  </emp>
  <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
   <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
  </emp>
 </store>
</root>
```

Recap of ORDER BY Capabilities

Hierarchical Ordering	With hierarchical structures, ordering must follow the structure of the nodes in the structure being queried
Multi-Level ORDER BY	Any number of fields can be ordered in any order in a node
Node ORDER BY Ordering	The order that the fields are specified in the ORDER BY has no significance, the node ordering is fixed to the structure being processed
Field ORDER BY Ordering	The field order specified does not affect the ordering of the nodes but does affect the relative ordering of the fields in each node
Display Ordering	In each node, the order of the fields follows their relative order in the SELECT list

Table 9: ORDER BY Use

10) Advanced WHERE Filtering Differences With ON Data Filtering

A more detailed example is necessary to show the difference between the WHERE clause and ON condition operation. This example will demonstrate first hand this difference and its importance to hierarchical processing. Example SQL 10.0 below produces a result with no filtering and established an unfiltered basis for the other results to be compared against. Example SQL 10.1 filters the query using the ON condition specifying EmpStatus='F' which filters out Employee Emp02's Dependent but still includes Emp02 in the XML result. This is similar to an XPath path filtering. Example SQL 10.2 filters the query using the WHERE clause specifying EmpStatus='F' and this causes the removal of the full path, Emp02 and Dpnd02. This is because the WHERE clause filters the entire structure. Looking at it another way, the ON clause operates during structure creation, while the WHERE clause is used after the structure is built. This is standard processing and corresponds to XQuery's WHERE clause filtering. The filtering ranges of the WHERE and ON operations are shown in Figure 10 below with arrows.

Figure 10: Range of Data Filtering

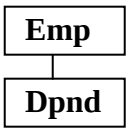
SQL 10.0: **SELECT * FROM Emp LEFT JOIN Dpnd ON EmpID = DpndEmpID**

```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
  <dpnd dpndid="Dpnd02" dpndempid="Emp02" dpndcode="N"/>
 </emp>
</root>
```

10.1) Linear Path Filtering and Business Rules Using ON Condition

The below SQL 10.1 query and result has employee "Emp02"s' dependent "Dpnd02" filtered out from the point on the path that the ON condition did not qualifying EmpStatus='F'. This is a path filtering very similar to XPath data filtering. In fact, the ON condition can refer back further up the active path within the active domain (i.e. active view). This path filtering can also be used to enforce business rules.

SQL 10.1: **SELECT * FROM Emp LEFT JOIN Dpnd ON EmpID = DpndEmpID**
AND EmpStatus='F'

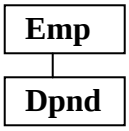



```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
 </emp>
</root>
```


10.2) Global Hierarchical Filtering WHERE Clause

The below SQL 10.2 query and result has both employee "Emp02" and its dependent Dpnd02 filtered out. This is because this global WHERE clause affects the entire query domain in a full nonlinear operation.

SQL 10.2: **SELECT * FROM Emp LEFT JOIN Dpnd ON EmpID = DpndEmpID**
WHERE Empstatus='F'




```
<root>
 <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01"
    empstatus="F">
  <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
 </emp>
</root>
```

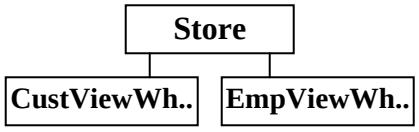
10.3) View Containing WHERE Clause Filtering

WHERE clauses in views are very useful being limited to their view domain, and they continue to operate in a nonlinear fashion for hierarchical views. We will test this by putting a WHERE clause filter into the CustView and one in the EmpView and join them using the Store table performing the same as our original StoreView. In this way we can verify if the result is the same as shown below in SQL 10.3.

CREATE VIEW **EmpViewWhere** AS
SELECT * FROM Emp LEFT JOIN Dpnd ON EmpID=DpndEmpID AND DpndCode = 'D'
LEFT JOIN Eaddr ON EmpCustID=EaddrCustID **WHERE EmpID='Emp01'**;

CREATE VIEW **CustViewWhere** AS
SELECT * FROM Cust LEFT JOIN Invoice ON CustID=InvCustID
LEFT JOIN Addr ON CustID=AddrCustID **WHERE CustID='Cust01'**;

SQL 10.3: **SELECT * FROM Store LEFT JOIN CustViewWhere ON StoreID=CustStoreID**
LEFT JOIN EmpViewWhere ON StoreID=EmpStoreID ;




```
<root>
 <store storeid="Store01">
  <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 empid="Emp01" empstoreid="Store01" empcustid="Cust01"
     empstatus="F">
   <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
   <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  </emp>
 </store>
</root>
```


10.4) Embedded View With Outer WHERE clause View

The identical result of the above SQL 10.3 query is also achieved from the following original StoreView with the equivalent WHERE clause added to it shown in SQL 10.4. This demonstrates the embedded WHERE clause in views continues to work in a correct nonlinear hierarchical fashion:

SQL 10.4: **SELECT * FROM StoreView WHERE CustID='Cust01' AND EmpID='Emp01'**

```
<root>
 <store storeid="Store01">
  <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 empid="Emp01" empstoreid="Store01" empcustid="Cust01"
     empstatus="F">
   <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
   <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  </emp>
 </store>
</root>
```

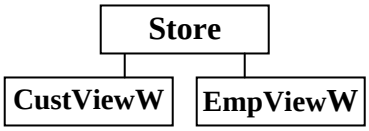
10.5) Embedded Views Each With a Piece of a Complex WHERE Clause

The identical result is also achieved by breaking the WHERE clause up and placing each part into its own view for the EmpView and CustView below and used in SQL 10.5. This tests the semantic soundness of hierarchical WHERE clauses.

CREATE VIEW **EmpViewW** AS SELECT * FROM EmpView **WHERE EmpID='Emp01'**;

CREATE VIEW **CustViewW** AS SELECT * FROM CustView **WHERE CustID='Cust01'**;

SQL 10.5: **SELECT * FROM Store LEFT JOIN CustViewW ON StoreID=CustStoreID**
LEFT JOIN EmpViewW ON StoreID=EmpStoreID ;




```
<root>
 <store storeid="Store01">
  <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 empid="Emp01" empstoreid="Store01" empcustid="Cust01"
     empstatus="F">
   <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
   <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  </emp>
 </store>
```


11) Automatic Detection of Ambiguous Query Structures

SQLfX® Is a SQL relational hierarchical processor which means that it operates only on hierarchical structures. Operating on non hierarchical structures would invalidate the hierarchical results. So ambiguous hierarchical structures are detected and trigger an error condition. The Ambiguous Query errors detected here when their containing SQL statement is executed are also detected immediately when executing a Create View containing one of these errors.

11.1) ON Clause OR Decisions

The following SQL 1.1 query models a network structure shown in Figure 11.1 because its OR condition defines two paths from two different nodes (Eaddr path OR Cust Addr path) to the same EmpView. Each path defines a different semantics (meanings) and this prevents nonprocedural query languages like SQLfX® from performing unambiguously and this needs to be detected and the user notified.

SQL 11.1: **SELECT * FROM CustView LEFT JOIN EmpView**
ON EaddrID=AddrID OR CustID=EmpCustID

The above ambiguous query produces the following error.

Semantic Error: OR inconsistent path error at or near CustID=EmpCustID

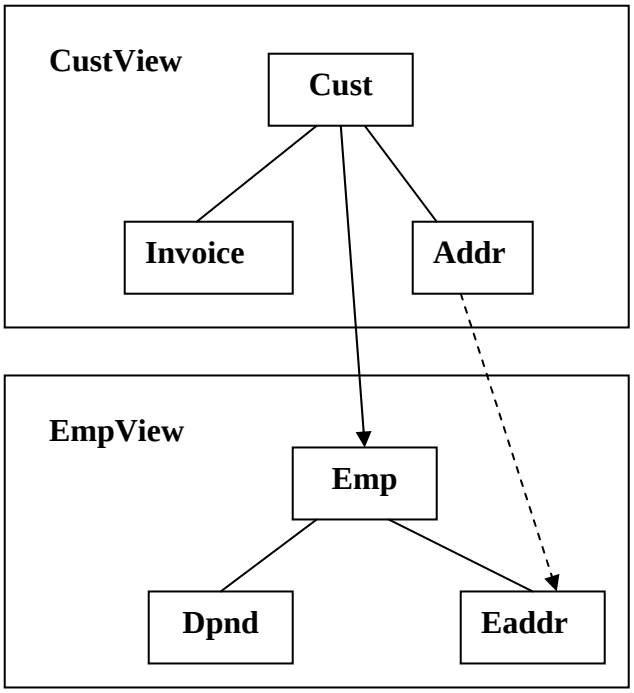


Figure 11: Ambiguous Network Structure

The Joining condition for CustView over EmpView was a complex ON condition that pointed to an OR choice of two different nodes that were in different paths as show with arrows in Figure 11. The real ambiguous problem here is that the Eaddr node is reachable from two different paths making this derived structure a network structure. Since each path has its own semantics and meaning, this nonprocedural query is ambiguous. Network structures like this one need to be procedurally navigated and lose their flexibility and ability to automatically utilize the powerful semantics in hierarchical structures.

11.2) ON Clause AND Conditions are More Tame

SQL 11.1 above is ambiguous because the ON clause OR test produced two separate paths by testing two different nodes. OR's operating on the same lower node are OK. By replacing the OR operation with an AND condition in SQL 11.1 the new query, SQL 11.2 below, becomes valid and produces the structure shown. The AND operation along a single path in the upper structure is valid and qualifies the path to its lowest level. An AND condition in the upper structure referencing two different nodes in the lower level structure is OK too since the root in the lower level structured remains the root regardless of where it was joined. This was covered in Section 6 Linking Below the Root. The derived structure is shown.

SQL 11.2: **SELECT * FROM CustView LEFT JOIN EmpView**
ON EaddrID=AddrID AND CustID=EmpCustID

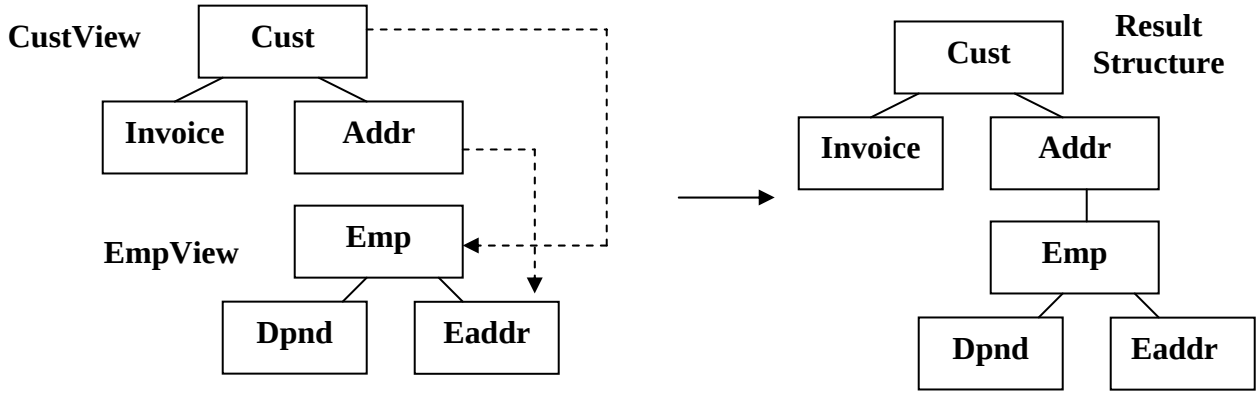


Figure 11.2 AND conditions are usually OK

11.3) ON Clause AND Conditions Can Still Cause Ambiguous Structures

AND and OR conditions can cause network structures when used on different upper level structure nodes. This is shown in Figure 11.3 with upper level references to Invoice and Addr in SQL 11.3.

SQL 11.3: **SELECT * FROM CustView LEFT JOIN EmpView**
ON EaddrID=AddrID AND InvCustID=EmpCustID

Semantic Error: Invalid network structure at or near InvCustID=EmpCustID

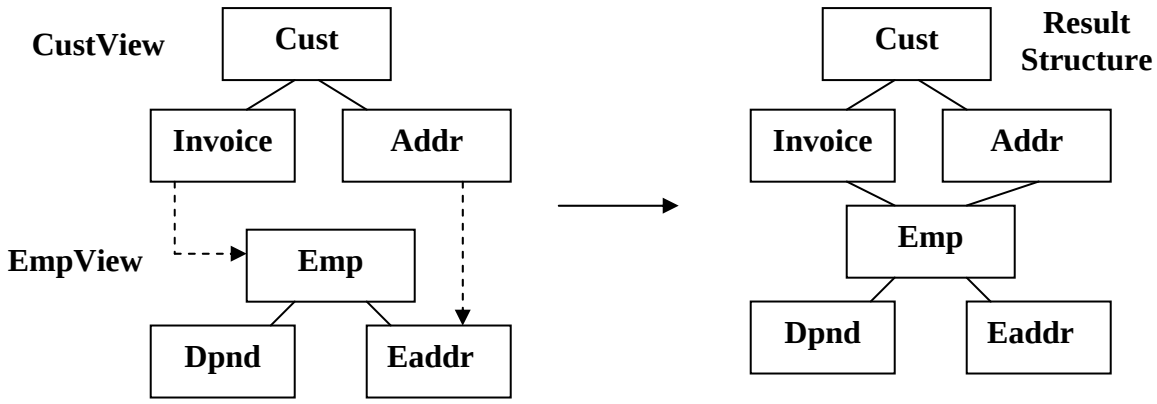


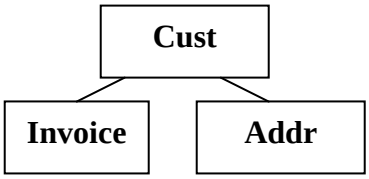
Figure 11.3 AND conditions can still cause invalid network structures

12) XML Input and Output

XML Input accesses XML efficiently hierarchically without simulating Left Outer Joins. XML support includes Attribute and Element formatted data input and output. SQLfX® can input and output mixed content. Mixed Content refers to XML documents that contain string data with attributes. The XML view definition for CustViewX is shown below. It contains Mixed Data where the string data is identified as ANY. The fields that will be identified with this string data are CustText, InvText and AddrText.

```
CREATE XML CustViewX
Cust(
  CustID Char(8),
  CustStoreID Char(8),
  CustText Char(100) ANY),
Invoice(
  InvID Char(8),
  InvCustID Char(8),
  InvStatus Char(8),
  InvText Char(100) ANY) Parent Cust,
Addr(
  AddrID Char(8),
  AddrCustID Char(8),
  AddrState Char(8),
  AddrText Char(100) ANY) Parent Cust
```

CustViewX



12.1) Mixed Content Input

String data can contain sub element tags intermixed within the string data for a particular node. This is shown in the sample CustViewX XML mixed content document below in. On retrieval by SQLfX®, the intermixed string data is concatenated. Cust01 contains a string comment at the Cust node level that spans its sub elements. Cust03 also has a string comment that spans its Cust node, and it also has an Addr string text inside the Cust node and needs to be kept separate from the Cust node when retrieved.

XML 12.1: **Sample CustViewX XML document With Mixed Input**

```
<root>
 <cust custid="Cust01" custstoreid="Store01"> Comment One,
  <invoice invid="Inv01" invcustid="Cust01" invstatus="P"/> Comment Two,
  <invoice invid="Inv02" invcustid="Cust01" invstatus="O"/> Comment Three,
  <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/> Comment Four
 </cust>
 <cust custid="Cust02" custstoreid="Store01">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O"/>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"/>
 </cust>
 <cust custid="Cust03" custstoreid="Store01"> Comment Five,
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV"> This is addr text
</addr> Comment Six
 </cust>
</root>
```


This XML saved in XML 12.1 is loaded by:
Load xml custviewx from 'file:examples/sqlfx/XML 12.1'

12.2) String Input Data Output as Mixed Content

For XML string data output in Mixed mode, the data is written out as Element string data. SQLfX® concatenates string data on input and places it in a single field. This is the way it is output in SQL 12.2.

SQL 12.2: **SELECT * FROM CustViewX FOR XML Mixed**

```
<cust custid="Cust01" custstoreid="Store01"> Comment One, Comment Two,
    Comment Three, Comment Four
 <invoice invid="Inv01" invcustid="Cust01" invstatus="P"></invoice>
 <invoice invid="Inv02" invcustid="Cust01" invstatus="O"></invoice>
 <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"></addr>
</cust>
<cust custid="Cust02" custstoreid="Store01">
 <invoice invid="Inv03" invcustid="Cust02" invstatus="O"></invoice>
 <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"></addr>
 <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"></addr>
</cust>
<cust custid="Cust03" custstoreid="Store01"> Comment Five, Comment Six
 <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV">This is addr
  text</addr>
</cust>
```

12.3) String Input Data Output as Attribute Formatted

XML string input can also be output in Attribute or Element format. Attribute format is shown below in SQL12.3. The string data is placed in an attribute using the string data's assigned name (CustText and AddrText) as defined in the XML view. Element Format not shown is also handled the same way with text data identified between its own XML tag names (CustText and AddrText).

SQL 12.3: **SELECT * FROM CustViewX FOR XML Attribute**

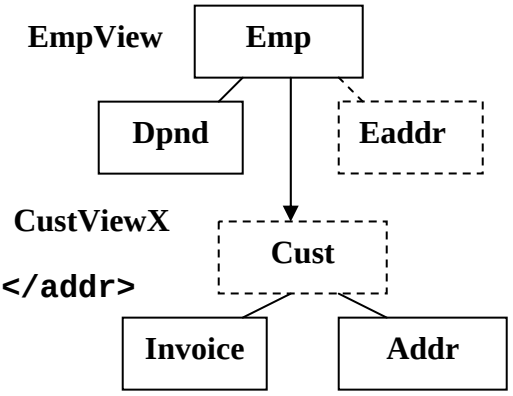
```
 <cust custid="Cust01" custstoreid="Store01" custtext="Comment One,
   Comment Two, Comment Three, Comment Four">
  <invoice invid="Inv01" invcustid="Cust01" invstatus="P" invtext=""/>
  <invoice invid="Inv02" invcustid="Cust01" invstatus="O" invtext=""/>
  <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA" addrtext=""/>
 </cust>
 <cust custid="Cust02" custstoreid="Store01" custtext="">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O" invtext=""/>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA" addrtext=""/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA" addrtext=""/>
 </cust>
 <cust custid="Cust03" custstoreid="Store01" custtext="Comment Five,
   Comment Six">
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV" addrtext="This is
    addr text"/>
 </cust>
```


12.4) Relational/XML Heterogeneous Example 1

Using our mixed XML document CustViewX, we will perform a heterogeneous data test placing our relational EmpView view over it to perform a heterogeneous complex join easily and transparently. The two data types integrate seamlessly. This is the same SQL request as in the SQL 5.1 example and produces the identical result except for the AddrText field added to CustViewX to test its XML Element string (mixed content) feature of XML. The Element string was given the name of AddrText which is not displayed in this example because it is formatted in Mixed Content output format shown in SQL12.4.

SQL 12.4: **SELECT EmpID, DpndID, InvID, AddrID, AddrText**
FROM EmpView LEFT JOIN CustViewX ON EmpCustID=CustID FOR XML Mixed

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

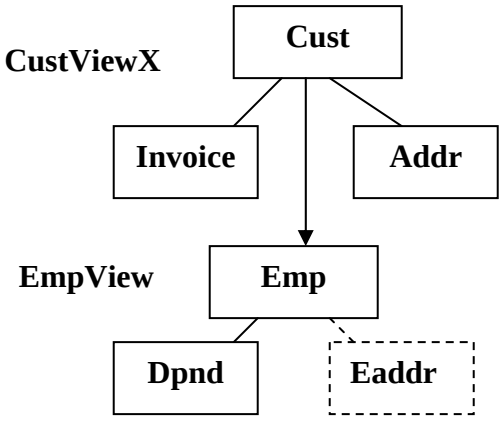


12.5) Relational/XML Heterogeneous Example 2

In a second heterogeneous Mixed Content test in 12.5, let's reverse the join and place the XML CustViewX on top of the relational EmpView. As a further test, let's display the result in XML Attribute format. Notice that the AddrText has been changed from string data to an attribute and placed correctly.

SQL 12.5: **SELECT AddrText, AddrID, CustID, Empid, DpndID, InvID**
FROM CustViewX LEFT JOIN EmpView ON EmpCustID=CustID FOR XML Attribute

