How ANSI SQL
Inherently Utilizes Hierarchical Structure Semantics
to Automatically Perform Nonlinear Hierarchical Processing

Every ANSI SQL Processor
Contains
A Powerful Nonlinear Hierarchical Processor

Technology that Extends the Life & Usefulness of ANSI SQL

**Nonlinear Hierarchical Structures** are full hierarchical structures with multiple legs making the data on different legs not linearly related, but nonlinearly related.

**Nonlinear Hierarchical Queries** have the ability to reference and semantically process any combination of multiple legs in a nonlinear hierarchical structure. This greatly increases query correctness and ease of use, and the meaningfulness and value of the nonlinearly related data.

**Nonlinear Hierarchical Processing** is the ability to process linear and nonlinear hierarchical queries nonprocedurally and without navigation. This is achieved by automatically analyzing the full nonlinear hierarchical structure taking into account all of its structure semantics together with the nonlinear query requirements. This advanced processing is automatically performed for the user dynamically increasing the data value and guaranteeing that the hierarchical query results are correct.
8) More on the Internal Processing of Hierarchical SQL
8.1) Right Sided View Nesting
8.2) Advanced Lowest Common Ancestor Logic
     8.2.1) Compound Lowest Common Ancestor Logic
     8.2.2) Common Ancestor Type 2
8.3) Logical and Physical Structure Processing Consistency
8.4) Nonlinear Internal Hierarchical Navigation
     8.4.1) Single Leg Navigation
     8.4.2) Multiple Leg Navigation
8.5) Data Structure Extraction (DSE) Technology
9) Advanced Hierarchical Topics
9.1) Network Structures
9.2) Lowest Common Ancestors in XML with Duplicate Node Types
10) SQLfX® Uses ANSI SQL’s Hierarchical Processing for XML
    10.1) Actual SQLfX® examples
Conclusion
1) Introduction

Nonlinear (multi-leg) hierarchical processing is not a new or invented technology, it has been used in database processing before and has continued in use in many other technical areas such as artificial intelligence and even bio tech. Hierarchical structures are very good at representing data meaningfully which can then be processed as a whole in a nonlinear fashion increase the value of the data to solve complex problems. This nonlinear processing is a solid principled technology whose use in database design and operation prematurely fell to the wayside with the introduction of relational technology three decades ago. With the advent of XML and its current limiting linear processing level, it is time to utilize this powerful hierarchical processing technology and get past our flat relational mindset.

A significant part of XML’s usefulness comes from its meaningful organization abilities and concise non-data replicated hierarchical structure which allows any unique path or combination of paths to be processed. Unfortunately, the standard level of knowledge about hierarchical data structures, their multi-leg semantics, and advanced database application is very low which may explain their lack of use today. In this same regard, information on this subject is very limited. Hierarchical structures inherently contain extremely powerful structural semantics. Properly utilized, this semantics can automatically supply fourth generation languages (4GLs) with the information needed to perform nonprocedural database navigation and semantically complex hierarchical queries logically and intuitively for the user.

Today there is no formal principled technology foundation being used in hierarchical processing, and it has been limited to single legs (linear processing) ignoring the vast amount of semantics that naturally exists between the different legs. Nonlinear hierarchical semantics can be used not only to automatically process queries involving any single leg query, but also any multi-leg query. It can even process queries that access data from one leg of the structure based on data from another leg. This multi-leg processing is a common requirement that is very useful and meaningful, allowing the most semantically complex queries to be easily nonprocedurally specified. This capability significantly increases the value of the data.

![Hierarchical organization chart example](image)

Any time you look at an organizational chart such as in Figure 1.0 above, you are usually analyzing the relationships across different legs of the structure based solely on the intuitive semantics of the hierarchical structure. This demonstrates the advanced level of semantic hierarchical processing automatically possible nonprocedurally and how it increases the value of the data. The current XML query languages today are not utilizing the vast goldmine of semantics that exists naturally between all legs of the hierarchical structure to be able to nonprocedurally query any combination of legs in a full hierarchical structure. These full hierarchical structures can be defined by a single global view with no processing overhead or penalties experienced before.

A significant advantage of using large global hierarchical views by the 4GL user is that they do not need to be aware of the hierarchical relationships of the structure they are querying. The result will automatically reflect the valid relationships and meaningful result. These powerful 4GL nonprocedural hierarchical querying capabilities were very popular before relational databases came into conventional use. As it turns out, standard SQL can perform these powerful full multi-leg hierarchical queries inherently as will be shown.
2) Review of Data Structures

2.1 Types of Data Structures

Some background into data structures is necessary to understand hierarchically structured data, its semantics, uses, and why it is so powerful. There are three common and basic types of data structures: flat, hierarchical and network as shown in Figure 2.1 below. Flat data structures have no apparent structure. They are stored at a single level with no hierarchy of data, hence the term flat. They are simple contiguous structures that are typically relational tables and flat files.

![Figure 2.1 Different types of data structures](image)

Hierarchical data structures are multilevel structures and can contain multiple legs as shown in Figure 2.2 below. They are composed of data nodes (usually flat, contiguous, closely related, normalized) in a hierarchical multi-path tree structure. As a tree structure, they have a single start node known as the root node, and all other nodes are only located on a single path (one entrance). They offer a single unambiguous view of the data such as describing a particular application data view. For this reason, hierarchical structures can be navigated automatically by 4GLs because there are no choices of multiple paths to take to get to a particular node. The different node relationships between each other are: child, parent, ancestor, descendent, sibling, or cousin. Siblings are related by the same parent, and cousins are related by a common ancestor. Sibling node order under the parent node is application dependent.

2.2 Node Types and Data Nodes

The term “node” may be used generally as in the above paragraph which may apply equally to the node type (its definition in the structure) or data node (a node data occurrence). Sometimes this distinction is important when describing or understanding hierarchical semantics and in these cases the distinction will be made, these can be seen in Figure 2.2 below. With this in mind, siblings are the different children node types of a parent node type, the legs of these nodes are known as sibling legs. Twins are the different node data occurrences of a parent node data occurrence. They are very tightly related and for this reason they are also usually directly linked together under their parent node data occurrence. Twins of the same node type but not related by the same parent data occurrence are not considered meaningfully related.

![Figure 2.2 Node types and data nodes, and their relationship](image)
of them which can be retrieved with the usually get next call at the top level. A root node data occurrence and all of its related data nodes retrieved are analogous to a data record.

2.3) Hierarchical Vs Network Structures

Network data structures, as opposed to hierarchical structures, are for storing and processing more complex interlinked nodes which do not represent a single way of representing the structure or its access. They usually require procedural navigation because their nodes can be reached from multiple paths with each path representing a different semantics. They are useful for storing multiple combinations of overlapping structures, such as defining all the combinations of nodes that represent the valid different application views for a given database, a compound view. This is demonstrated in Figure 2.3 below.

![Hierarchical Structure vs Network Structure](image)

**Figure 2.3 Network structures can represent multiple hierarchical structures**

Network structures because of their intersecting paths store nodes separately connecting these nodes with pointers while hierarchical structures have the choice of having their nodes stored separately connected hierarchically with pointers or stored contiguously in a hierarchical nested fashion. Regardless of how stored, hierarchical structures with their single view of relationships between nodes actually possess a high level of unambiguous hierarchical semantics between all nodes. This can be extremely useful for nonprocedural processing of multiple applications using the same global view. This is not the case for network structures with their nodes connected in a network because they do not represent any single unambiguous view. Because of this they require procedural navigation to handle different applications.

Another significant difference between network and hierarchical structures is hierarchical data preservation and hierarchical data inheritance. In a network structure, a given node can be in multiple paths, so as long as it exists in a path, it can not be deleted. Nodes in a hierarchical structure have a fixed parent-child relationship. In hierarchical structures, child data nodes can not exist without their single parent data occurrence existing. If a parent data node occurrence is deleted or filtered from the query, all related descendent data node occurrences are also deleted. This will cause cascading deletes down the leg producing variable length leg occurrences. It also allows for hierarchical data inheritance down the leg of a hierarchical structure. This means the data on a path above any data node occurrence is also available. These are important and useful hierarchical characteristics.

2.4) Entity Relationships Review

Hierarchical structures also have entity relationships between each adjoining node on a leg that can be independent of its hierarchy, but have an effect on the data structure semantics and data modeling. These relationships are one-to-many (1 to M), many-to-one (M to 1), and many-to-many (M to M or M to N).
2.4.1 One to Many and Many to One Relationships

One-to-many is your typical Department to Employee relationship where a single Department can have zero to many Employees. The opposite relationship, many-to-one can be represented by an Employee to Department relationship where many Employees can belong to the same Department. While physical hierarchical structures like XML primarily use one-to-many relationships (one parent can have many children), logical hierarchical structures, like relational, can also support many-to-one relationships through data duplication as shown below in Figure 2.4.1. This duplication is perfectly valid and necessary to represent the M to 1 semantics correctly and produce a valid hierarchical result that is different than the 1 to M relationship applied to the same data as shown below. Sharing a node in this case is not valid since the Dept node must stay independent by employee so that changing a department for one employee does not change it for all employees.

