ANSI SQL Hierarchical Processing Can Fully Integrate Native XML
(as published in the ACM SIGMOD Record, March 2003)

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

Abstract

Most SQL-based XML vendor support is through interoperation and not integration. One reason for this is that XML is inherently hierarchical and SQL is supposedly not. This paper demonstrates how ANSI SQL along with its relational Cartesian product model can naturally perform complete and flexible hierarchical query processing. With this ANSI SQL inherent hierarchical processing capability, native XML data can be fully and seamlessly integrated into SQL processing and operated on at a full hierarchical level. This paper will describe the basic stages involved in this hierarchical SQL processing: hierarchical data modeling, hierarchical working set creation, and hierarchical Cartesian product processing. These processes enable a complete relational, XML, and legacy data integration which maintains ANSI SQL compatibility even while performing the most complex multi-leg hierarchical processing, and includes the dynamic, direct, and controlled hierarchical joining of hierarchical structures. Also covered are ANSI SQL hierarchical support features: hierarchical SQL views, hierarchical data filtering, and hierarchical optimization. These make standard SQL a well rounded and complete hierarchical processor. With this full hierarchical level of processing established, it will be shown how the relational Cartesian product engine can be seamlessly replaced with a hierarchical engine, greatly increasing processing and memory utilization, and enabling advanced XML hierarchical processing capabilities.

1 Introduction

SQL is in trouble today because XML is becoming ubiquitous, used increasingly by the Internet, and no SQL vendor has found a solution to seamlessly integrate native XML processing into SQL. All SQL-based XML integration approaches have had to resort to using non standard, proprietary methods making them all far from seamless and incompatible with each other. Basically, these proprietary methods shred XML documents into table rows and columns [5]. The processing is still performed relationally and not hierarchically. The hierarchical semantics in XML are not being utilized, causing the hierarchical semantics to be discarded. This unacceptable level of XML integration by SQL, one of the most popular and important database interfaces to the Internet, may signal the downfall of SQL with its likely replacement being W3C's XQuery. This would require a huge effort in training and re-coding. XQuery requires procedural-like coding adding significantly to the learning and coding effort.

2 SQL-based XML integration wish list

As stated above, SQL-based XML vendor support today is limited to processing XML documents by flattening them into a relational table format. True native integration has remained an unsolvable problem because relational data is flat while XML is structured. If true integration is possible, the following capabilities and features would be very desirable.

2.1 ANSI standard, non proprietary integration

SQL users desire standard open integrated systems that are implemented seamlessly and require little or no additional training. They want standardized systems which already operate in a known and trusted way, and are not going to disappear overnight.

2.2 Ability to directly join XML structures

Being able to directly join hierarchical data structures with full control over how they are hierarchically combined is one of the most powerful and useful XML integration capabilities. It is also a test for seamless XML integration because of the implementation difficulties. Joining hierarchical data structures require a SQL syntax and semantics which specify exactly how to join data structures hierarchically together. This process should support dynamic queries and preserve the semantics of all the involved data structures.

2.3 Hierarchical data processing

To utilize the hierarchical semantics contained in XML documents, hierarchical data processing is necessary. SQL processing needs to know the hierarchical structure of the information being processed and how to utilize it. This includes semantic interpretation of the SQL query as it relates to the hierarchical data structure being accessed. An example is selecting data from one leg of a hierarchical structure based on data in another leg of the structure. This has a complex hierarchical semantics involving both query and data structure semantics.

2.4 Process advanced hierarchical structures

The embedding of meta data along with the data found in XML allows for advanced capabilities such as variable data structures, network structures, and duplicate named elements in the structure. SQL hierarchical processing should also handle non procedurally indicated mechanical operations such as node collection and promotion.

2.5 What is not on the wish list

SQL is a non procedural data processing language and should not be expected to handle all of the textual capabilities made possible with XML. These involve textual transformation and processing which requires a more procedural type processing. This is what XQuery was designed to handle. However, SQL should process non procedural specified structural transformations.