```
<cust custid="Cust01">
 <invoice invid="Inv01"/>
 <invoice invid="Inv02"/>
 <addr addrtext="" addrid="Addr01"/>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
 </emp>
</cust>
<cust custid="Cust02">
 <invoice invid="Inv03"/>
 <addr addrtext="" addrid="Addr02"/>
 <addr addrtext="" addrid="Addr04"/>
</cust>
<cust custid="Cust03">
 <addr addrtext="This is addr text" addrid="Addr03"/>
 <emp empid="Emp02">
 </emp>
</cust>
```

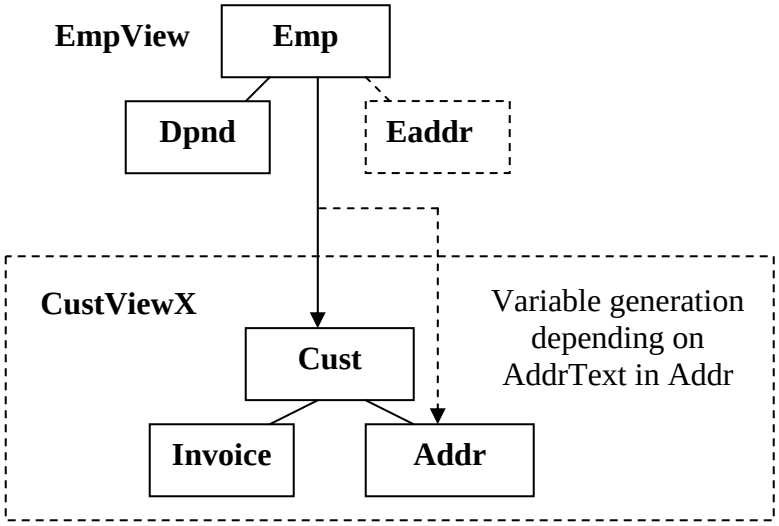


12.6) Look Ahead With "Like" Operation on String Mixed Data

This is a query where the Employee view is on top of the Customer view. An ON condition search string has been added to the lower level XML Customer view to search out mixed mode XML text data that matches data in a particular search criteria. If matched, the XML view occurrence will be included in the data returned otherwise it will be excluded without affecting the Employee view portion of the Query.

What is really special with this query is that the XML text being searched is within the lower level view being searched before being joined. This is allowed, as explained earlier in Section 6.3, because lower level views are materialized before being joined to the upper structure. This means the lower level view has its data accessible and defined before being joined. This has a very powerful feature for XML data allowing the same text to be searched and saved as the text tested. This is comparable to testing data before it has been access an already saved. This means the XML view portion is only generated when it has a certain matching text. This is a variable generation based on a look-ahead data condition further down the path than the current processing location. Compare this result from SQL 12.6 to the XML result example in 12.4 to see the data that is missing.

SQL 12.6: **SELECT CustID, EmpID, DpndID, InvID, AddrID, AddrText**
FROM EmpView LEFT JOIN CustViewX ON EmpCustID=CustID
AND AddrText LIKE '%addr%' FOR XML Mixed




```
<emp empid="Emp01">
 <dpnd dpndid="Dpnd01"/>
</emp>
<emp empid="Emp02">
 <cust custid="Cust03">
  <addr addrid="Addr03">This is addr text</addr>
 </cust>
</emp>
```


12.7) XML Content: Element, Attribute and Mixed

Mixed mode content was successfully tested in 12.1 which is actually a combination of Attribute mode and element discontinuous string data that can be interspersed between Element tags at its same level. Let's now test out inputting XML in Element mode content. We will use the Mixed mode XML 12.1 retrieved into CustViewX by SQL12.1 as the source data. To do this we will output the XML using SQLfX® to output Element Content Mode. But we will also change the natural key ordering of CustID, InvID, and AddrID to descending so that we can also run some tests on XML's ability to remain ordered. This is tested in SQL 12.7 below by seeing if the XML remains in descending order instead of being reordered to ascending by natural internal processing.

SQL12.7: **SELECT * FROM CustViewX Order BY CustID Desc, InvID Desc, AddrID Desc FOR XML Element**

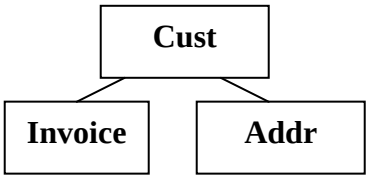
XML 12.7 Produced XML Element Mode Format with Ordering Descended

```
<cust>
 <custid>Cust03</custid>
 <custstoreid>Store01</custstoreid>
 <custtext>Comment Five,
   Comment Six</custtext>
 <addr>
  <addrid>Addr03</addrid>
  <addrcustid>Cust03</addrcustid>
  <addrstate>NV</addrstate>
  <addrtext>This is addr text
    </addrtext>
 </addr>
</cust>
<cust>
 <custid>Cust02</custid>
 <custstoreid>Store01</custstoreid>
 <custtext/>
 <invoice>
  <invid>Inv03</invid>
  <invcustid>Cust02</invcustid>
  <invstatus>O</invstatus>
  <invtext/>
 </invoice>
 <addr>
  <addrid>Addr04</addrid>
  <addrcustid>Cust02</addrcustid>
  <addrstate>CA</addrstate>
  <addrtext/>
 </addr>
 <addr>
  <addrid>Addr02</addrid>
  <addrcustid>Cust02</addrcustid>
  <addrstate>CA</addrstate>
  <addrtext/>
 </addr>
</cust>
<cust>
 <custid>Cust01</custid>
 <custstoreid>Store01</custstoreid>
 <custtext>Comment One,
   Comment Two,
   Comment Three,
   Comment Four</custtext>
 <invoice>
  <invid>Inv02</invid>
  <invcustid>Cust01</invcustid>
  <invstatus>O</invstatus>
  <invtext/>
 </invoice>
 <invoice>
  <invid>Inv01</invid>
  <invcustid>Cust01</invcustid>
  <invstatus>P</invstatus>
  <invtext/>
 </invoice>
 <addr>
  <addrid>Addr01</addrid>
  <addrcustid>Cust01</addrcustid>
  <addrstate>CA</addrstate>
  <addrtext/>
 </addr>
</cust>
```


Now Create CustViewE:

```
CREATE XML CustViewE
Cust(
  CustID Char(8),
  CustStoreID Char(8),
  CustText Char(100) ANY),
Invoice(
  InvID Char(8),
  InvCustID Char(8),
  InvStatus Char(8),
  InvText Char(100) ANY) Parent Cust,
Addr(
  AddrID Char(8),
  AddrCustID Char(8),
  AddrState Char(8),
  AddrText Char(100) ANY) Parent Cust
```

CustViewE



The XML saved in XML 12.7 is loaded into CustViewE by:
Load xml custviewe from 'file:examples/sqlfx/XML 12.7'

Output CustViewE in Element, Attribute, and Mixed Mode Content Formats

All three XML output mode formats are mapped identically on input and their output values should be the same formatted in any of the three XML mode output formats. These are all output correctly based on the Element mode formatted CustViewE input and their Descending input order was preserved as XML data should be.

12.7.1) XML Element Mode Output in its Natural Descending Order

SQL 12.7.1: **SELECT * FROM CustViewE FOR XML Element**

```
<cust>
 <custid>Cust03</custid>
 <custstoreid>Store01</custstoreid>
 <custtext>Comment Five,
   Comment Six</custtext>
 <addr>
  <addrid>Addr03</addrid>
  <addrcustid>Cust03</addrcustid>
  <addrstate>NV</addrstate>
  <addrtext>This is addr
text</addrtext>
 </addr>
</cust>
<cust>
 <custid>Cust02</custid>
 <custstoreid>Store01</custstoreid>
 <custtext/>
 <invoice>
  <invid>Inv03</invid>
  <invcustid>Cust02</invcustid>
  <invstatus>O</invstatus>
  <invtext/>
 </invoice>
 <addr>
  <addrid>Addr04</addrid>
  <addrcustid>Cust02</addrcustid>
  <addrstate>CA</addrstate>
  <addrtext/>
 </addr>
 <addr>
  <addrid>Addr02</addrid>
  <addrcustid>Cust02</addrcustid>
  <addrstate>CA</addrstate>
  <addrtext/>
 </addr>
</cust>
<cust>
 <custid>Cust01</custid>
```



```
 <custstoreid>Store01</custstoreid>
 <custtext>Comment One,
   Comment Two,
   Comment Three,
   Comment Four</custtext>
 <invoice>
  <invid>Inv02</invid>
  <invcustid>Cust01</invcustid>
  <invstatus>O</invstatus>
  <invtext/>
 </invoice>
 <invoice>
  <invid>Inv01</invid>
  <invcustid>Cust01</invcustid>
  <invstatus>P</invstatus>
  <invtext/>
 </invoice>
 <addr>
  <addrid>Addr01</addrid>
  <addrcustid>Cust01</addrcustid>
  <addrstate>CA</addrstate>
  <addrtext/>
 </addr>
</cust>
```

12.7.2) XML Attribute Mode Output in its Natural Descending Order

SQL 12.7.2: **SELECT * FROM CustViewE FOR XML Attribute**

```
<cust custid="Cust03" custstoreid="Store01" custtext="Comment Five,
   Comment Six">
 <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV" addrtext="This is addr
  text"/>
</cust>
<cust custid="Cust02" custstoreid="Store01" custtext="">
 <invoice invid="Inv03" invcustid="Cust02" invstatus="O" invtext=""/>
 <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA" addrtext=""/>
 <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA" addrtext=""/>
</cust>
<cust custid="Cust01" custstoreid="Store01" custtext="Comment One,
   Comment Two,
   Comment Three,
   Comment Four">
 <invoice invid="Inv02" invcustid="Cust01" invstatus="O" invtext=""/>
 <invoice invid="Inv01" invcustid="Cust01" invstatus="P" invtext=""/>
 <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA" addrtext=""/>
</cust>
```

12.7.3) XML Mixed Mode Output in its Natural Descending Order

SQL 12.7.3: **SELECT * FROM CustViewE FOR XML Mixed**

```
<cust custid="Cust03" custstoreid="Store01">
 Comment Five,
   Comment Six
 <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV">This is addr text</addr>
</cust>
<cust custid="Cust02" custstoreid="Store01">

 <invoice invid="Inv03" invcustid="Cust02" invstatus="O"></invoice>
 <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"></addr>
 <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"></addr>
</cust>
<cust custid="Cust01" custstoreid="Store01">
 Comment One,
   Comment Two,
   Comment Three,
   Comment Four
```



```
 <invoice invid="Inv02" invcustid="Cust01" invstatus="O"></invoice>
 <invoice invid="Inv01" invcustid="Cust01" invstatus="P"></invoice>
 <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"></addr>
</cust>
```

12.8) XML Content Order Preservation Test

XML Content order preservation needs to operate correctly with the internal operations of SQLfX® that rely on ordered of data. This testing is done here by first joining CustViewE over EmpView and testing it, and then joining CustViewE under EmpView and testing it. These examples will introduce a lot of internal data replications that need to be controlled accurately, and XML ordering involves addition internal methods that need to work with the replication processing logic. The XML input data order should remain the same. The results were found correct.

12.8.1) Output CustViewE Over Empview

SQL 12.8.1: **SELECT * FROM CustViewE LEFT JOIN EmpView ON CustID=EmpCustID**

```
<root>
 <cust custid="Cust03" custstoreid="Store01" custtext="Comment Five,
   Comment Six">
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV" addrtext="This is addr
   text"/>
  <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
   <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
  </emp>
 </cust>
 <cust custid="Cust02" custstoreid="Store01" custtext="">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O" invtext=""/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA" addrtext=""/>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA" addrtext=""/>
 </cust>
 <cust custid="Cust01" custstoreid="Store01" custtext="Comment One,
   Comment Two,
   Comment Three,
   Comment Four">
  <invoice invid="Inv02" invcustid="Cust01" invstatus="O" invtext=""/>
  <invoice invid="Inv01" invcustid="Cust01" invstatus="P" invtext=""/>
  <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA" addrtext=""/>
  <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
   <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
   <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  </emp>
 </cust>
</root>
```

12.8.2) Output CustViewE Under EmpView

SQL 12.8.2: **SELECT * FROM EmpView LEFT JOIN CustViewE ON CustID=EmpCustID**

```
<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" coststoreid="Store01" custtext="Comment One,
   Comment Two,
   Comment Three,
   Comment Four">
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O" invtext=""/>
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P" invtext=""/>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA" addrtext=""/>
  </cust>
 </emp>
 <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
  <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
  <cust custid="Cust03" coststoreid="Store01" custtext="Comment Five,
   Comment Six">
   <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV" addrtext="This is addr
    text"/>
  </cust>
 </emp>
</root>
```

12.9) XML Order Preservation with use of ORDER BY

XML Content order preservation also needs to operate correctly with the External ORDER BY operations that can selectively override XML's default ordering. Our ORDER BY testing performed here is by first overriding a parent and one child node combination of CustViewE, and then by overriding just the sibling child nodes of CustViewE. Not all of the XML default data order is changed and it must remain operating as before. So the combination of XML default ordering and explicitly supplied ORDER BY should work together producing the desired ordering and they do.

12.9.1) Explicitly Ordered CustID and InvID Parent and Child Node Combination

Explicitly order CustID and InvID parent/child combination ascending and leave other child AddrID naturally ordered descending using SQL 12.9.1.

SQL 12.9.1: **SELECT * FROM CustViewE ORDER BY CustID Asc, InvID Asc**

```
root>
 <cust custid="Cust01" coststoreid="Store01" custtext="Comment One,
   Comment Two,
   Comment Three,
   Comment Four">
  <invoice invid="Inv01" invcustid="Cust01" invstatus="P" invtext=""/>
  <invoice invid="Inv02" invcustid="Cust01" invstatus="O" invtext=""/>
  <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA" addrtext=""/>
 </cust>
 <cust custid="Cust02" coststoreid="Store01" custtext="">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O" invtext=""/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA" addrtext=""/>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA" addrtext=""/>
 </cust>
 <cust custid="Cust03" coststoreid="Store01" custtext="Comment Five,
   Comment Six">
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV" addrtext="This is addr
   text"/>
 </cust>
</root>
```


12.9.2) Explicitly Order InvID and AddrID Sibling Child Node Combinations

Explicitly order InvID and AddrID siblings ascending and leave parent CustID naturally ordered descending in SQL 12.9.2.

SQL 12.9.2: **SELECT * FROM CustViewE ORDER BY InvID Asc, AddrID Asc**

```
root>
 <cust custid="Cust03" coststoreid="Store01" custtext="Comment Five,
   Comment Six">
  <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV" addrtext="This is addr
   text"/>
 </cust>
 <cust custid="Cust02" coststoreid="Store01" custtext="">
  <invoice invid="Inv03" invcustid="Cust02" invstatus="O" invtext=""/>
  <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA" addrtext=""/>
  <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA" addrtext=""/>
 </cust>
 <cust custid="Cust01" coststoreid="Store01" custtext="Comment One,
   Comment Two,
   Comment Three,
   Comment Four">
  <invoice invid="Inv01" invcustid="Cust01" invstatus="P" invtext=""/>
  <invoice invid="Inv02" invcustid="Cust01" invstatus="O" invtext=""/>
  <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA" addrtext=""/>
 </cust>
</root>
```

Recap of XML Input and Output

XML Shredding	The processes of shredding XML into the SQL database and then operated on fully hierarchically along with the standard relational data works as designed.
Attribute Content Mode	Input and output of Attributed mode content format works
Element Content Mode	Input and Output of Element mode content format works
Mixed Content Mode	Input and Output of Mixed mode Content format works
Any Content Input to Any Content Output	Output Content mode is not depended on input content. Relational data can also be output as any XML output mode content type.
XML Natural Order Maintained	XML's Input data order is maintained throughout processing unless overridden by the Order By operation.
Heterogeneous Support	ALL input data formats including relational data are hierarchically mapped and processed naturally in ANSI SQL producing seamless heterogeneous support.
Native XML Support	Native XML in the form of realtime EII will be supported transparently just as XML shredding is in our initial product release. Both can be supported together.

Table 12: XML Input and Output Capabilities

13) Association Tables and M to M Relationship Usage
13.1) Creating Many to Many Hierarchical Structures

This will be an example of how two unrelated structures can be joined. Unrelated structures are structures that do not contain any data relationships used for direct joining, but they do contain relationships that are not related directly by existing data. For example, if we assume the CustView and EmpView have no direct relationships in their data values, we could relate them through a simple relational association table that contains these relationships. In addition, an advantage of this association table is that M to M relationships like Parts and Supplies can be defined. This also allows for the addition of intersecting data to be stored in the association table that can be different for each specific match relationships, such as a Part price for a specific Supplier when a part can have multiple suppliers and multiple suppliers can have the same part.

This example will demonstrate creating an external Association table CustEmpAssoc. It can be used to relate CustView over EmpView and visa versa. In our example below in Figure 13.1 CustView and EmpView can be relational or XML structures.

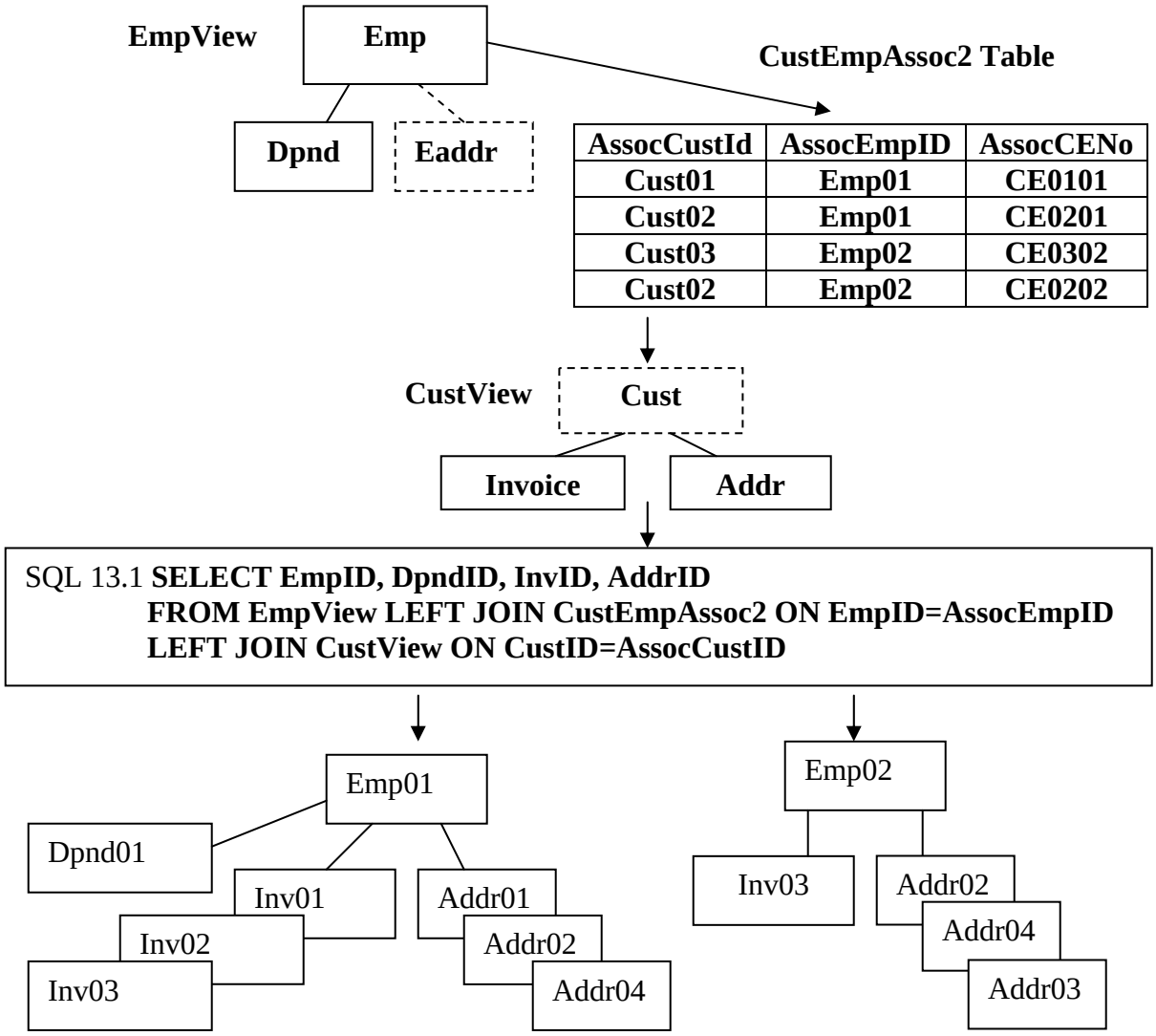
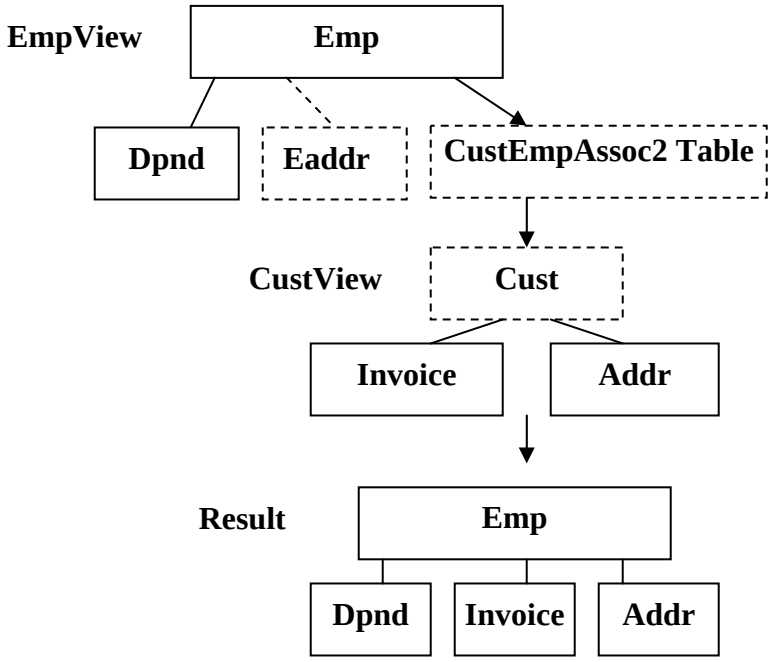


Figure 13.1 Join using Association table

With the CustEmp Association table in Figure 13.1 above, Emp01 is associated with two Customers and causes Emp01 to take on both of their information as shown in the XML below and Figure 13.1 above. And since the Customer node is not selected for output, its information is directly associated with Employee because of node promotion. Association tables usually remain invisible as in this SQL 13.1 example and their effect is the same as if the linking data values where in the joined structures.

SQL 13.1: **SELECT EmpID, DpndID, InvID, AddrID**
FROM EmpView LEFT JOIN CustEmpAssoc2 ON EmpID=AssocEmpID
LEFT JOIN CustView ON CustID=AssocCustID

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <invoice invid="Inv03"/>
  <addr addrid="Addr01"/>
  <addr addrid="Addr02"/>
  <addr addrid="Addr04"/>
 </emp>
 <emp empid="Emp02">
  <invoice invid="Inv03"/>
  <addr addrid="Addr02"/>
  <addr addrid="Addr04"/>
  <addr addrid="Addr03"/>
 </emp>
</root>
```



13.2) Including Intersecting Data in the XML Result

In the previous example SQL 13.1, the Association table was not selected for output, so its associated node was excluded from the XML result. To include the intersecting data (AssocCENo) in the result it is selected for output as in SQL 13.2 below. Notice how the node of the intersecting data properly encompasses the Invoice and Address nodes to which it applies to in the XML result represented below and in Figure 13.2 and the SQL 13.2 result.