![Figure 2.4.1 M to 1 and 1 to M representing the same data differently](image)

2.4.2 Many to Many Relationships

A many-to-many relationship is your typical Parts/Supplier relationship where a given Part can have many Suppliers while a given Supplier can carry many Parts as shown below in Figure 2.4.2. Physical databases have a difficult time supporting these bi-directional complex relationships while logical relational databases support them easily. This is performed by using an association table that contains all of the M to M bi-directional relationships so that the same association table can be used to represent the Part→Suppliers and Supplier→Parts relations. Using the association table, both of these relationships become a hierarchical 1 to M. This works by Part→Part/Supplier→Suppliers where Part/Supplier is the association table which can be turned around to support Supplier→Supplier/Part→Parts.

![Figure 2.4.2 M to M association table use example](image)

These association tables can also contain intersecting data at the intersecting point such as the price of a specific part from a specific supplier (Part/Price/Supplier). This intersecting data can be naturally attached to the lower level node (Part or Supplier) which has the proper number of data occurrences to hold
all the combinations because the association table has transformed the resulting relationship into a 1 to M. In this way the association table can be made transparent as shown above in Figure 2.4.2.

2.5) Logical and Physical Hierarchical Structures

Physical structures are fixed and must be modeled and accessed based on their fixed homogeneous structure. They do not need to be contiguously stored like XML which is nested, they can also be in separate pieces and related by physical pointers like IBM’s IMS database. Logical structures like relational structures are composed of multiple tables (rows are nodes) that are linked by logical data relationships. This means they can represent any number of logical structures which can be determined dynamically, thereby supplying data independence, Figure 2.5 below is an example of this.

Logical and physical hierarchical structures can both be represented in the same hierarchical structure because they have the same attributes and principles of operation. Having the same basic hierarchical structure, logical and physical hierarchical structures can be seamlessly combined into a larger heterogeneous logical structure. This means, for example, that two XML documents can be easily and seamlessly joined using a relational association table in the middle to support M to M relationships as shown below in Figure 2.5.1. The new structure is a heterogeneous logical structure.

2.5.1 Heterogeneous Association Table

Logical and physical hierarchical structures can both be represented in the same hierarchical structure because they have the same attributes and principles of operation. Having the same basic hierarchical structure, logical and physical hierarchical structures can be seamlessly combined into a larger heterogeneous logical structure. This means, for example, that two XML documents can be easily and seamlessly joined using a relational association table in the middle to support M to M relationships as shown below in Figure 2.5.1. The new structure is a heterogeneous logical structure.

2.6) Hierarchical Data Structure History

Relational databases became popular because of their flexible data independence that allows tables to be combined in any way. This put earlier hierarchical databases in a bad light because they prevented data independence because of their fixed hierarchical data structure. This mistakenly put all hierarchical data structures in a bad light. This unfortunately ignored the usefulness of logical hierarchical data structures which unlike physical hierarchical structures does not prevent data independence, it increases it.
Because of this mistaken popular belief, most commercial database software that used hierarchical data structures went into disfavor and eventually disappeared. This is the main reason there is so little information on hierarchical data structures and their advanced processing capabilities today. This also explains why there are so few Fourth Generation Languages that operate automatically based on the inherent semantics of the hierarchical structure being processed.

### 2.7) Single Vs Multiple Node Type Hierarchical Processing

Hierarchical processing in SQL today is performed by external hierarchical processing SQL functions. There are many ways to build an external hierarchical processing system on top of SQL. External hierarchical processing systems require programming and use procedural functions and external hierarchical navigation information stored in the database. Because of this additional external processing complexity, the structures are usually limited to a single node type. Single node type DB systems have no twin support, recursive structures are used to form a hierarchical structure. The adjacency list model is used to hold and control the simple single node type like the Employee Organization structure containing control data shown below in Figure 2.7.

**Adjacency List**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
<th>Parent</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Ben</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Fred</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Bert</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 2.7 Adjacency List model hierarchical structure**

The above characteristics mean that external single node type hierarchical systems are more limited and not general purpose. For all of these reasons, external hierarchical databases are usually limited to simple structures. SQL-99 does support nested tables which can represent hierarchical structures, but they also require procedural methods to be written and are not general purpose. These adjacency list model hierarchical processing systems are not integrated with SQL, they use external processing while the retrieval and filtering of the data is still performed by SQL external functions.

Nested relational internal hierarchical SQL database systems were popular for a while, but since they required new nonstandard syntax they were not seamless and did not catch on. In addition, they required a static schema definition limiting hierarchical flexibility and ad hoc processing. Fortunately, as will be explained in more detail, it has now been determined that SQL with its general purpose hierarchical processing based on ANSI SQL hierarchical join relationships automatically performs full nonlinear hierarchical processing.

The hierarchical join relationships enabling inherent hierarchical processing define multiple heterogeneous node types that support twin logic. These multiple node type database systems allow very complex hierarchical structures such as XML to be naturally modeled and processed internally hierarchically. This means that no SQL hierarchical processing functions need to be written, the navigation and processing is also performed automatically and nondeterministically by SQL inherently. It is a powerful general purpose solution, and being performed internally in SQL making it efficient and seamless. This is the type of hierarchical structure processing that this document concentrates on. By being naturally integrated into SQL, the data is treated hierarchically from the start as it is hierarchically retrieved, filtered, processed, and output.
2.8) Recursive Structures

Recursive structures are a special case of hierarchical structures where a piece of the structure can double back on itself to physically represent a new lower level substructure occurrence. This can happen indefinitely as needed as in a parts explosion where parts consist of other parts. XML supports the definition of complex nonlinear hierarchical recursive substructures composed of repeating multiple node type sequences easily in its hierarchical type structure as shown in Figure 2.8. SQL-99 supports flat linear recursive structures in its data controlled hierarchies. This is performed by a recursive operation that brings in data recursively through a loop which accesses recursively related rows and UNIONs them back into the result where they will also eventually go through the same process as shown in Figure 2.8. Recursive information indicators can be inserted into rows to help with the processing of the result.

![Figure 2.8 Recursion techniques](image)

From Figure 2.8 above, one problem is that SQL recursive hierarchical structures are limited by their linear row structure. On the other hand, nonlinear recursion can create very complex structures difficult or impossible to process nonprocedurally. The SQL hierarchical type hierarchies documented in this paper can change this by opening up more possibilities leading to a combination of both recursive methods.

3) Hierarchical 4GL Capabilities

Fourth generation languages are nonprocedural languages. They are also known as declarative languages because the user only needs to specify what they want and not how to get it. In order for this to work, the 4GL needs to know the hierarchical data structure being processed. Using the inherent meta information derived directly from hierarchical structures definitions, the 4GL can automatically determine the semantics in the data structure. This can be naturally applied to the processing of the query without having to be supplied with the hierarchical processing instructions or logic. This means that no procedural structure navigation is necessary, in fact navigation becomes transparent as shown in Figure 3.0. This is not true for XQuery and XPath. This automatic use of the structure semantics dynamically increases the value of the data.
With the advent and popularity of XML today, hierarchical data structures are coming back in a big way. Unfortunately, maybe because of the lack of knowledge about hierarchical structures and their capabilities, XML query languages have not taken full advantage of hierarchical structures and the powerful structural semantics they contain. This results in XML query languages having to be procedurally supplied with hierarchical semantic logic causing significant difficulty in specifying hierarchical queries that operate across multiple legs of the hierarchical structure as shown in the nonlinear query in Figure 3.0. The programming complexity level increases as the number of legs involved in processing the query increases. This can make semantically complex queries not worth the time or effort to specify procedurally. It also prevents the ability to specify ad hoc queries, and increases the chances of inaccurate results from faulty hierarchical processing logic.

Multi-leg queries greatly increase the hierarchical semantics involved and complexity level necessary for processing them. While more complex to process, multi-leg queries processed by nonprocedural 4GLs are easy and intuitive for the user to specify, are useful in decision support, and can avoid using procedurally complex queries. More importantly, being able to freely query the entire structure nonprocedurally allows any query to be processed regardless of the number of legs involved. This frees the ad hoc user or query developer of any concerns or knowledge of the structure. The 4GL user does not need to know the location of the data or hierarchical relationships of the structure being queried. The complex hierarchical operations which can be determined by the hierarchical data structure semantics are not well understood or documented today, but can be performed automatically by a 4GL.

3.1) Hierarchical Semantics and Lowest Common Ancestor

Full nonlinear hierarchical structures as described in this document require the use of multiple node types and twins. Twins mean that there can be multiple data occurrences related under their parent node data occurrence. Hierarchical structures built on top of relational databases do not usually support these capabilities. Twins (multiple data occurrences) are more tightly related than siblings (multiple legs) giving them a different semantics. Siblings affect the hierarchical structure model and twins do not. For this reason twin semantics are often overlooked in properly processing hierarchical data meaningfully. Data node occurrences not related by the same parent node occurrence are not usually considered meaningfully related and if treated as they are will usually produce an incorrect result. This is why the use of “occurrence” or “data occurrence” is used liberally and meaningfully in this paper.