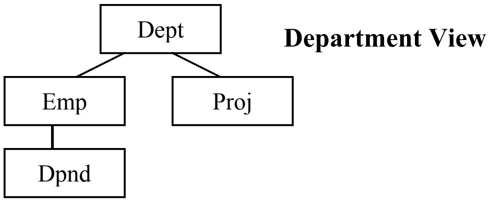
3 Standard SQL hierarchical processing

Standard SQL today contains all of the necessary capabilities to support full and complete hierarchical processing. SQL's hierarchical processing consists of three stages: hierarchical data modeling, hierarchical working set creation, and hierarchical Cartesian product processing. After completing the hierarchical data modeling stage, the hierarchical working set creation and hierarchical Cartesian product processing stages are automatically performed by the SQL engine. This sequence produces full hierarchical processing while at the same time observing valid relational processing.

3.1 Hierarchical data modeling

Hierarchical data modeling is specified naturally in SQL by the ANSI SQL Left Outer Join operation which inherently models hierarchical structures [2]. This Left Outer Join process operates left to right joining the left and right data argument values together. The resulting structure becomes the left argument to the following Left Outer Join operations which will each merge their right data argument in turn. At each join point, the left data argument is preserved even if there is no matching data, while the right data argument is not preserved if there is no data match. This behavior indicates that the left argument is hierarchically above the right argument because the left data argument can exist without a related right argument while the reverse is not true.

When the data modeling process described above is performed, the left data argument is combined into a unified hierarchical structure as it progresses left to right. The right data argument is hierarchically joined to the left data argument conforming to the Outer Join operation's On clause join criteria. The On clause is specified at each Outer Join point to specify how the two data arguments are hierarchically related [2]. These capabilities allow for any hierarchical structure to be modeled. Figure 3.1 demonstrates a data structure being modeled using the Left Outer Join syntax.



```
CREATE VIEW DeptView AS
   SELECT * FROM Dept
         LEFT OUTER JOIN Emp
                  ON DeptId=EmpDeptId
         LEFT OUTER JOIN Dpnd
                  ON EmpId=DpndEmpId
                       AND DpndAge<18
         LEFT OUTER JOIN Proj
                  ON DeptId=ProjDeptId
```

Figure 3.1 Hierarchical data modeling

The data modeling SQL in Figure 3.1 models the multi-level, multi-leg hierarchical data structure shown. The Left Outer Join operation controls the hierarchy of the data arguments and the On clause specifies the link points between the nodes in the data arguments, defining the hierarchical pathways. Multiple legs (paths) are created when the same upper level data node has been linked to multiple times as shown above with the Dept node. The SQL alias feature (not shown) can be used to include or model the same node type (name) in the structure multiple times. This is useful for defining network structures as hierarchical structures [1] and modeling XML IDREFs and duplicate named elements.

The On clause can also specify hierarchical filtering [1] which starts at its associated Left Outer Join's right argument's join node and affects only this node from its hierarchical structure position downward. An example is shown in Figure 3.1, where dependents over the age of eighteen will be excluded. The advantage of this hierarchical filtering is that removing all dependents for an employee will not also remove the employee. The On clause's pathway definition and its hierarchical filtering correspond very closely to XML's XPath operation.

3.2 Hierarchical working set creation

The semantics of the Left Outer Join operation controls how the relational working set is generated so that the row data values correspond to those in the hierarchical structure being modeled. In the Outer Join view in

Figure 3.1, this equates to employees and projects with no associated department being excluded, and the preserving of employees with no dependents. This hierarchical working set is shown in Figure 3.2.