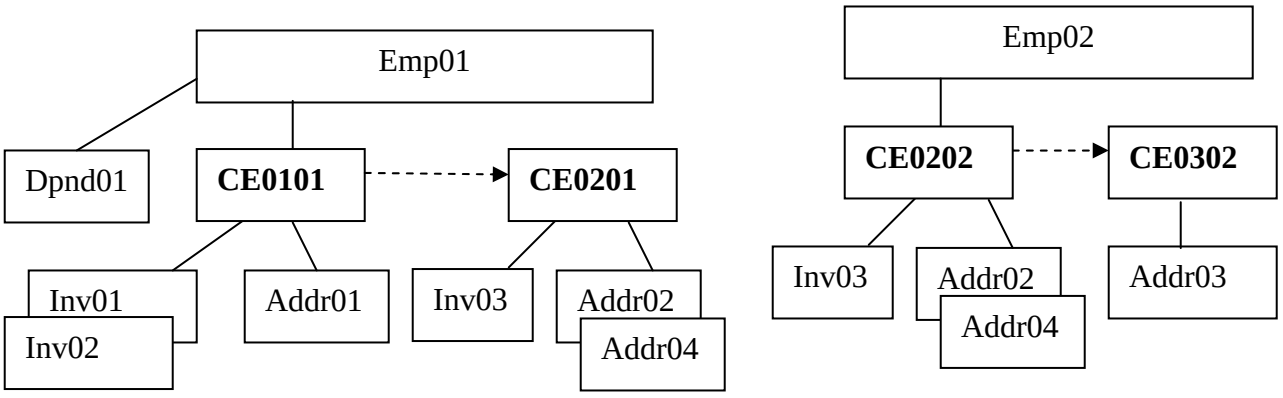
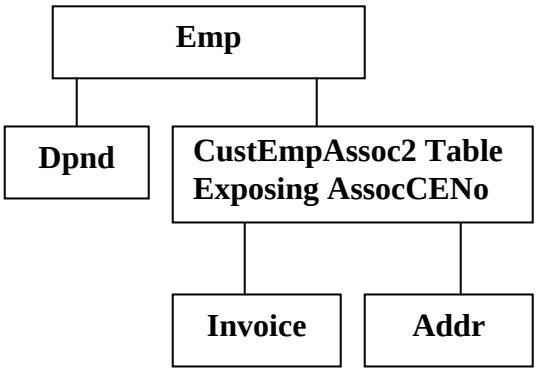


Figure 13.2 Selecting intersecting data

SQL 13.2: **SELECT EmpID, DpndID, InvID, AddrID, AssocCENo**
FROM EmpView LEFT JOIN CustEmpAssoc2 ON EmpID=AssocEmpID
LEFT JOIN CustView ON CustID=AssocCustID

```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <custempassoc2 assoceno="CE0101">
   <invoice invid="Inv01"/>
   <invoice invid="Inv02"/>
   <addr addrid="Addr01"/>
  </custempassoc2>
  <custempassoc2 assoceno="CE0201">
   <invoice invid="Inv03"/>
   <addr addrid="Addr02"/>
   <addr addrid="Addr04"/>
  </custempassoc2>
 </emp>
 <emp empid="Emp02">
  <custempassoc2 assoceno="CE0202">
   <invoice invid="Inv03"/>
   <addr addrid="Addr02"/>
   <addr addrid="Addr04"/>
  </custempassoc>
  <custempassoc2 assocempid="CE0302">
   <addr addrid="Addr03"/>
  </custempassoc2>
 </emp>
</root>
```

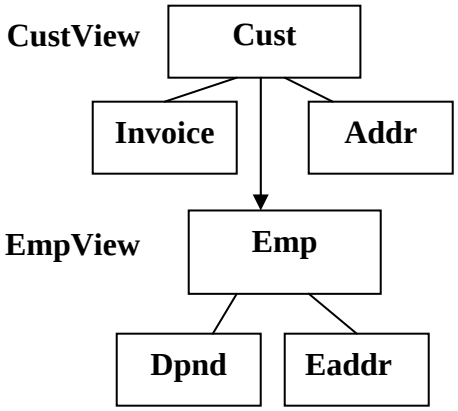


13.3) Reversing the Association Table

Let's reverse the structure using the Association table and put the CustView over the EmpView. This is shown in SQL 13.3 with its XML structured below it. We have added CustID to the SELECT list to demonstrate the full structure better, and removed the Association table from being output because it served its purpose to relate the structures transparently. You will notice that the M to M characteristics of the association table worked by treating this as a 1 to M association like the reverse structures above were also.

SQL 13.3: **SELECT CustID, EmpID, DpndID, InvID, AddrID**
FROM CustView LEFT JOIN CustEmpAssoc2 ON CustID=AssocCustID
LEFT JOIN EmpView ON EmpID=AssocEmpID


```
<root>
 <cust custid="Cust01">
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <addr addrid="Addr01"/>
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
  </emp>
 </cust>
 <cust custid="Cust02">
  <invoice invid="Inv03"/>
  <addr addrid="Addr02"/>
  <addr addrid="Addr04"/>
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
  </emp>
  <emp empid="Emp02">
  </emp>
 </cust>
 <cust custid="Cust03">
  <addr addrid="Addr03"/>
  <emp empid="Emp02">
  </emp>
 </cust>
</root>
```



Recap of Association Table Capabilities

Create External Relationships	An association table allows relationships between structures to be externally created and maintained. This is very flexible and easy for maintenance. These association tables can be invisible.
Many to Many Relationships	Association table can also be used to define M to M relationships. This is where customers can have many employees and employees may work for more than one customer. This is not usually possible to maintain in a hierarchical database. This can be used in either direction to form a 1 to M relationship.
Intersecting Data	Many-to-many relationships like Customers and Employees can each have their own data at their specific intersection point. For example, employee X working for customer Y, employee X working for customer Z, Customer Z working for employee W. This specific intersecting data could be hourly wage for example and it could be stored easily in the association table with every unique combination Cust and EMP.

Table 13: Association Table Use Recap

SQLfX® Beta Structure Transformation Capability

Hierarchical Structure Transformation Introduction

The next three sections, 14, 15 and 16, describe a breakthrough ANSI SQL nonprocedural nonlinear hierarchical conceptual structure transformation capability that includes our breakthrough any-to-any structure transformations technology. We start with structure restructuring in Section 14, then structure reshaping in chapter 15, and then demonstrate how both restructuring and reshaping capabilities can be combined to accomplish any linear or nonlinear structure transformation possible and do it accurately.

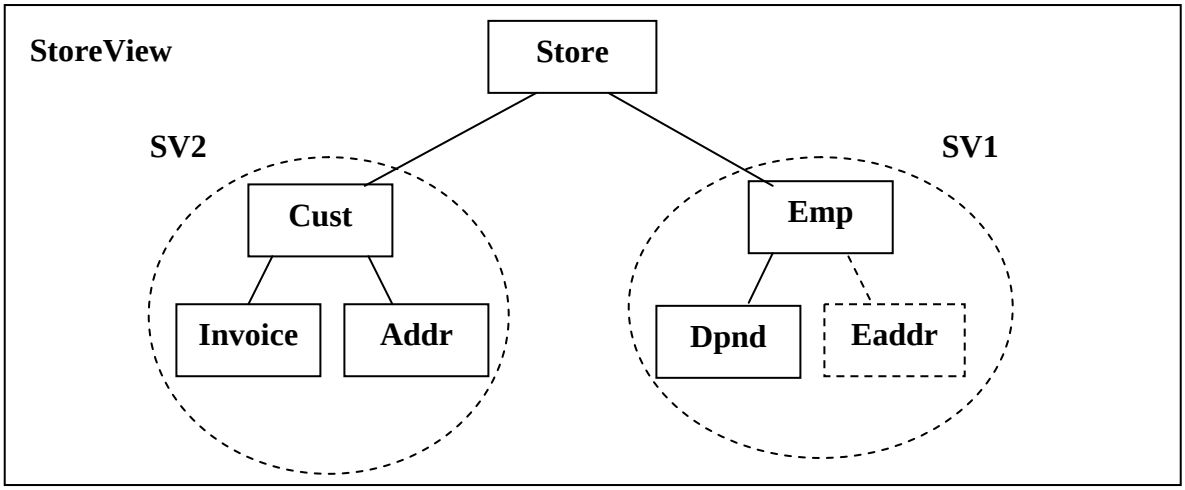
Today the structure transformation terminology of Restructuring and Reshaping are used interchangeably for XML structure transformation processes. There are two basic types of XML hierarchical structure transformations that need to be distinguished because they are different in meaning, results, and use. These are Restructuring controlled by existing relationships in the data, and Reshaping controlled by the semantics of the current data structure. Restructuring is performed by using new and unused relationships to restructure the data. On the other hand, Reshaping uses the semantics of the current structure to mold the structure into any other shape without requiring or relying on any relationships in the data.

Restructuring using data relationships can create a new structure and data with new semantics, while Reshaping using structure semantics alters the structure without changing the data and its semantics. Both have their use. Restructuring is usually used to match a structure to its application use, and Reshaping to map a structure to a desired structure format. Reshaping is also used in inverting structures when data relationships are not available in the data. This is often the case with XML data since foreign keys are not required because of their contiguous nested storage.

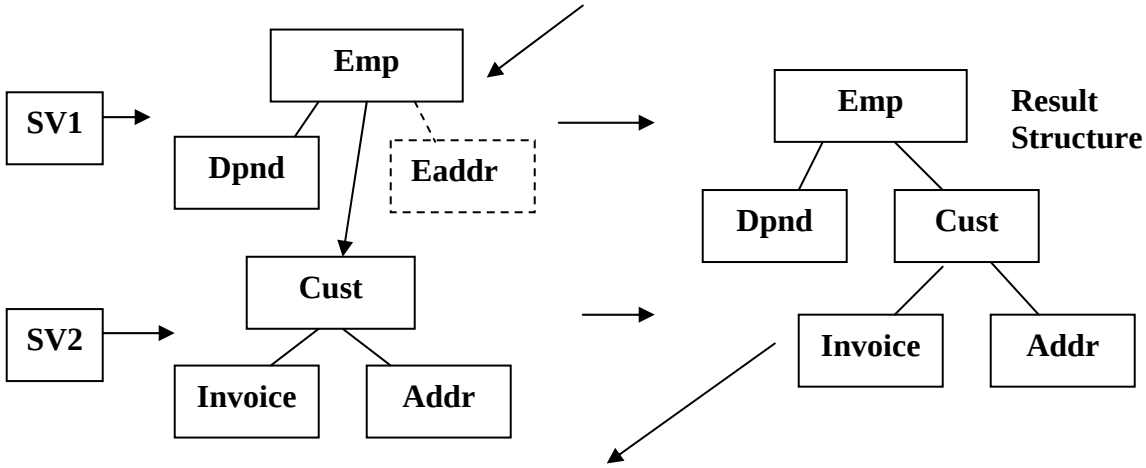
Up until now, even internally complex queries did not require much thought on how they need to be specified because their specification was straightforward. Structure transforms are still specified nonprocedural in standard SQL, but they do require some thought to how they are specified as you might expect for transformations. All transformation operations carried out will be performed semantically correct. This is because the operations are performed through the existing hierarchical structure strictly following the semantics in the structure. This aids the user specifying the transformations considerably. This makes it easier for the user to specify complex transformations without introducing any errors either from the user or the software.

14) Hierarchical Structure Restructuring Using Data Relationships

The following example in Figure 14.0 uses the StoreView to create a complex hierarchical structure and proceeds to transform it by isolating and manipulating structure fragments from the structure. A fragment is a related subset of a hierarchical structure that can be located from or below the root. SQL handles this naturally and automatically. It is controlled by what data fields are selected which in turn defines what nodes are selected from the structure. The natural operation of node promotion will cause the structure fragment to become contiguous enabling it to be easily manipulated and joined into the structure being constructed using available matching relationship values. This is done at a high hierarchical conceptual level new to SQL/XML processing. This is shown below in Figure 14.0 where the separate fragments are encircled in a dotted circle and is translated into SQL in SQL 14.1. If there are no required matching relationship values available, use Reshaping in Section 15 which operates using structure semantics.



SQL 14.1: **SELECT SV1.EmpID, SV1.DpndID, SV2.CustID, SV2.InvID, SV2.AddrID**
FROM StoreView SV1 LEFT JOIN StoreView SV2 ON SV1.EmpCustID=SV2.CustID



EmpID	DpndID	CustID	InvID	AddrID
Emp01	Dpnd01	Cust01	Inv01	Addr01
Emp01	Dpnd01	Cust01	Inv02	Addr01`
Emp02		Cust03		Addr03

Figure 14.0: Decomposition and Transformation

14.1) Basic Structure Restructuring

Hierarchical structure transformation described in this section is known as Structure Restructuring or simply as Restructuring. It is performed on physical or logical hierarchical structures by carving out separate fragments and reassembling them. This complex processing continues to be processed nonprocedurally. Pulling out separate fragments is possible by duplicating the structure so that it can be separately accessed to retrieve multiple fragments that can be independently manipulated, thereby transforming the original structure. Submit the SQL 14.1 statement below from Figure 14.0 to see the result which should resemble the structure and result found in previously in Figure 6. Notice how simple and intuitive this SQL query is to specify and the hierarchical power it demonstrates.

Decomposition and transformation of logical and physical data structures is possible with standard SQL using a combination of fragment processing and SQL alias processing that allows structure views to be accessed multiple times as shown above in Figure 14.0. It uses the StoreView view defined in Figure 2.1.2 as the single source of two separate fragments (encircled in a dotted circle in Figure 14.0) that reflect the data that comprises the CustView and EmpView. These fragments decompose the StoreView and then remodel it differently to duplicate the data model and data found previously in Figure 6. This remodeling operates by joining the structure fragments based on data value relationships in the structure.

While logical structures are free to be modeled in many different ways, physical structures must be modeled reflecting their actual structure. While the StoreView in the above example in Figure 14.0 is modeling a logical structure, it could also be a fixed structure for this example. Both are represented identically in the rowset. This example demonstrates that a physical structure can be rearranged by separately selecting fixed fragments from the rowset containing it which can be independently joined into the main structure (unified view) with the flexibility offered by logical structures. The alias feature gives a new table or view name to an object and in the SELECT list clause you need to use this new object name as a high level prefix for the data items in the different renamed objects to specify which renamed object you a referring to, since they have the same name in both objects. This is shown in Figure 14.0 above.

SQL 14.1: **SELECT SV1.EmpID, SV1.DpndID, SV2.CustID, SV2.InvID, SV2.AddrID**
FROM StoreView SV1 LEFT JOIN StoreView SV2 ON SV1.EmpCustID=SV2.CustID

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

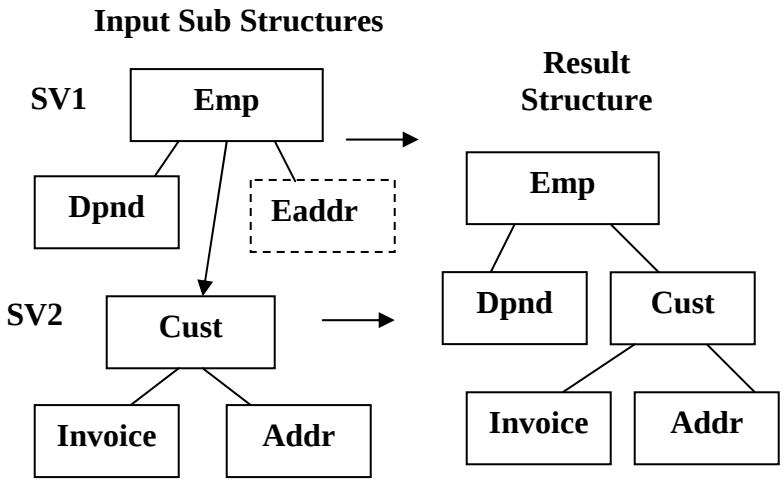


Figure 14.1 Structure and XML result of restructuring

14.2) Using Alias and Structure Restructuring In a View

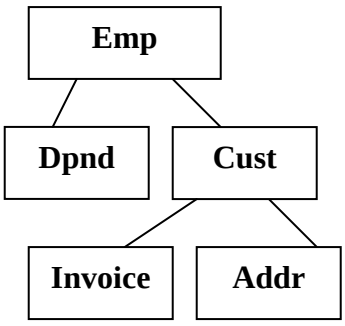
This example demonstrates a number of SQL capabilities applied to XML Transform processing. First, the Restructuring SQL query performed in SQL 14.1 above has been placed in an SQL view for abstraction and invoked from its view to demonstrate its easy flexible abstraction use. Second, the column names InvID and AddrID have been assigned the aliases Invoice and Address which are reflected in the generated XML from SQL 14.2.

CREATE View Transform AS
SELECT SV1.EmpID EmpID, SV1.DpndID DpndID, SV2.CustID CustID, SV2.InvID Invoice,
SV2.AddrID Address
FROM StoreView SV1 LEFT JOIN StoreView SV2 ON SV1.EmpCustID=SV2.CustID

SQL14.2: **SELECT * FROM Transform**

```
<root>
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
   <cust custid="Cust01">
    <invoice invoice="Inv01"/>
    <invoice invoice="Inv02"/>
    <addr address="Addr01"/>
   </cust>
  </emp>
  <emp empid="Emp02">
    <cust custid="Cust02">
    <addr address="Addr03"/>
   </cust>
  </emp>
 </root>
```

Transform View Structure

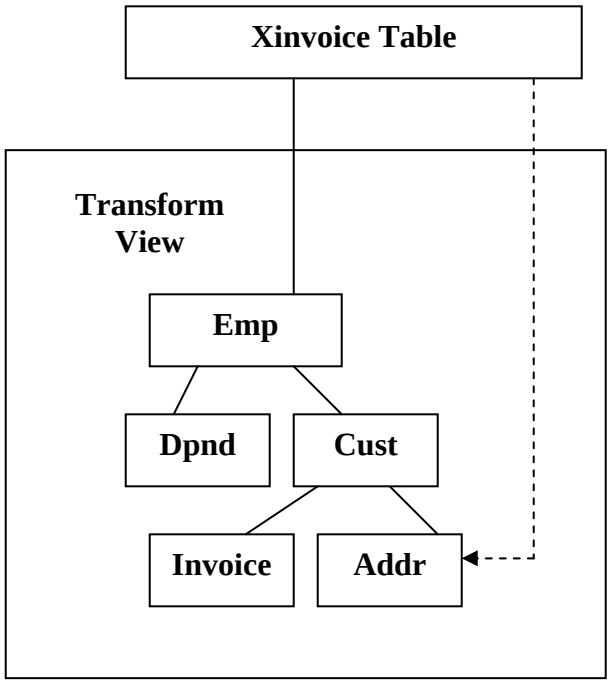


14.2.1) Using a Restructuring View in a Join

In this example the Restructuring view has been joined to the Invoice table to further test the transform being joined and show the proper replication of Employee information caused by have the Invoice table joined on top of it. The joined Invoice table has been renamed on the invoking SQL 14.2.1 statement so that it is not confused with the Invoice table in the Transform view.