The data representation below in Figure 3.1 represents one database record occurrence with all of its data occurrences. C1 and C2 are twins under data occurrence B1. C1 and C2 are directly related but since they are both C node data occurrences they usually occur only one at a time during hierarchical processing. They can be aggregated, but comparing them directly to one another is not usually supported in 4GLs. C1 and C3 are not meaningfully related because they have different parent occurrences. C1 and B1 are related but C1 and B2 are not meaningfully related because they have a different ancestor occurrence.
As stated previously, hierarchical data structures have an unambiguous semantics because they have only a single path to every node. This allows a nonprocedural query language to navigate them automatically and process the request utilizing the unambiguous semantics in the structure. The different legs of a hierarchical structure are independent of one another, but are related indirectly by the Lowest Common Ancestor (LCA) nodes that connect them. This makes the nodes on different legs indirectly related which is referred to here as cousins. Each common ancestor data node occurrence naturally coordinates the hierarchical control logic necessary for hierarchical processing across legs. This means that a single node data occurrence on one leg of the structure is also related to all other node data occurrences for sibling legs under their Lowest Common Ancestor node data occurrence.

As an example of this LCA logic in Figure 3.1, sibling E1 is related to both F1 and F2 because of their Lowest Common Ancestor data occurrence D1, and E1 and F3 are not meaningfully related because they belong to a different Lowest Common Ancestor D occurrence even though they are related under node A occurrence. The number of common ancestor nodes necessary to process the query increases with the number of legs referenced in a query.

As mentioned above, hierarchical data structure semantics today are currently being utilized only on a single leg at a time. A primary advantage of a full hierarchical structure is that it can define any combination of the legs necessary to process any multi-leg query. This semantics applied across the multiple legs means that the semantics between every node in the structure are meaningful and can be utilized for hierarchical processing. Typically a 4GL user querying data nonprocedurally from a full hierarchical structure is naturally going to reference multiple legs. This is because the user does not need to know the structure or be restricted by it. This multi-leg processing can be handled automatically by a 4GL utilizing the inherent semantics and principles that occur inherently between all nodes in the entire structure. Hierarchical semantics and operations remain consistent independent of how the structure is stored and represented physically or logically.

3.2) Hierarchical SQL Opportunity

Every SQL processor has the natural ability to perform full nonlinear hierarchical processing. SQL’s underutilized and overlooked inherent hierarchical processing capabilities offers proof of what was stated earlier about current underutilized hierarchical capabilities in XML. SQL can model hierarchical structures directly using the SQL-92 ANSI SQL Left Outer Join operation as shown in Figure 3.2 below. Tables and XML elements are nodes in the hierarchical structure. The Left Outer Join hierarchically preserves the left database object over the right database object and links them using the ON clause join criteria supplied at each specific join point.

© Copyright 2006, 2007 Advanced Data Access Technologies, Inc. All rights reserved. Page 12
When these data modeling outer joins in Figure 3.2 are directly processed by the relational engine, the engine automatically operates at a full hierarchical level. This means hierarchical processing is a valid subset of relational processing and shares its mathematical correctness. This makes XML integration in SQL seamless and capable of full hierarchical coverage carried out completely nonprocedurally and automatically. One of Advanced Data Access Technologies (ADAT) Inc.’s main goals is to utilize its hierarchical processing technology and expertise to fully utilize XML’s hierarchical processing capabilities in its innovative SQL products. These new capabilities further utilize or seamlessly extend SQL’s inherent hierarchical processing capabilities.

SQL’s nonprocedural processing utilizes the semantics naturally present in multi-leg hierarchical data structures. This drives the nonlinear hierarchical processing which can automatically correlate information throughout the entire multi-leg structure and can also be performed in ad hoc mode to support powerful decision support. This significantly increases the value of the data by automatically being able to dynamically process very semantically complex queries. This takes on staggering importance and value with the goldmine of hierarchical semantics naturally existing in XML data structures and is also available for legacy hierarchical data sources which can also be processed hierarchical by SQL.

4) ANSI SQL as a Hierarchical 4GL Query Processor

With ANSI SQL being a 4GL, its data modeling SQL syntax and semantics automatically instruct the SQL relational engine how to perform the hierarchical query processing. This is because all nodes are inherently related semantically and can be automatically processed hierarchically as stated earlier. This enables SQL to perform hierarchical processing naturally by following the semantics in the data structure being processed. The data structure is defined hierarchically by the Left Outer Join operation which unlike the standard and default Inner Join operates hierarchically. This enables complex hierarchical multi-leg queries to be processed nonprocedurally and automatically. For example, this allows an SQL query that selects data from one area of a structure based on data in another area of the structure to be performed automatically. The logic involved uses the Lowest Common Ancestor (LCA) nodes between the referenced legs to coordinate and correlate the multi-leg processing based on the hierarchical semantics.

A simple SQL query involving more than two legs would involve multiple Lowest Common Ancestors and their interrelationships making the internal logic increasingly complex. Fortunately, nonprocedural relational languages such as SQL can do this accurately and automatically regardless of the semantic complexity. This is because the relational hierarchically restricted Cartesian product processing naturally and automatically generates all valid hierarchical combinations between the legs of each Lowest Common Ancestor node data occurrence as shown in Figure 4.0. This automatically and accurately replaces the normal hierarchical tree walking in physical hierarchical structure required for LCA processing by the standard relational engine.
ANSI SQL Nonlinear Hierarchical Processing

which tests all valid hierarchical LCA combinations operating a row at a time. Current XML query languages such as XQuery are controlled procedurally by user supplied multi-leg hierarchical semantic logic. This can make them impracticable for semantically complex multi-leg hierarchical operations.

![Hierarchical Data Structure Rowset Data](image)

Figure 4.0 Rowset hierarchical representation

With the hierarchical structure data preservation from Left Outer Joins, each separate leg of a hierarchical structure can dynamically vary in number of data types from leg occurrence to leg occurrence. Surprisingly, this does not present a problem for fixed column relational data. Variable column legs in SQL relational databases are logically represented correctly hierarchically in form and length in standard fixed column relational rowsets. This is because the variable missing trailing columns of every leg occurrence are automatically padded with Nulls as a normal operation of the SQL Outer Join data preservation operation as shown in Figure 4.0. This allows a variable number of columns in hierarchical leg occurrences to be processed accurately in fixed length rowsets dynamically preserving the fixed alignment. This enables seamless and transparent operation of variable depth legs.

SQL’s automatic processing of full multi-leg structures has all the same operational characteristics and capabilities as any physical or logical hierarchical structure being processed by older proven nonprocedural hierarchical processors. This relationally backed processing also validates that this is the correct mathematically correct way of hierarchical processing. Hierarchical processing and its semantics operate the same for physical (XML) or logical (relational) structures. This enables a consistent and seamless operation across heterogeneous structure formats when SQL is automatically performing hierarchically. There will be further coverage on physical and logical structures later.

4.1) Hierarchical Query Specification

A typical SQL request specification uses the familiar SELECT, FROM, and WHERE clause syntax. The following description explains how this syntax naturally applies to hierarchical processing when SQL is processing hierarchically modeled structures. Since the SELECT clause relies on the data specified in the FROM clause, the FROM clause will be discussed first. The FROM clause specifies the location of data objects (i.e. SQL tables and XML elements) that form a pool of nodes consisting of relational tables and XML elements which will be placed in the input working set as needed. All required data types for processing the query must be included there. The Left Outer Join operation is specified in the FROM clause to specify how the nodes are related hierarchically into a single structure. This can be abstracted into separate SQL views representing meaningful substructures that can be combined in different ways.

The SELECT clause specifies which data items (i.e. table columns, element attributes) in the modeled structured defined in the FROM clause are to be returned. If any data item in a node is returned then that node is represented in the returned structure, otherwise that node is excluded in the returned structure. The
WHERE clause is used to return only the qualified data occurrences. If no WHERE clause is specified than no nonlinear data filtering is performed. This is shown in Figure 4.1 below.

![Figure 4.1 SQL hierarchical query specification and operation](image)

**4.2) FROM Clause Controls Hierarchical Data Modeling**

The data associated with the left argument of the Left Outer Join is hierarchically preserved over the right data argument because it is preserved even when the right argument’s data is not present. Using this capability, any hierarchical structure can be modeled and processed hierarchically by the relational engine performing the hierarchical semantics associated with the SQL hierarchical modeling syntax. This is shown below in the definition of the ViewX structure in Figure 4.2 below. You can notice how legs are created by modeling them going down the structure and how multiple legs are formed when ON clauses used to supply join criteria link back to nodes already with a formed or partial formed leg. This SQL hierarchical Left Outer Join view can be easily generated automatically from existing meta data sources such as XML schemas.

### Figure 4.2 SQL hierarchical data view

The SQL hierarchical data modeling syntax in Figure 4.2 above is also directly executable by SQL to perform hierarchical processing. This is possible because the associated SQL semantics of the data modeling SQL is specifying the basic principles of hierarchical structures and their processing. This makes for an extremely tight and seamless bond between SQL hierarchical data modeling and its associated hierarchical processing assuring data modeling accuracy, efficient processing, and an open and available data modeling language and its accompanying hierarchical processor. Also realize that this SQL data modeling syntax is a self defining data structure definition that accompanies the SQL were ever it may go. These powerful characteristics have very useful implications that will be examined further.

### 4.2.1) Joining Hierarchical Structures

In the same way that the hierarchical data modeling was performed a node at time using Left Outer Joins in Section 4.2, hierarchical substructures defined in SQL views can be joined (linked) 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 4.2.1 below. The ON clause takes on
added importance because it also specified the join points within each structure, in this case it is the C node linked to the X node.