DeptId	EmpId	DpndId	ProjId
DeptX	Emp1	Dpnd1	ProjX
DeptX	Emp2	Dpnd2	ProjX
DeptX	Emp1	Dpnd1	ProjY
DeptX	Emp2	Dpnd2	ProjY
DeptY	Emp3	NULL	ProjZ
DeptY	Emp3	NULL	ProjW

Figure 3.2 DeptView hierarchical working set

3.3 Hierarchical Cartesian product processing

The hierarchical working set shown in Figure 3.2 contains a restricted Cartesian product. It is restricted by the On clauses shown in Figure 3.1 to its hierarchically related combinations of data. This hierarchically related Cartesian product sets the final stage for processing. This final stage produces the hierarchical result set from the hierarchical working set. It utilizes the information gathered in the hierarchical working set for processing and Where clause data filtering. Unlike the On clause, the Where clause ranges over the entire record. This is also standard for hierarchical query processing.

Where clause filtering criteria applied to hierarchical structures can be quite powerful and complex. For example, selecting data from one leg of a hierarchical structure with filtering criteria based on another leg of the structure has a definite hierarchical and useful semantics. Based on the query shown in Figure 3.3 and its associated structure shown in Figure 3.1, it could be shown what dependents are in the same department as “ProjX”. The involved legs are related by their common ancestor node, Dept, and all dependents are selected under the qualifying common ancestor data occurrence of “DeptX” from the working set in Figure 3.2.

The above characteristics of the Cartesian product model allow the relational engine to apply complex Where clause filtering logic to the working set a single row at a time. How is it possible that the complex common ancestor hierarchical filtering logic can be determined a row at a time? This is possible because all of the necessary and valid hierarchical relationship combinations that make up the hierarchically related Cartesian product are represented in the working set shown in Figure 3.2. All the sibling combinations of related data automatically range under their qualified hierarchical ancestors because the relationships are hierarchical. This can be seen in Figure 3.3 which demonstrates such a multi-leg query applied against the hierarchical working set in Figure 3.2.

The query example in Figure 3.3 selects only rows with a ProjId of “ProjX” and outputs EmpId and DpndId values from a sibling leg under the common department node occurrence of “DeptX”. A selection based on “ProjY” would produce the same results accept for the ProjId of “ProjY”. This is possible because the data has been replicated hierarchically under the common ancestor data occurrence of “DeptX” which also has a project of “ProjY” as shown in Figure 3.2.

```
SELECT * FROM DeptView
   WHERE ProjId=”ProjX”
```

DeptId	EmpId	DpndId	ProjId
DeptX	Emp1	Dpnd1	ProjX
DeptX	Emp2	Dpnd2	ProjX

Figure 3.3 Multi-leg Where selection processing

The SQL query example in Figure 3.3 is a simple hierarchical query. A more complex SQL query could involve a data structure with many legs and many different common ancestor node types. The hierarchical Cartesian product working set for this query would have all the related hierarchical data combinations generated under each common ancestor node. This would still be handled equally well and automatically by the relational engine’s standard Cartesian product processing. This level of hierarchical processing by a relational processor may come as a surprise, but hierarchical processing is actually a subset of relational processing’s capabilities.

Relational processing can perform the most complex queries based on the data relationships specified and the relational Cartesian product engine will automatically match the semantics implied by the relationships. Depending on the type of relationships defined, the implied semantics may not always be logical or unambiguous but they will be performed as defined. Even network relationships can be defined [1] and processed. Hierarchically defined relationships are logical and unambiguous, producing logical and non ambiguous hierarchical results when processed by the relational Cartesian product engine.

The common ancestor Where clause filtering semantics and its processing logic becomes even more complex when the filtering criteria contains an Or operation. Normally with Or operations, if the first condition tests true, there is no need to test the second one. This is not true for processing hierarchical structures because multiple levels of qualification could cause the second condition to further qualify the result. This means that both conditions of the Or operation should be tested to insure the correct result. Figure 3.4 demonstrates this.

```
SELECT * FROM DeptView
   WHERE ProjId="ProjX"
         OR Emp="Emp3"
```

DeptId	EmpId	DpndId	ProjId
DeptX	Emp1	Dpnd1	ProjX
DeptX	Emp2	Dpnd2	ProjX
DeptY	Emp3	NULL	ProjZ
DeptY	Emp3	NULL	ProjW