SQL14.2.1: **SELECT InvID, EmpID, CustID, Address, Invoice**
FROM Invoice Xinvoice LEFT JOIN Transform ON Invid=Invoice


```
<root>
 <xinvoice invid="Inv01">
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
   <cust custid="Cust01">
    <invoice invoice="Inv01"/>
    <addr address="Addr01"/>
   </cust>
  </emp>
 </xinvoice>
 <xinvoice invid="Inv02">
  <emp empid="Emp01">
   <dpnd dpndid="Dpnd01"/>
   <cust custid="Cust01">
    <invoice invoice="Inv02"/>
    <addr address="Addr01"/>
   </cust>
  </emp>
 </xinvoice>
 <xinvoice invid="Inv03">
 </xinvoice>
</root>
```



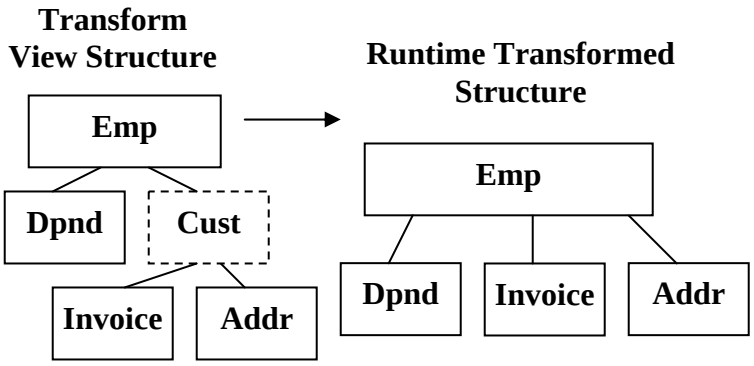
The above SQL 14.2.1 transformation view worked perfectly. The Xinvoice table joined over the Transform view has encompassed it at a higher level. Because Emp02 has no invoice it has been excluded. The joining logic follows the rules described earlier for joining below the root. Currently, alias names only carry over to the XML output when used with single nodes and not views as shown above.

14.2.2) Runtime Restructuring View Modification

Most XML Transformations are static and have to be modified if the input is changed by including or excluding a value in the output XML structure. The SQLfX® transform does not need the use of statically placed XML formatting functions in the Select list, allowing simple data item inclusion or exclusion as in the following SQL 14.2.2 example which does not Select the Cust node causing Cust to be removed and Invoice and Addr nodes to be node promoted around them, changing the already transformed structure.

SQL 14.2.2: **SELECT EmpID, DpndID, Invoice, Address FROM Transform**

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

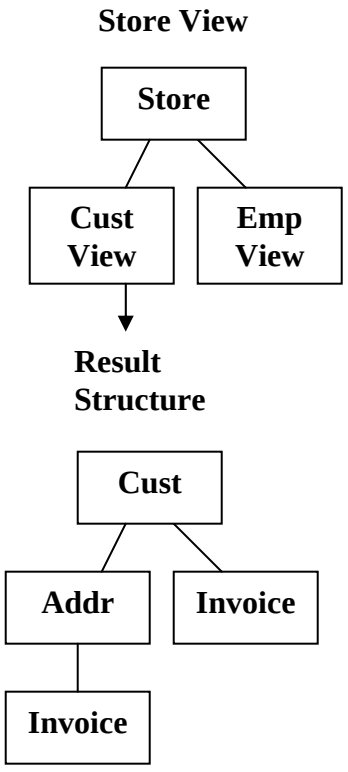


14.3) Changing Leg Order and Replicating Nodes

The SQL 14.3 code below, takes the StoreView data structure shown in Figure 2.1 in Part I with its full structure, and transforms it in a number of ways. The Store view used below in SQL14.3 is composed of two views, CustView and EmpView. This transform operation breaks the CustView out of the StoreView by isolating each of its nodes. Having done this, it reassembles Cust view and deliberately swaps its Addr and Invoice legs around by controlling the order the joins are performed. The replicated data caused by the combination of the EmpView and Custview data will be automatically removed by SQLfX®. In addition, an additional Invoice node is desired, and placed under the Addr node.

SQL 14.3: **SELECT Cust.custid, Invoice.invid, Invoice.invcustid, Addr.addrid,**
Addr.addrcustid, invoice2.invid NewInv
FROM Storeview Cust
LEFT JOIN StoreView Addr ON Cust.custid=Addr.addrcustid
LEFT JOIN StoreView Invoice ON Cust.custid=Invoice.invcustid
LEFT JOIN StoreView Invoice2 ON Addr.addrcustid=Invoice2.invcustid

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



The previous restructuring examples, SQL 14.1 and SQL 14.2, used fragments to move data around while this example operated at the node level. For complete control, node level works well, but fragments are easier and also offers an important capability that node level does not. Often nodes in a group are already in the structure desired and may not still contain the values (i.e. foreign keys) to enable rejoining. Fragments offer the solution by being able to be moved as a single group that remains intact.

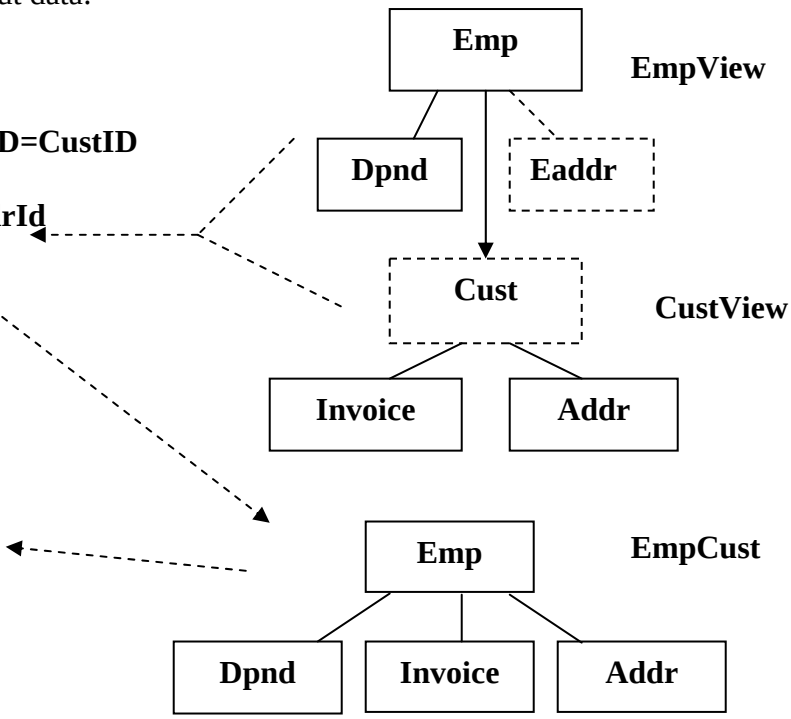
14.4) Restructuring Produces Properly Replicated Data

This example will create a new structure view, EmpCust, composed using CustView and EmpView again, only more tightly integrated. This transform example will take one of the lower nodes, Invoice, and make it the new root. First let's look at the input data:

```
CREATE VIEW EmpCust AS
SELECT * FROM Empview
LEFT JOIN CustView ON EmpCustID=CustID

SELECT EmpId, DpndId, InvId, AddrId
FROM EmpCust
```

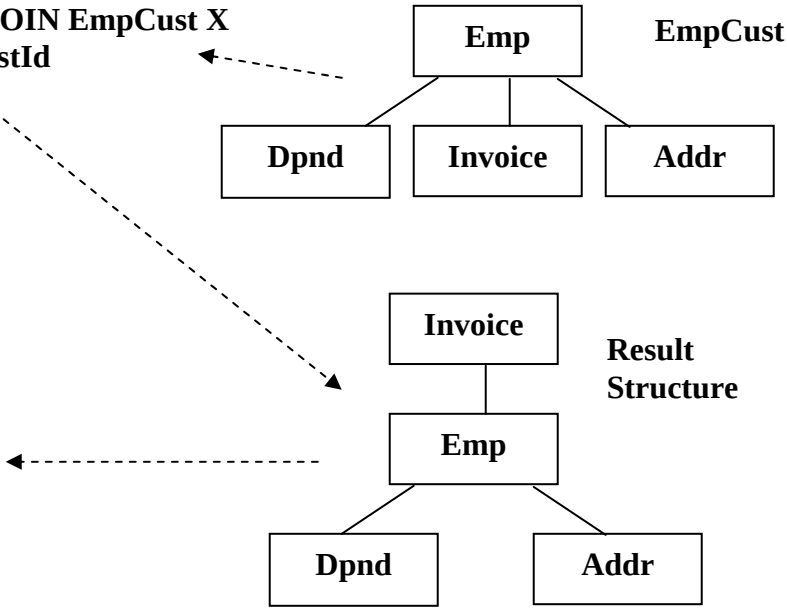
```
<root>
 <emp empid="Emp01">
  <dpnd dpndid="Dpnd01"/>
  <invoice invid="Inv01"/>
  <invoice invid="Inv02"/>
  <addr addrid="Addr01"/>
 </emp>
 <emp empid="Emp02">
  <addr addrid="Addr03"/>
 </emp>
</root>
```



The following SQL 14.4 Restructure operation slices out the Invoice node shown above, and makes it the new root. Notice how the two invoices in the result replicate the data separately under each below. This is the correct semantics for the new structure.

SQL 14.4: **SELECT X.EmpId, X.DpndId, Y.InvId, X.AddrId**
FROM EmpCust Y LEFT JOIN EmpCust X
ON Y.invCustId=X.EmpCustId

```
<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>
 </invoice>
</root>
```

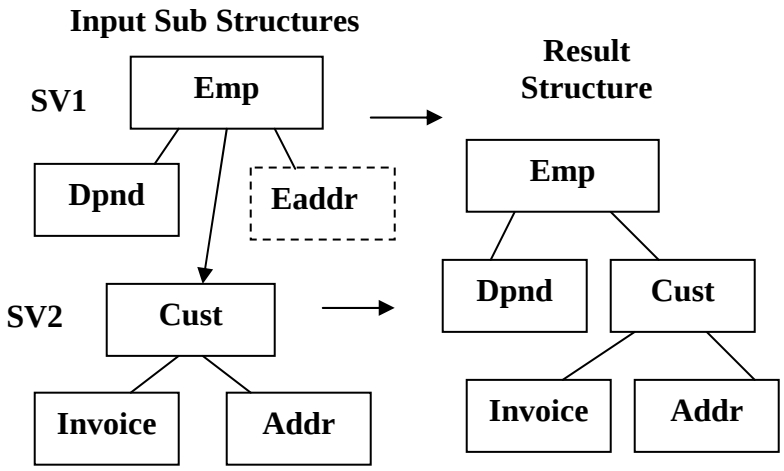


14.5) Restructuring With ON Condition Path Data Filtering

Another powerful capability of transforming structures by pulling them apart and reassembling them differently with joins, is that data filtering can be applied to the already joined data during this transform operation. This is performed by adding a data filtering criteria to the desired ON clause to be filtered. This has been covered earlier when modeling and building structures was covered. It is covered here to show that it also works for transforming data too, which may not be obvious. The first example in this section, SQL 14.1 will be used show the transform operation without the ON clause filtering criteria. The SQL 14.5 example below shows the same SQL 14.1 query with "**AND SV1.EmpStatus='F'**" added at the end to filter out Emp02's lower level data.

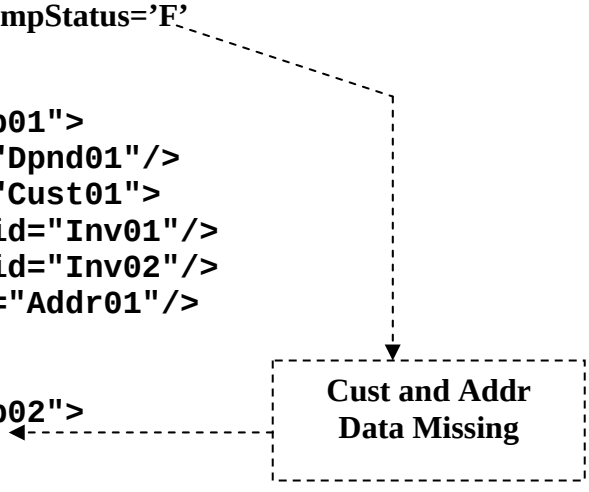
SQL 14.1: **SELECT SV1.EmpID, SV1.DpndID, SV2.CustID, SV2.InvID, SV2.AddrID**
FROM StoreView SV1 LEFT JOIN StoreView SV2 ON SV1.EmpCustID=SV2.CustID

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



SQL 14.5: **SELECT SV1.EmpID, SV1.DpndID, SV2.CustID, SV2.InvID, SV2.AddrID**
FROM StoreView SV1 LEFT JOIN StoreView SV2 ON SV1.EmpCustID=SV2.CustID
AND SV1.EmpStatus='F'

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



14.6) Restructuring With WHERE Clause Global Data Filtering

If the ON Condition can filter the Restructuring operations, then the WHERE clause should be able to perform its more global filtering. Let's check it out just to make sure it is operating correctly on restructuring operations. We will do this by replacing the ON condition on the previous SQL 14.5 with a WHERE clause performing the same filtering condition in SQL 14.6 below. If it works correctly it should remove Emp02 and all of its dependent data, which it did correctly.

SQL 14.6: **SELECT SV1.EmpID, SV1.DpndID, SV2.CustID, SV2.InvID, SV2.AddrID**
FROM StoreView SV1 LEFT JOIN StoreView SV2 ON SV1.EmpCustID=SV2.CustID
WHERE SV1.EmpStatus='F'

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

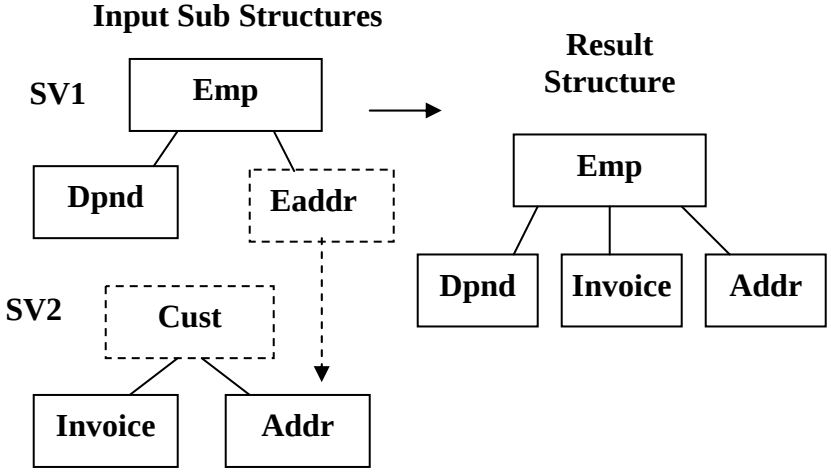
Emp02 and all of its dependent data have been filtered out

14.7) Restructuring Using Separate Fragment Groups

This is an example where two separate fragments, Invoice and Addr, are grouped by prefix SV2 in SQL 14.7 below, but are not defining a single structure since Cust is not selected in the SV2 group. Additionally, only Addr is being linked to. The question is, is this valid, if so how is Invoice handled? This is legal and operates on the same principles as linking below the root (see Section 6). In this example, Invoice and Addr are still related through Cust (LCA) and this is how Invoice can be brought along with Addr since it is the one being joined on. The XML is producing the correct results.

SQL 14.7: **SELECT SV1.EmpID, SV1.DpndID, SV2.InvID, SV2.AddrID**
FROM StoreView SV1 LEFT JOIN StoreView SV2
ON SV1.EaddrID= SV2.AddrID

```
<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>
```



Recap of Structure Restructuring

Basic Operation	Isolate and move fragments around by using duplicate views for each fragment and use relationships in data to re-join on.
Changing Leg Order	Change Leg order by changing join order.
Renaming, Duplicating, Changing & Splitting Nodes	Use SQL's Alias for supporting Renaming, Duplicating, Changing and Splitting Nodes.
Replicated Data	Basic transformation operation correctly replicates data.
Hierarchical WHERE and ON Clause Data Filtering	The hierarchical WHERE clause data filtering and ON path data filtering can be used with the Transformation.
Joining Under Root	Joining under the root greatly increases the number of relationships available to re-join on.
Transformation Can be Placed in View	Transformations can be stored in views and invoked with varying real-time query specifications like different data filtering and Selected node input.

Table 14: Structure Restructuring Capabilities

15) Any-to-Any Hierarchical Structure Reshaping Using Semantics

Section 14 described nonlinear Restructuring using SQL. This is performed by isolating structure fragments using Selection and Renaming (SQL Alias) with use of prefixes to separately group the fragments. These fragments were then reassembled into a different structure using unused and previously used data relationships in joins. This is a hierarchical high level transform structures, but it does rely on the data relationships in the structure being transformed limiting its range of transformations. XML's lack of need for foreign keys to define its contiguous structure adds to this lack of available data relationships.

A similar but more flexible approach to this data value relationship transformation is not to rely on any established relationships but to rely only on the structure's natural data semantics to enable Reshaping the structure into any other structure with the same node types. This is performed by duplicating the same structure as many times as necessary and link the occurrences of the structures together in any desired manner by matching on the same unique controlling fields in the joined copies of the structure. This increases the range of transformations allowing any structure to be shaped into any other hierarchical structure while preserving the data and maintaining or altering its semantics depending on the new shape. The renaming and splitting of nodes, and data filtering capabilities available with Restructuring in Section 14 are available in the same way with Reshaping, but are not shown.

Internally the new semantic associations between the selected nodes logically persist even after all other nodes are removed by lack of selection so that the existing nodes are still related to each other in the same way. The desired nodes' data is selected at its required level. Since this is the only level it is selected at, all other node levels for this node type are not selected for output and the resulting structure is nicely compressed to only select nodes by the natural process of node promotion discussed in Section 3.2.

The following example relationships represented in CustViewT below in Figure 15 are used to generate test data for reshaping examples to follow. They are mostly 1 to M (One to Many) the most common for hierarchical structures. When nodes become inverted by re-shaping these relationships will be changed to M to 1 which should flatten the data and possibly lose some of the associated data because of hierarchical preservation principles. The following examples will demonstrate these hierarchically principled operations, but first let's look at the example structures and their data.

View CustViewT below defines the full structure

```
CREATE View CustViewT AS
SELECT * FROM CustT Customer
LEFT JOIN InvoiceT Invoice ON CustID=InvCustID
LEFT JOIN EmpT Employee ON CustID=EmpCustID
LEFT JOIN DpndT Dependent ON EmpID=DpndEmpID
```

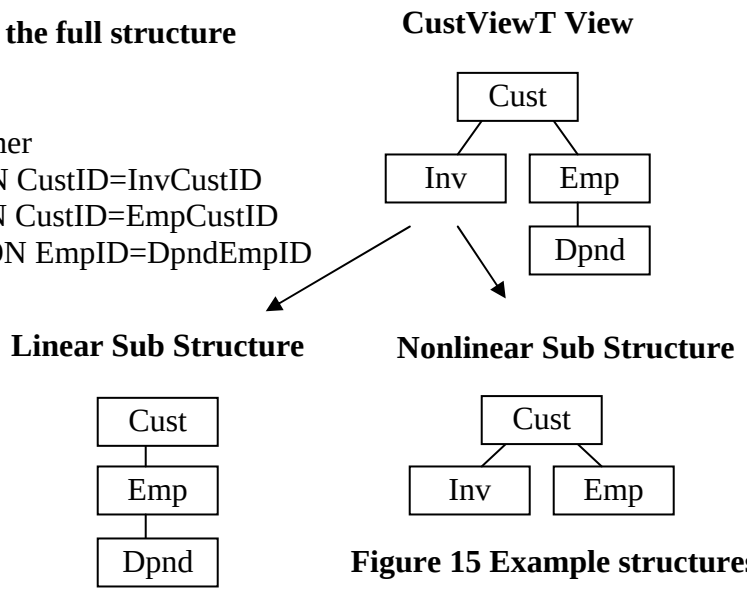


Figure 15 Example structures

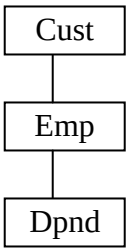
15.01 Linear Sub Structure Data

This linear structure data is achieved from the CustViewT view be Selecting data only from nodes that represents a linear structure. In this case, its data from the Cust, Emp and Dpnd nodes as shown below. The linear data listed below and in the examples is limited to only data from Store01 by a WHERE clause. This is done only to limit the data to a more manageable level. Foreign keys, though not utilized in the reshaping operation, are selected in the output to help verify the correctness of the structure.