![Diagram of hierarchical data structures](image)

**Figure 4.2.1 Joining hierarchical data structures conceptually**

The SQL input views in Figure 4.2.1 above are modeling hierarchical structures. Since logical and physical structures have the same semantics and operational principles, this means that an SQL view can hierarchically model relational or XML data. XML data can be retrieved and returned as a rowset which is mapped by its hierarchical SQL view. This allows it to be accessed seamlessly at the SQL data item level. This is why the data structures referenced in Figure 4.2.1 above can consist of relational and XML data. More information on XML access can be found in Section 5.1.

Since the SQL hierarchical modeling views are SQL and the invoking statement is SQL, they combine naturally into a unified SQL executable statement. This means the SQL joining of the Left Outer Join views (ViewX and ViewY) in Figure 4.2.1 above automatically expand into a unified virtual SQL view that correctly maps the combined heterogeneous hierarchical structure being processed in the SQL query. This enables seamless access and processing across the heterogeneous global virtual structure. Also notice that the level of data modeling is dynamic and at a very high conceptual hierarchical modeling level. It can manipulate structures as whole entities as can be seen visually above in Figure 4.2.1 and below in examples to follow.

### 4.2.2) Linking Below the Lower Level Structure’s Root

The joining of the two hierarchical structures in Figure 4.2.1 above is standard for hierarchical structures since the lower level structure was linked to its root node. The semantics of the combined hierarchical structure are very intuitive as shown. But, what would the semantics of the combined structure if the lower level structure was linked to its lower Y node instead of its root node X by replacing the ON clause with: ON C.pk=Y.fk as shown in Figure 4.2.2. Interestingly this capability is naturally supported in SQL and the combined new structure generated remains the same as if the root was linked to as shown above. This is easily understood and makes sense since the root still exhibits the same inheritance and subordination effect on its structure. As you might expect, data filtering is applied to the joined lower level structure at the lower link point, the Y node in this case. This significantly increases user friendliness by not requiring the user to know the structure and not imposing further restrictions on joining structures.

So any occurrences of Y that do not match the join criteria are filtered out along with all of their dependent Z node occurrences. Also note in Figure 4.2.2, that cousin nodes like W that qualify from common ancestor nodes persist also. More information on the linking below the root operation will be discussed in Section 8.1.
4.3) SELECT Clause Controls Node Promotion

Data selection, known as projection in relational terms, is specified by the SQL SELECT clause. It controls which data types from the input working set defined by the FROM clause are moved to the output structure (a fixed vertical/column kind of filtering). Nodes with no data selected are not moved to the output structure which is standard relational processing. This slicing out of unselected output nodes is also standard hierarchical processing and is known as node promotion because the removed node’s selected descendent nodes are preserved by being naturally promoted up and around the removed nodes following the hierarchical form of the data structure. This is shown in Figure 4.3 where node C was not selected for output.

The selected nodes in Figure 4.3 above also maintain their hierarchical semantics because the remaining nodes still exert the same hierarchical effect and semantics on each other (subordination is maintained). This is because the structure is naturally condensed in a hierarchically controlled manner. Also notice that nodes D and E with separate paths directly under node A is an example of node collection. All these operations are basic hierarchical processing and are represented in the relationally flat hierarchical structure in Figure 4.3.

4.3.1) Fragment Isolation Using Node Promotion

Fragments are pieces of a hierarchical structure similar to substructures but more dynamic. This definition is extended here to include node promotion caused by node exclusion. By extending upon the same data selection principles used in node promotion demonstrated above in Figure 4.3, it is possible to isolate a hierarchically structured fragment in a view. This is demonstrated below in Figure 4.3.1 where a fragment is isolated by only selecting on data in the C, D, and E nodes.

© Copyright 2006, 2007 Advanced Data Access Technologies, Inc. All rights reserved. Page 17
4.3.2) Structure Transformation Using Fragment Isolation

By utilizing fragment processing shown above in Section 4.3.1 and hierarchical structure joining in Section 4.2.1, it is possible to isolate multiple structure fragments in a single structure (view) and then manipulate them with join operations. This is fairly simple, but involves another level of complexity by renaming the view and using the new view names as qualifiers to distinguish the two fragments. This is demonstrated in Figure 4.3.2 below where two fragments are formed and separately reconstructed from the same SQL view. These are fragments A-B indicated by the V1 qualifier and C-D indicated by the V2 qualifier, which are used to independently manipulate and perform structure transformation by rejoining the fragments differently.

As shown in Figure 4.3.2 above, transforming data structures in standard SQL is possible by defining different fragments in the same structure by using the SQL high level prefix capability to keep them separate and distinct. Both logical and physical structures once retrieved to the working set in memory are stored as contiguously fixed structures. This allows any fragment in the working set to be naturally isolated and moved separately to the result set using standard hierarchical processing. This is possible because all accessed data types, such as logical relational, contiguous XML, and even linked IMS data forms are now stored in a contiguous homogenous rowset in the relational working set. This means the structured data in the working set can be treated as a fixed contiguous rowset and can be logically joined with other rowsets irregardless of the makeup of the source data format. See Section 8.3 for additional information on logical and physical structures.

4.4) WHERE Clause Controls Hierarchical Data Filtering

Hierarchical data filtering is controlled by the WHERE clause. It specifies which data row occurrences are filtered out based on their data content. This is a dynamic horizontal filtering of rows or path data.
occurrences. Data filtering will affect which data is in the result set on a row by row, path by path data occurrence basis. It is important to realize that the WHERE clause is applied (at least logically) after all of the joins and processing is performed in order to remove entire rows.

One of a hierarchical query’s most powerful capabilities is its hierarchical WHERE clause data filtering which is applied to the full hierarchical structure. To fully understand hierarchical query data filtering which is more involved than flat structure filtering, it is more easily understood if examined as data qualification. This is because it is simpler to demonstrate a positive rather than negative operation when demonstrating the natural hierarchical filtering operation. This means that when hierarchical data filtering is used, it is examined as qualifying data rather than specifically filtering out data. When no WHERE clause filtering is specified, all the queried data is qualified.

4.4.1) Hierarchical Data Qualification

WHERE clause data qualification operates on the entire hierarchical structure in a hierarchical fashion based on how all the node types relate to each other. This same hierarchical process is occurring in standard SQL and relational table processing, but takes on hierarchical meaning (semantics) when the entire structure is recognized and the result is examined against its hierarchical structure. This semantics is more easily traced through the hierarchical structure following the related data qualified by the WHERE clause which is described below.

ViewX below in Figure 4.4.1 consists of tables A, B, C, D, E joined into the hierarchical structure as shown. The tables become nodes in the structure when modeled in SQL. If “A”, “B”, “C”, “D” and “E” are selected from this structure qualified on some value for “C” as in: SELECT A,B,C,D, E FROM ViewX WHERE C.val=’C5’, what are the data selection semantics of this query? The qualification process starts at the WHERE clause condition that directly qualifies the C data node occurrence(s) with C.val=’C5’. Then all path data occurrences under the qualified C data node occurrence (D and E node occurrences) and the path data occurrence above each qualified C data node occurrence (A node related data occurrence) qualify.

![Figure 4.4.1 WHERE clause data qualification flow](image)

The above described WHERE clause in data selection logic in Figure 4.4.1 covers directly related qualification logic familiar today, but multi-leg qualification includes any node path data occurrences indirectly related across the legs. These are connected to a qualified node data occurrence such as the related B node cousin data occurrences related by the qualified A node data occurrence in ViewX above in Figure 4.4.1. This same hierarchical result exactly reflects the result found in the relational result, with the meaning obscured by the flat representation. The hierarchical semantics and associated data are still available for hierarchical use. It can enable structured XML output to be nonprocedurally produced.
4.4.2) Multi-leg Filtering Semantics with AND Logic

When multiple legs take part in data filtering as in \( \text{WHERE } D.\text{val}=D_1 \text{ AND } E.\text{val}=E_0 \) or \( \text{WHERE } D.\text{val}=.\text{val}E \), using the ViewX below in Figure 4.4.2, all combinations across the qualified data value occurrences under the Lowest Common Ancestor data occurrences of node C are processed for a matching combination that qualifies. If lower level nodes under the qualified LCA C node are selected, than all combinations that qualify must be processed since the lower level nodes can be qualified separately by different qualified combinations. This is also the same logic performed in hierarchical processing which is naturally reproduced by the relational Cartesian product processing.

![Figure 4.4.2 AND filtering across legs](image)

Many queries will have data filtering criteria applied to more than two legs which can produce more than one Lowest Common Ancestor node filtering processes as described above. Using the hierarchical structure ViewX above in Figure 4.4.2, data filtering as in \( \text{WHERE } D.\text{val}=D_1 \text{ AND } E.\text{val}=E_0 \text{ AND } B.\text{val}=B_3 \) will cause a more complex processing of data filtering. It requires nested processing of Lowest Common Ancestor logic in this compound WHERE clause. Common ancestor node C is derived from D and E nodes. It is located under the common ancestor node A node which is derived from B node and C node’s sub result of D and E nodes. This requires nested common ancestor processing of C under A node that naturally follows correct hierarchical processing logic.

Regardless of the number of common ancestors involved, standard SQL operating hierarchically will perform this hierarchical logic perfectly thanks to the hierarchically restricted Cartesian product processing controlled by the WHERE clause. It generates the correct combination of row values to automatically emulate this type of hierarchical nested Lowest Common Ancestor processing a single row at a time. More information of SQL’s automatic use of Lowest Common Ancestor logic can be found in Section 8.2.