Figure 3.4 Multi-leg Or logic

The query and result in Figure 3.4 shows that when the "ProjX" data occurrence condition is true, all the other leg occurrences under the qualified common ancestor occurrence "DeptX" qualify (these are "Emp1" and "Emp2"). The reverse situation is true when the "Emp3" data occurrence condition is true ("ProjZ" and "ProjW" qualify). When both sides of the Or operation are true, both sides will fully qualify. These are the correct hierarchical semantics. They will be performed automatically by the relational Cartesian product engine processing the working set shown in Figure 3.2 a single row at a time. The semantic correctness of these results can be proven by applying each side of the Or operation separately and unioning the results. The result will be semantically the same, proving this Or processing is valid. Most XML query processors can not handle this level of hierarchical processing non procedurally.

4 SQL hierarchical support capabilities

The SQL hierarchical processing described thus far does offer complete hierarchical processing, but SQL's inherent hierarchical processing does not stop here. There are other very useful and powerful hierarchical SQL support features that naturally compliment and extend SQL's inherent hierarchical processing. These are hierarchical SQL views and hierarchical optimization which are described below. They increase ease of use and efficiency, raising SQL's hierarchical processing to a first class level.

4.1 Hierarchical SQL views

Left Outer Joins that model hierarchical structures or portions of structures can be defined as standard SQL views which can be specified as substructures in Outer Join specifications that model hierarchical structures. This is shown in Figure 4.1. There are no limitations on these hierarchical views. They can be specified as the left or right data argument to Outer Join operations modeling hierarchical structures in the same manner as described previously in Section 3.1. This enables the full hierarchical joining of hierarchical data structures as shown in Figure 5.1 These hierarchically structured views can also be embedded to any depth. This natural hierarchical subview capability increases data abstraction and reuse significantly simplifying hierarchical processing.

Most notably, when these standard SQL hierarchical views naturally expand into a single homogenous SQL statement for processing, it precisely and accurately models the complete data structure. This automatically handles the combining of the representative data structures into a unified virtual hierarchical structure, performed naturally by standard SQL processing. This further supports and simplifies SQL's ability to naturally process hierarchical structures. Figure 4.1 shows a hierarchical Outer Join view expansion.

```
CREATE VIEW EmpView AS
   SELECT * FROM Emp  LEFT OUTER
       JOIN Dpnd ON EmpId=DpndEmpId
```

Embedded View:

```
SELECT * FROM Dept LEFT OUTER
  JOIN EmpView ON DeptId=EmpDeptId
```

View Expansion:

```
SELECT * FROM Dept LEFT OUTER
 JOIN LEFT OUTER JOIN Dpnd
        ON EmpId=DpndEmpId
 ON DeptId=EmpDeptId
```

Figure 4.1 Hierarchical SQL view usage

When Outer Join views expand, they automatically create right sided nesting which is demonstrated in Figure 4.1. The expanded view, EmpView, pushes the surrounding Outer Join's matching On clause to the right causing the current working set and its related Outer Join operation to be temporarily suspended during run time processing. This causes the expanded view to be performed using a new working set so that it does not adversely affect the working set(s) placed in suspension. When the expanded view operation completes, its working set naturally becomes the right data argument to the previous Outer Join operation placed in suspension. The expanded SQL syntax shown in Figure 4.1 is standard Outer Join syntax and correctly models the completed structure. Performed automatically, the SQL programmer is not aware of this nested operation.

What this nested Outer Join syntax does is insure that embedded views do not corrupt the data structure being constructed. For example, embedded Inner Join views would be destructive if they were not processed in this manner. This nested view feature allows symmetric join views used in modeling and constructing hierarchical structures to define a single logical node. This is possible because Inner Joins and Full Outer Joins being symmetric in operation model a flat structure and can be used to represent a single logical node in the hierarchical structure [1].