SQL15.01: **SELECT CustId, EmpID, EmpCustID, DpndID, DpndEmpID**
FROM CustViewT WHERE CustStoreID='Store01'

```
<root>
 <customer custid="Cust01">
  <employee empid="Emp01" empcustid="Cust01">
   <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
   <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
  </employee>
  <employee empid="Emp03" empcustid="Cust01">
  </employee>
  <employee empid="Emp07" empcustid="Cust01">
  </employee>
 </customer>
 <customer custid="Cust02">
  <employee empid="Emp02" empcustid="Cust02">
   <dependent dpndid="Dpnd02" dpndempid="Emp02"/>
   <dependent dpndid="Dpnd04" dpndempid="Emp02"/>
   <dependent dpndid="Dpnd12" dpndempid="Emp02"/>
  </employee>
  <employee empid="Emp04" empcustid="Cust02">
   <dependent dpndid="Dpnd05" dpndempid="Emp04"/>
   <dependent dpndid="Dpnd07" dpndempid="Emp04"/>
  </employee>
  <employee empid="Emp05" empcustid="Cust02">
   <dependent dpndid="Dpnd06" dpndempid="Emp05"/>
   <dependent dpndid="Dpnd08" dpndempid="Emp05"/>
  </employee>
  <employee empid="Emp06" empcustid="Cust02">
   <dependent dpndid="Dpnd01" dpndempid="Emp06"/>
  </employee>
  <employee empid="Emp08" empcustid="Cust02">
  </employee>
 </customer>
 <customer custid="Cust03">
  <employee empid="Emp09" empcustid="Cust03">
  </employee>
 </customer>
 <customer custid="Cust14">
 </customer>
</root>
```

Result Structure

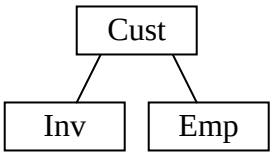


15.02 Nonlinear Sub Structure Data

This nonlinear data structure is achieved from the CustViewT view be Selecting data only from nodes that represents a nonlinear structure. In this case, its data from the Cust, Emp and Inv nodes as shown below. The Inv and Emp nodes are sibling nodes making the structure nonlinear with Cust as the common parent. You will notice that SQLfX® has automatically removed the duplicates caused by the relational Cartesian product which are usually prevalent between siblings. Foreign keys, though not utilized in the reshaping operation, are selected in the output to help verify the correctness of the structure. The nonlinear data listed below and in the examples is limited to only data from Store01 by a WHERE clause. This is done only to limit the data to a more manageable level.

SQL15.02: **SELECT CustId, InvID, InvCustID, EmpID, EmpCustID**
FROM CustViewT WHERE CustStoreID='Store01'

Result Structure




```
<root>
 <customer custid="Cust01">
  <invoice invid="Inv01" invcustid="Cust01"/>
  <invoice invid="Inv02" invcustid="Cust01"/>
  <invoice invid="Inv11" invcustid="Cust01"/>
  <invoice invid="Inv12" invcustid="Cust01"/>
  <invoice invid="Inv36" invcustid="Cust01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
 </customer>
 <customer custid="Cust02">
  <invoice invid="Inv03" invcustid="Cust02"/>
  <invoice invid="Inv04" invcustid="Cust02"/>
  <invoice invid="Inv05" invcustid="Cust02"/>
  <employee empid="Emp02" empcustid="Cust02"/>
  <employee empid="Emp04" empcustid="Cust02"/>
  <employee empid="Emp05" empcustid="Cust02"/>
  <employee empid="Emp06" empcustid="Cust02"/>
  <employee empid="Emp08" empcustid="Cust02"/>
 </customer>
 <customer custid="Cust03">
  <invoice invid="Inv06" invcustid="Cust03"/>
  <invoice invid="Inv31" invcustid="Cust03"/>
  <employee empid="Emp09" empcustid="Cust03"/>
 </customer>
 <customer custid="Cust14">
  <invoice invid="Inv25" invcustid="Cust14"/>
 </customer>
</root>
```


15.10 Linear Inversion Logic

Linear structures are simpler than nonlinear structures so we will start with them first. Our first reshaping example in Figure 15.10 will be the inversion of a two level structure of Cust over Emp. This can be performed by making a second copy of the structure and then joining one over the other in such a way that we can create and extract through joining across structures an Emp over Cust fragment. There are two ways to join the two structures by putting one node over the other. These are either using the join criteria of first.Cust=second.Cust or first.Emp=second.Emp. In Figure 15.10 below, the first example on the left using the Cust nodes to join on does not produce the right combinations for EMP over Cust. The second join operation joining on first.Emp=second.Emp does produce Emp over Cust from SQL15.10 below.

SQL15.10: **SELECT First.EmpID, First.EmpCustID, Second.CustID, Second.CustStoreID**
FROM CustViewT First LEFT JOIN CustViewT Second ON First.EmpID=Second.EmpID

The above join is aided by the capabilities and rules for joining a lower level structure below the lower level root described in Section 6 which states that the actual data modeling join point remains the root of the lower level structure, Cust in this case. The Select clause is used to select the correct combination of nodes as in: SELECT First.EmpID, Second.CustID, utilizing prefixes to specify the correct duplicate named nodes created by the self joins.

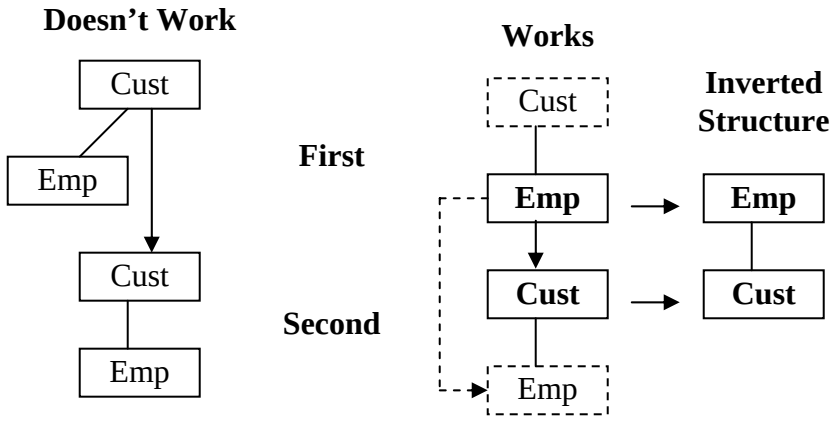


Figure 15.10: Simple two level linear inversion

The example in Figure 15.10 has been used to explain how structure reshaping is performed; we will not bother with showing these simple actual results. The following three level linear inversion will show live data examples. But as long as we are here, a review of the diagram symbols as used above would be a good idea. A dashed arrow represents the ON clause alignment node linkage points from the upper structure to the lower structure which could be below the root. If there is no dashed arrow, this means the solid arrow also species the ON clause linkage points. Solid arrow represents the interpreted data modeling structure linkage between the structures being joined. This is used to complete the unified data structure of all the structures joined. This represents the semantics of the structure which naturally controls the processing of the query. A solid box indicates a SELECTed node. A dashed box indicates an unselected node that is sliced out of the query returned result.

15.11 Inverting a 1 to M Linear Three Level Structure Reshaping

This is a longer linear structure inversion using Cust over Emp over Dpnd (a series of 1 to M relationships). This will demonstrate that the alignment joining starts at the bottom of the structures, and drives the inversion upward at each node as the selected inverted node from each level is selected: SELECT first: X.Dpnd, second: Y.Emp, third: Z.Dpnd. Only these three nodes are selected once each for output so they are squeezed together and keep their structure which is naturally inverted and is now M to 1 relationships. This is demonstrated in the XML results below by no multiple occurrences of data under a parent as occurred in the input data, but the XML results under SQL 15.11 below do have multiple occurrences of data being replicated in other areas of the structure such as Emps 1,2,4 and 5, and Cust 1 and 2. More importantly, Emps 3, 7 and 8 have been removed because they now have no higher level dependents, and for the same reason Custs 3 and 14 have been removed because they have no Dpnds. With this taken into account, the inverted XML structure is correct.

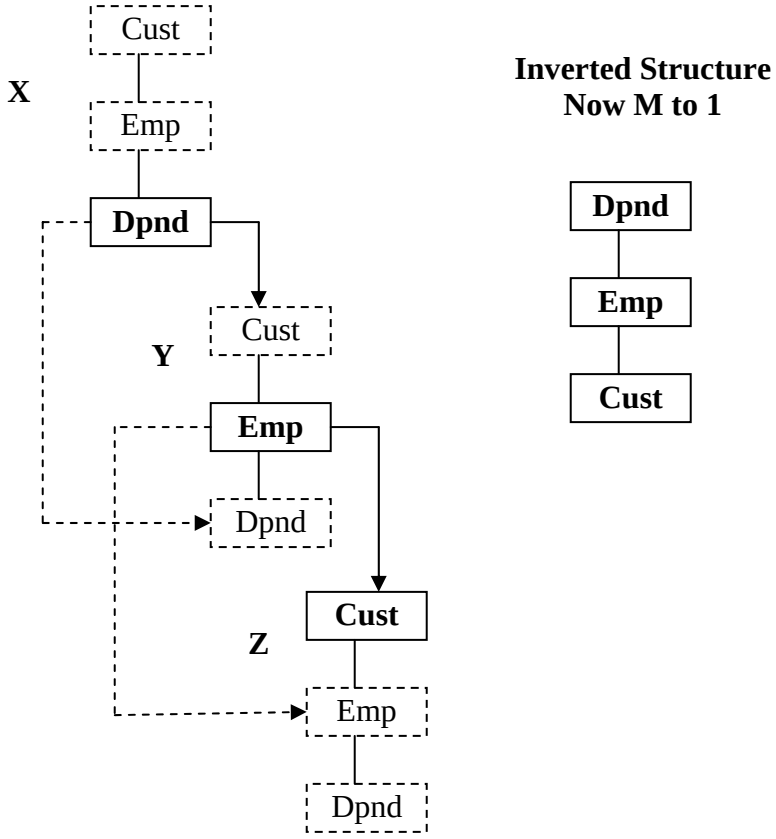


Figure 15.11: Complex linear inversion

CREATE VIEW CustViewInvert: **CREATE VIEW CustViewInvert AS**
SELECT X.DpndID, X.DpndEmpID, Y.EmpID, Y.EmpCustID, Z.CustID, Z.CustStoreID
FROM CustViewT X LEFT JOIN CustViewT Y ON X.DpndID=Y.DpndID
LEFT JOIN CustViewT Z ON Y.EmpID=Z.EmpID

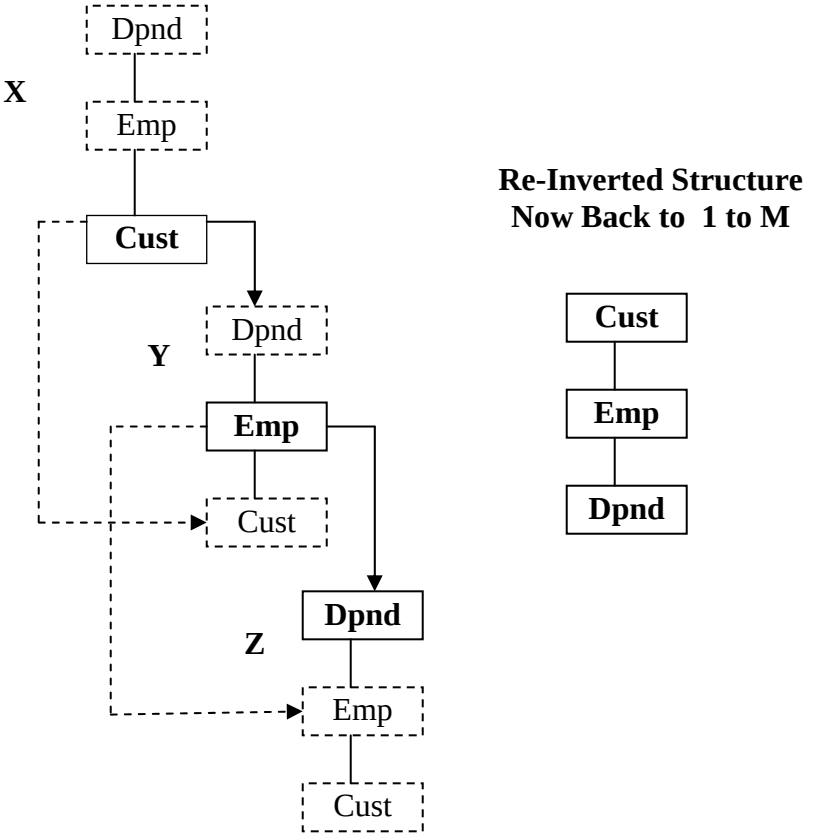
SQL 15.11: **SELECT * FROM CustViewInvert Where CustStoreID='Store01'**

```
<root>
 <dependent dpndid="Dpnd01" dpndempid="Emp06">
  <employee empid="Emp06" empcustid="Cust02">
```

```
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd02" dpndempid="Emp02">
  <employee empid="Emp02" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd03" dpndempid="Emp01">
  <employee empid="Emp01" empcustid="Cust01">
   <customer custid="Cust01" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd04" dpndempid="Emp02">
  <employee empid="Emp02" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd05" dpndempid="Emp04">
  <employee empid="Emp04" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd06" dpndempid="Emp05">
  <employee empid="Emp05" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd07" dpndempid="Emp04">
  <employee empid="Emp04" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd08" dpndempid="Emp05">
  <employee empid="Emp05" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd11" dpndempid="Emp01">
  <employee empid="Emp01" empcustid="Cust01">
   <customer custid="Cust01" custstoreid="Store01"/>
  </employee>
 </dependent>
 <dependent dpndid="Dpnd12" dpndempid="Emp02">
  <employee empid="Emp02" empcustid="Cust02">
   <customer custid="Cust02" custstoreid="Store01"/>
  </employee>
 </dependent>
</root>
```


15.12 Inverting a Linear M to 1 Three Level Structure

This structure inversion actually takes the output of the previous inversion which flattened the 1 to M structure into an M to 1 structure and re-inverts it. The previous inverted structure is re-created by the CustViewInvert view used in the SQL 15.12 example. This example should not only change the structure back into a 1 to M structure and remove the duplicates, but test if it is smart enough to also recognize the valid replications and return the structure in 1 to M hierarchically structure format which regroups the multiple occurrences with only a single parent occurrence (renormalize), which it did as shown below in the XML output from SQL 15.12. The missing data was introduced from the initial inversion as described in Section 15.11 above.



SQL 15.12: **SELECT X.CustID, X.CustStoreID, Y.EmpID, Y.EmpCustID, Z.DpndID, Z.DpndEmpID**
FROM CustViewInvert X LEFT JOIN CustViewInvert Y ON X.CustID=Y.CustID
LEFT JOIN CustViewInvert ON Y.EmpID=Z.EmpID
WHERE X.CustStoreID='Store01'

```
<root>
 <customer custid="Cust01" coststoreid="Store01">
  <employee empid="Emp01" empcustid="Cust01">
   <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
   <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
  </employee>
 </customer>
 <customer custid="Cust02" coststoreid="Store01">
```



```
  <employee empid="Emp02" empcustid="Cust02">
   <dependent dpndid="Dpnd02" dpndempid="Emp02"/>
   <dependent dpndid="Dpnd04" dpndempid="Emp02"/>
   <dependent dpndid="Dpnd12" dpndempid="Emp02"/>
  </employee>
  <employee empid="Emp04" empcustid="Cust02">
   <dependent dpndid="Dpnd05" dpndempid="Emp04"/>
   <dependent dpndid="Dpnd07" dpndempid="Emp04"/>
  </employee>
  <employee empid="Emp05" empcustid="Cust02">
   <dependent dpndid="Dpnd06" dpndempid="Emp05"/>
   <dependent dpndid="Dpnd08" dpndempid="Emp05"/>
  </employee>
  <employee empid="Emp06" empcustid="Cust02">
   <dependent dpndid="Dpnd01" dpndempid="Emp06"/>
  </employee>
 </customer>
</root>
```


15.2 Linear to Nonlinear Reshaping Logic

Interestingly, linear structures can be reshaped into nonlinear structures and the replication of input structures used in this section can be used to demonstrate this.

15.21 Linear to Nonlinear Preserved Semantics Reshaping

In this example the linear structure Cust over Emp over Dpnd can be used to generate the structure Emp directly over the siblings Dpnd and Cust. Two copies of the Input structure are necessary and are joined by their common Emp node since it is the starting node to building the new structure. Only two copies are necessary because the first copy can be used to move two nodes, Emp over Dpnd, which are already in place. This places first.emp over the first.Dpnd and first.Cust siblings. This creates the nonlinear structure desired and these same node values will be selected to create the nonlinear structure desired. By placing Emp over Cust, Cust is naturally and correctly replicated, notice that Emp01 and Emp03 both contain Cust01.

The previous linear examples and this nonlinear example have not lost the semantics of the input structure in the new structure because the semantics have been kept the same or inverted. This means the nodes have basically kept attached to the same nodes as shown below in the derived result structure. Emp over Dpnd remains the same while Emp over Cust has been inverted. The SQL 15.21 below result is accurate and models the result structure shown.

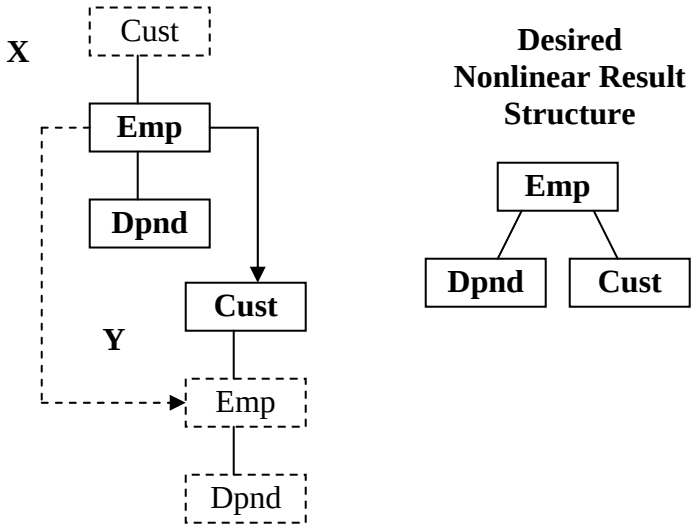


Figure 15.21: Linear to nonlinear reshaping with same semantics

SQL15.21: **SELECT X.EmpID, X.EmpCustID, Y.CustID, Y.CustStoreID,**
X.DpndID, X.DpndEmpID
FROM CustViewT X LEFT JOIN CustViewT Y ON X.EmpID=Y.EmpID
WHERE Y.CustStoreID='Store01'

```
<root>
 <employee empid="Emp01" empcustid="Cust01">
  <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
  <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
```



```
  <customer custid="Cust01" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp02" empcustid="Cust02">
  <dependent dpndid="Dpnd02" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd04" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd12" dpndempid="Emp02"/>
  <customer custid="Cust02" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp03" empcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp04" empcustid="Cust02">
  <dependent dpndid="Dpnd05" dpndempid="Emp04"/>
  <dependent dpndid="Dpnd07" dpndempid="Emp04"/>
  <customer custid="Cust02" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp05" empcustid="Cust02">
  <dependent dpndid="Dpnd06" dpndempid="Emp05"/>
  <dependent dpndid="Dpnd08" dpndempid="Emp05"/>
  <customer custid="Cust02" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp06" empcustid="Cust02">
  <dependent dpndid="Dpnd01" dpndempid="Emp06"/>
  <customer custid="Cust02" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp07" empcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp08" empcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp09" empcustid="Cust03">
  <customer custid="Cust03" custstoreid="Store01"/>
 </employee>
</root>
```


15.22 Linear to Nonlinear Preserved Semantics Reverse Legs Reshaping

This is the same reshaping as the previous 15.21 query example except the sibling legs are reversed. Unfortunately because of the placement of the data in the input structure, the proper structure combinations do not become available with only a single structure duplication. This time it takes three copies of the structure to have Cust be the left sibling shown in SQL 15.22 below. But this is a good example that any number of copies can be used until the desired structure can be modeled and produced with no side effects. Sometimes multiple reshaping moves can be performed at a single level, sometimes only one reshaping move. Keep in mind that any reshaping move can be automatically following a chain of many intervening nodes without incurring any overhead of having to recreate the joins since they have already been performed in memory.