4.4.3) Multi-leg Variable Semantics with OR Logic

As a further example of the subtleties and complexities of hierarchical processing, the automatic hierarchical query internal semantic processing can become more powerful and complex with OR decision processing in the WHERE clause. This enhances and complicates the natural semantics of hierarchical data filtering processing, making it dynamically variable. Using the hierarchical structure from ViewX in Figure 4.4.3 below, what would the semantics of the query: \( \text{SELECT } D.\text{val}, E.\text{val} \text{ From ViewX WHERE } D.\text{val}=D_1 \text{ OR } E.\text{val}=E_0 \) ? If \( D.\text{val}=D_1 \) is true and \( E.\text{val}=E_0 \) is not, then every data occurrence of E and Y nodes under the qualified Lowest Common Ancestor C node occurrence is qualified (thanks to \( D.\text{val}=D_1 \) being true) and only the D node data occurrence with the qualified \( D.\text{val}=D_1 \) and its associated X node data occurrence qualifies since their qualification is more specific.
As you would expect from the data qualification results directly above in Figure 4.4.3, the opposite results occur if only E is true and D is not. In this case, every occurrence of D and X under the qualified lowest common ancestor C node occurrence is qualified and only the E node data occurrence that tested true and its associated Y data occurrence qualifies. If both conditions are true, then all occurrences of the D, X, Y, and E nodes qualify because of cross qualification. These results also demonstrate why it is easier to examine data filtering as data qualification, because both sides of the OR operation have an additive affect. While these output qualification semantics are complex, the results they produce are logical and intuitive to the user.

The WHERE clause OR processing semantics described above means that both sides of the OR condition must always be tested because the qualification semantics can dynamically change depending on which side of the OR operation is true. SQL handles this advanced hierarchical processing automatically with its Cartesian product processing that generates all combinations of hierarchical relations which automatically checks both sides of the OR operation. The semantic correctness of this operation can be logically verified by replacing the OR operation with separate filtering operations on two queries and unioning the results.

This sophisticated and intuitive variable hierarchical data filtering is performed naturally in relational SQL Cartesian product processing. This natural hierarchical processing is quite remarkable and meaningful for mathematically correct relational hierarchical processing. It reinforces and validates that the hierarchical processing known and used before relational processing became popular was and is correct. This is because it is the same as the hierarchical processing produced logically from relational processing operating under hierarchical relationships.

### 4.4.4 Single Path Data Filtering (Data Model Rules)

WHERE clause hierarchical data filtering is very powerful operating intuitively on the entire multi-leg structure, but you may still need a more restrictive (path only) type filtering very similar to XPath. This is where ON clause data filtering can be used. While the ON clause is used to specify the join condition, it can also include a data filtering condition. Being on the ON clause, data filtering only affects the join operation of the node it is specified on and the related lower level data nodes which can not exist without its parent data occurrence existence (causing a cascading delete). All other nodes on other legs are not affected in any way. As you probably realize, all ON clauses are processed before the WHERE clause is processed since the WHERE clause affects the entire structure which requires all the ON clauses to have been processed (at least logically).

The above differences in the operation of the WHERE and ON Clauses are extremely noteworthy and useful. Using the structure Department over Employee over Dependent for example, a qualification on a dependent under the age of twenty-one on a WHERE clause can cause entire structure occurrences to be
removed from the result when there are no data node occurrences that match the qualification. Placed on the ON clause of a Dependent node, this filtering can only remove the Dependent node occurrences it is used on, and any of its related descendent nodes. This is a cascading delete and an example of subordination. This is very useful for specifying a business rule that can be included in the hierarchical data model defined in the SQL view since it is better associated with the data model and not the specific query. Without this capability, employees with no dependents under twenty-one years old would be removed and if this causes departments with no employees, the departments would be removed and so on up the structure to the root. This is shown below in Figure 4.4.4.

```
SELECT Emp, Dpnd FROM Emp LEFT JOIN Dpnd ON EmpId=DpndEmpId…
WHERE DpndAge<18       ON…AND DpndAge<18          Join without filtering
↓
↓
Results:    Emp2  Dpnd2    Emp1   Null
           Emp2   Dpnd2
```

Figure 4.4.4 Difference between WHERE and ON clause data filtering

### 4.5) Order By for Hierarchical Data

The SQL Order By operation used in hierarchical processing needs to be restricted to ordering that is within the hierarchical order. This is because ordering out of the hierarchical order of the data being processed can easily change the structure inadvertently and cause invalid results if not detected. For example in the simple structure Dept over Emp, ordering Emp before Dept changes the structure, but ordering within Dept, or EMP, or both is OK. If ordering within Dept and Emp, Dept must be ordered before Emp.

Why does the hierarchical structure change when ordered in a different order than its natural structure? This happens for example when Department over Employee, a very standard 1 to M type structure, is ordered by Employee first. The resulting structure is inadvertently changed as shown below in Figure 4.5. Since Employee is given more significance than Dept, it can be assumed that it is now hierarchically above Dept. After performing this join, Employee would probably appear in a report above Department. Even if Dept were still treated as the higher level, the additional semantics added by the ordering would prevent the 1 to M structure from operating correctly. So ordering out of hierarchical order introduces many problems. Notice how Dept is out of order because of Emp’s ordering, forcing a need to be duplicated and inverted.

```
SQL * From DeptEmp
Order BY Emp, Dept
```

Figure 4.5 How ordering hierarchical data can change its structure
4.6) SQL Update from Hierarchical Data

With SQL, updating of relational databases is standard and is possible using data retrieved from hierarchical sources. This is accomplished easily using the standard SQL INSERT operation and retrieving the data from an SQL sub query that represents a hierarchical structure as described in this document. The desired data is pulled from a particular data path node occurrence navigated by the WHERE clause of the subselect.

![Diagram of DeptView hierarchy](image)

**INSERT INTO** Target (DeptID, EmpID, DpndID, Dpndname, DpndBD)
**SELECT** DeptNo, EmpNo, DpndNo, DpndName, DpndBday
**FROM** DeptView
**WHERE** DeptNo=124 and EmpNo=628 and DpndNo=456

**Figure 4.6 SQL Insert operation using a hierarchical data source**

The DeptView above in Figure 4.6 has been defined hierarchically in SQL as described in this document. To located 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 and the data up the path (identified in the SELECT list) are available to the Insert operation.

4.7) Namespace Capability

XML has its namespace capability to avoid data field naming conflicts caused by multiple fragments of documents pulled together from different documents. SQL has similar capabilities used for handling tables and views that are pulled together by being embedded in other views or in the query directly. In Figure 4.7, below there is a case where two invoice source objects need to be accessed together and they have the same named fields. SQL has three ways of qualifying the fields in the data source objects which can be tables or views. The example uses the “Cost” field as the field to be qualified to avoid ambiguity.

<table>
<thead>
<tr>
<th>Type of Qualification</th>
<th>SQL Usage of Qualification Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner Qualified:</td>
<td>SELECT Inv07.Cost, Inv09.Cost</td>
</tr>
<tr>
<td></td>
<td>FROM Inv07, Inv09</td>
</tr>
<tr>
<td></td>
<td>WHERE Inv07.Cost &gt; Inv09.Cost</td>
</tr>
<tr>
<td>Alias Qualified:</td>
<td>SELECT Purchased.Cost, Sold.Cost</td>
</tr>
<tr>
<td></td>
<td>FROM Inv07 AS Purchased, Inv09 AS Sold</td>
</tr>
<tr>
<td></td>
<td>WHERE Purchased.Cost &gt; Sold.Cost</td>
</tr>
</tbody>
</table>

**Figure 4.7 Types of SQL data name qualification**
From Figure 4.7 above, Owner Qualified is simply qualifying the data item by preceding it with the Table or View name it is to be associated with. The Alias Qualified method is similar to the Qualifier and Owner method except an SQL alias (known as a correlation name) is used. It renames the data source object and then the new name is used as in the Owner Qualified method.

4.7.1 Renaming Data Fields

Both of these Qualifications work, they have slightly difference limitations or advantages. The Owner Qualified method can not be used to reference multiple uses of the same table or view and the Alias use may offer advantages outside of qualification use. One thing may be apparent from the example in Figure 4.7 above, that the user may find this method verbose or difficult to use the qualification names. This can be avoided when used in embedded views by renaming the qualified names. These Select list data names can be assigned aliases as shown below in Figure 4.7.1. Their names now become “Input” instead of “Purchased.Cost” and “Output” instead of “Sold.Cost”. This renaming ability works with all three qualification methods.

```
CREATE VIEW EasyView AS
SELECT Purchased.Cost AS Input, Sold.Cost AS Output
FROM Purchased.Inv07, Sold.Inv09
WHERE Purchased.Cost>Sold.Cost
```

Figure 4.7.1 Renaming qualified data items

5) Other Hierarchical Based Capabilities

5.1) XML Joinless Access

While logical structures composed of relational tables use joining operations when retrieved, physical structures like XML do not need to simulate costly relational joins in the retrieval process when they are modeled by Outer Joins views. This is because the Left Outer Join hierarchical data modeling syntax for physical structures only represents the data structure metadata semantics. This means physical structures can be accessed and processed directly without joins because the hierarchical semantics of the Left Outer Join are reflected naturally in the physical hierarchical structures themselves. So they do not require expensive and needless joins to physically model their hierarchical structure. The rowsets can be built directly without joins as shown in Figure 5.1.
Query languages that simulate physical join views to integrate with relational data relationally will have join processing overhead for XML to support SQL/XML integration. But because the natural solution described in this document utilizes ANSI SQL’s hierarchical processing to perform SQL/XML integration at a hierarchical level, it can efficiently process physical hierarchical structures hierarchically. This allows constructing the hierarchical preserved rowset that models the Left Outer Join data modeling without performing expensive relational joins. This is not usually performed.