4.2 Hierarchical optimization

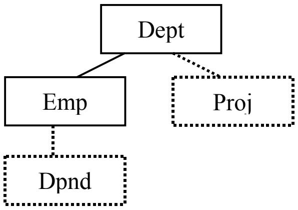
To insure view consistency, conventional Inner Join view materialization always accesses all data sources specified in the view regardless of what data is required. This is because any data source specified in the view can affect the result because of the way the Inner Join operation processes missing data. This has significant overhead, and often results in multiple tailored variations of views being defined for efficiency which defeats their purpose of reuse and data abstraction.

Outer Join views that model hierarchical structures can be optimized at query invocation to access only the data necessary for the current query [1]. This is because missing data is processed differently with Left Outer Joins and follows the semantics of hierarchical structures. Unlike Inner Joins, missing data outside the range of the query will not affect the query, and does not need to be accessed. Only required data and data on the path to required data needs to be accessed. This enables hierarchical views to be dynamically optimized at view invocation based on what data is necessary for the query being processed. This is shown in Figure 4.2 where the Dpnd and Proj nodes are temporarily excluded. In this way fewer alternative view definitions are necessary, which increases data abstraction and reuse, further simplifying hierarchical processing and greatly increasing efficiency.

Because the hierarchical optimization in Figure 4.2 has dynamically removed dependents and projects from the Outer Join view, they are not accessed. This also means that the project replications that can be seen in the working set in Figure 3.2 are not present to cause the unnecessary replications of EmpId values that would have been present with an Inner Join operation.

SQL vendors have yet to take advantage of hierarchical optimization because they are either not aware of it or they mistakenly believe it does not follow the ANSI specification. The ANSI SQL specification defines the Outer Join operation in terms of a simulation using Inner Joins. This presents a problem when it is used as a model to implement the Outer Join operation. This is because it will unnecessarily access every data source in an Outer Join view to take into account the affect of missing data described earlier. True Outer Join operations are not influenced by missing data and do not need to test for missing data in a view.

SELECT EmpId FROM DeptView



Working Set

DeptId	EmpId
DeptX	Emp1
DeptX	Emp2
DeptY	Emp3

Result

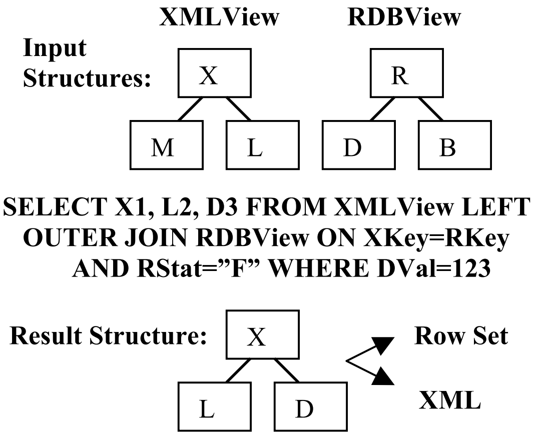
EmpId
Emp1
Emp2
Emp3

Figure 4.2 Hierarchical view optimization

5 XML and legacy data support

As described in section 4.1, SQL views can be used to define hierarchical structures which can be joined naturally with other hierarchical views into a unified hierarchical structure. To support the heterogeneous processing of hierarchical data such as XML, these SQL structured views can represent logical or physical hierarchical data sources. The work on XML-Related Specifications (SQL/XML) [3] consisting of SQL/XML mappings and XML Select list functions can be utilized in these views also. These SQL structured views will enable seamlessly access to the hierarchical data source, returning row set data that exactly matches the Outer Join specification modeling it in the view. This makes the support of XML and other legacy data sources completely seamless as shown in Figure 5.1.