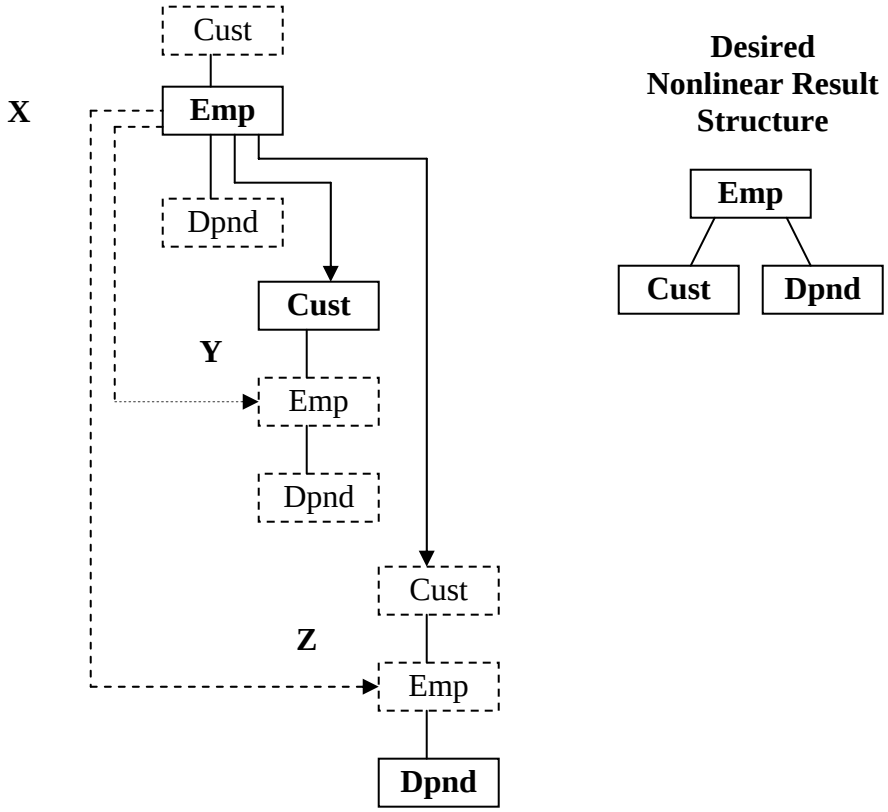


Figure 15.22: Linear to nonlinear reshaping with same semantics

SQL15.22: **SELECT X.EmpID, X.EmpCustID, Y.CustID, Y.CustStoreID, Z.DpndID, Z.DpndEmpID**
FROM CustViewT X LEFT JOIN CustViewT Y ON X.EmpID=Y.EmpID
LEFT JOIN CustViewT Z ON X.EmpID=Z.EmpID
WHERE Y.CustStoreID='Store01'

```
<root>
 <employee empid="Emp01" empcustid="Cust01">
  <customer custid="Cust01" coststoreid="Store01"/>
  <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
```



```
  <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
 </employee>
 <employee empid="Emp02" empcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
  <dependent dpndid="Dpnd02" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd04" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd12" dpndempid="Emp02"/>
 </employee>
 <employee empid="Emp03" empcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp04" empcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
  <dependent dpndid="Dpnd05" dpndempid="Emp04"/>
  <dependent dpndid="Dpnd07" dpndempid="Emp04"/>
 </employee>
 <employee empid="Emp05" empcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
  <dependent dpndid="Dpnd06" dpndempid="Emp05"/>
  <dependent dpndid="Dpnd08" dpndempid="Emp05"/>
 </employee>
 <employee empid="Emp06" empcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
  <dependent dpndid="Dpnd01" dpndempid="Emp06"/>
 </employee>
 <employee empid="Emp07" empcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp08" empcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
 </employee>
 <employee empid="Emp09" empcustid="Cust03">
  <customer custid="Cust03" custstoreid="Store01"/>
 </employee>
</root>
```


15.23 Linear to Nonlinear Indirectly Related Semantic Reshaping

The difference with this linear to nonlinear SQL 15.23 example from the previous SQL 15.22 example is that the semantics of the result structure have been changed. Cust over Emp is the same semantics but Cust over Dpnd is indirectly related through Emp as can be seen in the input structure below. With Emp relocated in a different location in the output structure there is no natural link between Cust over Dpnd in the result structure. This does not invalidate reshaping since Cust was related to Dpnd the same as if the Emp node was not selected for output and Dpnd was node promoted up directly under Cust.

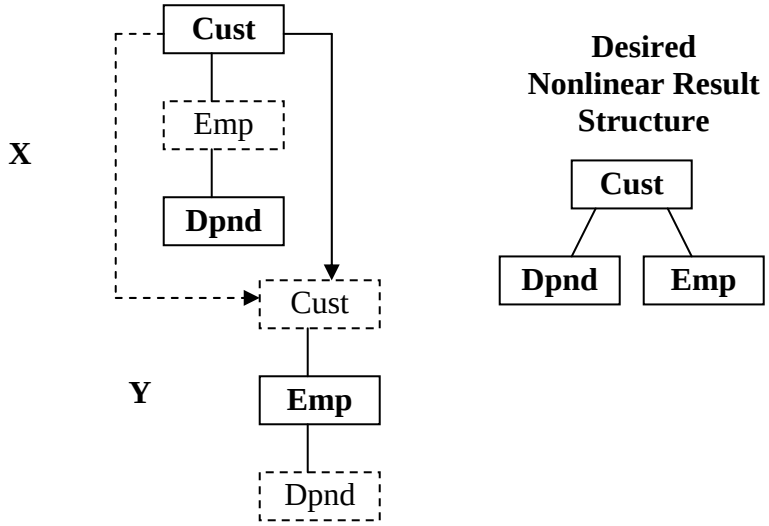


Figure 15.23: Linear to nonlinear reshaping with changed semantics

SQL15.23: **SELECT X.CustID, X.CustStoreID, X.DpndID, X.DpndEmpID,**
Y.EmpID, Y.EmpCustID
FROM CustViewT X LEFT JOIN CustViewT Y ON X.CustID=Y.CustID
WHERE X.CustStoreID='Store01'

```
<root>
 <customer custid="Cust01" coststoreid="Store01">
  <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
  <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
 </customer>
 <customer custid="Cust02" coststoreid="Store01">
  <dependent dpndid="Dpnd01" dpndempid="Emp06"/>
  <dependent dpndid="Dpnd02" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd04" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd05" dpndempid="Emp04"/>
  <dependent dpndid="Dpnd06" dpndempid="Emp05"/>
  <dependent dpndid="Dpnd07" dpndempid="Emp04"/>
  <dependent dpndid="Dpnd08" dpndempid="Emp05"/>
  <dependent dpndid="Dpnd12" dpndempid="Emp02"/>
```



```
  <employee empid="Emp02" empcustid="Cust02"/>
  <employee empid="Emp04" empcustid="Cust02"/>
  <employee empid="Emp05" empcustid="Cust02"/>
  <employee empid="Emp06" empcustid="Cust02"/>
  <employee empid="Emp08" empcustid="Cust02"/>
 </customer>
 <customer custid="Cust03" custstoreid="Store01">
  <employee empid="Emp09" empcustid="Cust03"/>
 </customer>
 <customer custid="Cust14" custstoreid="Store01">
 </customer>
</root>
```


15.3) Nonlinear to Linear and Nonlinear Reshaping

Nonlinear structures as input can also be used to build linear and nonlinear structures. In fact, nonlinear structures offer more flexibility in how they are utilized because their multiple legs offer more opportunity to find the correct reshaping being sought. This means less input copies need to be used.

15.31 Nonlinear to Linear Reshaping

The below nonlinear input structure can be duplicated to create a linear structure reshaping of itself using SQL 15.31. Since we are starting with Inv as the root, this will be the first matching link. Cust becomes available to SELECT in the second level structure which is valid since it is related to Inv to the related link point. Emp can be selected from the same structure copy since it is already located under Cust. In effect, it is possible to reach Emp from Inv directly in memory.

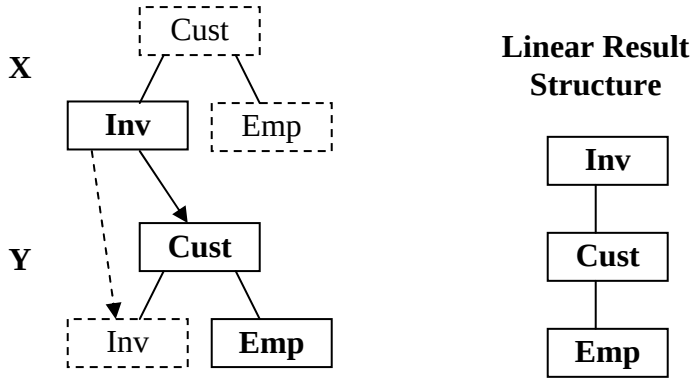


Figure 15.31: Nonlinear to linear reshaping example

SQL15.31: **SELECT Y.CustID, Y.CustStoreID, X.InvID, X.InvCustID, Y.EmpID, Y.EmpCustID**
FROM CustViewT X LEFT JOIN CustViewT Y ON X.InvID=Y.InvID
WHERE X.CustStoreID='Store01' AND X.InvID<'Inv13'

```
<root>
 <invoice invid="Inv01" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01">
   <employee empid="Emp01" empcustid="Cust01"/>
   <employee empid="Emp03" empcustid="Cust01"/>
   <employee empid="Emp07" empcustid="Cust01"/>
  </customer>
 </invoice>
 <invoice invid="Inv02" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01">
   <employee empid="Emp01" empcustid="Cust01"/>
   <employee empid="Emp03" empcustid="Cust01"/>
   <employee empid="Emp07" empcustid="Cust01"/>
  </customer>
 </invoice>
 <invoice invid="Inv03" invcustid="Cust02">
```



```
  <customer custid="Cust02" custstoreid="Store01">
   <employee empid="Emp02" empcustid="Cust02"/>
   <employee empid="Emp04" empcustid="Cust02"/>
   <employee empid="Emp05" empcustid="Cust02"/>
   <employee empid="Emp06" empcustid="Cust02"/>
   <employee empid="Emp08" empcustid="Cust02"/>
  </customer>
 </invoice>
 <invoice invid="Inv04" invcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01">
   <employee empid="Emp02" empcustid="Cust02"/>
   <employee empid="Emp04" empcustid="Cust02"/>
   <employee empid="Emp05" empcustid="Cust02"/>
   <employee empid="Emp06" empcustid="Cust02"/>
   <employee empid="Emp08" empcustid="Cust02"/>
  </customer>
 </invoice>
 <invoice invid="Inv05" invcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01">
   <employee empid="Emp02" empcustid="Cust02"/>
   <employee empid="Emp04" empcustid="Cust02"/>
   <employee empid="Emp05" empcustid="Cust02"/>
   <employee empid="Emp06" empcustid="Cust02"/>
   <employee empid="Emp08" empcustid="Cust02"/>
  </customer>
 </invoice>
 <invoice invid="Inv06" invcustid="Cust03">
  <customer custid="Cust03" custstoreid="Store01">
   <employee empid="Emp09" empcustid="Cust03"/>
  </customer>
 </invoice>
 <invoice invid="Inv11" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01">
   <employee empid="Emp01" empcustid="Cust01"/>
   <employee empid="Emp03" empcustid="Cust01"/>
   <employee empid="Emp07" empcustid="Cust01"/>
  </customer>
 </invoice>
 <invoice invid="Inv12" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01">
   <employee empid="Emp01" empcustid="Cust01"/>
   <employee empid="Emp03" empcustid="Cust01"/>
   <employee empid="Emp07" empcustid="Cust01"/>
  </customer>
 </invoice>
</root>
```


15.32 Nonlinear to Nonlinear Reshaping

This nonlinear to nonlinear example is very similar to the previous nonlinear to linear SQL 15.31 example where the linear structure Inv over Cust over Emp was produced easily using SQL 15.32. This is an example demonstrating that nonlinear structures can produce nonlinear structures so this example copies the previous example but places Emp not under Cust but under Inv making Cust and Emp siblings for this structure. This requires a third copy of the input structure also matched to Inv because that is where Emp is being attached to. Emp is accessed indirectly from Inv up to Cust down to Emp a powerful related semantic reshape. While this produces the same result as the previous example, it is correct. In both structures, this one and the previous one, Emp is or was indirectly related to Inv. Basically, the second structure and additional join in this example was necessary to move Emp from under Cust to under Inv.

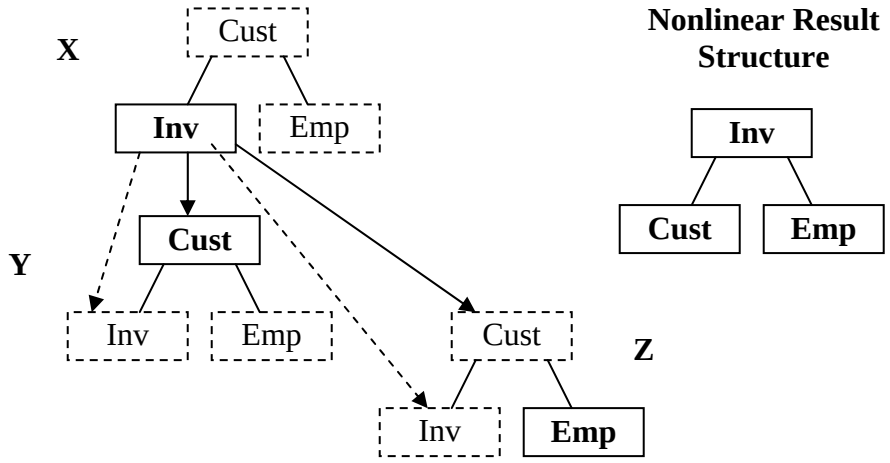


Figure 15.32: Nonlinear to nonlinear reshaping example

SQL15.32: **SELECT Y.CustID, Y.CustStoreID, X.InvID, X.InvCustID, Z.EmpID, Z.EmpCustID**
FROM CustViewT X LEFT JOIN CustViewT Y ON X.InvID=Y.InvID
LEFT JOIN CustViewT Z ON Y.InvID=Z.InvID
WHERE X.CustStoreID='Store01' AND X.InvID<'Inv13'

```
<root>
 <invoice invid="Inv01" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
 </invoice>
 <invoice invid="Inv02" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
 </invoice>
 <invoice invid="Inv03" invcustid="Cust02">
```



```
  <customer custid="Cust02" custstoreid="Store01"/>
  <employee empid="Emp02" empcustid="Cust02"/>
  <employee empid="Emp04" empcustid="Cust02"/>
  <employee empid="Emp05" empcustid="Cust02"/>
  <employee empid="Emp06" empcustid="Cust02"/>
  <employee empid="Emp08" empcustid="Cust02"/>
 </invoice>
 <invoice invid="Inv04" invcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
  <employee empid="Emp02" empcustid="Cust02"/>
  <employee empid="Emp04" empcustid="Cust02"/>
  <employee empid="Emp05" empcustid="Cust02"/>
  <employee empid="Emp06" empcustid="Cust02"/>
  <employee empid="Emp08" empcustid="Cust02"/>
 </invoice>
 <invoice invid="Inv05" invcustid="Cust02">
  <customer custid="Cust02" custstoreid="Store01"/>
  <employee empid="Emp02" empcustid="Cust02"/>
  <employee empid="Emp04" empcustid="Cust02"/>
  <employee empid="Emp05" empcustid="Cust02"/>
  <employee empid="Emp06" empcustid="Cust02"/>
  <employee empid="Emp08" empcustid="Cust02"/>
 </invoice>
 <invoice invid="Inv06" invcustid="Cust03">
  <customer custid="Cust03" custstoreid="Store01"/>
  <employee empid="Emp09" empcustid="Cust03"/>
 </invoice>
 <invoice invid="Inv11" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
 </invoice>
 <invoice invid="Inv12" invcustid="Cust01">
  <customer custid="Cust01" custstoreid="Store01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
 </invoice>
</root>
```


15.4) Transform New Structure Recognition Tests

Structure transformation from restructuring or reshaping changes the existing structure rather than just adding to the existing structure being built. SQLfX® has patented technology that is aware of the structure of the hierarchical structure being built allowing it to perform its XML support transparently. Transformations of the structure increase the complexity of keeping track of the current hierarchical being built. These need to be tested against the most structure sensitive hierarchical operations which are the hierarchical Order By and the LCA logic in the WHERE clause. These are tested directly below.

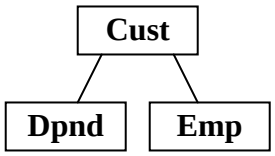
15.41 ORDER BY Hierarchical Structure Transform Recognition Test

This is the previous SQL 15.23 query with an Order By added to reverse its natural ordering to descending. The goal of this example is to make sure that the Order By understands its new data structure. You will notice in the XML below that CustID, DpndID, and EmpID order has been changed to descending correctly for its new structure.

SQL15.41: **SELECT X.CustID, X.CustStoreID, X.DpndID, X.DpndEmpID,**
Y.EmpID, Y.EmpCustID
FROM CustViewT X LEFT JOIN CustViewT Y ON X.CustID=Y.CustID
WHERE X.CustStoreID='Store01'
ORDER BY X.CustID Desc, X,DpndID Desc, Y.EmpID Desc

```
<root>
 <customer custid="Cust14" coststoreid="Store01">
 </customer>
 <customer custid="Cust03" coststoreid="Store01">
  <employee empid="Emp09" empcustid="Cust03"/>
 </customer>
 <customer custid="Cust02" coststoreid="Store01">
  <dependent dpndid="Dpnd12" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd08" dpndempid="Emp05"/>
  <dependent dpndid="Dpnd07" dpndempid="Emp04"/>
  <dependent dpndid="Dpnd06" dpndempid="Emp05"/>
  <dependent dpndid="Dpnd05" dpndempid="Emp04"/>
  <dependent dpndid="Dpnd04" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd02" dpndempid="Emp02"/>
  <dependent dpndid="Dpnd01" dpndempid="Emp06"/>
  <employee empid="Emp08" empcustid="Cust02"/>
  <employee empid="Emp06" empcustid="Cust02"/>
  <employee empid="Emp05" empcustid="Cust02"/>
  <employee empid="Emp04" empcustid="Cust02"/>
  <employee empid="Emp02" empcustid="Cust02"/>
 </customer>
 <customer custid="Cust01" coststoreid="Store01">
  <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
  <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
  <employee empid="Emp07" empcustid="Cust01"/>
```

New Nonlinear Result Structure




```
  <employee empid="Emp03" empcustid="Cust01"/>
  <employee empid="Emp01" empcustid="Cust01"/>
 </customer>
</root>
```

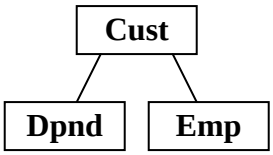
15.42 LCA Hierarchical Logic Structure Transform Recognition Test

This is the previous SQL 15.23 query with a different WHERE clause to test out the correct use of LCA logic. The goal of this example is to make sure that the LCA reflects its new data structure. Using the new structure shown below, WHERE qualified "Emp03" qualifies only "Cust01" which qualifies all of its dependents which is shown below. "Cust01" is the LCA occurrence.

SQL15.42: **SELECT X.CustID, X.CustStoreID, X.DpndID, X.DpndEmpID,**
Y.EmpID, Y.EmpCustID
FROM CustViewT X LEFT JOIN CustViewT Y ON X.CustID=Y.CustID
WHERE X.EmpID='Emp03'

```
<root>
 <customer custid="Cust01" coststoreid="Store01">
  <dependent dpndid="Dpnd03" dpndempid="Emp01"/>
  <dependent dpndid="Dpnd11" dpndempid="Emp01"/>
  <employee empid="Emp03" empcustid="Cust01"/>
 </customer>
</root>
```

New Nonlinear Result Structure



15.5 Polymorphic Reshaping

Polymorphic reshaping does not rely on the structure of the input structure. The advanced reshaping capability shown in Section 15 does support polymorphic reshaping when only one node is moved per join. The example in Section 15.21 is not polymorphic because it relies on a specific structure by moving two related nodes at one time while the same basic reshaping performed in Section 15.22 is polymorphic because it does not rely on the input structure by locating and moving one node at a time. The choice of using reduced steps or polymorphic solution is up to the user.

Recap of Structure Reshaping

Basic Operation	Isolate and move desired nodes in order necessary to build desired structure using key to same key to synchronize matching views.
Changing Leg Order	Change Leg order by changing join order.
Renaming, Duplicating, Changing and Splitting Nodes	Use SQL's Alias capability to support Renaming, Duplicating, Changing and Splitting Nodes.
Replicated Data	Basic transformation operation correctly replicates data when necessary to represent the new structure.
ON and WHERE Clause Data Filtering	The hierarchical WHERE clause global filtering and ON path filtering can be used with the reshaping.
Joining Under Root	Joining under the root greatly increases the number of relationships available to re-join on.
Transformation Can be Placed in View	Transformations can be stored in views and invoked with varying query specifications, different data filtering and Select list node input.
Any-to-Any Hierarchical Structure Reshaping	By not relying on any relationships, and using an unrestrictive enabling reshaping capability, any-to-any structure reshaping is possible.
Polymorphic Reshaping	Polymorphic reshaping is supported in this technique as long as only one node per join is located and moved.

Table 15: Structure Reshaping Capabilities

16) Multi-type and Multiple Structure Transformation

We have seen in the two sections that preceded this one that there are two methods for structure transformation. These were restructuring using existing relationships in the data, and reshaping where no prior relationships are needed because the structure semantics are used. Each had their own use. Restructure introducing new active relationships producing new semantics and data to match it to a possible application, and reshaping preserving the semantics and data to match a desired data structure. They have different uses and different capabilities. At times it may be useful to combine the two structure transformation operations to derive the desired result. This section demonstrates this multi-type structure transform.

All of the previous structure transformations used only a single input source structure. Reshaping operation usually implies operating on the same structure, but restructuring has no barriers to utilizing multiple input source structures which could be quite useful. Combining this multiple structure transform with multi-type structure transformation produces an extremely powerful structure transformation capability with unlimited transformation capabilities. The example in Figure 16 below demonstrates these two powerful structural transformation capabilities in the same single example.

The structure transformation SQL 16 example in Figure 16 involves the two separate structures CustView and EmpView. It starts off by performing a reshaping on the CustView to invert its structure without the use of data relationships in the data except for self referencing relationships, InvID in this case to synchronize the two copies of CustView. Then the second structure, EmpView is referenced using the CustID to EmpCustID relationship to access and move the EmpID node data in the second structure using restructuring.