5.2) Natural Distributed Hierarchical Processing

The performing of distributed hierarchical processing automatically when distributed processing is performed is extremely powerful and simple. When the hierarchical data modeling Left Outer Joins are broken up and sent to remote sites for processing, the hierarchical substructures they represent will automatically be performed hierarchically. The returned results have been naturally processed hierarchical 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 5.2.

![Figure 5.2 Automatic hierarchical distributed processing](image)

5.3) Full Ad Hoc Nonprocedural Processing Supported

It is important to point out that all other SQL/XML integration solutions require proprietary solutions and XML centric syntax that is procedural. A side effect of this is that ad hoc processing is really not possible if allowed at all. The 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 completely ad hoc for the reasons listed in Figure 5.3.

<table>
<thead>
<tr>
<th>SQL Ad Hoc Ability</th>
<th>Other XML Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT list: Parameter driven</td>
<td>Selected items placed in code</td>
</tr>
<tr>
<td>FROM: Nonprocedural processing</td>
<td>Physical procedural coding</td>
</tr>
<tr>
<td>WHERE: Automatic nonlinear filtering</td>
<td>No automatic LCA processing</td>
</tr>
<tr>
<td>Hierarchical views: Flexible metadata</td>
<td>Less flexible physical views</td>
</tr>
</tbody>
</table>

![Figure 5.3 SQL ad hoc querying advantages](image)
SQL hierarchical processing maintains its pure parameter drive 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 hierarchical filtering without the need to specify complex hierarchical Lowest Common Ancestor (LCA) filtering logic. SQL hierarchical views further increase the ease of use and reuse.

5.4) Hierarchical Structure Construction Order

Hierarchal structures can generally be built or processed in any order without changing their semantics or result. This processing can be top-down, left to right, bottom-up, or in any combination. This is also true of the hierarchical structure’s data modeling definition. As shown in Figure 5.4, the same hierarchical structure can be modeled in a top-down, left to right, or bottom-up order which controls the order built. The completed hierarchical structure has the same semantics or derives the same results in whatever order it is built or processed. As an indication of SQL’s hierarchical processing capability and flexibility, all of these same capabilities and operational characteristics remain the same as hierarchical processing because the semantics are identical.

![Hierarchical structure construction order, down and up](image)

Generally speaking, hierarchical structures are most easily defined and built top-down as all the previous examples have intuitively demonstrated. Some automatic SQL data modeling definition or view expansion processes may change the standard data modeling order while remaining semantically the same. This can produce inefficiencies that cause throwaway data from dangling tuples (no matching row) that are avoided with top-down processing. This is avoided by rewriting the outer join for top-down processing at runtime.

6) SQL/XML Hierarchical Integration Opportunity

With ANSI SQL performing full hierarchical processing, it can naturally be used to integrate with hierarchical XML at a full nonlinear hierarchical level. This can be at a transparent level where SQL tables and XML elements both represent nodes in a nonlinear hierarchical structure.

6.1) XML ETL Integration with Hierarchical EII Processing

There are two basic ways for SQL to access XML hierarchical documents. The first is where XML is shredded into relational tables where they can be accessed as relational data. This is a batch method that accommodates large amounts of data (many documents) but the data can 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 flattened dynamically 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 and SQL EII for smaller amounts of XML accessed dynamically. SQL nonlinear hierarchical processing can not only access and process native XML dynamically, it can also model and access ETL shredded XML data and process it hierarchically. It can handle both at the same time in the same query as indicated in Figure 6.1 below. In this
way XML ETL and hierarchical SQL EII can be integrated seamlessly and 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.

Figure 6.1 The Integration of batch and dynamic XML hierarchical processing

6.2) Processing XML’s Unconventional Hierarchical Structures

XML hierarchical structures are not quite as fixed as conventional hierarchical structures. This is because XML is composed of semistructured data where their structure is defined in the data allowing it to change dynamically, define logical network data structures or even support unpredictable nesting for markup use. These present problems for relational processing. Relational solutions to these cases based on SQL hierarchical processing are described below. While it is nice to support all XML capabilities derived from its dynamically flexible structure definition, no XML processor is capability of supporting all possible capabilities. Each has their own strengths and limitations. For this reason, SQL should not be expected to support all XML capabilities as any other XML processor wouldn’t either.

XML defines hierarchical data structures, it does not specify in any way how they are to be processed. Even the SAX and DOM parsers sometimes interpret the same XML differently. There is no basic or solid principled foundation implied or associated with how the XML is hierarchically processed. XML also has two basic uses, database and markup and they require different strategies or processing. They require different processors.

XML structures sometimes do not conform to standard conventional hierarchical structures. We have already seen how they can resemble network structures and can be of variable structure which can be handled seamlessly. But structures in XML can also be formed where element types can appear in any order and element text can have elements inserted in the middle of the text. These are capabilities that are not useful for defining hierarchical data structures because they do not represent hierarchical data structures. These capabilities used together support text markup and are supported because XML was originally designed as a text markup language (the “M” in XML). XML as a hierarchical database storage format was actually an afterthought. The problem is that the two uses need to be treated differently. It is not useful to
parse markup into a database hierarchical data structure. Markup needs to be processed in its contiguous form and stored and acted upon as a single string field value which SQL can do in a function.

6.2.1) Variable Structure Generation

Lets examine XML unconventional database capabilities that make sense to support. Variable structures are hierarchical structures whose structure can change dynamically between different structure occurrences and even within a given structure occurrence. This is allowed in XML. SQL can support variable structures by using variable values in the ON clause that hierarchically model the structures differently depending on the value of the variable. Since separate ON clauses are used at each join point they can dynamically control whether the join at any point in the structure building process is performed or not. By having a test condition based on a higher level data value located higher on the current path occurrence (hierarchical data inheritance is supported in SQL), the variable generation of the data structure can be dynamically controlled. This is shown below in Figure 6.2.1 where either a D node or an E node data occurrence is generated, but not both, controlled by the value in the C.x data field. This is similar to COBOL’s Depending ON clause in its File Definition section. This is demonstrated below with an overly simple example, but can give you an idea of how this capability is supported and utilized.

![Figure 6.2.1 Variable structures controlled by data](image)

6.2.2) Mapping Network Structures to Hierarchical Structures

Some of XML’s advanced features require or create a logical network structure. Network structures unlike hierarchical structures allow a node type to be accessed from more than one path. This is demonstrated directly below in the XML IDRef Structure in Figure 6.2.2. This makes XML hierarchical structures ambiguous for nonprocedural, navigationless languages like SQL because a node can have multiple path entries each with its own semantics. A similar problem occurs with the use of duplicate element type nodes which is also permitted in XML and shown below. It is possible to model these network type structures as unambiguous hierarchical structures in SQL. SQL has an alias/rename ability that can enable a network to hierarchical structure mapping capability as shown below in Figure 6.2.2.

![Figure 6.2.2 Hierarchical mapping solutions for network structures](image)
While the underlying storage of the Addr node is different in both usages (shared or separate) above in Figure 6.2.2, the semantics of both structures are basically the same and can be mapped into the same remodeled hierarchical structure which is unambiguous as shown above.

7) Hierarchical SQL View Use and its Importance

Hierarchical outer join definitions of logical relational table structures and physical XML documents can be placed in standard SQL views. These ANSI SQL hierarchical views have unique capabilities that come together synergistically producing even more powerful capabilities making SQL hierarchical processing more powerful and very user friendly as described below.

7.1) Hierarchical View Advantages

Hierarchical SQL views naturally map logical relational and physical XML structures using the Left Outer Join operations. This makes hierarchical processing flexible, reusable, and intuitive because of these hierarchical views’ powerful hierarchical view abstraction. These hierarchical views can be embedded or hierarchically joined with other hierarchical SQL views. This is performed the same way tables are by using the Left Outer Join directly on views to naturally form larger hierarchical views as in FROM ViewX LEFT JOIN ViewY ON X.x=Y.y as discussed in Section 4.2.1. This joining of hierarchical views preserves and combines the hierarchical semantics and can even be performed dynamically. The views alias (correlation name) ability serves the same purpose as XML namespaces.

7.2) Hierarchical Query Optimization

The Left Outer Join’s natural hierarchical data preservation operation described earlier allows separate views or 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. This 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 7.2 where nodes B and D from ViewX are not accessed.

You can also see in Figure 7.2 that node C while not referenced is still required for navigating from node A to node E. Node C will be removed from the final result 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 can be easily performed by dynamically or logically removing the unneeded Left Outer Joins from the view at runtime as indicated in Figure 7.2 above. This optimization has been previously suggested in Universal Relation theory.
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 (single view feature), 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 powerful hierarchical optimization is not a replacement for the standard Inner or Outer join optimization. Nor does it have to be integrated into the current Inner or Outer join optimization. This powerful hierarchical optimization is simply applied before the standard optimization is performed. It can be applied externally and dynamically by rebuilding the query to eliminate the unneeded nodes.