Hierarchical structured views can be used at three levels. These levels are physical, logical, and external. Physical hierarchical views define physical hierarchical data structures such as XML and legacy data using the Left Outer Join. They can be defined automatically from their data definitions. Logical hierarchical views are made up of physical views, logical views, and Left Outer Joins allowing for maximum flexibility and data abstraction. The external view is the topmost SQL specification used to invoke the query. It can be comprised of logical views, physical views, and Left Outer Joins. This external specification can be specified dynamically for ad hoc processing which can include the hierarchical joining of data structures. All three of these view levels are demonstrated in Figure 5.1. They all use the standard Outer Join data modeling SQL, so they automatically expand seamlessly into a single seamless SQL specification that exactly models the combined hierarchical structure. This greatly simplifies heterogeneous access and assures seamless operation.



```
SELECT X1, L2, D3 FROM XMLView LEFT
OUTER JOIN RDBView ON XKey=RKey
AND RStat="F" WHERE DVal=123
```

Figure 5.1 ANSI SQL-based XML integration

6. Full native hierarchical support

At this point it has been shown how relational, XML, and hierarchical legacy data sources can be accessed and seamlessly processed at a full hierarchical processing level directly in standard SQL. With this capability established, SQL's relational engine could be seamlessly extended by making it aware of the hierarchical data structure being processed. In this way it can utilize the hierarchical semantics in the data structure to improve SQL's hierarchical processing. Some examples are hierarchical optimizations, improved structured input and output, and support for XPath expressions. The process needed to extract the structure meta data from data modeling SQL has been developed by Advanced Data Access Technologies, Inc.

There is a significant innovation that can be used when queries are limited to hierarchical structures. Since SQL hierarchical processing is operating seamlessly under ANSI standard SQL syntax and semantics, the relational Cartesian product model and engine could be replaced seamlessly with a true hierarchical engine. This would greatly increase the memory and processing efficiency avoiding Cartesian product explosions and processing [4], and extend the hierarchical processing capabilities. These capabilities include unlimited multi-leg data ordering, avoiding flattening hierarchical data on input, joins performed hierarchically, and increasing the accuracy and efficiency when producing a relational row set or fully structured XML document on output.

With a hierarchical engine powering SQL, XML's irregular structures and semistructured capabilities can be supported. These include recursive structures, variable structures, and node collection. These advanced XML capabilities may require non standard SQL syntax additions, but this is tempered because they operate based on ANSI SQL's inherent hierarchical processing capability. This makes these additions more seamless, efficient, and easily accepted.

7 Conclusion

This paper has identified SQL's inherent and greatly under utilized hierarchical processing capabilities and shown how they combine synergistically to perform unsurpassed hierarchical query processing. This enables SQL to naturally and seamlessly integrate native XML and legacy data without the use of proprietary language constructs. This is shown with the standard SQL query in Figure 5.1 which seamlessly performs the following hierarchical capabilities described in this paper and remains consistent with ANSI SQL's specifications:

- Physical and logical data structure modeling
- Hierarchical multi-leg data structure processing
- Hierarchical Where clause filtering
- Hierarchical structure view support
- Native XML hierarchical integration
- Dynamic hierarchical joining of data structures
- Hierarchical access optimization
- Hierarchical node promotion
- Hierarchical data filtering (RStat="F")
- Hierarchical engine can process this query
- Result has hierarchical semantics preserved

With SQL's inherent hierarchical processing capability fully utilized, SQL's future will continue to look bright. And the Internet will have a standard well known SQL interface that interfaces at a hierarchical level that can take full advantage of XML.

References

[1] M. David. *Advanced ANSI SQL Data Modeling and Structure Processing*. Artech House Publishers, 1999.

[2] M. David. Advanced Capabilities of the Outer Join. ACM SIGMOD Record, Vol. 21, No.1, March 1992.

[3] A. Eisenberg, J Melton. SQL/XML is Making Good Progress. SIGMOD Record, Vol. 31, No. 2, June 2002.

[4] M. Fernandez, A. Morishima, D. Suciu. Efficient Evaluation of XML Middle-ware Queries. Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, May 2001.

[5] J. Shanmugasundaram et al. A General Technique for Querying XML Documents using a Relational Database System. SIGMOD Record, Vol. 30, No.3, Sept. 2001