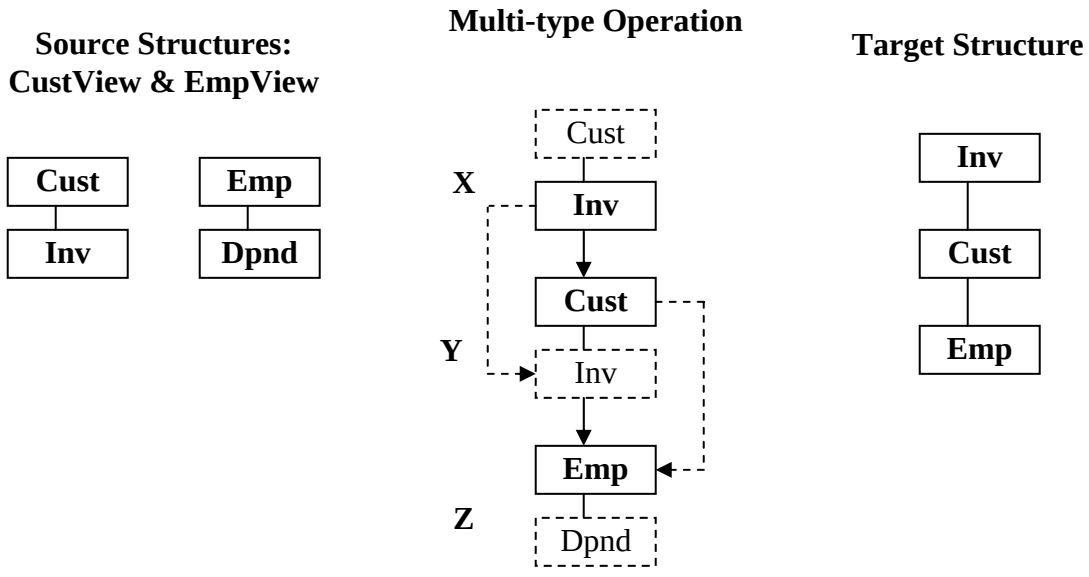


Figure 16: Multi-type and Multiple Structure Transform

SQL 16: **SELECT X.InvID, Y.CustID, Z.EmpID**
FROM CustView X LEFT JOIN CustView Y ON X.InvID=Y.InvID
LEFT JOIN EmpView Z ON Y.CustID=Z.EmpCustID


```
<root>
 <invoice invid="Inv01">
  <cust custid="Cust01">
   <emp empid="Emp01"/>
  </cust>
 </invoice>
 <invoice invid="Inv02">
  <cust custid="Cust01">
   <emp empid="Emp01"/>
  </cust>
 </invoice>
 <invoice invid="Inv03">
  <cust custid="Cust02">
  </cust>
 </invoice>
</root>
```

16 Hierarchical Reshaping and Restructuring Conclusion

This section has shown and proven that reshaping can transform any structure into any other structure with the same nodes types. Reshaping implies that no relationships between nodes are needed or utilized. Restructuring transformations (Section 14) use different relationships in the data while Reshaping (Section 15) uses the data semantics in the hierarchical structure to perform structure transformations. Both transformation methods support transformations that are not possible in the other, so both are necessary for complete coverage of structure transformation. Restructuring is used more to map to a required application to make its use and access easier, while Reshaping is used to map to a desired predefined structure possibly defined by an XML Schema. Reshaping is also used in inverting structures when data relationships are not available in the data. This is often the case with XML data since foreign keys are not required because of their contiguous nested storage.

The last structure transformation example demonstrated that restructuring and restructuring operations can be combined and that multiple input structures could be utilized, allowing unlimited structural transformations. These complex transformations are performed nonprocedurally at a high hierarchical conceptual level and can then be abstracted in a view. This transformation view will be automatically hierarchically optimized at run time based on the specified output, as any view will be in SQLfX®. It is assumed that in SQL processors that the use of identical views as used in the structural transforms described here, it will be optimized by caching. These transformations can also support Replicating, Renaming and Splitting nodes as shown in Section 5.3.

The required replication of structures used in our SQLfX® technology to perform any-to-any structure reshaping transformations is not necessary outside of SQL processing. In this regard, our commercial product release will also contain a global nonprocedural view-to-view (any-to-any structure) reshaping capability on the entire unified virtual expanded view requiring no joins, and will support explicit formatting and renaming controls. This capability will be available at the end of query processing and does not require any operational instruction at all. Other mapping and renaming capabilities will be available also.

Recap of Multi-type and Multiple Structure Transform

All Features of Restructure	Structure transform by relationships. Creates new semantics and data.
All Features of Reshaping	Structure transformation by structure semantics. Any to Any structure capability. Does not rely on relationships in data.
Multiple Structures	Transformation not restricted to a single structure. Multiple source structures can be used.
Renaming, Duplicating, Changing and Splitting Nodes	Use SQL's Alias capability to support Renaming, Duplicating, Changing and Splitting Nodes.
Replicated Data	Basic transformation operation correctly replicates data when necessary.
WHERE Clause Global Hierarchical Data Filtering	WHERE clause filtering performs global hierarchical filtering on entire hierarchical structure.
ON Clause Path Filtering	ON clause filtering only occurs down a path. Can be applied to structure transformation.
Transformation Can be Placed in View	Transformations can be stored in views and invoked with varying query specifications, different data filtering and Select list node input.

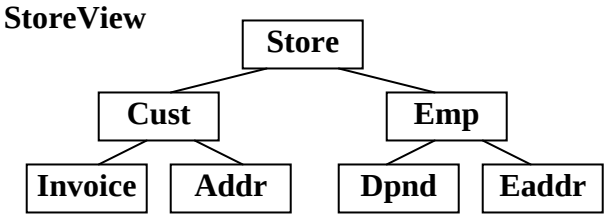
Table 16: Multi-type and Multiple Structure Transform Features

17 Global Hierarchical Queries

Global queries were mentioned a few times previously with some examples in this document, but global queries real importance and use was not really gone into. XML today is accessed via procedural navigation, even from high level languages like XQuery. This has limited XML processing to small portions of the total structure being accessed. In addition, LCA multi-leg operation is not being automatically utilized today, basically limiting access to a single leg of the structure.

SQLfX® is navigationless, nonprocedural, and automatically supports LCA processing logic. This means it can easily process requirements involving the entire hierarchical structure. For example a common need is to perform a filtering on the entire structure based on a single data item or a complex filtering based on multiple items in multiple legs. This can involve hierarchical filtering rippling through the entire structure. SQLfX® can do this with a simple query using a SELECT * to indicate all the different data items are to be selected and output and a WHERE clause is also used that automatically performs hierarchical filtering of the entire structure based on the SELECT clause's selected items.

First let's list out the entire StoreView structure to see the full structure for comparing against our global filtering examples and show the StoreView hierarchical structure again below.



SQL 17.0: **SELECT * FROM StoreView**

```
<root>
 <store storeid="Store01">
  <cust custid="Cust01" coststoreid="Store01">
   <invoice invid="Inv01" invcustid="Cust01" invstatus="P"/>
   <invoice invid="Inv02" invcustid="Cust01" invstatus="O"/>
   <addr addrid="Addr01" addrcustid="Cust01" addrstate="CA"/>
  </cust>
  <cust custid="Cust02" coststoreid="Store01">
   <invoice invid="Inv03" invcustid="Cust02" invstatus="O"/>
   <addr addrid="Addr02" addrcustid="Cust02" addrstate="CA"/>
   <addr addrid="Addr04" addrcustid="Cust02" addrstate="CA"/>
  </cust>
  <cust custid="Cust03" coststoreid="Store01">
   <addr addrid="Addr03" addrcustid="Cust03" addrstate="NV"/>
  </cust>
  <emp empid="Emp01" empstoreid="Store01" empcustid="Cust01" empstatus="F">
   <dpnd dpndid="Dpnd01" dpndempid="Emp01" dpndcode="D"/>
   <eaddr eaddrid="Addr01" eaddrcustid="Cust01" eaddrstate="CA"/>
  </emp>
  <emp empid="Emp02" empstoreid="Store01" empcustid="Cust03" empstatus="">
   <eaddr eaddrid="Addr03" eaddrcustid="Cust03" eaddrstate="NV"/>
  </emp>
 </store>
```


17.1) Simple Global Filtering

This example filters the entire structure filtering out all data that is not related to Inventory ID "Inv01". You will notice that only "Inv01" and its hierarchically related information are present. This is a global hierarchical filtering of the entire structure specified easily and performed correctly automatically. Even though the StoreView structure shown above is not that complex a structure, it does involve four separate legs to procedurally navigate by procedural processors. The LCA processing for the different pairs of legs could get extremely involved to get correct with procedural processing.

SQL 17.1: **SELECT * FROM StoreView WHERE InvID='Inv01'**

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

17.2) Complex Global Filtering

This example is a more complex; it filters out all data that is not related to both Inventory ID "Inv01" and Employees without the status of "F" for Fulltime. This filtering logic is much more complex than the previous example SQL 17.1 because the two different values being tested on this SQL 17.2 example need to be hierarchically coordinated to keep the results meaningful. This result has further filtered out employees who are not fulltime.

SQL 17.2: **SELECT * FROM StoreView WHERE InvID='Inv01' AND EmpStatus='F'**

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


A) Powerful Automatic Features

Many Complex and powerful SQLfX® features are performed automatically and transparently whenever they are needed or useful to improve processing such as optimization or to enhance processing such as automatically regrouping (normalizing) data to its correct hierarchical shape to produce correct results.

A1) Powerful Hierarchical Query Optimization and Efficiency

The Left Outer Join's natural hierarchical data preservation operation referred to earlier allows separate views and the entire unified heterogeneous view to be optimized hierarchically at runtime. This is an optimization to access only the nodes referenced or on a path to a referenced node, thereby saving on unnecessary data access without affecting the integrity of the view. (Inner joins because they have no data preservation must always access all table in a view or query.) This optimization also significantly cuts down on data explosions caused by semantically incorrect and confusing data replication, and the inefficiency they cause in memory and CPU usage. This is demonstrated below in Figure A1 where nodes B and D from ViewX are not accessed.

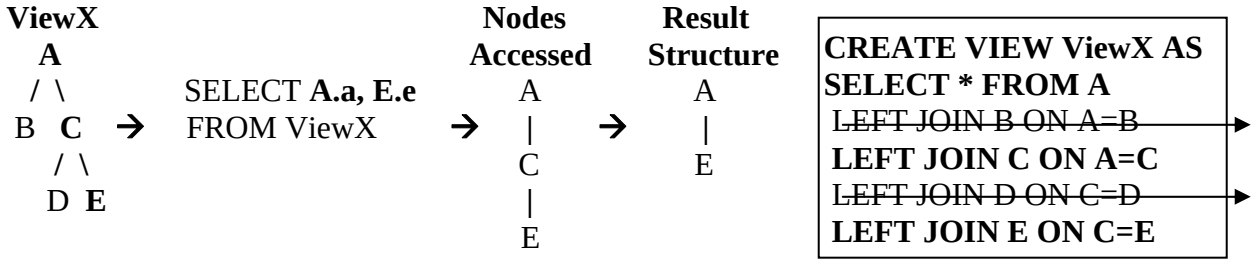


Figure A1: Hierarchical access optimization

You can also see in Figure A1 that node C while not referenced is still required for navigating from node A to node E. Node C will be removed from processing since it was not selected for output. The lack of access of the optimized out nodes (B and D) has no negative influence on the result because of Left Outer Join hierarchical preservation and improves upon the semantic accuracy of the result by reducing unnecessary data replications. This optimization is performed by SQLfX® transparently by dynamically and logically removing the unneeded Left Outer Joins from the view at runtime by simulating the view so it has control over what tables get processed.

This optimization's power is significantly increased by how easily the SQL SELECT clause can be changed and the view can automatically adapt by eliminating unnecessary nodes from the defined structure. This is a form of semantic optimization driven by the data structure's metadata instead of physical views that use procedural programming instructions with smaller optimization windows.

Global views of entire structures become very useful with this hierarchical optimization. Global views result in fewer specialized views, further increasing the reuse and hierarchical data abstraction for the user not needing to be concerned with details of the structure being processed. This also allows global views to be used without incurring any overhead. In a nonprocedural 4GL like SQL, global views automatically increase the database domain of the queries using these large views. This makes them very user friendly by eliminating the use of small views and having to know when and how to use which view.

This database access optimization is based on the metadata structure information represented in the Left Outer Joins. This means that physical hierarchical structures can use this path elimination optimization

also. In addition, since the outer joins represent the hierarchical structure, the access strategy for physical hierarchical structures does not need to simulate expensive outer joins. The specific access procedure for each hierarchical structure type can access the hierarchical structure in the most efficient way for its type and its result is still Left Outer Join compatible. This is truly a global optimization because the entire multi-view hierarchical query is optimized using this optimization technique.

A1.1 Hierarchical Optimization Underlying Principle Demonstrated

Normally with Inner joins, if one of the matching sides of the join is missing the other side is also removed even if it exists. The following SQL A1.1 proves the Left Outer Join does preserve the left side if the right side is missing. In this example, Cust is joined with Emp. You will notice that Cust02 is preserved even though it has no matching EMP.

SQL A1.1 **SELECT CustID, EmpID FROM Cust LEFT JOIN Emp ON CustID=EmpCustID**

```
<root>
 <cust custid="Cust01">
  <emp empid="Emp01"/>
 </cust>
 <cust custid="Cust02">
 </cust>
 <cust custid="Cust03">
  <emp empid="Emp02"/>
 </cust>
</root>
```

Now we will examine selecting CustIds only from the Cust table SQL A1.2. We do not join it with the Emp table since we did not SELECT any Emps. The result for displaying Emps excluding Emps matches the previous example SQLA1.1.

SQL A1.2 **SELECT CustID FROM Cust**

```
<root>
 <cust custid="Cust01">
 <cust custid="Cust02">
 <cust custid="Cust03">
</root>
```

Now lets simulate a view that is joining Cust with Emp in SQL A1.3, but the user is only access Cust. The data preservation we saw in A1.1 with Cust02 having no Emp should mean that this result should be the same as the previous one, SQL A1.2 and it is. This means our optimization principle for removing Table access has held up and is valid. This means there is no overhead for using global views or using smaller view for efficiency reasons. This also makes operation for the user easier and more user friendly.

SQL A1.3 **SELECT CustID FROM Cust LEFT JOIN Emp ON CustID=EmpCustID**

```
<root>
 <cust custid="Cust01">
 <cust custid="Cust02">
 <cust custid="Cust03">
</root>
```

A2) Full Ad hoc Interactive Operation

It is important to point out that all other SQL/XML integration solutions require proprietary solutions and procedural XML centric syntax. A down side of this is that ad hoc interactive processing is really not realistically possible if allowed at all. The SQLfX® ANSI SQL solution presented in this document is nonprocedural and transparent. It also supports full dynamic ad hoc processing support of XML and other forms of hierarchical data format such as legacy data. SQL remains ad hoc capable for the reasons listed in the Figure below.

SQL Ad Hoc Ability	Other XML Processors
SELECT list: Parameter driven	**Selected items placed in code**
FROM : Nonprocedural processing	**Physical procedural coding**
WHERE: Automatic nonlinear filtering	**No automatic LCA processing**
Hierarchical views: Flexible metadata	**Less flexible physical views**
Nonlinear processing: Increases data value	**Procedure code more error prone**

Figure A2: SQL ad hoc querying advantages

SQL hierarchical processing maintains its pure parameter driven SELECT list allowing the returned values to be specified without specifying or including them in processing logic. The FROM clause specifies the input data and its hierarchical relationships in a metadata form that allows the structure to be automatically navigated and globally optimized. The WHERE clause specifies nonlinear (multi-leg) hierarchical filtering which includes linear processing without the need to specify complex hierarchical filtering logic. SQL hierarchical views further increase the ease of use required for interactive processing.

A3) Hierarchical View Capabilities

SQLfX®'s XML and Relational hierarchical views are interchangeable as was shown in the SQL 12.4 and SQL 12.5 examples. They both represent the defined hierarchical structure's metadata and can be used as any other ANSI SQL view. Being treated as metadata and self contained, these ANSI SQLfX® views combine and transform naturally keeping track of the changing hierarchical structure. They also adapt to the query. This means the SELECT clause can automatically influence how the structures materialize by simple adding or removing a data item. This is not possible with the ANSI SQL SQL/XML centric functions or XQuery's function's representing structures. Both of these methods have the logic of the query embedded in their procedural XML generation.

This also makes XQuery's optimization difficult because of the procedural logic and database navigation. This limits optimization to chunks of logic code. SQLfX®'s metadata views represent hierarchical metadata which means large views can be used with no overhead. They are optimized based on what data is required at query execution time. This means that users can avoid using many smaller views previously optimized for each query, and can use larger views that will be optimally optimized at execution.

These larger views are also expanded naturally into a single seamless heterogeneous unified virtual ANSI SQL view which is used directly by the SQL processor. This unified view has also been hierarchically optimized consistently across the different database types. This produces the most heterogeneous and seamless processing possible with any processor.

A4) Duplicate Data Removal and Replicated Data Re-Normalization

Cartesian products created by relational joins create an explosive amount of duplicated data. This duplicated data is data created as relational placeholders for the row duplication caused by joining and needs to be removed when the relational rowset is transformed into a rowset or hierarchical result. Otherwise the results are not hierarchically correct and can lead to invalid results. This is a difficult operation that must be performed automatically, there is no user intervention, it must be performed transparently and accurately.

Duplicate data is different than duplicated data described directly above. Duplicate data is input and should represent duplicate data that is significant. It should not be removed because it is a duplicate. This gets tricky when there are no unique keys in the incoming data. This is possible with XML data where the data order may also require being unchanged from input.

Replicated data is similar to duplicated data as describe above, but its generation and use represents valid significant data occurrences not placeholders, its use represent distinct meaningful data occurrences. This replicated data occurs during data modeling which can also be triggered by structure transformations. For example, inverting a typical 1-to-M structure such as Department over Employee where a department can have many employees, to a M-to 1 structure of Employee over Department where Each employee know has its own department copy. This department copy is significant for each employee. Re-inverting this M-to-1 structure back to a 1-to-M structure (Department over Employee) should re-normalize (regroup) all the employees back under their single Department occurrence. This should also be done transparently; the user should not be concerned with this operation. This insures correct results.

XML query products today require that the user keep track and procedurally specify the data processing. This means duplicate, duplicated, and replicated data handling must be initiated by the user specifying the proper handling functions when necessary. In SQLfX® these are performed automatically to guarantee hierarchical processing accuracy. Additionally, XML Query products today are limited to linear processing. One of the reasons for this may be that full nonlinear hierarchical processing requires much more complex data handling for duplicated and replicated data handling. SQLfX® has solved this nonlinear processing problem and it automatically supports these capabilities today with its full nonlinear hierarchical processing.

A5) Reuse and Reusability

SQLfX® has huge reuse potential. Reuse was saved for the end of this document because it relies on many of the basic and powerful capabilities described previously. These are: outer join meta data and optimization; view structures; view structure with WHERE clause; global view structures; variable SELECT lists; variable structure generation; joining view structures in unlimited ways; global queries; linking below root of lower structure; dynamic control of XML Output; and reusing structure transformations.

A5.1) View Structures and Joining View Structures

Hierarchical structures created by Left Outer Joins can be placed in standard SQL views. This can be seen in Section 2.3. These views are easily reused by embedding and joining in many different ways. This can be seen in Section 5.1.

A5.2) Embedded View Structure Controlled by WHERE Clause

Structure views can be embedded and controlled by an external WHERE clause. This allows the view to be reused by changing its data content from an external WHERE clause. This was shown in Section 10.4.

A5.3) Variable SELECT List and Global View Structures

Variable SELECT lists are extremely powerful in changing the operation and specification of views as described in Section 2.3.1 The Left Outer Join operation can represent structure metadata and along with its hierarchical data preservation allow for a power hierarchical access optimization so there is no overhead for large global views that can service many different uses. This optimization is described in Section A1. Variable Selection can also be used with structure transformation views to dynamically select which nodes and data are output after the transform.

A5.4) Variable Structure Generation

The capability to generate variable structures from variable structures views opens the door for much reuse of these variable structure views. They generate the data structure dynamically based on data values in the data. This was demonstrated in Section 7.

A5.5) Linking Below Root

The capability of linking below the root of view structures increases their ability to be reused by many times over. This capability is shown in Section 6.1.

A5.6) Dynamic XML Output Control

The selection of the output format can be specified at query execution. So the same query processing can be reused to generate different XML formats and data content. See Section 3.7 for FOR XML dynamic output functions.

A5.7) Query Structure Independence

Since SQLfX® is navigationless and nonprocedural like standard SQL is, the same query could utilize different views that have different structures. This is shown below in Figure A5.7. This is a form of reuse. The same SQL query is depicted using a view that could have one of two structures and in this case each produce a valid result that is different depending on the structure operated on. This allows the user to query the structure without knowing the data structure.