This hierarchical optimization can also be applied to accessing physical hierarchical structures described Section 5.1. It can be used to indicate paths that can avoid database access. This use made of this will vary for the type of physical structures and their access mechanisms. Structures comprised of physical pointers like IBM’s IMS database versus nested contiguous structures like XML and Structured VSAM will use different access optimizations that can benefit from the hierarchical optimization described here.

### 7.3) Heterogeneous Unified Virtual Views

When all the specified hierarchical logical and physical SQL views expand at runtime they automatically form into a seamless and consistent unified heterogeneous virtual hierarchical view. Defined naturally in SQL, this takes on significant synergistic proportions making hierarchical processing very user friendly. When all views expand into a heterogeneous unified virtual view, it can also be hierarchically optimized.

![Diagram of expanded heterogeneous unified virtual view](Figure 7.3 Expanded heterogeneous unified virtual view)
8) More on the Internal Processing of Hierarchical SQL

There are a number of automatic powerful internal ANSI SQL operations that play an essential role in ANSI SQL hierarchical processing that have not been fully covered yet. This was done to keep the previous descriptions more easily understood. At this point these powerful features can now be examined.

8.1) Right Sided View Nesting

In section 4.2.2, Linking Below the Root, closer examination of the expanded SQL may clear up a possible perceived problem. The simplified example used below contains View1:A LEFT JOIN B ON A.a=B.b and View2: X LEFT JOIN Y ON X.x=Y.y. It joins them together with View1 over View2 with a link to View2 at a node below the root: ViewAB LEFT JOIN ViewXY ON B.Key=Y.key. The expanded view and new derived structure follows in Figure 8.1.

![Figure 8.1 Right sided view is nested](image)

The apparent problem with linking below the root of the lower level joined structure (ViewXY) is that it may appear in the unexpanded view that node Y node is linked to before it is joined to root X. This would be logically invalid since node X determines node Y. On a closer look at the expansion in Figure 8.1, you will notice that the LEFT JOIN between ViewAB and ViewXY is delayed because its matching ON clause has been pushed to the far side of ViewXY which contains its own ON clauses. This causes expanded ViewXY to be nested (right sided view nesting) which causes it to be fully performed in isolation before being joined to ViewAB. This means that all nodes of ViewXY are available to be referenced when joined to ViewAB. Any depth of nested views can be automatically and transparently handled this way. The SQL coder does not even need to be aware of this powerful nesting taking place.

Each nested view is allocated to a new and separate working set while being processed. This also protects all other working sets from any destructive operation the active working set may perform. This means for example that destructive Inner Joins can be performed in views without causing data to be discarded in other working sets. This is a very powerful capability that can be exploited in many ways and acts as a safety valve that keeps everything running smoothly with the SQL data modeling, adding powerful SQL recombinant capabilities. In addition, the ANSI outer join also maintains the specified join order. This is because with the data preservation of the outer join, changing the join order could change the results. This also offers more control for hierarchical processing.

Section 5.1, XML Joinless Access, explained why physical structures do not need to be accessed by performing joins. The physical structure processing operation defined by the Left Outer Join view also requires the natural processing of the Right Sided View Nesting. This is because a full hierarchical physical structure must be fully materialized before it can be processed below the root. The Right Sided View Nesting enables this to happen. While directly accessing the root node of a physical structure does not present a problem, hierarchically locating other nodes out of physically stored order can be a problem.
because of the dynamic navigation required of physical structures. This is why physical structures need to be materialized before being processed and the Right Sided Nesting does this automatically to make all nodes available at the time of the join. The access is out of order, but is necessary for the previous reason, to make all nodes of the view structure available. This does not cause problems for processing hierarchical structures, because hierarchical structures can be built in any order, bottom-up, top-down or even a mixture.

### 8.2) Advanced Lowest Common Ancestor Logic

The concept of Lowest Common Ancestor (LCA) introduced in Section 3.1 deserves closer examination in its intrinsic use in ANSI SQL hierarchical processing. LCA logic is used to help interpret and utilize the semantics between nodes in different legs of hierarchical structures. Finding the Lowest Common Ancestor of two nodes is not a trivial process and papers have put forth fast algorithms. Interestingly, SQL’s automatic and intrinsic use of LCA avoids this problem. Queries often reference more than two legs and this causes multiple LCAs. This will require more complex compound LCA logic which is performed automatically.

#### 8.2.1) Compound Lowest Common Ancestor Logic

The ANSI SQL hierarchical processing presented in this document is an inherent operation that was not intentionally designed into ANSI SQL. So if the LCA logic is necessary to hierarchical processing how is it being performed (spontaneously) if it was not designed or coded into ANSI SQL? The hierarchically restricted Cartesian product produced from a multi-table Left Outer Join that models the data structure automatically and naturally performs the logic of the LCA. It builds restricted Cartesian products under each Lowest Common Ancestor node at each hierarchical join point. This means the logic of determining the LCA for node pairs is naturally built into the Cartesian product processor of the relational processor.

In Figure 8.2.1 below, the predicate WHERE E=C would test all combinations of E=C generated under Lowest Common Ancestor node A, while WHERE F=G would test all combinations of F=G under Lowest Common Ancestor node C. The combination of WHERE E=1 AND F=G would still test the correct combinations of each LCA thanks to the hierarchically restricted Cartesian product and its underlying inherent LCA logic.

![Figure 8.2.1 Compound LCAs automatically form](image)

These LCA operations enable the WHERE and SELECT clauses hierarchical qualification to qualify a single row at a time which is required for relational processing and avoids a tree walking logic normally required for LCA logic. Lowest Common Ancestor (LCA) is also known as Nearest Common Ancestor (NCA) and Closest Common Ancestor. “Lowest” for lowest level common ancestor node in the structure, “Nearest” and “Closest” for the nearest/closest common ancestor node. They all derive the same LCA node. Since many scientific fields (Bio tech, AI, Vision) use hierarchical structures heavily, LCA processing is used there too.
8.2.2) Lowest Common Ancestor Logic Type 2

If we look deeper into SQL’s automatic processing of LCA logic, it turns out that there is another type of LCA logic at play we will call LCA logic Type 2. This is one where the SELECT and WHERE clauses take on the different legs that determine the LCA. Below in Figure 8.2.2 the Selected B node and the Where Clause derived C Node determine the LCA A node which controls the data qualification of the B node for data selection. The C node LCA was derived from the WHERE clause showing an example of nested LCA processing of Type 1 and Type 2 LCA use.

```
SELECT B FROM ViewX WHERE F=G
```

![Figure 8.2.2 Type 2 Lowest Common Ancestors](image)

The same Select data qualification was shown in section 4.4.1. The principles remain the same. This is a deeper exploration explaining why the basic hierarchical rules operate the way they do. Unlike WHERE clause Type 1 LCA’s which generate all data combinations under the LCA, SELECT lists Type 2 LCA’s do not generate a Cartesian product, the LCA is used to qualify all the data occurrences under it.

8.3) Logical and Physical Structure Processing Consistency

There is a hierarchical consistency that exists between logical and physical hierarchical structures, including between all the different types of physical hierarchical structures. The commonality is that the hierarchical structure maintains its basic principles and how it is logically operated upon irregardless of its makeup. The Left Outer Join syntax can model any conventional hierarchical structure and the associated semantics defines how it is operated on and the subsequent semantics of its result. This means all hierarchical structures can be defined in a global logical structure and accessed consistently by SQL as shown in Figure 8.3 for seamless heterogeneous processing.

The access of each different type hierarchical structure will require its own access routine use and the result would be converted to a relational rowset that defines the hierarchical structure modeled by its Left Outer Join operation as shown in Figure 8.3. The Left Outer Join hierarchical operation performed on tables does this automatically by preserving the hierarchical structure by inserting nulls to keep the variable length leg segments aligned. These same data modeling Left Outer Joins can model physical structures and their access routines will duplicate the Left Outer Join format in their returned rowset. In this way the entire unified heterogeneous hierarchical structure is seamlessly defined by the fully expanded SQL as shown in Figure 8.3. SQL operates consistently across all the joined rowsets because their make up is physically the same while their physical source may not have been.
These rowsets add relational flexibility that may not have been previously recognized. Section 4.3.2 demonstrated structure transformation. This may seem simple to expect of relational data, but transforming physical data may seem a lot more difficult, having to pull these different data formats apart, filter, and reassembling it. But since the data from logical and physical structures is now also in contiguous (at least logically) rowsets and strung together in the result set as in Figure 8.3, it can also be easily accessed and reassembled by SQL as was shown in Figure 4.3.2.

8.4) Nonlinear Internal Hierarchical Processing Navigation

Outside of relational hierarchical processing, nonlinear hierarchical data processing usually requires a combination of hierarchical processing and navigation. This internal processing is shown here to demonstrate how this is performed outside of relational processing on physical nonlinear hierarchical structures.

8.4.1) Single Leg Processing Navigation

To access a hierarchical structure, a series of Get First (GF) and Get Next (GN) operations are performed to access the hierarchical structure top to bottom. Get First and Get Next operate on nodes of the structure. Hierarchical navigation requires establishing the database positioning when accessing the hierarchical structure top-to-bottom. This means when accessing a node (except for the root), the parent node type must be already in position with its current data node occurrence. When accessing the first new occurrence of a node under its parent, a Get First is required, then Get Next is used to continue reading all occurrences under the established parent’s data node occurrence. When its data node occurrences are exhausted, the parent node’s next data occurrence is accessed, then the child node can be accessed again under its new parent data occurrence starting with a Get First followed by a series of Get Next operations shown below in Figure 8.4.1.
Figure 8.4.1 Navigating a single row of a hierarchical structure

Figure 8.4.1 above demonstrates how a single path in a database record can be read (navigated) sequentially. A process step has been included to indicate logical places to process the data collected. It is similar to rows of a relational table where only one data occurrence is available at a time.

8.4.2) Multiple Leg Processing Navigation

Navigating multiple paths in a hierarchical structures becomes more complex. Most nonlinear (multi-leg) hierarchical navigation systems support multiple positioning. This means that positioning in multiple paths can be maintained simultaneously and automatically. This allows accessing another leg without losing data positioning in other legs. This allows navigating multi-leg queries such as one that requests data from one leg of the structure based on data from another leg of the structure to be processed more easily across the different legs.

Figure 8.4.2 Navigating a multi-leg hierarchical structure
This multi-leg query logic requires LCA logic which can require Cartesian product type processing between legs (generating all combinations). Figure 8.4.2 performs this Cartesian product processing between the left and right leg shown below. All D’s are accessed every time a C node is read.

Nonlinear hierarchical query processing would use this type of internal hierarchical processing underneath. But interestingly SQL does not use this internal hierarchical processing because its Cartesian processing and natural modeling hierarchical processing manages to do the processing within SQL’s natural capabilities. On the other hand, the internal processing could be significantly improved with an actual internal hierarchical engine and the hierarchical capabilities could be improved with no hierarchical limitations. The SQL hierarchical syntax does make a powerful conceptual hierarchical nonprocedural query language. Its hierarchical syntax interpretation could be kept and the relational Cartesian product engine could be replaced seamlessly with a hierarchical engine.

8.5) Data Structure Extraction Technology

Almost all of the hierarchical processing capabilities are occurring automatically in ANSI SQL because of the Left Outer Join data modeling support. The ANSI SQL relational engine is not even aware of this hierarchical processing. This means it can not automatically extend this naturally capability to other areas such as seamless access to external hierarchical data sources to automatically take advantage of the natural hierarchical processing. The relational engine would also need to be cognitive of the dynamically modeled hierarchical structure being accessed to automatically format the data to hierarchically structured output data types like XML. The patented Data Structure Extraction (DSE) technology developed by Advanced Data Access Technologies can automatically derive the dynamically generated structure of the expanded unified hierarchical structure being processed and return it in an Edge Table format for easy processing as shown below in Figure 8.5. With this information, advanced capabilities can be naturally added to the SQL engine to extend its inherent nonprocedural hierarchical processing naturally and transparently to XML and legacy data as discussed in this document.

**Data Structure**

```
   A
  / \
 B   E
/ \ / \ /
C D F G
```

**Node Edge Table**

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Node No.</th>
<th>Hier Level</th>
<th>Parent No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>G</td>
<td>7</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8.5 Node edge table representing data structure
9.0 Advanced Hierarchical Topics

These advanced hierarchical topics are areas to be studied further for deriving new features and capabilities.

9.1) Network Structures

Networks were briefly covered in Section 2.1. Networks, from a hierarchical point of view, can have multiple paths to a node. This presents a problem for nonprocedurally querying this type of structure. This is because this structure is inherently ambiguous for nonprocedurally queries. Each path to a given node has a different semantic meaning. With nonprocedural 4GL navigationless access, there is no way to specify which path to use for this structure type. This is why procedural language and navigation is necessary to access and process network structures. Interestingly, SQL can also naturally model and process network structures also using Left Outer joins as it does for hierarchical structures. This might be an interesting capability in SQL to explore further.

<table>
<thead>
<tr>
<th>Network Structure</th>
<th>SQL Network Structure Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept</td>
<td>CREATE VIEW DeptView AS</td>
</tr>
<tr>
<td></td>
<td>SELECT * FROM Dept</td>
</tr>
<tr>
<td></td>
<td>LEFT JOIN Emp ON DeptKey=EmpFKey</td>
</tr>
<tr>
<td></td>
<td>LEFT JOIN Proj ON DeptKey=ProjFKey</td>
</tr>
<tr>
<td></td>
<td>LEFT JOIN Mgr ON MgrKey=EmpEKey</td>
</tr>
<tr>
<td></td>
<td>OR MgrKey=EmpPKey</td>
</tr>
</tbody>
</table>

Figure 9.1 Network structure definition using SQL

The SQL network definition in Figure 9.1 defines a hierarchical structure by joining the Mgr node (table) to both the Emp and Proj nodes using an OR logic operator. This gives access to the Mgr node from either the Emp path or the Proj path. SELECT Emp, Mgr FROM DeptView WHERE Emp EXISTS will still not always produce the desired result of always using the Emp path to get to Mgr. Just because Emp exists, it may not connect with Mgr while the Proj path does satisfying this query and returning EMP and MGR. On the other hand, SELECT Dept, Mgr FROM DeptView would be meaningful if it does not matter which path was taken.

Using an AND condition to create a network structure is less ambiguous but its network structure does cause implementation problems and more complex rules and semantics to work out.

9.2) Lowest Common Ancestors in XML with Duplicate Node Types

XML’s duplicate Node types were covered in Section 6.2.2 where it was shown how SQL could use them nonprocedurally in SQL. Other (still) academic research projects are experimenting with nonprocedural navigationless nonlinear hierarchical processing. They utilize multiple node types such as those shown in Figure 9.2 below. The problem is that duplicate node types have very unstable Lowest Common Ancestors that change frequently during a query and range from being very meaningful to not so meaningful. Most meaningful LCAs will be used when available during the query, but missing data occurrences can cause less meaningful LCAs to be used. When the LCA that would have qualified as the most meaningful has missing data occurrences, then the next strongest LCA will be used. This happens dynamically during execution so these LCAs are constantly changing. This may seem like a useful feature for Markup, but not for stable database access as the next paragraph demonstrates on the following figure 9.2.
Figure 9.2 above shows an example of a store that sells books and records. Common to both books and records are their titles and authors which are kept separately for each of them. To support nonprocedural queries like `SELECT Record WHERE Title="Advanced SQL"` the context of the query is used. This query makes sense because in the context of the query is for records making the Record node the most meaningful LCA. But if Title is missing under Record, the Store node becomes the next most meaningful LCA and the Title under book is used to qualify the query. This is usually not the intention of this database query. In addition, since Store is now the LCA, all Record data occurrences under this store occurrence are qualified and Selected.

10) SQLfX® Uses ANSI SQL’s Hierarchical Processing for XML

Advanced Data Access Technologies, Inc., ADAT, has utilized the inherent ANSI SQL hierarchical processing capabilities shown in this nonlinear hierarchical processing tutorial to support ANSI SQL transparent native XML Integration. None of the nonlinear hierarchical principles shown in this tutorial have been invented or designed, they are all naturally occurring hierarchical principles. As further proof, it was shown how ANSI SQL automatically and naturally reproduced these same hierarchical principles when nonlinear, multi-leg queries were modeled using the Left Outer Join operation. Left Outer Joins can naturally and precisely model nonlinear hierarchical structures using their semantics as defined in the ANSI SQL 1992 specification. ADAT’s ANSI SQL for XML product, SQLfX®, performs hierarchically from input, through processing, to XML structured output automatically and transparently. Input can be relational or XML data that is modeled hierarchically in SQL. SQL tables and XML elements are both treated as nodes in the hierarchical structure being processed hierarchically. The SQLfX® product actually utilizes a standard ANSI SQL processor as its hierarchical engine. This also has many advantages for the customer making SQLfX® truly transparent.

10.1) Actual SQLfX® examples

The SQLfX® product is in Beta testing and producing hierarchically correct and fully hierarchically structured XML results from examples that test out all the nonlinear hierarchical processing operations shown in this tutorial. Check out the SQLfX® Beta Example document at [http://www.adatinc.com/images/Verifying_SQLfX_Current.pdf](http://www.adatinc.com/images/Verifying_SQLfX_Current.pdf) to see actual examples which include multi-leg (LCA) queries, dynamic joins of hierarchical structures, structure transformations, and automatic structured XML output all from ANSI standard SQL processing.
Conclusion

ANSI SQL makes a very powerful, complete, and flexible nonprocedural 4GL nonlinear hierarchical querying language for relational and XML native data. It is unique because it automatically utilizes the semantics in the data structure to automatically process the most semantically complex multi-leg hierarchical queries correctly without extending the relational model. This means the user does not have to know the data structure or supply the hierarchical semantic processing logic, increasing the value of the data. The results are simultaneously hierarchically and relationally accurate making the result mathematically correct. This allows it to integrate logical or physical, and heterogeneous hierarchical structures transparently at a full interactive conceptual hierarchical processing level. SQL naturally performs hierarchical processing so accurately that it can be used as a model for determining correct hierarchical processing. This has already been done to determine and validate the semantics of linking below the root of the lower level structure when joining hierarchical data structures.