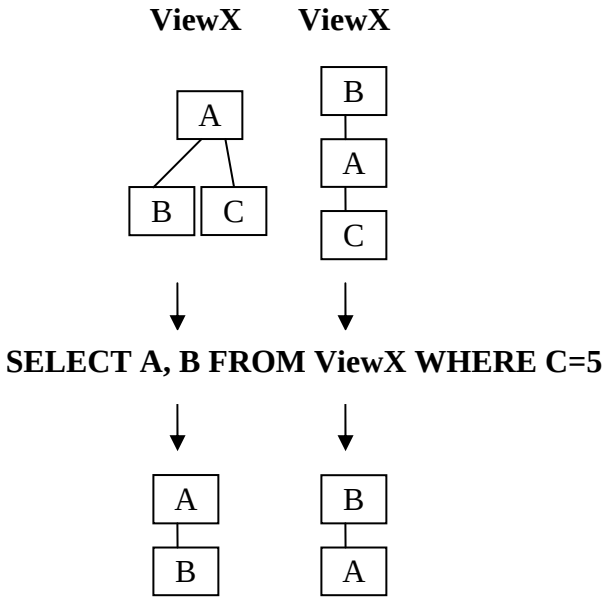


Figure A5.7: Query structure independence

A5.8) Reusable Polymorphic Reshaping

Polymorphic reshaping was described in Section 15.5. Polymorphic reshaping does not depend on the source structure's hierarchical structure to operate. This means that either version of ViewX from Section A5.7 above could be used as hierarchical source input in a polymorphic reshaping and it would produce the same transformed target structure in either case. This is shown below in Figure A5.8. This flexible polymorphic structure transform is another form of reuse by reusing polymorphic reshaping specifications for multiple different source structures requiring the same target structure.

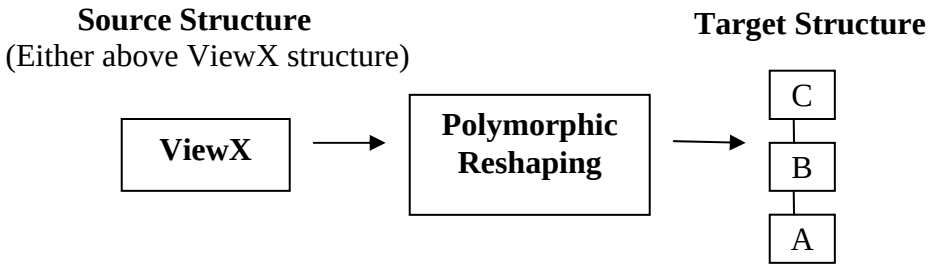


Figure A5.8: Reusable polymorphic reshaping

A6) Navigationless and Transparent Access

Transparent navigationless access is one of the most important capabilities of SQLfX® which is often overlooked just because they are seamless and transparent. They allow non technical users to use SQLfX® and to perform the most complex operations which are not feasible in procedural products like XQuery. This capability also guarantees the hierarchical processing to be accurate.

A7) Hierarchical Structure Aware

SQLfX®'s ability to automatically know the hierarchical structure at all times and act on it dynamically is one of the most important transparent operations it has. It controls hierarchical optimization, XML output hierarchical structure format, and SQL/XML mapping functions. Structure joining, structure data modeling, and transformations will dynamically modify the current data structure being processed.

A8) Automatic Distributed Hierarchical Processing

The performing of distributed hierarchical processing is automatic when distributed processing is supported and performed by the SQLfX® customer's SQL processor. When the hierarchical data modeling Left Outer Joins are broken up and sent to remote sites for processing, the hierarchical substructure fragments they represent will automatically be performed hierarchically. The returned results have been naturally processed hierarchically at the remote sites and still remain correctly hierarchically mapped at the local site. The final result remains fully hierarchically processed. This happens automatically because the hierarchical data modeling Left Outer Join specification is self contained in the SQL and each ANSI SQL site's ANSI SQL processor naturally performs the hierarchical processing defined by the data modeling SQL as shown in Figure A8.

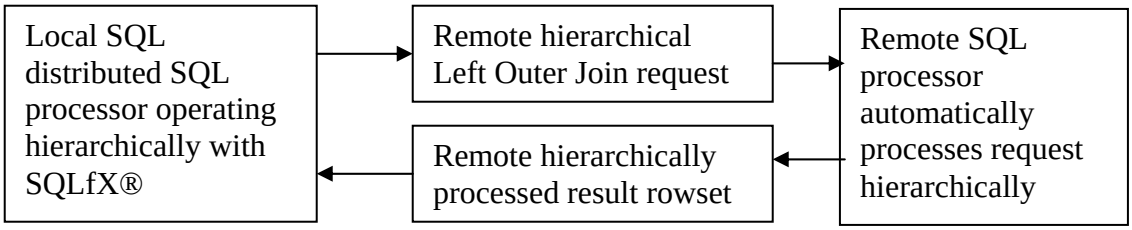


Figure A8 Automatic hierarchical distributed processing

B) Capabilities to be Completed for the Initial Release

These are other capabilities which enhance the SQLfX® operation. They operate naturally under SQLfX®'s inherent hierarchical operation and enhance or extend its operation.

B1) Additional FOR XML Options

These are for adding additional global type formatting and operational keyword options such as supplying a collector node dynamically only when it is needed. A Choice for how empty elements are specified or used could be useful. This could be a static setting or one that is dynamic dependent on some internal data condition. An XML output spacing control would be nice too. FOR XML could also be used to invoke internal tracing or to specify an XML output storage location.

B2) View-to-view Explicit Transformation

Our SQL any-to-any structure reshaping breakthrough described in Section 15 does require some thought in how it need to be expressed. We will also offer a view-to-view post processing nonprocedural explicit transformation capability making it easier to specify and having more options. It will be similar to current templates technology today, but requires no procedural coding or looping logic and is not static. This means it can automatically adapt to the input virtual structure which can change from query to query. Fixed format templates must be modified to handle different queries that change the data variables being retrieved. This is because with XQuery and ANSI SQL/XML functions, the SELECT list is embedded in the physical output template and loses its independent flexibility.

B3) Using XML's Duplicate and Shared Node Capability

Some of XML's unconventional features like duplicate nodes and shared nodes can be automatically tamed so that their defined XML elements can still be accessed and their data utilized by ANSI SQL SQLfX®. Duplicate and Shared Node capabilities are unconventional features which are closely related. Their integration solutions rely on the same ANSI SQL capabilities are described in this section.

B3.1) Duplicate Nodes

Duplicate node types are allowed in XML. These are used in XML by XPath to take advantage of this capability by navigating to the next closest duplicate node type if a closer duplicate node type from another location is missing. SQL usage does not make much sense of this for nonprocedural use because the multiple possible paths meanings are ambiguous. These duplicate node types can be used by SQLfX® by renaming them on input so that each one has a different name. This will keep the SQL query unambiguous as shown in Figure B3 below with a separate unambiguous path to the shared node. The SQLfX® XML view as shown in Section 12.1 will support aliases for nodes to enable duplicate nodes to be supported in SQL as shown in Figure B3 below.

B3.2) Shared Nodes

Sharing a node type using XML's IDRef causes network structures to be created in XML. SQL nonprocedural hierarchical processing can not handle network structures because they are ambiguous and require procedural navigation. ANSI SQL's Alias (rename capability) allows the IDref path to a shared node to be renamed another name than the shared node. This removes the network ambiguity and restores

the structure to a valid hierarchical form while allowing the shared node to remain shared as shown in Figure B3 with a separate unambiguous path to the shared node. The SQLfX® XML view as shown in section 12.1 will support the XML IDref and aliases to enable shared nodes to be supported in SQL as shown in Figure B3 below.

B3.3) Network Structures

Network structures unlike hierarchical structures can have multiple paths to any given node. This presents a problem for nonprocedural languages like SQL because network structures processed nonprocedurally results in ambiguous queries. XML with its duplicate named nodes and shared node capability supports a form of networks as shown below in Figure B3. This is the reason that all XML processors today require navigation and can not be nonprocedural. Since SQLfX® is a hierarchical product that maintains hierarchical accuracy; it will not support network structures directly. But it provides a way to transparently map them to hierarchical structures where the data can be unambiguously accessed nonprocedurally as can be seen in Figure B3. This allows hierarchical solid unambiguous principles to remain intact.

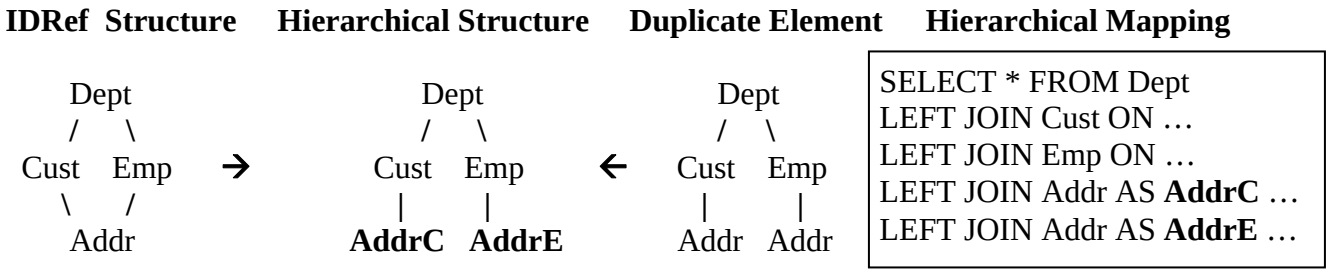


Figure B3: Hierarchical mapping solutions for network structures

The shared node capability does not provide any XML capabilities beyond our mapping solution above. But the duplicate node capability does offer a new unconventional capability of having the navigation not always pick the same node type at the same hierarchical location. While this capability is useful for processing markup, it is not that useful for database data. In fact the processing becomes unpredictable. So the SQLfX® solution is quite satisfactory and keeps SQL operating predictably.

B4) Add Real-time EII and Integrate With Batch ETL XML

There are two basic ways for SQL to access XML hierarchical documents. The first is where XML is shredded into relational tables in mass where it can be accessed at a later time relationally. This is a batch method that accommodates large amounts of data (many documents occurrences) but the data may become stale. The other approach is to have SQL access native XML dynamically (supporting EII) a document at a time. With this approach, the data is dynamically flattened to resemble relational rowsets where the document is accessed relationally. The two methods complement each other. Batch ETL for large amounts of stable XML data shredded into relational tables before hand, and SQL EII for smaller amounts of XML accessed dynamically and presented as a rowset. The batch method accesses, shreds and saves all the data fields identified in the identifying XML view so they can be used when needed, while the dynamic EII method accesses and retrieves only the fields necessary for the specific query being processed.

Both methods will use dynamic hierarchical access optimization as already described. SQLfX® can handle both XML access methods at the same time in the same query as indicated in Figure B4 below. In

this way XML ETL and hierarchical SQLfX® EII can be integrated seamlessly and both at a single high nonlinear hierarchical processing level. This also allows the DBA to transparently switch between batch or dynamic hierarchical operation or a balance of the two methods in a single query. The initial SQLfX® Beta version handles XML the batch ETL way. The dynamic EII way will also be in the initial product. Both methods operate the same in SQLfX® producing the same hierarchical result and are interchangeable.

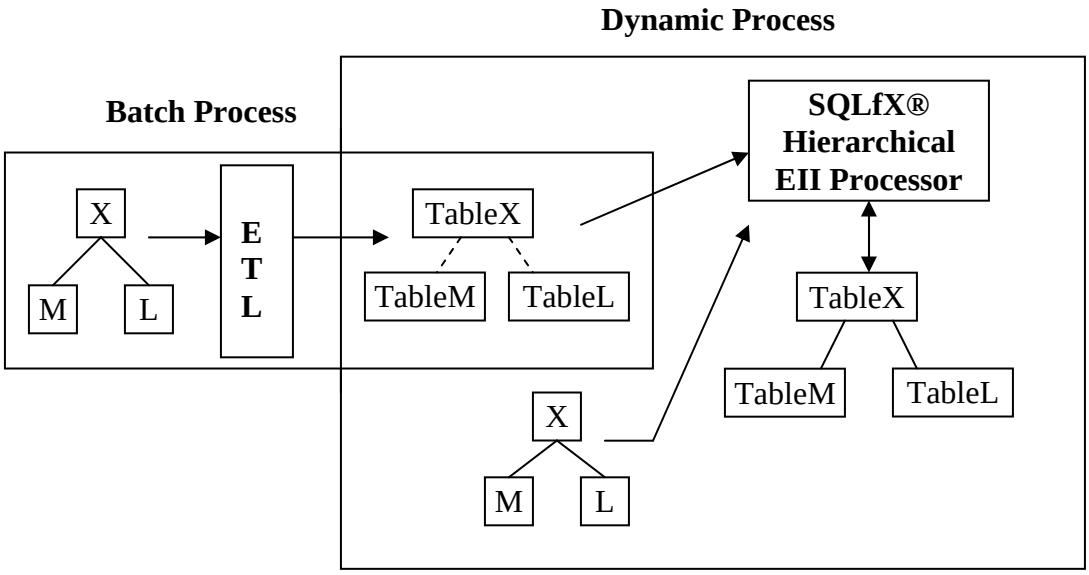


Figure B4 The Integration of batch and dynamic XML hierarchical native processing

B5) SQL Update Using Native XML Data

SQL updating of relational databases is possible involving data retrieved from hierarchical sources in the SQLfX® subquery. This is accomplished easily using the standard SQL INSERT operation and retrieving the data from a SQLfX® subquery that represents a hierarchical structure as previously defined in this document. The desired data is pulled from a particular data path node occurrence navigated by the WHERE clause of the sub query as shown below.

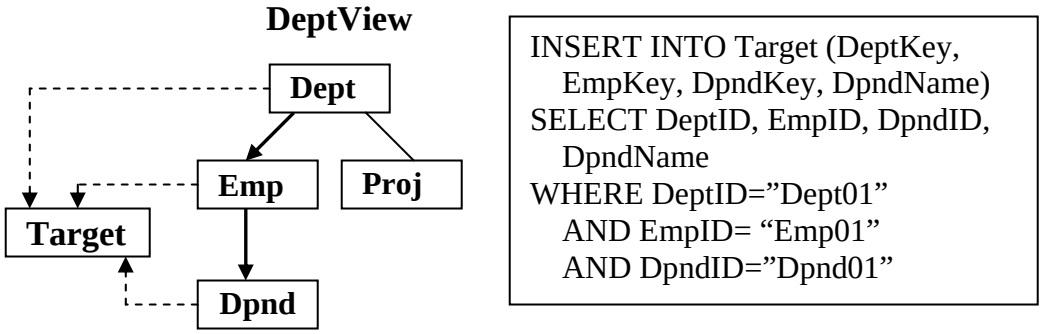


Figure B5: SQL Insert operation navigating a hierarchical data source

The DeptView above in Figure B5 has been defined hierarchically in SQL as described in this document. To locate the desired data path from the hierarchical DeptView is performed by using the WHERE clause to isolate the data path to the desired Dpnd node occurrence. The needed data in the isolated Dpnd node occurrence and the data up the path (identified in the SELECT list) are available to the Insert operation.

B6) Automatic Generation of Hierarchical Views

A utility for generating SQLfX® XML hierarchical views from an XML Schema and a GUI to assistance in producing relational hierarchical views will be provided. These views define the entire hierarchical structure so they will work for any query involving the structure. There is no efficiency penalty for these global structure views because the SQLfX® hierarchical optimization optimally optimizes every query so that there is no additional overhead for global queries, just additional ease of use.

B7) XML SQL View Definition Supports ON clause

XML SQL view definition supports Left Outer Join Statement that represents XML document. This is necessary to support the ON clause which is used for additional capabilities such as variable structure control.

B8) Support View.*

This view select all capability was not included in the Beta version. Its flexibility is needed.

C) Future Capabilities

Future capabilities are second level priorities and still need to be explored further. They may still make it into release 1.

C1) Capturing Additional XML Document Information

These include XML Comments, Instructions, Prolog and Prefix information.

C2) Grouping and Range Control

Hierarchical Grouping range control operates within a node and is controlled by data value. Structure range control is used with Existential operations and summary functions which operate across nodes and within the nonlinear structure.

C2.1) Hierarchical Data Driven GROUP BY and Summary Functions

Similar to the standard ORDER BY, The GROUP BY and Aggregation operations will require modifying the syntax to fit nonlinear hierarchical structures. Hierarchical and data driven groupings will be explored. Summary functions will be supported such as Min, Max, Avg, and Count

C2.2) Hierarchical Structure Driven Existential and Summary Functions

"ANY" and "ALL" Existential predicate modifiers and Summary Range specifier will be added as standard syntax that can specify the hierarchical range of their operation based on the structure being processed. These will more easily simulated sub query usage while naturally taking advantage of the hierarchical structure processing.

C3 Recursive Rollup

Nonlinear, multi-path Recursive Rollup support will be explored.

C4) Parallel Processing Capabilities

SQLfX®'s strict hierarchical structure processing and access makes it perfect for automatic parallel processing and this can be performed without requiring hints from the coder. One of the first uses could be applied to quad core chip technology.

C5) Legacy Data Hierarchical Integration

SQLfX®'s basic operation is not so much XML processing as it is generic hierarchical principled processing. For this reason, adding hierarchical legacy access such as IBM's IMS and structured VSAM (with its nested structure format) is very generic and can be easily added at a full hierarchical processing level. This hierarchical and user friendly level of SQL access for legacy data is not currently available.

C6) Denormalized Table Support

Denormalized table support can be added in the same manner that native XML flat rowsets are supported hierarchically with the addition of additional metadata for the denormalized data.

C7) Union Support

Appends XML results of multiple sequential SELECT operations and optionally place under same root.

C8) Advanced Drill Down

These include capability to use previous result structure in current query. This may include saving current query results that can be accessed in future queries.

C9) Positional Operations

Positional operations such as First, Last and [N] will be explored.

C10) XML Access Operations

Such as Xlink

C11) SQL Pass-through Capability

This allows specified SQL statements to be unconditionally passed down to underlying SQL processor for processing.

C12) XML and Hierarchical Processing Functions

These functions can be functions that supply information about the underlying structure metadata being processed, or functions that perform processing on the hierarchical data being processed and can optionally return data from the function operation. Such as controlling sibling leg order via new syntax.

C13) Hierarchical Keyword Search

SQLfX® performs database hierarchical filtering automatically, but does handle markup processing which is not conducive to database processing which is more fixed and unambiguous. Keyword search does require markup that is flexible and fuzzy. To handle this capability without affecting SQLfX®'s internal operation, we can introduce a text search function that handles this fuzzy requirement.

C14) More Data Types Support

Currently we treat all data as character and we need to support other types of data explicitly.

D) Our New SQL Technology Discoveries

Our SQLfX® technology is unique in that we have uncovered and utilize new capabilities in ANSI SQL processing that are were not 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.

D1) SQL Full 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 to define and model hierarchical structures while its semantics define its hierarchical processing. SQL is not aware of this hierarchical processing occurring.

D2) SQL Inherent 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.

D3) 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 using any other node than the lower level structure's root. During our research of SQL's natural hierarchical processing we discovered SQL could naturally handle this powerful new capability of linking below the lower level structure's root 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 active track of.

D4) Powerful Hierarchical Access Optimization

ANSI SQL does not inherently support hierarchical access optimization since SQL does not know it is performing hierarchical structure 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. This enables global views.

D5) 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 Reshaping to perform any-to-any structure transformations without the use of existing relationships in the data structure.

E) Capabilities Not to be Supported In SQLfX®

SQL has incredible power operating on hierarchical structures as this document has hopefully shown and proved. It has gained its control over hierarchical structures by performing only valid hierarchical operations based on solid hierarchical operations and semantics. This is a case of less is more. This means that not all XML capabilities can be supported without SQLfX® giving up its principled hierarchical operation.

E1) XML "ANY" Markup Processed by SQL Syntax

XML was designed for flexible markup and not for Database use. Processing Markup is a much less precise operation than database data. It has since been adapted for database use. Database structures while variable in some cases is still much more stable than the unpredictable hierarchical structures used for markup. This is why XML has the ability to create variable structures with no limits, "ANY" in XML. SQLfX® must impose XML processing limitations for capabilities used for markup to insure correct hierarchical processing. XQuery makes no claims for correct hierarchical processing, SQLfX® does. This does not preclude SQL functions from supporting markup processing operations.

E2) Processing Duplicate Node Types Without Aliasing

Processing XML duplicate named node types in a single structure has two problems for SQL processing. First, SQL being a nonprocedural language with self navigation needs every node type identified by an unambiguous name. This is handled by renaming (aliasing) the offending node in SQL. The second problem is that XML's support of duplicate node names is used as a feature when searching for a node name using XPath. This is because different node locations with the same name can be inconsistently selected making for a very ambiguous query. XML query academic products that support LCA processing have adapted this behavior to support LCAs that can dynamically change positions during query processing. Again, this is applicable for processing markup, but not database data.

SQLfX® Beta Document 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)

SQLfX® Beta Index

(Numbers are Section Numbers)

XML Transparency: 1.4.1
XPath Type Processing: See ON clause

